# Pattern Lattice Traversal by Selective Jumps

Osmar R. Zaïane Mohammad El-Hajj
Department of Computing Science, University of Alberta Edmonton, AB, Canada
{zaiane, mohammad}@cs.ualberta.ca

## ABSTRACT

Regardless of the frequent patterns to discover, either the full frequent patterns or the condensed ones, either closed or maximal, the strategy always includes the traversal of the lattice of candidate patterns. We study the existing depth versus breadth traversal approaches for generating candidate patterns and propose in this paper a new traversal approach that jumps in the search space among only promising nodes. Our leaping approach avoids nodes that would not participate in the answer set and reduce drastically the number of candidate patterns. We use this approach to efficiently pinpoint maximal patterns at the border of the frequent patterns in the lattice and collect enough information in the process to generate all subsequent patterns.

## 1. INTRODUCTION

Discovering frequent patterns is a fundamental problem in data mining. Many efficient algorithms have been published in this area in the last 10 years. The problem is by no means solved and remains a major challenge, in particular for extremely large databases. The idea behind these algorithms is the identification of a relatively small set of candidate patterns, and counting those candidates to keep only the frequent ones. The fundamental difference between the algorithms lies in the strategy to traverse the search space and to prune irrelevant parts. For frequent itemsets, the search space is a lattice connecting all combinations of items between the empty set and the set of all items. Regardless of the pruning techniques, the sole purpose of an algorithm is to reduce the set of enumerated candidates to be counted. The strategies adopted for traversing the lattice are always systematic, either depth-first or breadth-first, traversing the space of itemsets either top-down or bottom-up. Among these four strategies, there is never a clear winner, since each one either favors long or short patterns, thus heavily relying on the transactional database at hand. Our primary motivation here is to find a new traversal method that neither favors nor penalizes a given type of dataset, and at the same time allows the application of lattice pruning for the minimization of candidate generation. Moreover, while discovering frequent patterns can shed light on the content and trends in a transactional database, the discovered patterns can outnumber the transactions themselves, making the analysis of the discovered patterns impractical and even useless. New attempts toward solving such problems are made by finding the set of frequent closed itemsets (FCI) [8, 10, 11]. A frequent itemset $X$ is closed if and only if there is no $X'$ such that $X$ is a subset of $X'$ and $X'$ is contained in every transaction containing $X$. Finding only the closed item patterns reduces dramatically the size of the result sets without losing relevant information. Closed itemsets reduce indeed the redundancy already in the set of frequent itemsets. From the closed itemsets one can derive all frequent itemsets and their counts. Directly discovering or enumerating closed itemsets can lead to huge time saving during the mining process. But in cases where databases are made of long frequent patterns, finding the set of closed itemsets is not always feasible. In such cases finding the set of maximal frequent patterns is the alternative [2, 3, 5, 6], where a frequent itemset is said to be maximal if there is no other frequent itemset that subsumes it. Frequent maximal patterns are a subset of frequent closed patterns, which are a subset of all frequent patterns. While we can derive the set of all frequent itemsets directly from the maximal patterns, their support cannot be obtained without counting with an additional database scan. Nonetheless, discovering maximal patterns has interesting significance, especially in pattern clustering applications where frequent patterns are important, not their exact support. Formally the problem is stated as follows: Let $I = \{i_1, i_2, ...i_m\}$ be a set of literals, called items and $m$ is considered the dimensionality of the problem. Let $\mathcal{D}$ be a set of transactions, where each transaction $T$ is a set of items such that $T \subseteq I$. A transaction $T$ is said to contain $X$, a set of items in $I$, if $X \subseteq T$. An itemset $X$ is said to be *frequent* if its *support* $s$ (i.e. ratio of transactions in $\mathcal{D}$ that contain $X$) is greater than or equal to a given minimum support threshold $\sigma$. A frequent itemset $\mathcal{M}$ is considered maximal if there is no other frequent set that is a superset of $\mathcal{M}$. A frequent itemset $X$ is is said to be closed if and only if there is no $X'$ such that $X \subseteq X'$ and the support of $X$ equals the support of $X'$. Clearly the set of maximal patterns is subsumed by the set of closed patterns, which is in turn subsumed by the set of all frequent patterns. However, from the set of closed itemsets, one can directly determine all frequent patterns with their respective supports, but the maximal patterns can only help

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*KDD'05* August 21–24, 2005, Chicago, Illinois, USA
Copyright 2005 ACM 1-59593-135-X/05/0008 ...$5.00.

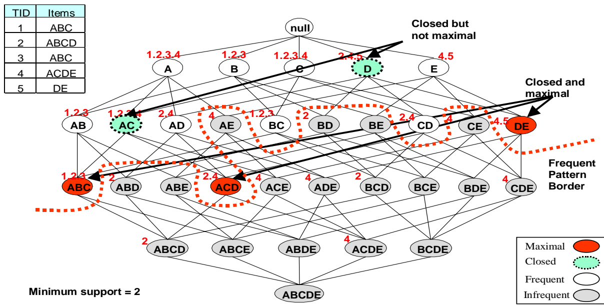


**Figure 1: Lattice with frequent pattern border**

enumerate the set of all frequent patterns without hinting to their exact support counts. What is relevant here is the efficient counting of some specific itemsets.

Figure 1 presents the pattern lattice of a token example with a dataset of 5 transactions made of 5 frequent items using an absolute support threshold of 2. The frequent pattern border can be drawn to separate frequent from non-frequent patterns, where all non-frequent patterns lie below that border. Closed and maximal patterns are also identified on that figure. Maximals are directly above the border.

In this paper a new traversal approach for the itemset lattice, called leap-traversal, is proposed. We show that using this traversal we can find the set of maximal patterns, and collect enough information in the process to derive subsequent patterns, closed and all, without having to recount.

## 2. TRAVERSAL APPROACHES

Existing algorithms use either breadth-first-search or depth-first-search strategies to find candidates that will be used to determine the frequent patterns. Breadth-first-search traverses the lattice level-by-level: where it uses frequent patterns at level k to generate candidates at level k+1 before omitting the non-frequent ones and keeping the frequent ones to be used for the level k+2, and so on. This approach usually uses many database scans, and it is not favored while mining databases that are made of long frequent patterns. When traversing the same lattice as in Figure 1 using a breadth-first strategy, frequent 1-itemsets are first generated, then used to generate longer candidates to be tested from size two to above. In our token example, this approach would test 18 candidates to finally discover the 13 frequent ones. Five were unnecessarily tested. On the contrary depth-first-search tries to detect the long patterns at the beginning and only back-tracks to generate the frequent patterns from the long ones that have already been declared as frequent. For longer patterns, depth-first-search indeed outperforms the breadth-first method. But in cases of sparse databases where long candidates do not occur frequently, the depth-first-search is shown to have poor performance. Using the depth-first approach with the same lattice as in Figure 1, 23 candidates are tested, 10 of them unnecessarily.

It is true that many algorithms have been published for enumerating and counting frequent patterns, and yet all algorithms still use one of the two traversal strategies (depth-first vs. breadth-first) in their search. They differ only in their pruning techniques and structures used. No work has been done to find new traversal strategies, such as greedy ones, or best first, etc. We need a new greedy approach that jumps in the lattice searching for the most promising nodes and based on these nodes it would generate the set of all frequent patterns.

### 2.1 Leap Traversal Approach: Candidate Generation vs. Maximal generation

Most frequent itemset algorithms follow the candidate generation first approach, where candidate items are generated first and only the candidate with support higher than the predefined threshold are declared as frequent while others are omitted. One of the main objectives of the existing algorithms is to reduce the number of candidate patterns. In this work, we propose a new approach to traverse the search space for frequent patterns that is based on finding two things: the set of maximal patterns, and a data-structure that encodes the support of all frequent patterns that can be generated from the set of maximal frequent patterns. Since maximal patterns alone do not suffice to generate the subsequent patterns, the data structure we use keeps enough information about frequencies to counter this deficiency. The basic idea behind the leap traversal approach is that we try to identify the frequent pattern border in the lattice by marking some particular patterns (called later path bases). Simply put, the marked nodes are those representing complete sub-transactions of frequent items. How these are identified and marked will be discussed later. If those marked patterns are frequent, they belong to the border (i.e. they are potential maximal) otherwise their subsets could be frequent, and thus we jump in the lattice to patterns derived from the intersection of infrequent marked patterns in the anticipation of identifying the frequent pattern border. The intersection comes from the following intuition: if a marked node is not a maximal, a subset of it should be maximal. However, rather than testing all its descendants, to reduce the search space we look at descendent's of two non-frequent marked nodes at a time, hence the pattern intersection. The process is repeated until all currently intersected marked patterns are frequent and hence the border is found. Before we explain the Leap-Traversal approach in detail, let us define the Frequent-Path-Bases (FPB). Simply put, these are some particular patterns in the itemset lattice that we mark and use for our traversal. An FPB if frequent could be a maximal. If infrequent, one of its subsets could be frequent and maximal. A frequent-path-base for an item $A$, called $A$-Frequent-Path-Base, is a set of frequent items that has the following properties:

1. At maximum one $A$-FPB can be generated from one transaction.
2. All frequent items in an $A$-frequent path base have support greater than or equal to the support of $A$;
3. Each $A$-FPB represents items that physically occur in the database with item $A$.
4. Each $A$-FPB has its branch-support, which represents the number of occurrences for this $A$-FPB in the database exactly alone (i.e. not as subset of other FPBs). In other words, the branch support of a pattern is the number of transactions that consist of this pattern, not the transactions that include this pattern along with other frequent items. The branch support is always less or equal to the support of a pattern.

As an example for the leap-traversal, assuming we have an oracle to generate for us the Frequent-Pattern-Bases from

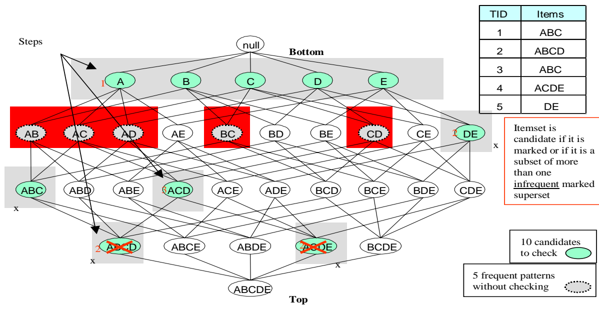


**Figure 2: Leap-Traversal**

the token database in Figure 1 , Figure 2 illustrates the process. With the initial FPBs ABC, ABCD, ACDE and DE (given by our hypothetical oracle) we end-up testing only 10 candidates to discover 13 frequent patterns, two were tested unnecessarily (ABCD and ACDE) but 5 patterns were directly identified as frequent without even testing them: AB, AC, AD, BC, CD. From the initially marked nodes, ABC and DE are found frequent, but ABCD and ACDE are not. The intersection of those two nodes yields ACD. This newly marked node is found frequent and thus maximal. From the maximals ACD, DE and ABC, we generate all the subsequent patterns, some even without testing (AB, AC, AD, BC and CD). The supports of these patterns are calculated from their superset FPBs. For example, AC has the support of 4 since ABC occurs (alone) twice, ABCD and ACDE occur each alone once.

Frequent pattern bases that have support greater than the predefined support (i.e. frequent patterns) are put aside as they are already known to be frequent and all their subsets are also known to be frequent. Only infrequent ones participate in the leap traversal approach, which consists of intersecting non-frequent FPBs to find a common subset of items shared among the two intersected patterns. The support of this new pattern is found as follows without revisiting the database: the support of $Y$ where $Y = FPB1 \cap FPB2$, is the summation of the branch support of all FPBs that are superset of $Y$. For example if we have only two frequent path bases ABCD: 1, and ABEF: 1, by intersecting both FPBs we get AB that occurs only once in ABCD and once in ABEF which means it occurs twice in total. By doing so, we do not have to traverse any candidate pattern of size 3 as we were able to jump directly to the first frequent pattern of size 2, which can be declared de-facto as a maximal pattern. Hence the name leap-traversal. Consequently, all its subsets are also frequent, which are A and B with support of 2 as they occur only once in each of the frequent path bases.

The Leap-Traversal approach starts by building a lexicographic tree of intersections among the Frequent-Pattern-Bases FPBs. It is a tree of possible intersections between existing FPBs ordered in a lexicographic manner. The size of this tree is relatively big as it has a depth equal to the number of FPBs. It is also unbalanced to the left since intersection is commutative. The number of nodes in a lexicographic tree equals to $\sum_{i=1}^{n}\binom{n}{i}$ where $n$ is the number of Frequent Pattern Bases. It is obvious that the more FPBs we have, the larger the tree becomes. Thus, pruning this tree plays an important role for having an efficient algorithm.

Four pruning techniques can be applied to the lexicographic tree of intersections. These pruning strategies can be explained by the following theorems:

**Theorem 1:** $\forall X, Y \in FPBs$ ordered lexicographically, if $X \cap Y$ is frequent then there is no need to intersect any other elements that have $X \cap Y$., i.e, all children of $X \cap Y$ can be pruned. **Proof:** $\forall A, X, Y \in FPBs$, $A \cap X \cap Y \subset X \cap Y$. If $X \cap Y$ is frequent then $A \cap X \cap Y$ is also frequent (apriori property) as a subset of a frequent pattern is also frequent.
**Theorem 2:** $\forall X, Y, Z, W \in FPBs$ ordered lexicographically, if $X \cap Y = X \cap W$ and $Y \ll W$ (i.e $Y$ is left of $W$ in the lexicographic tree) then there is no need to explore any children of $X \cap Y$. Since $Z$ is left of $W$ (or equal to $W$) in the lexicographical order, all children of $X \cap Y$ will also be children of $X \cap Z$ or $X \cap W$. **Proof:** To prove this theorem we need to show that any children of $X \cap Y$ are repeated under another pattern $X \cap Z$ that always exists. Since $X \cap Y = X \cap W$, then $X \cap Y \cap Z = X \cap Z \cap W$ (intersection is commutative) and $X \cap Z \cap W$ always exists in the lexicographic tree of intersections because of the order. Then, we can prune $X \cap Y$.
**Theorem 3:** $\forall X, Y, Z \in FPBs$ ordered lexicographically, if $X \cap Y \subset X \cap Z$ then we can ignore the subtree $X \cap Y \cap Z$. **Proof:** Assume we have $X, Y, Z \in FPBs$, Since $X \cap Y \subset X \cap Z$ then $X \cap Y \cap Z = X \cap Y$. This means we do not get any additional information by intersecting $Z$ with $X \cap Y$. Thus, the subtree under $X \cap Y$ suffices.
**Theorem 4:** $\forall X, Y, Z \in FPBs$, if $X \cap Y \supset X \cap Z$ then we can ignore the subtree of $X \cap Z$ as long $X \cap Z$ is not frequent. **Proof:** following the proof of Theorem 3 we can conclude that $X \cap Z$ is included in $X \cap Y$.
**Lemma 1:** At each level of the lexicographic tree of intersections, consider each item as a root of a new subtree:
(A) Intersect the siblings for each node with the root
(B) If a set exists and it is not frequent then we can prune that node. **Proof:** Assume we have $X, Y, Z \in FPBs$, if $X$ is a parent node then if $X \cap Y \cap Z$ exists and is not frequent then any superset for this intersected node is also not frequent (apriori property) that is why any intersection of $X$ with any other item is also not frequent.

## 2.2 Heuristics used for building and traversing the lexicographic tree

**Heuristic 1:** The lexicographic tree of intersections of FPBs needs to be ordered. Four ways of ordering could be used which are: order by support, support branch, pattern length, and random. Ordering by support yields the best results, as intersecting two patterns with high support in general would generate a pattern with higher support than intersecting two patterns with lower support. Ordering the tree by assigning the high support nodes at the left increases the probability of finding early frequent patterns in the left and by using Theorem 1, a larger subtree can be pruned.

**Heuristic 2:** The second heuristic deals with the traversal of the lexicographic tree. The breadth-traversal of the tree is better than the depth-traversal. This observation can be justified by the fact that the goal of the lattice Leap-Traversal approach is to find the maximal patterns, which means finding longer patterns early is the goal of this approach. Thus, by using the breadth-first approach on the intersection tree, we detect and test the longer patterns early before applying too many intersections that usually lead to smaller patterns.

## 3. TREE STRUCTURES USED

The Leap-Traversal approach we discuss consists of two main stages: the construction of a Frequent Pattern tree (HFP-tree); and the actual mining for this data structure by building the tree of Intersected patterns.

### 3.1 Construction of the Frequent Pattern Tree

The goal of this stage is to build the compact data structure called Frequent Pattern Tree, which is a prefix tree representing sub-transactions pertaining to a given minimum support threshold. This data structure compressing the transactional data was contributed by Han et al. in [7]. The tree structure we use, called HFP-tree is a variation of the original FP-tree. However, we will start introducing the original FP-tree before discussing the differences with our data structure. The construction of the FP-tree is done in two phases, where each phase requires a full I/O scan of the database. A first initial scan of the database identifies the frequent 1-itemsets. After the enumeration of the items appearing in the transactions, infrequent items with a support less than the support threshold are weeded out and the remaining frequent items are sorted by their frequency. This list is organized in a table, called header table, where the items and their respective support are stored along with pointers to the first occurrence of the item in the frequent pattern tree. The actual frequent pattern tree is built in the second phase. This phase requires a second complete I/O scan of the database. For each transaction read, only the set of frequent items present in the header table is collected and sorted in descending order according to their frequency. These sorted transaction items are used in constructing the FP-tree. Each ordered sub-transaction is compared to the prefix tree starting from the root. If there is a match, the support in the matched nodes is simply incremented, otherwise new nodes are added for the items in the suffix of the transaction to continue a new path, each new node having a support of one. During the process of adding any new item-node to the FP-Tree, a link is maintained between this item-node in the tree and its entry in the header table. The header table holds one pointer per item that points to the first occurrences of this item in the FP-Tree structure.

Our tree structure is the same as the FP-tree except for the following differences. We call this tree Headerless-Frequent-Pattern-Tree or HFP-tree.

1. We do not maintain a header table, as a header table is used to facilitate the generation of the conditional trees in the FP-Growth model;
2. We do not need to maintain the links between the same itemset across the different tree branches;
3. The links between nodes are bi-directional to allow top-down and bottom-up traversals of the tree;
4. All leaf nodes are linked together as the leaf nodes are the start of any pattern base and linking them helps the discovery of frequent pattern bases;
5. In addition to *support*, each node in the HFP-tree has a second variable called *participation*. *Participation* plays a similar role in the mining process as the *participation* counter in the COFI-tree [4].

Basically, the support represents the support of a node, while participation represents, at a given time in the mining process, the number of times the node has participated in already counted patterns. Based on the difference between the two variables, *participation* and *support*, the special patterns called *frequent-path-bases* are generated. These are simply the paths from a given node $x$, with participation smaller than the support, up to the root (i.e. nodes that did not fully participate in frequent patterns yet). Algorithm 1 shows

**Algorithm 1** HFP-Leap: Leap-Traversal with Headerless FP-tree

```
Input: D (transactional database); σ (Support threshold); Type (Maximal, Closed or All).
Output: Type patterns with their respective supports.

Scan D to find the set of frequent 1-itemsets F1
Scan D to build the Headerless FP-tree HFP
FPB ← FindFrequentPatternBases(HFP)
Maximals ← FindMaximals(FPB, σ)
if Type ==Maximal then
  Output Maximals
else
  Patterns ← GeneratePatterns(FPB, Maximals, Type)
  Output Patterns
end if
```

**Algorithm 2** FindFrequentPatternBases: Marking nodes in the lattice

```
Input: HFP (Headerless FP-Tree).
Output: FPB (Frequent pattern bases with counts)

ListNodesFlagged ← ∅
Follow the linked list of leaf nodes in HFP
for each leaf node N do
  Add N to ListNodesFlagged
end for
while ListNodesFlagged ≠ ∅ do
  N ← Pop(ListNodesFlagged) {from top of the list}
  fpb ← Path from N to root
  fpb.branchSupport ← N.support - N.participation
  for each node P in fpb do
    P.participation ← P.participation + fpb.branchSupport
    if P.participation < P.support AND ∀c child of P,
    c.participation = c.support then
      add P in ListNodesFlagged
    end if
  end for
  add fpb in FPB
end while
RETURN FPB
```

the main steps in our approach. After building the Headerless FP-tree with 2 scans of the database, we mark some specific nodes in the pattern lattice using *FindFrequentPatternBases*. Using the FPBs, the leap-traversal in *FindMaximals* discovers the maximal patterns at the frequent pattern border in the lattice. *GeneratePatterns* is called to produce all subsequent patterns (closed or all) if needed using the information collected along with the FPBs (i.e. their branch support).

Algorithm 2 shows how patterns in the lattice are marked. The linked list of leaf nodes in the HFP-tree is traversed to find upwards the unique paths representing sub-transactions. If frequent maximals exist, they have to be among these complete sub-transactions. The participation counter helps reusing nodes exactly as needed to determine the frequent path bases.

Algorithm 3 is the actual leap-traversal to find maximals. It starts by listing some candidate maximals stored in *PotentialMaximals* which is initialized with the frequent pat-

tern bases that are frequent. All the non-frequent FPBs are used for the jumps of the lattice leap-traversal. These FPBs are stored in the list *List* and intermediary lists *NList* and *NList2* will store the nodes in the lattice that the intersection of FPBs would point to, in other words, the nodes that may lead to maximals. The nodes in the lists have two attributes: *flag* and *startpoint*. For a node $n$, *flag* indicates that a subtree in the intersection tree should not be considered starting from the node $n$. For example, if node $(A \cap B)$ has a flag $C$, then the subtree under the node $(A \cap B \cap C)$ should not be considered. For a given node $n$, *startpoint* indicates which subtrees in the intersection tree, descendants of $n$, should be considered. For example, if a node $(A \cap B)$ has the startpoint $D$, then only the descendent's $(A \cap B \cap D)$ and so on are considered, but $(A \cap B \cap C)$ is omitted. Note that $ABCD$ are ordered lexicographically. At each level in the intersection tree, when *NList2* is updated with new nodes, the theorems are used to prune the intersection tree. In other words, the theorems help avoid useless intersections (i.e. useless maximal candidates). The same process is repeated for all levels of the intersection tree until there are no other intersections to do (i.e. *NList2* is empty). At the end, the set potential maximals is cleaned by removing subsets of any sets in *PotentialMaximals*.

It is obvious in the Leap-traversal approach that superset checking and intersections plays an important role. We found that the best way to work with this is by using the bit-vector approach where each frequent item is represented by one bit in a vector. In this approach, intersection is nothing but applying the OR operation between two vectors, and subset checking is nothing but applying the AND operation between two victors. If $A \cap B = A$ then A is a subset of B.

## 3.2 Closed and All frequent patterns

The main goal of the Leap-traversal approach is to find the set of maximal patterns. From this set we can generate all the subset patterns where all subsets are nothing but the set of all frequent patterns, and some of them are the set of closed patterns. The only challenge in this process is to compute the support of these patterns. The computation for the support for these patterns is encoded in the branch support of the existing FPBs already generated from the HFP-tree, where the support of any generated pattern is the summation of the branch support of all its supersets of FPBs.

As can be seen in Algorithm 4, all relevant patterns are generated from the set of maximals. Using the definition of maximals, all subsets of a maximal are de facto frequent. Once their support is computed using the branch support of FPBs as described above, pinpointing closed patterns is simply done using the definition of closed itemsets (i.e. no other frequent pattern subsumes it and has the same support).

# 4. RELATED WORK

There is a plethora of algorithms proposed in the literature to address the issue of discovering frequent itemsets. The most important, and at the basis of many other approaches, is *apriori* [1]. The property that is at the heart of *apriori* and forms the foundation of most algorithms simply states that for an itemset to be frequent all its subsets have to be frequent. This anti-monotone property reduces the candidate itemset space drastically. However, the generation of candidate sets, especially when very long frequent patterns exist, is still very expensive. Moreover, *apriori* is heavily I/O bound. Another approach that avoids generating and testing itemsets is FP-Growth [7]. FP-Growth generates, after only two I/O scans, a compact prefix tree representing all sub-transactions with only frequent items called FP-tree. However, mining the FP-tree using FP-Growth strategy requires significant memory, and large databases quickly blow out the memory stack.

**Algorithm 3** FindMaximals: The actual leap-traversal

```
Input: FPB (Frequent Pattern Bases); σ (Support threshold).
Output: Maximals (Frequent Maximal patterns)

List ← FPB; PotentialMaximals ← ∅
for each i in List do
  Find support of i {using branch supports}
  if support(i) > σ then
    Add i to PotentialMaximals
    Remove i from List
  end if
end for

Sort List based on support
NList ← List; NList2 ← ∅
∀i ∈ NList initialize i.flag ← NULL AND i.startpoint ← index
of i in NList
while NList ≠ ∅ do
  for each i in NList do
    g ← Intersect(i, j) {where j ∈ List AND i ≪ j (in lexico-
    graphic order) AND not j.flag}
    g.startpoint ← j; Add g to NList2
  end for

  for each i in NList2 do
    Find support of i {using branch supports}
    if support(i) > σ then
      Add i to PotentialMaximals
      Remove all duplicates or subsets of i in NList2; Remove
      i from NList2
    else
      Remove all duplicates of i in NList2 except the most
      right one ; Remove i from NList2
      Remove all non frequent subsets of i from NList2
      if ∃j ∈ NList2 AND j ⊇ i then
        i.flag ← j
      end if
      for all j in List do
        if j ≫ i.startpoint (in lexicographic order) then
          n ← Intersect(i, j)
          Find support of n {using branch supports}
          if support(n) < σ then
            Remove i from NList2
          end if
        end if
      end for
    end if
  end for
  NList ← NList2; NList2 ← ∅
end while
Remove any x from PotentialMaximals if (∃M ∈
PotentialMaximals AND x ⊂ M)
Maximals ← PotentialMaximals
RETURN Maximals
```

MaxMiner [2], GenMax [5], & MAFIA [3] are state-of-the-art algorithms that specialize in finding frequent maximal patterns. MaxMiner is an apriori-like algorithm that performs a breadth-first traversal of the search space. At the same time it performs intelligent pruning techniques to eliminate irrelevant paths of the search tree. A look-ahead

strategy achieves this, where there is no need to further process a node if it, with all its extensions, is determined to be frequent. To improve the effectiveness of the superset frequency pruning, MaxMiner uses a reorder strategy. MAFIA, which is one of the fastest maximal algorithms, uses many pruning techniques such as the look-ahead used by the MaxMiner, checking if a new set is subsumed in another existing maximal set, and other clever heuristics. GenMax is a vertical approach that uses a novel strategy, progressive focusing, for finding supersets. In addition, it counts supports faster using diffsets [12].

Finding the set of closed frequent patterns has also been extensively studied. A-Close [8] is an *apriori*-like algorithm. This algorithm mines directly for closed frequent itemsets. It uses a breadth-search strategy. This algorithm is one-order of magnitude faster than *apriori*, when mining with a small support. This algorithm shows, however, poor performance compared to *apriori* when mining with high support especially when we find a small set of frequent patterns as it consumes most of its computation power in computing the closure of itemsets. CLOSET+ [10] is an extension of the FP-Growth algorithm. It builds recursively conditional trees that cause this algorithm to suffer when mining for low support thresholds. CHARM [11] uses a vertical representation of the database. It adopts the diffset technique to reduce the size of intermediate tidsets.

**Algorithm 4** GeneratePatterns: Extending the maximals

**Input:** $FPB$ (Frequent pattern bases); $Maximals$ (Set of frequent maximals); *Type* (Closed or All).
**Output:** *Patterns* (Either closed or all frequent patterns with their supports)

```
for each M in Maximals do
  FP ← Generate all sub-patterns of M
end for
for each p in FP do
  p.support = ∑ ∀X.branchSupport {Where X ∈ FBS AND
  p ⊂ X}
  if Type ==Closed then
    if not(p.support == g.support AND g ∈ FP AND g ⊃ p)
    then
      add p in Patterns
    end if
  else
    add p in Patterns
  end if
end for
RETURN Patterns
```

# 5. PERFORMANCE EVALUATIONS

To evaluate our leap-traversal approach, we conducted a set of different experiments. First, to measure its effectiveness compared to depth-first and breadth-first, we evaluated the number of candidates generated by all the three methods at different support levels and on different datasets. Second, we compared our method with FP-Growth and FP-MAX in terms of efficiency when discovering all frequent patterns. Since the strategy of our leap-traversal approach is to first discover the maximal patterns before generating all patterns, we also compared our algorithm with other state-of-the-art algorithms solely to discover those maximal patterns, in terms of speed, memory usage and scalability.

The contenders we tested against are MAFIA [3], FP-MAX [6] and FP-Growth [7]. The implementations of these algorithms were all provided to us by their original authors. All our experiments were conducted on an IBM P4 2.6GHz with 1GB memory running Linux 2.4.20-20.9 Red Hat Linux release 9. We have tested these algorithms using both real and synthetic datasets.

## 5.1 Number of generated candidates

To better differentiate between our novel leap-traversal method and the other typical traversal methods, breadth-first and depth-first, we conducted a set of experiments to count the number of candidate itemsets generated. Generating candidates and counting their support is what takes time and what all algorithms are trying to minimize. A candidate pattern is any pattern that the algorithm generates and counts even if it turns out later to be infrequent. To do so, we implemented three versions of lattice traversal: breadth-first, depth-first (without any look-ahead strategy) and leap traversal. We tested many databases to study the effect of changing the traversal strategies. The effect is relative to the length of the frequent patterns, and thus, we report the results with a UCI dataset [9], namely *chess*, with relatively long frequent patterns and one with mixed size frequent patterns, *T10I4D100K*. In most cases using the leap traversal approach reduces drastically the number of generated candidate patterns.

In Figure 3.A, with the *Chess* real dataset, we can see that the leap traversal approach was able to generate almost 10 times less candidate patterns than the depth-search and almost 14 times less than the breadth-search strategies. At 70% support, both depth and breadth-first strategies generate one order of magnitude more candidates than necessary, while our leap traversal method generates almost as many candidates as the actual number of frequent patterns. Many were unnecessarily generated, but many more were directly (and correctly) generated without checking and counting. Similar discriminating results using the *T10I4D100K* dataset can be seen in Figure 3.B. The Leap-traversal approach considers significantly less candidates to effectively find exactly the same frequent patterns as its rival methods. Since the patterns are very small in size, breadth-first can stop early and thus need not generate too many candidates.

| Support % | # of frequents | Generated Candidates | | | Directly Generated Frequent patterns by Leap | Unnecessarily generated patterns | | |
|---|---|---|---|---|---|---|---|---|
| | | Breadth | Depth | Leap | | Breadth | Depth | Leap |
| 95 | 78 | 165 | 90 | 33 | 66 | 87 | 12 | 21 |
| 90 | 628 | 2768 | 1842 | 70 | 591 | 2140 | 1214 | 33 |
| 85 | 2690 | 20871 | 9667 | 380 | 2571 | 18181 | 6977 | 261 |
| 80 | 8282 | 91577 | 49196 | 630 | 8056 | 83295 | 40914 | 404 |
| 75 | 20846 | 292363 | 160362 | 1100 | 20357 | 271517 | 139516 | 611 |
| 70 | 48939 | 731740 | 560103 | 1780 | 48041 | 682801 | 511164 | 882 |

( A )

| Support % | # of frequents | Generated Candidates | | | Directly Generated Frequent patterns by Leap | Unnecessarily generated patterns | | |
|---|---|---|---|---|---|---|---|---|
| | | Breadth | Depth | Leap | | Breadth | Depth | Leap |
| 0.75 | 561 | 637 | 745 | 204 | 462 | 76 | 184 | 105 |
| 0.5 | 1073 | 1867 | 1620 | 1009 | 912 | 794 | 547 | 848 |
| 0.025 | 7703 | 37369 | 36684 | 19940 | 7101 | 29666 | 28981 | 19338 |
| 0.01 | 27532 | 265020 | 458650 | 175265 | 25223 | 237488 | 431118 | 172956 |
| 0.005 | 53385 | 1130154 | 4924098 | 1026095 | 41330 | 1076769 | 4870713 | 1014040 |

(B)

**Figure 3: Candidate generated using 3 approaches**

## 5.2 Efficiency Evaluation

The goal in the following experiments is to measure the advantage of first discovering maximals in order to discover all frequent itemsets. FP-Growth finds directly all frequent patterns, while our HFP-Leap uses the discovered maximals

and the branch supports of FPBs to derive all frequent patterns. To have a fair comparison, we also tried to find the maximals with FPMAX and derive all frequent patterns and count their supports from the in memory FP-tree structure. We refer to this version of the algorithm as FPMAX+. The times reported in Figure 4.A include the time to generate maximals, for HFP-Leap and FPMAX+, as well as the time needed to derive all frequent patterns with their respective supports. The experiment reported here was using a synthetic dataset with 50K transactions using 1000 items with and average of 36 items per transaction (i.e. relatively dense). As can be seen in Figure 4.A, HFP-Leap outperforms FP-Growth and FPMAX+ by a large margin. FPMAX+ discovers all frequent patterns faster than FP-Growth despite the fact that the supports of all patterns still needed to be computed. This shows the advantage of initially finding maximals. Unfortunately since maximals are a "lossy" representation of frequent patterns FPMAX+ still has to compute supports resorting to the FP-tree representation. Keeping extra relevant information within the frequent pattern bases, gives an edge to HFP-Leap.

## 5.3 Scalability

Since our algorithm starts by finding maximal patterns and has an advantage over its contenders like FP-Growth by collecting extra information to generate subsequent patterns, we conducted experiments for discovering exclusively the maximal patterns on UCI and synthetic datasets. For lack of space we could not presents all the results we came upon. In summary, while mining small synthetic datasets, we found that the HFP-Leap algorithm outperforms algorithms such as MAFIA and GENMAX by as much as two orders of magnitudes. The difference between FPMAX and HFP-Leap was not always clear, as each one of them was a winner in many cases and a runner up in others. To Distinguish the subtle differences between both strategies, we conducted our experiments on extremely large datasets. In these series of experiments we used three synthetic datasets made of 5M, 25M, and 50M transactions, with a dimension equals to 100K items, an average transaction length equals to 24. All experiments were conducted using a support of 0.05%. In mining 5M transactions both algorithms show similar performance: HFP-Leap finished its work in 320 second while FPMAX finished in 375 seconds. At 25M transactions the difference starts to increase. In the final test, mining a transactional database with 50M transactions, HFP-Leap discovered all patterns in 1980 seconds while FPMAX finished in 2985 seconds. The results, averaged on many runs, are depicted in Figure 4.B. From these experiments we see that the difference between FPMAX and HFP-Leap while mining synthetic datasets becomes clearer once we deal with extremely large datasets as HFP-Leap saves at least one third of the execution time compared to FPMAX. This is due to the reduction in candidate checking.

## 5.4 Memory Usage

We have also tested the memory usage by FPMAX, MAFIA and HFP-Leap. In many cases we have noticed that HFP-Leap consumes one order of magnitude less memory than both FPMAX and MAFIA. Figure 4.C illustrates a sample of the experiments that we conducted where the transaction size, the dimension and the average transaction length are respectively 1000K, 5K and 12. The support was varied from 0.1% to 0.01%. This memory usage is due to the fact that HFP-Leap generates the maximal patterns directly from its HFP-tree. Also the intersection tree is never physically built. FPMAX, however, uses a recursive technique that keeps building trees for each frequent item tested and thus uses much more memory.

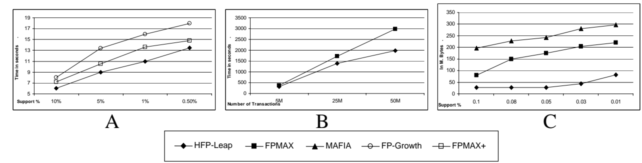


**Figure 4: A. Advantage of finding maximals first. B. Scalability with very large datasets. C. Memory usage.**

# 6. CONCLUSION

We presented a new way of traversing the pattern lattice to search for pattern candidates. The idea is to first discover maximal patterns and keep enough intermediary information to generate from these maximal patterns all types of patterns with their exact support. Our new lattice traversal approach dramatically minimized the size of candidate list because it selectively jumps within the lattice toward the frequent pattern border. It also introduces a new method of counting the supports of candidates based on the supports of other candidate patterns, namely the branch supports of FPBs. Our performance studies show that our approach outperforms the state of the art methods that have the same objective: discovering maximal and all patterns by, in some cases, two orders of magnitude. This algorithm shows drastic saving in terms of memory usage as it has a small footprint in the main memory at any given time.

# 7. REFERENCES


[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB*, 1994.

[2] R. J. Bayardo. Efficiently mining long patterns from databases. In *ACM SIGMOD*, 1998.

[3] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *IEEE ICDE*, 2001.

[4] M. El-Hajj and O. R. Zaïane. Inverted matrix: Efficient discovery of frequent items in large datasets in the context of interactive mining. In *ACM SIGKDD*, 2003.

[5] K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *IEEE ICDM*, pages 163–170, 2001.

[6] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *FIMI*, 2003.

[7] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM SIGMOD*, 2000.

[8] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT*, 1999.

[9] Uci machine learning repository. http://www.ics.uci.edu/~mlearn/MLRepository.html.

[10] J. Wang, J. Han, and J. Pei. Closet+: Searching for the best strategies for mining frequent closed itemsets. In *ACM SIGKDD*, 2003.

[11] M. Zaki and C.-J. Hsiao. ChARM: An efficient algorithm for closed itemset mining. In *SIAM SDM*, 2002.

[12] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. In *ACM SIGKDD*, 2003.