Bifold Constraint-Based Mining by Simultaneous Monotone and Anti-Monotone Checking

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

Abstract

Mining for frequent itemsets can generate an overwhelming number of patterns, often exceeding the size of the original transactional database. One way to deal with this issue is to set filters and interestingness measures. Others advocate the use of constraints to apply to the patterns, either on the form of the patterns or on descriptors of the items in the patterns. However, typically the filtering of patterns based on these constraints is done as a post-processing phase. Filtering the patterns post-mining adds a significant overhead, still suffers from the sheer size of the pattern set and loses the opportunity to exploit those constraints.

In this paper we propose an approach that allows the efficient mining of frequent itemsets patterns, while pushing simultaneously both monotone and anti-monotone constraints during and at different strategic stages of the mining process. Our implementation shows a significant improvement when considering the constraints early and a better performance over Dualminer which also considers both types of constraints.

1 Introduction

Frequent Itemset Mining (FIM) is a key component of many algorithms which extract patterns from transactional databases. For example, FIM can be leveraged to produce association rules, clusters, classifiers or contrast sets. This capability provides a strategic resource for decision support, and is most commonly used for market basket analysis. One challenge for frequent itemset mining is the potentially huge number of extracted patterns, which can eclipse the original database in size. In addition to increasing the cost of mining, this makes it more difficult for users to find the valuable patterns. Introducing constraints to the mining process helps mitigate both issues. Decision makers can restrict discovered patterns according to specified rules. By applying these restrictions as early as possible, the cost of mining can be constrained. For example, users may be interested in purchases whose total price exceeds \$100, or whose items cost between \$50 and \$100.

Constraint based mining is an ongoing area of research. Two important categories of constraints are *monotone* and *anti-monotone* [13]. Anti-monotone constraints are constraints that when valid for a pattern, they are consequentially valid for any subset subsumed by the pattern. Monotone constraints when valid for a pattern are inevitably valid for any superset subsuming that pattern. The straightforward way to deal with constraints is to use them as a filter post-mining. However it is more efficient to consider the constraints during the mining process. This is what is refereed to as "*pushing the constraints*" [14]. Most existing algorithms leverage (or push) one of these types during mining and postpone the other to a post-processing phase. New algorithms such as Dualminer apply both types of constraints at the same time. [7]. It considers these two types of constraints in a double process, one mirroring the other for each type of constraint, hence its name. However, monotone and anti-monotone constraints do not necessarily apply in duality. Especially when considering the mining process as a set of distinct phases, such as the building of structures to compress the data and the mining of these structures, the application of these constraints differ by type. Moreover, some constraints have different properties and should be considered separately. For instance, minimum support and maximum support are intricately tied to the mining process itself while constraints on item characteristics, such as price, are not. There is no existing algorithm that pushes both types of constraints early in the mining process and neither traverses the lattice of patterns top-down nor bottom-up. We introduce herein an algorithm that pushes both monotone and anti-monotone constraints by wisely jumping several levels in the pattern lattice from the bottom and top, and cleverly reducing the unnecessary constraint checking while considering the intricacies and properties of the constraints and the patterns sought after.

1.1 Problem Statement

The problem of mining association rules over market basket analysis was introduced in [1, 2]. The problem consists of finding associations between items or itemsets in transactional data. Formally, the problem is stated as follows: Let $I = \{i_1, i_2, ...i_m\}$ be a set of literals, called items. Each item is an object with some predefined attributes such as price, weight, etc. 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 $X \subseteq T$. A constraint ζ is a predicate on itemset X that yields either *true* or *false*. An itemset X satisfies a constraint ζ if and only if $\zeta(X)$ is *true*. An itemset X has a *support s* in the transaction set $\mathcal{D}$ if *s%* of the transactions in $\mathcal{D}$ contain X. Two particular constraints pertain to the support of an itemset, namely the *minimum support* constraint and the *maximum support* constraint. An itemset X is said to be *infrequent* if its *support* s is smaller than a given minimum support threshold σ; X is said to be *too frequent* if its *support* s is greater than a given maximum support Σ; and X is said to be *large* or *frequent* if its *support* s is greater or equal than σ and less or equal than Σ.

1.2 Motivation and Contribution

The problem of discovering all frequent itemsets that satisfy constraints is a difficult one. The difficulty stems from the fact that, firstly, testing for minimum support and maximum support can not be done simultaneously, since when valid, one is always true for subsets while the other is always true for supersets. Secondly, despite their selective power, some constraints cannot be checked to filter candidate itemsets until a very late stage of the mining process depending upon the type of constraint and the search space traversal strategy used.

We introduce a frequent itemset mining algorithm with the following properties:

- A *leap traversal* strategy is used to apply constraints from selected nodes in the lattice, in contrast to bottom-up or top-down traversals.
- Both monotone and anti-monotone constraints are pushed efficiently by placing and timing their respective evaluation strategically.
- Regions where one constraint needs not be evaluated are identified quickly using proven theorems.
- Previously known data structures, such as FP-tree [12] and COFI-tree [10], are used, but new algorithms exploiting these structures are proposed.
- Constraints are used not only to extract the valid frequent itemsets but also concurrently to obtain the valid frequent closed and maximal patterns along with their respective supports.

The remainder of the paper is organized as follows: The relevant types of constraints, monotone and anti-monotone, are discussed in Section 2 with illustrative examples. Section 3 presents a new algorithm for frequent itemset mining using the COFI-tree idea but instead of a bottom-up or top-down approach, selectively jumps within the pattern lattice to find those patterns that satisfy the minimum support threshold [17]. How to push both monotone and anti-monotone constraints and where these constraints are evaluated in this new approach is presented in Section 4. A selection from a battery of tests for performance evaluation is presented in Section 5. In particular, we compare our approach to Dualminer [7]. Section 6 presents related work. Finally, Section 7 concludes the paper.

MONOTONE	ANTI-MONOTONE
$min(S) \leq v$	$min(S) \geq v$
$max(S) \geq v$	$max(S) \leq v$
$count(S) \geq v$	$count(S) \leq v$
$sum(S) \geq v(\forall a \in S, a \geq 0)$	$sum(S) \leq v(\forall a \in S, a \geq 0)$
$range(S) \geq v$	$range(S) \leq v$
$support(S) \leq v$	$support(S) \geq v$

Table 1. Commonly used *monotone* and *anti-monotone* constraints

2 Constraints

It is known that algorithms for discovering association rules generate an overwhelming number of those rules. While many new efficient algorithms were recently proposed to allow the mining of extremely large datasets, the problem due to the sheer number of rules discovered still remains. The set of discovered rules is often so large that it becomes useless. Different measures of interestingness and filters have been proposed to reduce the discovered rules, but one of the most realistic ways to find only those interesting patterns is to express constraints on the rules we want to discover. However, filtering the rules post-mining adds a significant overhead and misses the opportunity to reduce the search space using the constraints. Ideally, dealing with the constraints should be done as early as possible during the mining process.

2.1 Categories of Constraints

A number of types of constraints have been identified in the literature [13]. In this work, we discuss two important categories of constraints – *monotone* and *anti-monotone*.

Definition 1 (*Anti-monotone constraints*)
A constraint ζ is *anti-monotone* if and only if an itemset X violates ζ, so does any superset of X. That is, if ζ holds for an itemset S then it holds for any subset of S.

Many constraints fall within the *anti-monotone* category. The minimum support threshold is a typical anti-monotone constraint. As an example, $sum(S) \leq v(\forall a \in S, a \geq 0)$ is an *anti-monotone* constraint. Assume that items A, B, and C have prices \$100, \$150, and \$200 respectively. Given the constraint ζ = ($sum(S) \leq \$200$), then since itemset AB, with a total price of \$250, violates the ζ constraint, there is no need to test any of its supersets (e.g. ABC) as they also violate the ζ constraint.

Definition 2 (*Monotone constraints*)
A constraint ζ is *monotone* if and only if an itemset X holds for ζ, so does any superset of X. That is, if ζ is violated for an itemset S then it is violated for any subset of S.

An example of a *monotone* constraint is $sum(S) \geq v(\forall a \in S, a \geq 0)$. Using the same items A, B, and C as before, and with constraint ζ = ($sum(S) \geq 500$), then knowing that ABC violates the constraint ζ is sufficient to know that all subsets of ABC will violate ζ as well.

Table 1 presents commonly used constraints that are either *anti-monotone* or *monotone*. From the definition of both types of constraints we can conclude that *anti-monotone* constraints can be pushed when the mining-algorithm uses the bottom-up approach, as we can prune any candidate superset if its subset violates the constraint. Conversely, the *monotone* constraints can be pushed efficiently when we are using algorithms that follow the top-down approach as we can prune any subset of patterns from the answer set once we find that its superset violates the *monotone* constraint.

Algorithm 1 COFILeap: Leap-Traversal with COFI-tree

```
  Input: $D$ (transactional database); $\sigma$ (Support threshold).
  Output: Type patterns with their respective supports.
  $F1 \leftarrow$ Scan $D$ to find the set of frequent 1-itemsets
  $FPT \leftarrow$ Scan $D$ to build the FP-tree using $F1$
  $GlobalMaximals \leftarrow \emptyset$
  for each item $I$ in Header($FPT$) in increasing support do
    $LF \leftarrow FindlocalFrequentWithRespect(I)$
    if Not $(I \cup LF) \subseteq GlobalMaximals$ then
      $ICT \leftarrow$ Build COFI-Tree for $I$
      $FPB \leftarrow$ FindFrequentPathBases($ICT$)
      $LocalMaximals \leftarrow Frequent(FPB)$
      $InFrequentFPB \leftarrow notFrequent(FPB)$
      for each pair $(A, B) \in InFrequentFPB$ do
        $Pattern \leftarrow A \cap B$
        if $Pattern$ is frequent and not $\emptyset$ then
          Add $Pattern$ in $LocalMaximals$
        else
          Add $Pattern$ in $InFrequentFPB$ IF not $\emptyset$
        end if
      end for
      for each pattern $P$ in $LocalMaximals$ do
        if $P$ is not a subset of any $M \in GlobalMaximals$
        then
          Add $P$ in $GlobalMaximals$
        end if
      end for
    end if
  end for
  $Patterns \leftarrow$ GeneratePatterns($FPB, GlobalMaximals$)
  Output $Patterns$
```

2.2 Bi-directional Pushing of Constraints

Pushing constraints early means considering constraints while mining for patterns rather than postponing the checking of constraints until after the mining process. Given the intrinsic characteristics of existing algorithms for mining frequent itemsets, either going over the lattice of candidate itemsets top-down or bottom-up, considering all constraints while mining is difficult. Most algorithms attempt to push either type of constraints during the mining process hoping to reduce the search space in one direction: from subsets to supersets or from supersets to subsets. Dualminer [7] pushes both types of constraints but at the expense of efficiency. Focusing solely on reducing the search space by pruning the lattice of itemsets is not necessarily a winning strategy. While pushing constraints early seems conceptually beneficial, in practice the testing of the constraints can add significant overhead. If the constraints are not selective enough, checking the constraint predicates for each candidate can be onerous. It is thus important that we also reduce the checking frequency. While the primary benefit of early constraint checking is the elimination of candidates which can not pass the constraint, it can also be used to identify candidates which are guaranteed to pass the constraint and therefore do not need to be re-checked. In summary, the goal of pushing constraints early is to reduce the itemset search space, eliminating unnecessary processing and memory consumption, while at the same time limiting the amount of constraint checking performed.

3 COFI with Leap

Most existing algorithms traverse the itemset lattice top-down or bottom-up, and search using a depth first or breadth first strategy. In contrast, we propose a *leap traversal* strategy that finds a superset of pertinent itemsets by "leaping" between promising nodes in the itemset lattice. In addition to finding these relevant candidate itemsets, sufficient information is gathered to produce the frequent itemset patterns, along with their supports. Here, we use leap traversal in conjunction with the complementary COFI idea [10], where the locally frequent itemsets of each frequent item are explored separately. This creates additional opportunities for pruning. What is the COFI idea and what is this set of pertinent itemsets with their additional information? This set of pertinent itemsets is the set of maximals. We will first present the COFI tree structure, then we will introduce our algorithm *COFILeap* which mines for frequent itemsets using COFI trees and jumping in the pattern lattice. In the next section, this same algorithm will be enhanced with constraint checking to produce our algorithm *BifoldLeap*.

3.1 COFI-trees

The COFI-tree idea was introduced in [10] as a means to reduce the memory requirement and speed up the mining strategy of FP-growth [12]. Rather than recursively building conditional trees from the FP-tree, the COFI strategy was to create COFI-trees for each frequent item and mine them separately. Conditional trees are FP-trees conditioned on the existence of a given frequent item. The FP-tree [12] is a compact prefix-tree representation of the sub-transactions in the input data. Here, sub-transactions are the original transactions with infrequent items removed. The FP-tree contains a header and inter-node links which facilitate downward traversal (forward in the itemset pattern) as well as lateral traversal (next node representing a specific item).

Building COFI-trees based on the FP-tree. For each frequent item in the FP-tree, in order of increasing support, one COFI-tree is built [10]. This tree is based on sub-transactions which contain the root item and are composed only of items locally frequent with the root item that have not already been used as root items in earlier COFI-trees. The COFI-tree is similar to the FP-tree, but includes extra links for upward traversal (earlier in the itemset pattern), a new *participation* counter in each node, and a data structure to allow traversal of all leaves in the tree. This participation counter is used during the mining process to count up the participation of each node in the generation of a frequent pattern.

Algorithm 2 BifoldLeap: Pushing P() and Q()

```
**Input:** $D$ (transactional database); $\sigma; \Sigma$; *P()*; *Q()*.
**Output:** Frequent patterns satisfying P(), Q()
$PF1 \leftarrow$ Scan $D$ to find the set of frequent P1-itemsets
$FPT \leftarrow$ Scan $D$ to build FP-tree using $PF1$ and $Q()$
$PGM(PGlobalMaximals) \leftarrow \emptyset$
**for** each item $I$ in Header($FPT$) **do**
  $LF \leftarrow FindlocalFrequentWithRespect(I)$
  **if** (Not $Q(I \cup LF)$) **then** break
  **if** ($P(I \cup LF)$) **then** Add $(I \cup LF)$ to $PGM$ and break
  **if** Not $(I \cup LF) \subseteq PGM$ **then**
    $ICT \leftarrow$ Build COFI-Tree for $I$
    $FPB \leftarrow$ FindFrequentPathBases($ICT$)
    $PLM(PLMaximals) \leftarrow \{P(FPB)$ and frequent$\}$
    $InFrequentFPB \leftarrow notFrequent(FPB)$
    **for** each pair $(A, B) \in InFrequentFPB$ **do**
      $header \leftarrow A \cap B$
      Add $header$ in $PLM$ and Break IF ($P(header)$
      AND is frequent and not $\emptyset$)
      Delete $header$ and break IF (Not $Q(header)$)
      $tail \leftarrow$ Intersection(FPBs not in $header$)
      delete $header$ and break IF (Not $P(header \cap tail)$)
      Do not check for $Q()$ in any subset of $header$ IF
      ($Q(header \cap tail)$)
    **end for**
    **for** each pattern $P$ in $PLM$ **do**
      Add $P$ in $PGM$ IF (($P$ not subset of any $M \in$
      $PGM$)
    **end for**
  **end if**
**end for**
PQ-Patterns $\leftarrow$ GPatternsQ($FPB, PGM$)
Output PQ-Patterns
```

3.2 COFILeap algorithm

COFILeap is different than the algorithm presented in [10] in the sense that it generates maximal patterns, where a pattern is said to be maximal if there is no other frequent pattern that subsumes it. *COFILeap* rather than traversing the pattern lattice top-down it leaps from one node to the other in search of the support border where maximals sit. Once maximals are found, with the extra information collected, all other patterns can be generated with their respective support.

Following is a brief summary of the *COFILeap* algorithm. First, a frequent pattern FP-tree [12] is created, using two scans of the database. Second, for each frequent item, a COFI-tree is created including all co-occurant frequent items to the right (i.e. in order of decreasing support). Each COFI-tree is generated from the FP-tree without returning to the transactional database for scans. Unique sub-transactions in the COFI-tree along with their count (called *branch support*) are obtained from the COFI-tree. These unique sub-transactions are called *frequent path bases* (FPB). These can be obtained by traversing upward from each leaf node in the COFI-tree, updating the participation counter to avoid over-counting nodes. Clearly, there is at most one FPB for each sub-transaction in the COFI-tree. Frequent FPBs are declared candidate maximals. Infrequent FPBs are intersected iteratively, producing subsets which may be frequent. When an intersection of infrequent FPBs results in a frequent itemset, that itemset is declared as a candidate maximal and is not subject to further intersections. When an intersection is infrequent, it participates in further intersections looking for maximals. This is indeed how the leaps in the lattice are done. The result of the intersection of FPBs indicates the next node to explore. How is the support of a pattern calculated? Given the set of frequent path bases along with their branch supports, it is possible to count the support of any itemset. This is done by finding all FPBs which are supersets of the target itemset, and summing their branch supports. For example, if there are two FPBs, ABC and $ABCD$, each with branch support 1, ABC has support 2, and $ABCD$ has support 1.

Algorithm 1 shows the main steps of *COFILeap*. Notice that COFI-trees are not generated systematically for all frequent 1-itemsets. There is no need to look for maximals locally with respect to an item I, if I and its locally frequent items are already subset of known global maximals. Finally, in the function $GeneratePatterns$ the set of candidate maximal patterns is used along with the frequent path bases to produce the set of all frequent itemsets that satisfy the constraints along with their supports. Maximal itemsets can be found by filtering the candidate maximals to remove subsets. Supports for the candidate maximal patterns were computed as part of the intersection process (to discover that they were frequent), and therefore do not need to be recomputed. Once the maximal itemsets have been found, the all frequent itemsets can be found by iterating over all subsets of the maximals, suppressing duplicates resulting from overlap with other maximal patterns. Support counting for the frequent itemsets is done as for the FPBs, i.e. by summing the branch supports of all FPBs which are supersets of the pattern.

4 Leap with constraints

The conjunction of all anti-monotone constraints comprises a predicate that we call $P()$. A second predicate $Q()$ contains the conjunction of the monotone constraints. A common approach is to include the ubiquitous minimum support constraint as part of $P()$. Similarly, the monotone maximum support constraint can be included as part of $Q()$. In this way, a frequent itemset mining algorithm can be extended to push $P()$ deeply by replacing checks for minimum support with checks for $P()$. In our algorithm, we separate the constraints on the support from other constraints. Thus, the minimum support constraint and the maximum support constraint are extracted from $P()$

and $Q()$ respectively. This is because checking for support is an integral part of the frequent itemset enumeration, while other constraints on item attributes are used for search space pruning. The algorithm *COFILeap* offers a number of opportunities to push the monotone and anti-monotone predicates, $P()$ and $Q()$ respectively. We start this process by defining two terms which are head (H) and tail (T) where H is a frequent path base or any subset generated from the intersection of frequent path bases, and T is the itemset generated from intersecting all remaining frequent path bases not used in the intersection of H. The intersection of H and T, $H \cap T$, is the smallest subset of H that may yet be considered. Thus Leap focuses on finding frequent H that can be declared as local maximals and candidate global maximals. *BifoldLeap* extends this idea to find local maximals that satisfy $P()$. We call these P-maximals.

Although we further constrain the P-maximals to itemsets which satisfy $Q()$, not all subsets of these P-maximals are guaranteed to satisfy $Q()$. To find the itemsets which satisfy both constraints, the subsets of each P-maximal are generated in order from long patterns to short. When a subset is found to fail $Q()$, further subsets do not need to be generated for that itemset, as they are guaranteed to fail $Q()$ also.

There are three significant places where constraints can be pushed: (a) while building the FP-tree, (b) while building the COFI-trees, and (c) while intersecting the frequent path bases, which is the main phase where both types of constraints are pushed at the same time (Algorithm 2).

Constraint pushing opportunities during FP-tree construction. First, $P()$ is applied to each 1-itemset. Items which fail this test are not included in FP-tree construction. Second, we use the idea from FP-Bonsai [5] where sub-transactions which do not satisfy $Q()$ are not used in the second phase of the FP-tree building process. The supports for the items within these transactions are decremented. This may result in some previously frequent items becoming infrequent. Such items will not be used to construct COFI-trees in the following phase.

Constraint pushing opportunities during COFI-tree construction. Let X be the set of all items that will be used to build the COFI-tree, i.e. the items which satisfy $P()$ individually but have not been used as the root of a previous COFI-tree. If X fails $Q()$, there is no need to build the COFI-tree, as no subset of X can satisfy $Q()$. Alternatively, if X satisfies $P()$, there is also no need to build the COFI-tree, as X is a candidate P-maximal.

Constraint pushing opportunities during intersection of Frequent-Path-Bases. There are two high-level strategies for pushing constraints during the intersection phase. First, $P()$ and $Q()$ can be used to eliminate an itemset or remove the need to evaluate its intersections with additional frequent path bases. Second, $P()$ and $Q()$ can be applied to the "head intersect tail" ($H \cap T$), which is the smallest subset of the current itemset that can be produced by further intersections. These strategies are detailed in the following four theorems.

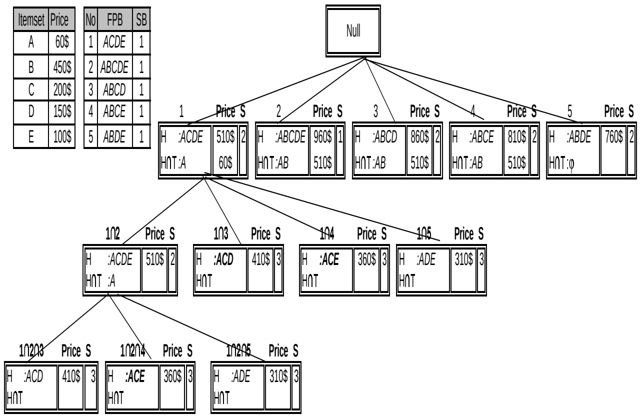
Theorem 1: If an intersection of frequent path bases (H) fails $Q()$, it can be discarded, and there is no need to evaluate further intersections with H.

Proof: If an itemset fails $Q()$, all of its subsets are guaranteed to fail $Q()$ based on the definition of monotone constraints. Further intersecting H will produce subsets, all of which are guaranteed to violate $Q()$.

Theorem 2: If an intersection of frequent path bases (H) passes $P()$, it is a candidate P-maximal, and there is no need to evaluate further intersections with H.**Proof**: Further intersecting H will produce subsets of H. By definition, no P-maximal is subsumed by another itemset which also satisfies $P()$. Therefore, none of these subsets of H are potential new P-maximals.

Theorem 3: If a node's $H \cap T$ fails $P()$, the H node can be discarded, and there is no need to evaluate further intersections with H. **Proof**: If an itemset fails $P()$, then all of its supersets will also violate $P()$ from the definition of anti-monotone constraints. Since a node's $H \cap T$ represents the subset of H that results from intersecting H with all remaining frequent path bases, H and all combinations of intersections between H and remaining frequent path bases are supersets of $H \cap T$ and therefore guaranteed to fail $P()$ also.

Theorem 4:If a node's $H \cap T$ passes $Q()$, $Q()$ is guaranteed to pass for any itemset resulting from the intersection of a subset of the frequent path bases used to generate H plus the remaining frequent path bases yet to be intersected with H. $Q()$ does not need to be checked in these cases. **Proof**: $Q()$ is guaranteed to pass for all of these itemsets because they are generated from a subset of the intersections used to produce the $H \cap T$ and are therefore supersets of the $H \cap T$.



Itemset	Price
A	60$
B	450$
C	200$
D	150$
E	100$

No	FPB	SB
1	ACDE	1
2	ABCDE	1
3	ABCD	1
4	ABCE	1
5	ABDE	1

Figure 1. Pushing P() and Q().

The following example, shown in Figure 1, illustrates how *BifoldLeap* works. An A-COFI-tree is made from five items, A, B, C, D, and E, with prices \$60, \$450, \$200, \$150, and \$100 respectively. In our example, this COFI-tree generates 5 frequent path bases, $ACDE$, $ABCDE$, $ABCD$, $ABCE$, and $ABDE$, each with branch support one. The anti-monotone predicate, $P()$, is $Sum(Prices) \leq \$500$, and the monotone predicate, $Q()$, is $Sum(prices) \geq \$100$. Intersecting the first FPB with the second produces $ACDE$ which has a price of \$510, and therefore violates $P()$ and passes $Q()$. Next, we examine the $H \cap T$, the intersection of this node with the remaining three FPBs, which yields A with price \$60, passing $P()$ and failing $Q()$. None of these constraint checks provide an opportunity for pruning, so we continue intersecting this itemset with the remaining frequent path bases. The first intersection is with the third FPB, producing ACD with price \$410, which satisfies both the anti-monotone and monotone constraints. The second intersection produces ACE, which also satisfies both constraints. The same thing occurs with the last intersection, which produces ADE. Going back to the second frequent path base, ABCDE, we find that the $H \cap T$, AB, violates the anti-monotone constraint with price \$510. Therefore, we do not need to consider $ABCDE$ or any further intersections with it. The remaining nodes are eliminated in the same manner. In total, three candidate P-maximals were discovered. We can generate all of their subsets while testing only against $Q()$. Finally, the support for these generated subsets can be computed from the existing frequent path bases.

5 Performance Evaluation

To evaluate our BifoldLeap algorithm, we conducted a set of experiments to test the effect of pushing monotone and anti-monotone constraints separately, and then both in combination for the same datasets. To quantify scalability, we experimented with datasets of varying size. We also measured the impact of pushing versus post-processing constraints on the number of evaluations of $P()$ and $Q()$. Like in [7], we assigned prices to items using both uniform and zipf distributions. Our constraints consisted of conjunctions of tests for aggregate, minimum, and maximum price in relation to specific threshholds.

We compared our algorithm with Dualminer [7]. Based on its authors' recommendations, we built the Dualminer framework on top of the MAFIA [8] implementation provided by its original authors. Our experiments were conducted on a 3GHz Intel P4 with 2 GB of memory running Linux 2.4.25, Red Hat Linux release 9. The times reported also include the time to output all matching itemsets. We have tested these algorithms using both real datasets provided by [11] and synthetic datasets generated using [3]; we used 'retail' as our primary real dataset reported here. A dataset with the same characteristics as the one reported in [7] was also generated.

We received an FP-Bonsai code (base on FP-Growth) from its original authors [5]. Unfortunately, not all pruning and clever constraint considerations suggested in their FP-Bonsai paper were implemented in this code. Moreover, the implementation as received produced some false positives and false negatives. This is why we opted not to add it to our comparison study. Although, with simple and only monotone constraints, the received FP-Bonsai implementation was indeed fast. FP-Bonsai as described in the paper has merit but because of lack of time we could not implement it ourselves (albeit adding implementation bias) or fix the received code.

5.1 Impact of P and Q Selectivity on BifoldLeap and Dualminer

To differentiate between our novel BifoldLeap algorithm and Dualminer, we experimented against the retail dataset. In the first experiment (Figure 2.A), we pushed $P()$, then $Q()$, and finally $P() \wedge Q()$. We used the zipf distribution to assign prices to items. Both $P()$ and $Q()$ consisted of constraints on the sum of the prices. The constraint thresholds were chosen to not be very selective. Figure 2.B presents the same experiment with more selective constraints.

Figure 2.C presents pushing extremely selective constraints, using anti-monotone and monotone constraints on the sum of the prices, and on the minimum and maximum item price. In this experiment, we found that BifoldLeap in most cases outperforms Dualminer and in some cases by more than one order of magnitude. The most interesting observation we found from this experiment was that if we push one type of constraint, e.g. $P()$, that takes $T1$ seconds and the other type of constraint, $Q()$, takes $T2$ seconds where $T1 \leq T2$, in Dualminer pushing both constraints together will take $T3$ seconds, where $T3$ is always between $T1$ and $T2$. In contrast, BifoldLeap always takes less time with the conjunction of the constraints than with either constraint in isolation. Monotone and anti-monotone constraints can indeed be mutually assisting each other in the selectivity. BifoldLeap took better advantage of this reciprocal assistance in the pruning.

5.2 Scalability Tests

Scalability is an important issue for frequent itemset mining algorithms. Synthetic datasets were generated with 50K, 100K, 250K, and 500K transactions, with 5K or 10K distinct items. In this experiment, BifoldLeap demonstrated extremely good scalability versus increasing dataset size. In contrast, Dualminer reached a point where it consumed almost three orders of magnitude more time than that needed by BifoldLeap. Figure 3.A depicts one of these results while mining datasets with only 5K unique items. As another experiment example, we tested both algorithms on datasets with up to 50 Million transactions and 100K items. Dualminer finished the 1M dataset in 8534 seconds while BifoldLeap finished in 186s, 190s, 987s and 2034s for the 1M, 5M, 25M and 50M transactions datasets respectively.

5.3 Constraint Checking: Pushing Constraints versus Post-processing

One of the major challenging issues for constraint mining is reducing the number of evaluations of $P()$ and $Q()$. In the following experiment, we generated a synthetic dataset with the same characteristics as the one reported in [7]. Specifically, it was generated with 10,000 transactions, an average transaction length of 15, an average maximal pattern length of 10, 1000 unique items, and 10,000 patterns. We found that Dualminer was indeed good on this dataset as reported in [7]. However, BiFoldLeap outperformed it with the same order of magnitude as the tests on timing. This shows that the predicate checking is indeed a significant overhead and BiFoldLeap outperforms Dualminer in time primarily because it does significantly less predicate checks.

The goal of these experiments was to test the number of evaluations and the effect of pushing constraints early versus post-processing them. We ran our experiments using this

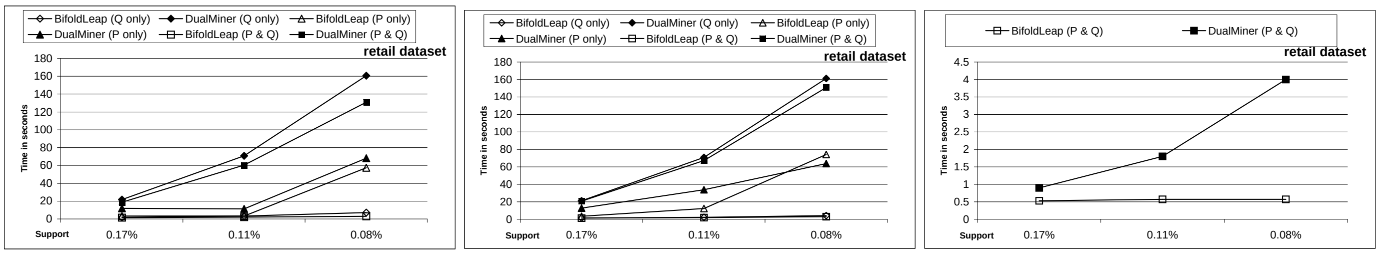


Figure 2. (A) Pushing $P()$, $Q()$, and $P() \wedge Q()$. (B) More selective constraints. (C) Extremely selective constraints

dataset with absolute support equal to 25, 50, and 75 using the two different distributions. We used a modified version of MAFIA with post-processing as the post-processing counterpart to Dualminer. Our implementation of Dualminer always tests minimum support and P() together, while BifoldLeap's minimum support checks occur at different times and do not contribute to the count for $P()$. Figure 4 depicts the results of these experiments. Our first observation is that Dualminer performs a huge number of constraint evaluations as compared to BifoldLeap. Even in cases where we only generated 255 patterns, Dualminer needed more than 50 thousand evaluations for both $P()$ and $Q()$, compared to almost 6 thousand needed by BifoldLeap. Our second observation is that MAFIA with post-processing requires fewer constraint evaluations than Dualminer.

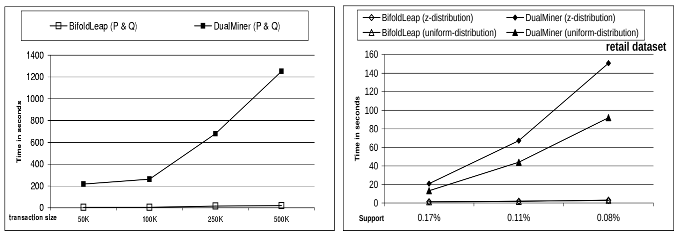


Figure 3. (A) Scalability test. (B) Effect of changing the price distribution

5.4 Different Distributions

All of our experiments were conducted using uniform and/or zipf price distributions. In most of the experiments, we found that the effect of changing the distribution on Dualminer was greater than for BifoldLeap. This can be justified by the effectiveness of the pruning techniques used by BifoldLeap that also reduce the number of candidate checks which consequently affected its performance. Figure 3.B depicts one of these results for the retail dataset.

6 Related work

Mining frequent patterns with constraints has been studied in [13] where the concept of *monotone* and *anti-monotone* and

Uniform-distribution

dmT10K1KD15L	Absolute support = 25		Absolute support = 50		Absolute support = 75	
	Early Push	Post Proc.	Early Push	Post Proc.	Early Push	Post Proc.
BifoldLeap (#P)	2266	3166	1483	1581	1212	1319
BifoldLeap (#Q)	4156	4650	299	351	166	189
DualMiner (#P)	25722	25649	18389	18028	14038	13720
DualMiner (#Q)	24946	298	17602	187	13221	160

# of generated patterns	255	158	134

(A)

Z-distribution

dmT10K1KD15L	Absolute support = 25		Absolute support = 50		Absolute support = 75	
	Early Push	Post Proc.	Early Push	Post Proc.	Early Push	Post Proc.
BifoldLeap (#P)	1814	2184	1428	1778	1207	1436
BifoldLeap (#Q)	420	625	125	312	99	254
DualMiner (#P)	11971	11722	8019	7790	6148	5950
DualMiner (#Q)	11185	197	7178	119	5282	100

# of generated patterns	130	76	62

(B)

Figure 4. No. of $P()$ and $Q()$ evaluations, using constraint pushing vs. post-processing

succinct were introduced to prune the search space. Jian Pei et al. [14, 15] have also generalized these two classes of constraints and introduced a new convertible constraint class. In their work they proposed a new algorithm called FIC^M which is an FP-Growth based algorithm [12]. This algorithm generates most frequent patterns before pruning them. Its main contribution is that it checks for monotone constraints early and once a frequent itemset is found to satisfy the monotone constraint, then all itemsets having this item as a prefix are sure to satisfy the constraint and consequently there is no need to apply further checks. Dualminer [7] is the first algorithm to mine both types of constraints at the same time. Nonetheless, it suffers from many practical limitations and performance issues. First, it is built on the top of the MAFIA [8] algorithm which produces the set of maximal patterns, and consequently all frequent patterns generated using this model do not have their support attached. Second, it assumes that the whole dataset can fit in main memory which is not always the case. FP-Growth and our approach use a very condensed representation, namely FP-Tree, which uses significantly less memory [12]. Third, their top-down computation exploiting the monotone constraint often performs many useless tests for relatively large datasets, which raises doubts about the performance gained by pushing constraints in the Dualminer algorithm. In a recent study of parallelizing Dualminer [16], the authors showed that by mining relatively small sparse datasets consisting of 10K transactions and 100K items, the sequential version of Dualminer took an excessive amount of time. Unfortunately, the original authors of Dualminer did not show any single experiment to depict the execution time of their algorithm but only the reduction in predicate executions [7]. A recent strategy dealing with monotone and anti-monotone constraints suggests reducing the transactional database input via pre-processing by successively eliminating transactions that violate the constraints and then applying any frequent itemset mining algorithm on the reduced transaction set [4, 6]. The main drawback of this approach is that it is highly I/O bound due to the iterative process needed in rewriting the reduced dataset to disk. This algorithm is also sensitive to the results of the initial monotone constraint checking which is applied to full transactions. In other words, if a whole transaction satisfies the monotone constraint, then no pruning is applied and consequently no gains are achieved even if parts of this transaction do not satisfy the same monotone constraint. To overcome some of the issues in [4], the same approach has been tested against the FP-Growth approach in [5] with new effective pruning heuristics.

7 Conclusion

Since the introduction of association rules a decade ago and the launch of the research in efficient frequent itemset mining, the development of effective approaches for mining large transactional databases has been the focus of many research studies. Furthermore, it is widely recognized that mining for frequent items or association rules, regardless of its efficiency, usually yields an overwhelming, crushing number of patterns. This is one of the reasons it is argued that the integration of data mining and database management technologies is required [9]. These large sets of discovered patterns could be queried. Expressing constraints using a query language could indeed help sift through the large pattern set to identify the useful ones.

We argue that pushing the consideration of these constraints at the mining process before discovering the patterns is an efficient and effective way to solve the problem. This does not exclude the integration of data mining and database systems, but suggests the need for data mining query languages intricately integrated with the data mining process.

In this paper we address the issue of early consideration of monotone and anti-monotone constraints in the case of frequent itemset mining. We propose a leap traversal approach, *BifoldLeap*, that traverses the search space by jumping from relevant node to relevant node and simultaneously checking for constraint violations. The approach we propose uses existing data structures, FP-tree and COFI-tree, but introduces new pruning techniques to reduce the search costs. We conducted a battery of experiments to evaluate our constraint-based search and report a fraction of them herein for lack of space. The experiments show the advantages of pushing both monotone and anti-monotone constraints as early as possible in the mining process despite the overhead of constraint checking. We also compared our algorithm to Dualminer, a state-of-the-art algorithm in constraint-based frequent itemset mining, and showed how our algorithm outperforms it and can find all frequent itemsets, the closed and the maximal patterns that satisfy constraints along with their exact supports.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. 1993 ACM-SIGMOD Int. Conf. Management of Data*, pages 207–216, Washington, D.C., May 1993.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 1994 Int. Conf. Very Large Data Bases*, pages 487–499, Santiago, Chile, September 1994.

[3] I. Almaden. Quest synthetic data generation code. http://www.almaden.ibm.com/software/quest/Resources/index.shtml.

[4] F. Bonchi, F. Giannotti, A. Mazzanti, and D. Pedreschi. Examiner: Optimized level-wise frequent pattern mining with monotone constraints. In *IEEE ICDM*, Melbourne, Florida, November 2004.

[5] F. Bonchi and B. Goethals. Fp-bonsai: the art of growing and pruning small fp-trees. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'04)*, pages 155–160, 2004.

[6] F. Bonchi and C. Lucchese. On closed constrained frequent pattern mining. In *IEEE International Conference on Data Mining (ICDM'04)*, Brighton, UK, November 2004.

[7] C. Bucila, J. Gehrke, D. Kifer, and W. White. Dualminer: A dual-pruning algorithm for itemsets with constraints. In *Eight ACM SIGKDD Internationa Conf. on Knowledge Discovery and Data Mining*, pages 42–51, Edmonton, Alberta, August 2002.

[8] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *ICDE*, pages 443–452, 2001.

[9] S. Chaudhuri. Data mining and database systems: Where is the intersection? *Bulletin of the Technical Committee on Data Engineering*, 21, March 1998.

[10] M. El-Hajj and O. R. Zaïane. Non recursive generation of frequent k-itemsets from frequent pattern tree representations. In *Proc. of 5th International Conference on Data Warehousing and Knowledge Discovery (DaWak'2003)*, September 2003.

[11] Frequent itemset mining implementations repository. http://fimi.cs.helsinki.fi/.

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

[13] L. Lakshmanan, R. Ng, J. Han, and A. Pang. Optimization of constrained frequent set queries with 2-variable constraints. In *ACM SIGMOD Conference on Management of Data*, pages 157–168, 1999.

[14] J. Pie and J. Han. Can we push more constraints into frequent pattern mining? In *ACM SIGKDD Conference*, pages 350–354, 2000.

[15] J. Pie, J. Han, and L. Lakshmanan. Mining frequent itemsets with convertible constraints. In *IEEE ICDE Conference*, pages 433–442, 2001.

[16] R. M. Ting, J. Bailey, and K. Ramamohanarao. Paradualminer: An efficient parallel implementation of the dualminer algorithm. In *Eight Pacific-Asia Conference, PAKDD 2004*, pages 96–105, Sydney, Australia, May 2004.

[17] O. R. Zaïane and M. El-Hajj. Pattern Lattice Traversal by Selective Jumps. In *In Proc. 2005 Int'l Conf. on Data Mining and Knowledge Discovery (ACM SIGKDD), Chicago, IL, USA.*, pages 729–735, August 2005.