Mining with Constraints by Pruning and Avoiding Ineffectual Processing

Mohammad El-Hajj and Osmar R. Zaïane

Department of Computing Science, University of Alberta,
Edmonton AB, Canada
{mohammad,zaiane}@cs.ualberta.ca

Abstract. It is known that algorithms for discovering association rules generate an overwhelming number of those rules. While many new very 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. In this paper we propose a new way of pushing the constraints in dual-mode based from the set of maximal patterns that is an order of magnitude smaller than the set of all frequent patterns.

1 Introduction

Frequent pattern discovery has become a common topic of investigation in the data mining research area. Its main theme is to discover the sets of items that occur together more than a given threshold value defined by the decision maker. In most cases when the support threshold is low and the number of frequent patterns "explodes", the discovery of these patterns becomes problematic. To reduce the effects of such problem new methods need to be investigated such as fast traversal techniques to reduce the search space or using constraints that lessen the output size whilst directly discovering patterns that are of interest to the user.

In short, our contributions in this paper is presenting the DPC-COFI (**D**ual-**P**ushing of **C**onstraints in COFI) algorithm that is based on our previous work COFI* where we push both types of constraints at the same time starting from the set of maximal patterns that is much smaller than the set of all frequent patterns. The algorithm does not only focus on the candidate itemset search space reduction, but also the elimination of pointless processing and the lessening of constraint predicate testing. The remainder of this paper is organized as follows: The types of constraints are explained in Section 2. We illustrate in Section 3 how we can integrate the constraint checking in our COFI approach to produce the DPC-COFI algorithm. Experimental results are given in Section 4.

2 Category of constraints

Let $I = \{i_1, i_2, ...i_m\}$ be a set of literals, called items. 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 is greater or equal than a given minimum support threshold σ. An itemset $\mathcal{M}$ is also called maximal if there is no other frequent pattern that is a superset of $\mathcal{M}$. In addition to the transactions, other tables could describe the items in the transactions in terms of attributes such as price, weight, etc. A constraint ζ is a predicate on itemset X that yields either *true or false* and is typically expressed in terms of the items' attributes. An itemset X satisfies a constraint ζ if and only if $\zeta(X)$ is *true*. 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. 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. The conjunction of all *anti-monotone* constraints forms the predicate $P()$, and the conjunction of all *monotone* constraint forms the predict $Q()$

3 Pushing Constraints

Our strategy to discover the frequent patterns that satisfy the given constraints is depicted in Figure 1. The major idea is that we first discover the set of maximal patterns, which generalizes the set of all frequent patterns and is significantly smaller than the set of all frequent itemsets, but can be used to straightforwardly generate them all. Initially, we only consider the anti-monotone constraint $P()$ to keep the singleton items that satisfy $P()$. We use $P()$ and $Q()$ to prune the COFI-trees [3] and reduce the search space. When generating the frequent itemsets from the maximals, we consider again both the monotone ($Q()$) and the anti-monotone constraints ($P()$). Before presenting the details, let us introduce the necessary preliminaries. The COFI* algorithm that we use in this work is an extension of our COFI algorithm [3] and consists of two main stages. Stage one is the construction of the Frequent Pattern tree (FP-tree) and stage two is the actual mining for this data structure by successively building other smaller tree structures (COFI-trees) that generate the set of maximal patterns with special data structures that encode the supports of all the subsets of these maximal patterns which are indeed the set of all frequent patterns. For lack of space, we refer the reader to our work in [4] for more details on how we can generate the set of maximals and consequently the set of all frequent patterns with their support using the COFI* algorithm that makes use of the COFI-trees.

3.1 Mining COFI-trees with constraints: The DPC-COFI algorithm

We adapted our COFI* algorithm by considering P() and Q() to reduce the sizes of the trees. Pushing P() early is adopted in COFI* as follows: **1.** Remove individual items that do not satisfy P() (i.e from the FP-tree). **2.** For each A-COFI-tree, remove any locally frequent item B, where AB does not satisfy P(). This reduces the A-COFI-tree size. **3.** There is no need for constraint checking for any A-COFI-tree if the itemset X, representing all locally frequent items with

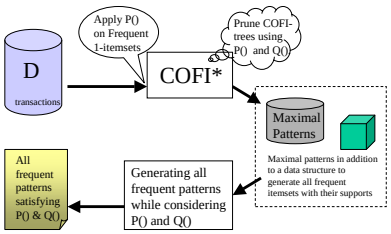

Fig. 1. Pushing P() and Q() using the COFI* approach.

respect to A, satisfies P() constraint. Similarly pushing Q() in COFI* is done as follows: **1.** No need to create an A-COFI-tree, if all its local items X with the A itemset violate Q(). This reduces the number of COFI-trees to mine. **2.** No need for constraint checking for any A-COFI-tree if the item A satisfies Q(), since any item with A in the A-COFI-tree will also satisfy this constraint. These strategies allow the pruning of the search space, the reduction of the data structures to mine and the direct pinpointing of patterns that satisfy the constraints without testing for those constraints (i.e. less checking is done). COFI* generates the set of Maximals with a special data structure to encode the supports of all frequent patterns. DPC-COFI is a framework built on top of COFI* during the subset generation. In DPC-COFI not all subset are part of the answer set as in COFI*, only those subsets that satisfy P() and Q(). Because of this we do not have to find the support for any frequent item unless we verify that it satisfies both P() and Q(). In Summary, during the subset generation of DPC-COFI, the following rules are tested: **1.** If an itemset satisfies both P() and Q() then generate its support and add it to the results sets. **2.** if an itemset satisfies P(), then there is no need to test P() for any of its subsets as all will satisfy P(). **3.** If an itemset does not satisfy P() then generate its subsets but do not add it (the initial itemset) to the results set. **4.** If an itemset does not satisfy Q() then prune it from the subset generation process. (i.e. none of its subsets will satisfy Q()).

4 Performance Evaluation

To evaluate our DPC-COFI 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 $Q()$. Due to the lack of space we could not depicts all our results in this work. We compared our algorithm with Dualminer [1] which is the only known algorithm to effectively mine both types of constraints at the same time. Based on its authors' recommendations, we built the Dualminer framework on top of the MAFIA [2] 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. We have tested these algorithms using both real datasets provided by [5] and synthetic datasets generated using [6]; we used 'retail' as our primary real dataset reported here. A dataset with the same characteristics as the one reported in [1] was also generated.

4.1 Impact of P and Q Selectivity on DPC-COFI and Dualminer

To differentiate between our novel DPC-COFI 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. Figure 2.B presents the same experiment with more selective constraints.

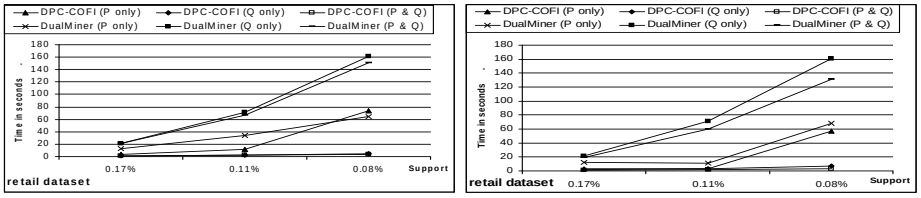

Fig. 2. (A)Pushing $P()$, $Q()$, and $P() \wedge Q()$. (B) Pushing more selective constraints.

5 Conclusion

In this paper we introduced an efficient approach allowing the consideration of both monotone and anti-monotone constraints during the mining process. It uses a new strategy of finding first the set of maximals that satisfy the constraints and applying later the constraints again during the subset generation of these maximals. Our experiments show significant performance gains compared to dualminer.

References

1. C. Bucila, J. Gehrke, D. Kifer, and W. White. Dualminer: A dual-pruning algorithm for itemsets with constraints. In *SIGKDD-2002*, pages 42–51, August 2002.
2. D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *ICDE*, pages 443–452, 2001.
3. M. El-Hajj and O. R. Zaïane. Non recursive generation of frequent k-itemsets from frequent pattern tree representations. In *(DaWak'2003)*, September 2003.
4. M. El-Hajj and O. R. Zaïane. Cofi approach for mining frequent itemsets revisited. In *(DMKD-04)*, pages 70–75, June 2004.
5. B. Goethals and M. Zaki. Advances in frequent itemset mining implementations: Introduction to fimi03. In *(FIMI'03) in conjunction with IEEE-ICDM*, 2003.
6. IBM_Almaden. Quest synthetic data generation code. http://www.almaden.ibm.com/cs/quest/syndata.html.