DualMiner: A Dual-Pruning Algorithm for Itemsets with Constraints

Cristian Bucilă, Johannes Gehrke and Daniel Kifer
({cristi,johannes,dkifer}@cs.cornell.edu*)*
Department of Computer Science
Cornell University

Walker White*
*(*wmwhite@udallas.edu*)*
Department of Mathematics
University of Dallas

Abstract. Recently, constraint-based mining of itemsets for questions like "find all frequent itemsets whose total price is at least \$50" has attracted much attention. Two classes of constraints, monotone and antimonotone, have been very useful in this area. There exist algorithms that efficiently take advantage of either one of these two classes, but no previous algorithms can efficiently handle both types of constraints *simultaneously.* In this paper, we present DualMiner, the first algorithm that efficiently prunes its search space using both monotone and antimonotone constraints. We complement a theoretical analysis and proof of correctness of DualMiner with an experimental study that shows the efficacy of DualMiner compared to previous work.

* Research conducted at Cornell University with the Intelligent Information Systems Institute

1. Introduction

Mining frequent itemsets in the presence of constraints is an important problem in data mining [5, 16, 17, 18, 20]. (We assume that the reader is familiar with the terminology from the association rules literature [2].) The problem can be stated abstractly as follows. Let M be a finite set of items from some domain (for example, products in a grocery store). All the items have a common set of descriptive attributes (i.e., the name, brand, or price of the item). A predicate (or constraint) over a set of items is a condition that the set has to satisfy. We can assume, without loss of generality, that each item has the same single descriptive attribute, and for convenience we will associate an item with its attribute value. Thus by "a predicate over a set of items X," we always mean a predicate over the associated set of attribute values of the items in X. For example, when we talk about the average of a set M, we are referring to the average of the values of the items in M. The goal of constraint-based market basket analysis is then: *given a set of predicates $P_1, P_2, \ldots, P_n$, find all subsets of M that satisfy $P_1 \wedge P_2 \wedge \cdots \wedge P_n$* .

Important classes of constraints, most notably *monotone* and *antimonotone*, have been introduced by Ng et al. [17, 16, 5, 18, 20]. There exist algorithms that take advantage of each class of constraints. However, their main deficiency is that they each handle only one class of constraints efficiently. More recently, Raedt and Kramer [22] have generalized these algorithms to allow several types of constraints, but this generalization only handles one type of constraint at a time. In this paper, we present DualMiner, a new algorithm which efficiently mines constraint-based itemsets by simultaneously taking advantage of *both* monotone and antimonotone predicates.

Previous work has shown that the search space of all itemsets forms a lattice. Actually, this search space forms a special type of lattice called an algebra. Algebras can succinctly represent the output in much the same way that maximal frequent itemsets succinctly represent the set of all frequent itemsets [4]. The algebra representation adds to the efficiency of DualMiner and naturally leads to the following cost metrics for comparing different algorithms: number of nodes in the search space that are examined, number of evaluations of the antimonotone predicate and number of evaluations of the monotone predicate.

Not all predicates have the same cost structure. For example, given an itemset X, the predicate "$\min(X) \geq c$" can be evaluated in constant time if the size of X is bounded (a safe assumption) regardless of the number of transactions in the database. However, the cost of counting the support of an itemset X may depend on the current node being examined. For example, the cost can depend on the number of transactions in the database (this is the case when using bitmaps as in the MAFIA algorithm [8]). As we shall see, different cost structures require different strategies for traversal through the search space. By design, DualMiner is traversal strategy agnostic, and thus can accommodate the traversal strategy that is best for the dataset at hand.

Preliminaries

First, let us formally introduce the notion of an antimonotone constraint.

Definition 1. Given a set M, a predicate P defined over the powerset of M is *antimonotone* if

$$\forall S, J : (J \subseteq S \subseteq M \ \wedge \ P(S)) \Rightarrow P(J)$$

That is, if P holds for S then it holds for any subset of S.

The most popular antimonotone constraint is *support*. Consider a database $\mathcal{D}$ that consists of nonempty subsets of M called *transactions*. Given a set $X \subseteq M$, we define the function support(X) to be the number of transactions in $\mathcal{D}$ which contain X [1]. Note that frequent itemset mining is really a special case of constraint-based market analysis.

The advantage of an antimonotone predicate P is that if a set X does not satisfy P, then no superset of X can satisfy P. Such a set X can let us prune away a large part of the search space. This pruning can substantially improve performance. This technique is effective because, for any small set A, there is a significant probability that any given larger set is actually a superset of A. This is a consequence of the fact that the search space forms a lattice and hence the larger sets are simply the unions of the smaller sets.

For example, consider the itemset lattice illustrated in Figure 1 (with the set of items $M = \{A, B, C, D\}$). A typical frequent itemset algorithm looks at one level of the lattice at a time, starting from the empty itemset at the top. In this example, we discover that the itemset $\{D\}$ is not frequent. Therefore, when we move on to successive levels of the lattice, we do not have to look at any supersets of $\{D\}$. Since

[1] We will drop the dependency on $\mathcal{D}$ from all our definitions.

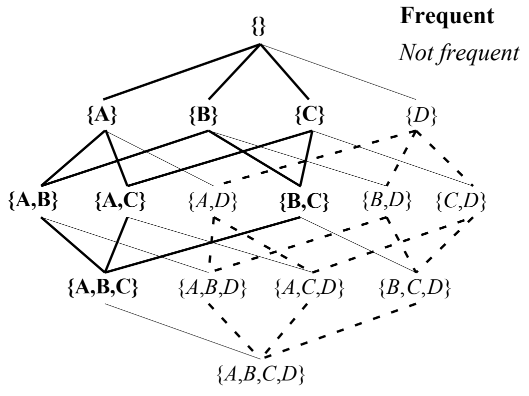

Figure 1. Frequent Itemsets for $M = \{A, B, C, D\}$

half of the lattice is composed of supersets of $\{D\}$, this is a dramatic reduction of the search space.

The collection of all itemsets does not just form a lattice; it forms a *Boolean algebra* (which we will call an *algebra*). Like a lattice, an algebra has union and intersection. In addition, it also has complementation. In the algebra of itemsets, the complement of a set is formed by subtracting it from the set of all items. For example, in the algebra illustrated in Figure 1, the complement of $\{D\}$ is the set $\overline{\{D\}} = \{A, B, C\}$.

In any algebra, complementation introduces a notion of *duality*. Every property or operation has a dual, and given any algorithm, we can construct a new algorithm by replacing everything with its dual. Figure 1 illustrates this duality. Suppose we wanted to find the itemsets of our algebra that are not frequent. We can again use a level-wise algorithm, this time starting from the maximal itemset $\overline{\emptyset} = \{A, B, C, D\}$ at the bottom.

As we move up levels in our algebra by removing elements from our itemsets, we can eliminate all subsets of a frequent itemset. For example, the itemset $\{A, B, C\}$ is frequent so we can remove half of the algebra from our search space just by inspecting this one node. Hence we see that infrequent is the dual of frequent, that subset is the dual of superset, and that the dual of any set is its complement.

In the previous example, we took advantage of the fact that the predicate "support$(X) \leq c$" for infrequent itemsets is *monotone*.

Definition 2. Given a set M, a predicate Q defined over the powerset of M is *monotone* if

$$\forall S, J : (S \subseteq J \subseteq M \ \wedge \ Q(S)) \Rightarrow Q(J)$$

That is, if Q holds for S then it holds for any superset of S.

While it may seem unnatural to search for infrequent itemsets, monotone predicates do occur naturally. For example, the constraint "$\max(X) > b$" is monotone. Other constraints, while not monotone themselves, have monotone approximations that are useful in pruning the search space. Pei and Han [18] have suggested several monotone approximations for the constraints "$\text{avg}(X) \leq a$" and "$\text{avg}(X) \geq b$."

The fact that many constraints are monotone or have monotone approximations motivates the need for an algorithm to find all sets that satisfy a conjunction of antimonotone and monotone predicates. Clearly, the conjunction of antimonotone predicates is antimonotone and the conjunction of monotone predicates is monotone. So our algorithm need only consider a predicate of the form $P(X) \wedge Q(X)$, where $P(X)$ is antimonotone (a conjunction of antimonotone predicates) and $Q(X)$ is monotone (a conjunction of monotone predicates).

While a similar problem has been considered by Raedt and Kramer [22], their algorithm performs a level-wise search with respect to the antimonotone predicate, followed by a level-wise search on the output with respect to the monotone predicate. If the monotone predicate is highly selective, this approach unnecessarily evaluates a large portion of the search space. Furthermore, the output of the first pass need not have a nice algebraic structure, and may be difficult to traverse. To the best of our knowledge, our algorithm is the first to use the structures of *both* P and Q simultaneously to avoid unnecessary evaluations of these potentially costly predicates. Thus our cost metrics are the number of sets $X \in 2^M$ for which we evaluate P and also the number of sets for which we evaluate Q.

Summary of our contributions:

– We introduce DualMiner, an algorithm that can prune using *both* monotone and antimonotone constraints (Sections 2 and 3).

– We give several non-trivial optimizations to the basic algorithm (Section 4).

– We analyze the complexity of our algorithm and compare it to other algorithms from the literature (Section 5).

- We present an improvement of the algorithm so that it outputs only non-mergeable subalgebras.
- In a thorough experimental study, we show that DualMiner significantly outperforms previous work (Section 7).

2. Overview of the Algorithm

2.1. Subalgebras

One of the advantages of the MAFIA algorithm is that it only searches for maximal frequent itemsets [8]. Not only does this permit several optimizations in the algorithm, it also provides a very concise representation of the output. For example, in Figure 1, the maximal frequent itemset $\{A, C, D\}$ uniquely defines the collection of all frequent itemsets in this algebra.

Because our algorithm considers the conjunction of an antimonotone predicate with a monotone predicate, the itemsets output by the algorithm are no longer closed under subset. For example, let the prices of A, B, C, D be $1, 4, 3, 2$, respectively, and suppose we apply the predicate "$\max(X.\text{price}) < 4 \wedge \min(X.\text{price}) < 2$" to the algebra in Figure 1. As can be seen in Figure 2, the set $\{A, C, D\}$ satisfies this predicate but the subset $\{C\}$ does not. Therefore, it is not enough to search for only maximal itemsets (or, similarly, for minimal itemsets).

However, there exists a suitable analogue of a maximal frequent itemset. Our search space is the powerset of M, which we will refer to as 2^M. This space is an algebra with maximal (top) element M, minimal (bottom) element $\emptyset$, the binary operations $\cap$ and $\cup$, and the complementation operator $\overline{}$. Given any collection of elements $\Gamma \subseteq 2^M$ that is closed under $\cap$ and $\cup$, we can define an algebra for Γ using the following definitions.

1. $T = \bigcup_{X \in \Gamma} X$ is the top element of Γ.
2. $B = \bigcap_{X \in \Gamma} X$ is the bottom element of Γ.
3. For any $A \in \Gamma$, $\overline{A} = T \setminus A$.

Given this fact, we define the notion of a *subalgebra* appropriately.

Definition 3. A *subalgebra* of 2^M is any collection of sets $\Gamma \subseteq 2^M$ closed under $\cap$ and $\cup$.

Due to the properties of P and Q, the collection of all sets $X \in 2^M$ for which $P(X) \wedge Q(X)$ is true can be represented as a collection of subalgebras. Furthermore, these subalgebras are complete in the following sense: if B is the bottom element of the subalgebra, and T is the top, then the subalgebra contains every element X of 2^M for which $B \subseteq X \subseteq T$. More specifically, if both $P(T)$ and $Q(B)$ are true then any superset X of B which is also a subset of T satisfies $P(X) \wedge Q(X)$. So we can compactly represent our subalgebras as pairs (B, T) where B is the bottom of the subalgebra and T is the top. We will refer to this collection of subalgebras as *good* subalgebras to distinguish them from subalgebras whose members do not necessarily satisfy P and Q. A *maximal* good subalgebra is a good subalgebra that is not contained in any other good subalgebra.

Since our subalgebras are defined by their top and bottom, our algorithm simultaneously works from both ends of the algebra 2^M. We can do this fairly efficiently, because the combinatorial explosion of the algebra occurs in the middle and not at the ends. Furthermore, while the sets on one end may be quite large, we can easily code them by the elements of M that they are missing instead of the elements that they contain. This will not affect our ability to evaluate most constraints. If we know both the average and size of M, then computing the average and size of A is no more difficult than computing the average and size of its complement $\overline{A}$; this fact is true of most statistical functions.

Our algorithm prunes subalgebras using P on one end and Q on the other. As an example, consider the algebra illustrated in Figure 2. At the first level, we see that $\{B\}$ does not satisfy P, and hence we remove all supersets of $\{B\}$. Furthermore, we see that $\overline{\{A\}} = \{B, C, D\}$ does not satisfy Q and so we remove all subsets of $\overline{\{A\}}$. We are left with the subalgebra $(\{A\}, \{A, C, D\})$. We can then repeat the algorithm on this subalgebra, but this is unnecessary. Since Q is monotone and $\{A\}$ satisfies Q, we know every element of this subalgebra satisfies Q. Similarly, since $\{A, C, D\}$ satisfies P, all elements of this subalgebra satisfy P. Thus we can determine that $(\{A\}, \{A, C, D\})$ is the only maximal good subalgebra without evaluating any of the interior nodes!

Note that if our query requires itemsets to be frequent, the large sets that we test with Q will probably not be frequent itemsets. However, for any single element x, the sets not containing x comprise half of the algebra. Therefore, there is some advantage to applying our constraints to very large sets, even though they may not satisfy P.

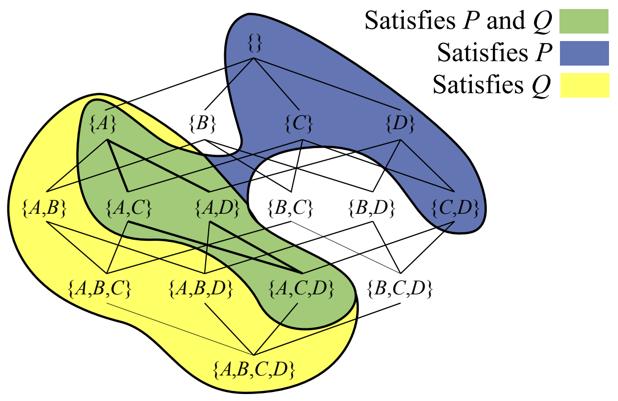

Figure 2. Simultaneously Pruning with P and Q

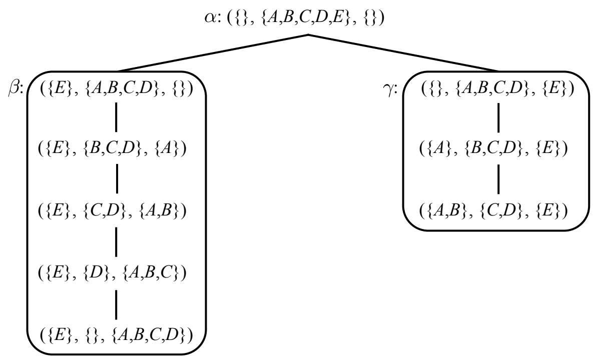

Figure 3. Evaluation Tree

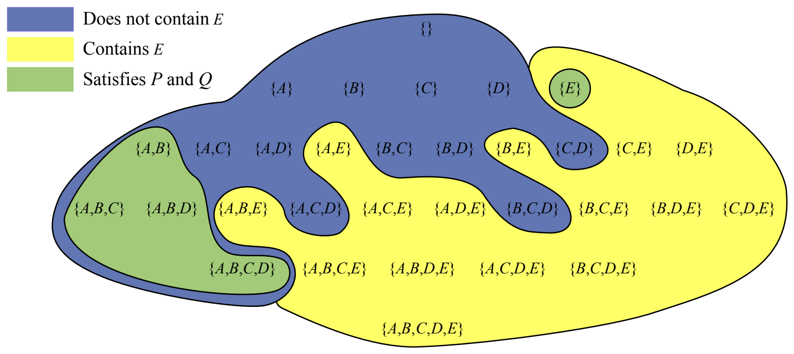
Does not contain E 🟦
Contains E 🟨
Satisfies P and Q 🟩
{}
{A} {B} {C} {D} {E}
{A,B} {A,C} {A,D} {A,E} {B,C} {B,D} {B,E} {C,D} {C,E} {D,E}
{A,B,C} {A,B,D} {A,B,E} {A,C,D} {A,C,E} {A,D,E} {B,C,D} {B,C,E} {B,D,E} {C,D,E}
{A,B,C,D} {A,B,C,E} {A,B,D,E} {A,C,D,E} {B,C,D,E}
{A,B,C,D,E}

Figure 4. Using the Evaluation Tree

2.2. Example Run-through

Suppose the database consists of the following two transactions: $\{A, B, C, D\}$ and $\{E\}$. The prices of A and B are both 26, the prices of C and D are both 1 and the price of E is 100. Furthermore, let P be the predicate "support$(X) \geq 1$" and let Q be the predicate "total_price$(X) > 50$." Figure 3 shows the evaluation tree of DualMiner.

Each node τ in the tree has a state that evolves as the algorithm runs. In this figure, the evolution of the state of each node is represented as a series of ordered triples of the form (X, Y, Z). We shall refer to the set X as IN(τ), Y as CHILD(τ) and Z as OUT(τ). When it is unambiguous, we shall just refer to IN, CHILD, and OUT. For each node τ, IN(τ) gives a lower bound on the bottom element of the subalgebra represented by the subtree rooted at τ; OUT(τ) represents a subset of the complement of the top element of that subalgebra; and CHILD(τ) represents the rest of the items. The evolution of a node occurs as it finds items that have already been placed in IN(τ) and OUT(τ). When the algorithm finds that no *single* item is either required to be in IN(τ) or required to be in OUT(τ), it chooses an item x from CHILD(τ) and considers both extensions to the state represented by the now-fully-evolved τ: that in which x is in IN(τ), or that in which x is in OUT(τ).

Initially we are at the root node α with state $(\{\}, \{A, B, C, D, E\}, \{\})$. Since every element is frequent, we cannot prune with P. We also cannot prune with Q since no single element is required to be in a set in order for its total price to exceed 50 (had Q been "total_price$(X) > 99$" then clearly a set must contain E in order to satisfy Q).

Note that $\{A, B, C, D, E\}$ is infrequent, so our good subalgebras cannot contain everything. Thus we have to make a choice. We choose an element, such as E, from CHILD(α) and explore what happens when a set contains this element. This leads to the creation of node β with state $(\{E\}, \{A, B, C, D\}, \{\})$. Since no itemset contains both E and A (i.e. $\{A, E\}$ does not satisfy P), we can remove A from consideration and move it from CHILD(β) to OUT(β). Now β has state $(\{E\}, \{B, C, D\}, \{A\})$. The same is done with B, C and D and β ends up in state $(\{E\}, \{\}, \{A, B, C, D\})$. We can interpret this to mean that every set that contains E and does not contain any of the items A, B, C, D satisfies P and Q. So the first algebra that we find is the singleton algebra $(\{E\}, \{E\})$.

After examining the sets that contain E (in node β) we go to node γ to explore sets that do not contain E. For this reason γ has the initial state $(\{\}, \{A, B, C, D\}, \{E\})$. We cannot prune using P, but now we see that a set not containing E has no hope of satisfying Q unless it contains A. Thus the state of γ changes to $(\{A\}, \{B, C, D\}, \{E\})$. Another iteration of pruning shows that any superset of A not containing E does not satisfy Q unless it also contains B. The state of γ becomes $(\{A, B\}, \{C, D\}, \{E\})$. At this point we cannot prune with any of our predicates, and so γ has reached its final state. We discover that $\{A, B\}$ satisfies Q while $\overline{\{E\}} = \{A, B, C, D\}$ satisfies P. Therefore we have found our second subalgebra $(\{A, B\}, \{A, B, C, D\})$. Thus the algorithm completes with the output $(\{E\}, \{E\}), (\{A, B\}, \{A, B, C, D\})$. This is illustrated in Figure 4.

In the next two sections, we will refer back to this example and Figure 3 for illustrative purposes. Unless otherwise mentioned, when we refer to the states of β and γ we are referring to their final states.

3. DualMiner

The basic algorithm dynamically builds a binary tree $\mathcal{T}$. Each node $\tau \in \mathcal{T}$ corresponds to a subalgebra of 2^M, which we refer to as SUBALG(τ). Note that these are not necessarily "good" subalgebras. Each subalgebra is associated with the following objects.

1. $\tau_{\text{new\_in}} \subseteq 2^M$ is one of the two (implicit) labels associated with the edge coming in to τ. If τ is the root, then $\tau_{\text{new\_in}}$ is defined to be $\emptyset$. Otherwise, if σ is the parent of τ, then $\tau_{\text{new\_in}}$ is generated from σ and represents items that should be included in every element of SUBALG(τ) but are not necessarily contained in *every* element of SUBALG(σ). In our example, $\beta_{\text{new\_in}} = \{E\}$, $\gamma_{\text{new\_in}} = \{A, B\}$.

2. $\tau_{\text{new\_out}} \subseteq 2^M$ is the other label associated with the edge coming in to τ. If τ is the root, then $\tau_{\text{new\_out}}$ is defined to be $\emptyset$. Otherwise, if σ is the parent of τ, then $\tau_{\text{new\_out}}$ is generated from σ and represents items that should *not* be included in *any* element of SUBALG(τ) but which are included in some element of SUBALG(σ). In our example, $\beta_{\text{new\_out}} = \{A, B, C, D\}$, $\gamma_{\text{new\_out}} = \{E\}$.

3. IN(τ) is the minimal set (bottom) of this subalgebra. Every element in SUBALG(τ) is a superset of IN(τ). IN(τ) is thus defined as:

$$\text{IN}(\tau) = \bigcup_{\rho \preceq \tau} \rho_{\text{new\_in}}$$

Here $\preceq$ is the standard ancestral relation; $\rho \preceq \tau$ if ρ is either τ or an ancestor of τ. In our example, IN(β) $= \{E\}$, IN(γ) $= \{A, B\}$.

4. $\mathrm{OUT}(\tau)$ is the complement of the maximal set (top) of this subalgebra. Every element in $\mathrm{SUBALG}(\tau)$ is a subset of $\overline{\mathrm{OUT}(\tau)}$. As with $\mathrm{IN}(\tau)$,
$$\mathrm{OUT}(\tau) = \bigcup_{\rho \preceq \tau} \rho_{\text{new\_out}}$$
In our example, $\mathrm{OUT}(\beta) = \{A, B, C, D\}$, $\mathrm{OUT}(\gamma) = \{E\}$. We will maintain the invariant that $\mathrm{IN}(\tau) \cap \mathrm{OUT}(\tau) = \emptyset$.

5. $\mathrm{CHILD}(\tau)$ is a macro for $\overline{\mathrm{IN}(\tau) \cup \mathrm{OUT}(\tau)}$ and so is the set of items representing the atoms of $\mathrm{SUBALG}(\tau)$. That is, every nontrivial element of $\mathrm{SUBALG}(\tau)$ is a union of some sets of the form $\mathrm{IN}(\tau) \cup \{x\}$ (where $x \in \mathrm{CHILD}(\tau)$). In our example, $\mathrm{CHILD}(\gamma) = \{C, D\}$.

Collectively, we refer to these objects as the *state* of τ.

As we dynamically build our tree, we classify nodes as either *determined* or *undetermined* and the classification of each node will change during the course of the algorithm. By default, every node starts out as undetermined. The state of an undetermined node may change. Because of this, we define the *s^{th}-iteration* of a node τ to be the s^{th} state assigned to it during the algorithm. For convenience, we refer to this as τ^s; hence the s^{th} version of $\mathrm{IN}(\tau)$ is $\mathrm{IN}(\tau^s)$. If we simply write τ, then we mean the most recent iteration of τ in our algorithm. Note that we do not consider τ^s to be a parent of τ^{s+1}. In our example, the node γ has three iterations and $\mathrm{IN}(\gamma^1) = \{A\}$ (because we start counting at 0).

At each stage, we visit a node that is undetermined and make it determined. We continue until all the nodes of $\mathcal{T}$ are determined. Our traversal strategy is irrelevant; any traversal strategy that visits parents before children is acceptable. This allows our algorithm some flexibility for the sake of optimizations. Therefore, at the highest level, we have DUAL_MINER, illustrated in Algorithm 1.

Algorithm 1 : DUAL_MINER

```
1: $\lambda_{\text{new\_in}}, \lambda_{\text{new\_out}} \leftarrow \emptyset$
2: $\mathcal{R} \leftarrow \emptyset$
3: while There are undetermined nodes do
4:    Traverse to the next undetermined node $\tau$.
5:    $G \leftarrow$ EXPAND_NODE($\tau$)
6:    $\mathcal{R} \leftarrow \mathcal{R} \cup G$
7:    Mark $\tau$ as determined.
8: end while
9: Answer = $\mathcal{R}$
```

Let λ be the initial node. $\mathrm{IN}(\lambda) = \mathrm{OUT}(\lambda) = \emptyset$. Note that by our definition of λ, $\mathrm{SUBALG}(\lambda)$ is the algebra 2^M. The algorithm EX-

PAND_NODE(τ) expands the tree $\mathcal{T}$ to break the algebra SUBALG(τ) into smaller, disjoint subalgebras that may be more easily searched. This is done by adding children to τ that specify further subalgebras of SUBALG(τ). It is important that the children of τ represent disjoint subalgebras of SUBALG(τ). This is not difficult because our algorithm ensures that no undetermined node has any children. All we have to do is chose some item $x \in$ CHILD(τ), and split SUBALG(τ) into those sets containing x and those sets not containing x. The result is EXPAND_NODE, shown in Algorithm 2.

Algorithm 2 : EXPAND_NODE(τ)

```
Require: $\tau$ is an undetermined node.
Returns: $G$, a good subalgebra or $\emptyset$.
1: $G \leftarrow$ PRUNE($\tau$)
2: if CHILD($\tau$) is not empty then
3:     Choose some $x \in$ CHILD($\tau$).
4:     $\rho_{\text{new\_in}}, \eta_{\text{new\_out}} \leftarrow \{x\}$
5:     $\rho_{\text{new\_out}}, \eta_{\text{new\_in}} \leftarrow \emptyset$
6:     Add $\rho$ and $\eta$ as children of $\tau$.
7: end if
8: return $G$
```

As mentioned before, the pruning strategy in PRUNE uses both ends of our algebra, evaluating both P and Q to generate successive states for τ. When we prune with respect to P, we look at each item $x \in$ CHILD(τ). If IN(τ)$\cup\{x\}$ does not satisfy P, then the antimonotone property of P implies that no superset of IN(τ) $\cup \{x\}$ satisfies P. This is equivalent to saying that no element of SUBALG(τ) containing x satisfies P. Therefore, we can put x into $\tau_{\text{new\_out}}$, further restricting SUBALG(τ). Putting this all together, we get MONO_PRUNE, which is shown in Algorithm 3.

An analogous result holds for pruning with respect to the predicate Q. If we replace everything in the algorithm MONO_PRUNE by its dual notion, we get ANTI_PRUNE, the algorithm for pruning with respect to Q. This is shown in Algorithm 4.

Both of these algorithms assume that SUBALG(τ) is an interesting algebra. It is possible that, as ANTI_PRUNE puts elements into $\tau_{\text{new\_in}}$, IN(τ) no longer satisfies P. In this case, no element of the subalgebra satisfies P, so we will not need to do further pruning or to construct children for this node. The easiest way to signify this is to empty CHILD(τ) by adding CHILD(τ) to $\tau_{\text{new\_out}}$. We represent this straightforward action as EMPTY_CHILD.

Algorithm 3 : MONO_PRUNE(τ^s)

```
Require: IN($\tau^s$) satisfies $P$
Returns: $\tau^{s+1}$
1: $\tau^{s+1}_{\text{new\_in}} \leftarrow \tau^{s}_{\text{new\_in}}$
2: $\tau^{s+1}_{\text{new\_out}} \leftarrow \tau^{s}_{\text{new\_out}}$
3: for all $x \in$ CHILD($\tau^s$) do
4:   if IN($\tau^s$) $\cup \{x\}$ does not satisfy $P$ then
5:     $\tau^{s+1}_{\text{new\_out}} \leftarrow \tau^{s+1}_{\text{new\_out}} \cup \{x\}$
6:     CHILD($\tau^{s+1}$) $\leftarrow$ CHILD($\tau^{s+1}$) $- \{x\}$
7:   end if
8: end for
9: return $\tau^{s+1}$
```

Algorithm 4 : ANTI_PRUNE(τ^s)

```
Require: $\overline{\text{OUT}(\tau^s)}$ satisfies $Q$
Returns: $\tau^{s+1}$
1: $\tau^{s+1}_{\text{new\_in}} \leftarrow \tau^{s}_{\text{new\_in}}$
2: $\tau^{s+1}_{\text{new\_out}} \leftarrow \tau^{s}_{\text{new\_out}}$
3: for all $x \in$ CHILD($\tau^s$) do
4:   if $\overline{\text{OUT}(\tau^s) \cup \{x\}}$ does not satisfy $Q$ then
5:     $\tau^{s+1}_{\text{new\_in}} \leftarrow \tau^{s+1}_{\text{new\_in}} \cup \{x\}$
6:     CHILD($\tau^{s+1}$) $\leftarrow$ CHILD($\tau^{s+1}$) $- \{x\}$
7:   end if
8: end for
```

It is clear that these pruning algorithms affect each other. The output of MONO_PRUNE is determined by IN(τ), which is in turn modified by the algorithm ANTI_PRUNE. Similarly, MONO_PRUNE modifies OUT(τ), which determines the output of ANTI_PRUNE. Therefore, it makes sense to interleave these algorithms until we reach a fixed point. The resulting pruning algorithm PRUNE(τ) is shown in Algorithm 5.

Note that PRUNE is the most extreme pruning strategy. It will not affect the correctness of our algorithm to do less pruning. We may choose only to do a fixed number of passes of MONO_PRUNE and ANTI_PRUNE. We may even choose to skip one or both of them altogether. This allows us some flexibility for optimization, as discussed in Section 4.

The correctness of this algorithm should be somewhat clear from the accompanying discussion. However, for a more rigorous proof, we present the following.

Algorithm 5 : PRUNE(τ)

```
Returns: A good subalgebra or $\emptyset$.
1:  $s \leftarrow 0$ (Note, at this point $\tau = \tau^0$ by definition)
2:  repeat
3:    if IN($\tau^s$) satisfies $P$ then
4:      $\tau^{s+1} \leftarrow$ MONO_PRUNE($\tau^s$)
5:    else
6:      $\tau^{s+1} \leftarrow$ EMPTY_CHILD($\tau^s$)
7:    end if
8:    $s \leftarrow s + 1$ // Pruning has changed the iteration
9:    if $\overline{\text{OUT}(\tau^s)}$ satisfies $Q$ then
10:     $\tau^{s+1} \leftarrow$ ANTI_PRUNE($\tau^s$)
11:   else
12:     $\tau^{s+1} \leftarrow$ EMPTY_CHILD($\tau^s$)
13:   end if
14:   $s \leftarrow s + 1$ // Pruning has changed the iteration
15: until $\tau^s_{\text{new\_in}} = \tau^{s-1}_{\text{new\_in}}$ and $\overline{\tau^s_{\text{new\_out}}} = \tau^{s-1}_{\text{new\_out}}$
16: if IN($\tau^s$) satisfies $Q$ and $\overline{\text{OUT}(\tau^s)}$ satisfies $P$ then
17:   $\tau^{s+1} \leftarrow$ EMPTY_CHILD($\tau^s$) (At this point $\tau = \tau^{s+1}$)
18:   return (IN($\tau^s$), $\overline{\text{OUT}(\tau^s)}$)
19: else
20:   return $\emptyset$
21: end if
```

Definition 4. The *depth* of a node τ^s is an ordered pair (p, s) where p is the number of nodes (excluding τ) whose descendants include τ, and s is the most recent iteration of τ. We define an ordering $\geq_\eta$ on depth as follows: $(p_1, s_1) \geq_\eta (p_2, s_2)$ if $p_1 > p_2$ or $(p_1 = p_2) \wedge (s_1 \geq s_2)$.

Theorem 1. *A set A satisfies $P \wedge Q$ if and only if A is an element of a subalgebra returned by EXPAND_NODE at some point in the algorithm.*

Proof. Note that EXPAND_NODE returns the same subalgebra that it receives from PRUNE. We will prove the direction assuming that A satisfies $P \wedge Q$; the other direction is clear. Define the predicate INCLUDED$_\tau$ as:

$$\text{INCLUDED}_\tau(A) = \text{IN}(\tau) \subseteq A \subseteq \overline{\text{OUT}(\tau)} \tag{1}$$

Note that INCLUDED$_{\lambda^0}(A)$ is true (where λ is the root node of $\mathcal{T}$). Let τ^s be the node of greatest depth for which (1) holds. If IN(τ^s) satisfies Q and $\overline{\text{OUT}(\tau^s)}$ satisfies P then clearly we are done. Otherwise, after calculating IN(τ^s) and $\overline{\text{OUT}(\tau^s)}$ in PRUNE, the algorithm would either call EMPTY_CHILD(τ^s), go through another pruning iteration, or add two children to τ^s in EXPAND_NODE.

If the algorithm called EMPTY_CHILD(τ^s) then either IN(τ^s) fails to satisfy P or $\overline{\mathrm{OUT}(\tau^s)}$ fails to satisfy Q. In the first case, since P is antimonotone, A also fails to satisfy P, a contradiction. The second case is similar because of symmetry.

If we go through another pruning iteration, then we get τ^{s+1} from either MONO_PRUNE or ANTI_PRUNE. In either case, it is clear that $\mathrm{INCLUDED}_{\tau^{s+1}}(A)$ holds, which contradicts the maximality of the depth of τ^s.

If we add two children to τ^s then let $x \in$ CHILD(τ) be the item defining the two children of τ in EXPAND_NODE. Assume that $x \in A$(the case that $x \notin A$ is symmetric), and let ρ be the child of τ such that $x \in \rho_{\text{new\_in}}$. Then $\mathrm{INCLUDED}_{\rho^0}(A)$ holds, which is also a contradiction. $\square$

4. Optimizations

The algorithm outlined above is in its most primitive form in order to make it easy to follow. There are several places in which it can be optimized. The most obvious optimization is to remove calls to MONO_PRUNE or ANTI_PRUNE that we know will not actually do any pruning. For example, if $\tau^s_{\text{new\_in}}$ is empty and σ is the parent of τ, then $\mathrm{IN}(\tau^s) = \mathrm{IN}(\sigma)$. Therefore, there is nothing to be gained calling MONO_PRUNE on τ^s when $\tau^s_{\text{new\_in}}$ is empty. A similar result holds for ANTI_PRUNE when $\tau^s_{\text{new\_out}}$ is empty.

Similarly, in the non-degenerate case, we run the same pruning algorithm on τ^{s+2} that we run on τ^s. Hence there is no point pruning τ^{s+2} with MONO_PRUNE if $\tau^{s+2}_{\text{new\_in}} = \tau^s_{\text{new\_in}}$. Part of this rationale is captured by the **repeat-until** loop in PRUNE. However, this loop continues until both $\tau_{\text{new\_in}}$ and $\tau_{\text{new\_out}}$ achieve a fixed point.

Another optimization issue depends on the cost structure of our predicates. Certain predicates may be very expensive on large sets, so we may wish to avoid computing P on $\overline{\mathrm{OUT}(\tau^s)}$ as we do in line 16 of PRUNE. In order to avoid this computation, we conservatively assume that $\overline{\mathrm{OUT}(\tau^s)}$ does not satisfy P and thus return the empty set. This is equivalent to replacing lines 16–21 of PRUNE with "return" (we don't care about the return value since it would always be the empty set). We will refer to this alternate version of PRUNE as PRUNE$'$. We then continue exploration of the subalgebra and use the observation that if τ has no children and IN(τ) satisfies P, then so will IN(τ') for any

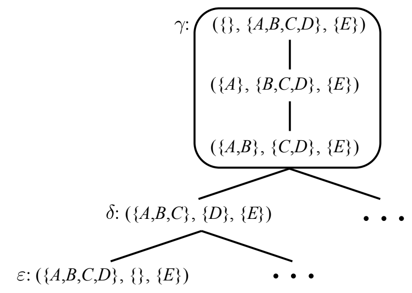

Figure 5. An Extended Evaluation Tree

ancestor τ' of τ. This is done by replacing EXPAND_NODE with the following algorithm, EXPAND_NODE^P.

Algorithm 6 : EXPAND_NODE$^P(\tau)$

```
Require: $\tau$ is an undetermined node.
Returns: a good subalgebra or $\emptyset$.
1: PRUNE$'(\tau)$
2: if CHILD$(\tau)$ is not empty then
3:     Choose some $x \in$ CHILD$(\tau)$.
4:     $\rho_{\text{new\_in}}, \eta_{\text{new\_out}} \leftarrow \{x\}$
5:     $\rho_{\text{new\_out}}, \eta_{\text{new\_in}} \leftarrow \emptyset$
6:     Add $\rho$ and $\eta$ as children of $\tau$.
7:     return $\emptyset$
8: else if IN$(\tau)$ satisfies $P$ then
9:     $\sigma \leftarrow$ the ancestor of $\tau$ of minimal depth satisfying $Q$.
10:    return $(\text{IN}(\sigma), \overline{\text{OUT}(\tau)})$
11: end if
```

To see how this alters the original algorithm, consider the example from Section 2.2. Instead of checking whether γ gives us a good subalgebra (which it does), we further expand it to nodes δ and ϵ as shown in Figure 5. Once we reach ϵ, we discover that $\{A, B, C, D\}$ satisfies P and hence every subset of it must also satisfy P. As $\gamma_{\text{new\_in}}$ satisfies Q, we see that $(\{A, B\}, \{A, B, C, D\})$ is a good subalgebra.

The proof that this new algorithm is correct is not that different from before. If A satisfies $P \wedge Q$, then it is an element of some subalgebra returned by EXPAND_NODE^P. Again, let τ be the node of greatest depth for which (1) in the proof of Theorem 1 holds (i.e. $\text{IN}(\tau) \subseteq A \subseteq \overline{\text{OUT}(\tau)}$). Subsequent calls to PRUNE$'$ will not prune A because the modified version of PRUNE and the old version still do the same pruning. This is because the affected lines (16-21 of PRUNE) are not involved in pruning the search space — they only delay the output of good subalgebras. Furthermore, we cannot expand the node τ, as one of the children would satisfy (1). Therefore, CHILD(τ) must be empty and so

$$A = \mathrm{IN}(\tau) = \overline{\mathrm{OUT}(\tau)}$$

Therefore A is output in EXPAND_NODE^P in line 10.

For the other half of correctness, we introduce the following definition.

Definition 5. A *complete left chain* is a sequence of nodes $\{\tau_k\}_{k \leq n}$ such that the following all hold.

1. CHILD(τ_0) $= \emptyset$
2. τ_{k+1} is the parent of τ_k for all $k < n$.
3. $(\tau_k)_{\mathrm{new\_out}} = (\tau_n)_{\mathrm{new\_out}}$ for all $k < n$.

Looking at Figure 5, we see that $\{\epsilon, \delta, \gamma\}$ forms a complete left chain. If we replace $(\tau_k)_{\mathrm{new\_out}}$ with $(\tau_k)_{\mathrm{new\_in}}$ in the previous definition, we get a *complete right chain.*

Note that for any ancestor σ of τ, $\mathrm{IN}(\sigma) \subseteq \mathrm{IN}(\tau)$ and $\overline{\mathrm{OUT}(\tau)} \subseteq \overline{\mathrm{OUT}(\sigma)}$. Therefore, since P is antimonotone and Q is monotone, we have the following result giving us the correctness of our new algorithm with EXPAND_NODE^P.

Proposition 2. *If $\{\tau_k\}_{k \leq n}$ is a complete left chain, every element of* SUBALG(τ_n) *satisfies P. Furthermore, if additionally* IN(τ_n) *satisfies Q, then every element of* SUBALG(τ_n) *satisfies $P \wedge Q$. A similar result holds when $\{\tau_k\}_{k \leq n}$ is a complete right chain.*

This means that once we find a node τ such that CHILD(τ) $= \emptyset$, we need only find the least $\sigma \preceq \tau$ such that

1. There is a complete left chain from τ to σ.
2. IN(σ) satisfies Q.

These two properties imply that all of SUBALG(σ) satisfy $P \wedge Q$. In this case, we should consider every descendant of σ to be determined, and choose the sibling of σ to be our next node in our traversal strategy.

A problem with the new algorithm is that, as written, EXPAND_-NODE^P outputs multiple subsets of a subalgebra. In the example above, the descendants of γ other than ϵ will also give us subsets of the same subalgebra, $(\{A, B\}, \{A, B, C, D\})$. To solve this problem, we can generalize the HUT strategy from the MAFIA algorithm [8]. We will call this strategy Dual HUT. In a standard depth-first traversal (of a frequent itemset algorithm), we know that if every element of the leftmost branch satisfies P, then everything to the right must also satisfy P (i.e. everything to the right is a subset of a set in the leftmost branch). Similarly, if $\sigma \preceq \tau$, then SUBALG(τ) $\subseteq$ SUBALG(σ). Therefore, if we can find a complete left chain from τ to σ, we can output SUBALG(σ) and mark all of the descendent's of σ as determined.

An analogous statement is true for complete right chains. Complete right chains are useful when the evaluation of Q is expensive while the evaluation of P is relatively cheap. To take advantage of such knowledge we can derive EXPAND_NODE^Q from EXPAND_NODE^P by appropriately modifying lines 10-12. We will refer to this technique for complete left and right chains as the Dual HUT strategy.

We are interested in complete left chains from τ to σ even if IN(σ) does not satisfy Q. We still know that every element of SUBALG(σ) satisfies P. Hence we no longer need to evaluate MONO_PRUNE for nodes in SUBALG(σ) even though we have to traverse them. A similar argument holds for Q if IN(σ) satisfies Q.

Because we value complete right chains as much as complete left chains in our algorithm, it may be advantageous to take a "steady state" approach to our traversal strategy. In this approach, if we are visiting the left child of a node (i.e. a child τ such that $\tau^0_{\text{new\_out}} = \emptyset$), we should continue choosing the left child as we descend the tree. Similarly, if we are visiting the right child of a node, we should continue choosing the right child as we descend the tree.

Now we can see how the traversal strategy may depend on the cost structure of P. For example, suppose P is a support constraint and support counting is done using bitmaps, as in the MAFIA algorithm. If we continue to visit left children (starting from a node σ), then the total cost of evaluating P on IN(τ) (for each node τ that we expand) is almost the same as the cost of evaluating P on $\overline{\text{OUT}(\sigma)}$. Therefore, there is no point in checking if $\overline{\text{OUT}(\sigma)}$ satisfies P, since we will find this out during our traversal of left children at almost the same cost. However, if P is a constraint of the form "$\min(X) > c$" then evaluating P takes constant time and so we can evaluate P on $\overline{\text{OUT}(\sigma)}$ in the hope that this will result in fewer nodes being expanded.

Other MAFIA optimizations [8], such as keeping a partial list of maximal frequent itemsets (or minimal sets that satisfy Q) may also be used to reduce the number of predicate evaluations.

Sometimes, evaluations of MONO_PRUNE or ANTI_PRUNE may be very inefficient, especially if they do not result in any pruning. Usually we can't predict these occurrences, but knowledge of the domain and knowledge of extra properties of P and Q can be utilized to develop heuristics. For example, suppose we are given the constraint total_price(X) $> c$ and the prices of items are low and relatively uniform. If CHILD(τ) has many items and IN(τ) has few (and so would have a small total price) we would not expect much pruning to be done. In this case, a suitable heuristic would avoid pruning with Q until the size of CHILD(τ) falls beneath some threshold. This would save many unnecessary evaluations of Q.

At the same time, some chances to prune with Q may be missed. It would be unfortunate if this results in an incorrect algorithm. This turns out not to be a problem with DualMiner. The proof of correctness in Theorem 1 is general enough to apply to any deterministic pruning strategy that satisfies the following properties:

1. The pruning phase is *safe* (i.e. no element is incorrectly moved to OUT(τ))
2. Once the pruning phase at a node stops, the algorithm performs a choice step (as in lines 3-6 of EXPAND_NODE) and subsequently it considers both children of that node.

With these guidelines, several types of heuristics are available to us:

Traversal Strategy: As hinted in line 4 of DUAL_MINER (Algorithm 1), nodes can be visited in almost any order. For example, nodes can be traversed in a depth-first Mafia-style order or a breadth-first Apriori-style order. They can even be visited using a hybrid traversal strategy. The choice of this heuristic depends on which implementation is considered to be more efficient.

Pruning Order Heuristics: These heuristics select the order in which elements from CHILD(τ) are selected for pruning tests (see line 3 of MONO_PRUNE and ANTI_PRUNE)

Choice Order Heuristics: Similar to the pruning order heuristics, they choose which element from CHILD(τ) to branch on (see line 3 of EXPAND_NODE).

Stop Heuristics: These heuristics decide when to stop performing pruning tests on elements in CHILD(τ). For example, stop when all the elements have been tested (this is what DualMiner does by default).

Control Heuristics: Control Heuristics decide the order of calls to MONO_PRUNE and ANTI_PRUNE and also decide when to end the pruning phase at a node. By default, DualMiner alternates pruning with P and Q and ends the pruning phase when no more changes can be made.

Once again, suppose Q is the constraint "total_price(X) $> c$." We can use the pruning and choice order heuristics to order the elements of CHILD(τ) by decreasing value. A stop heuristic would stop Q pruning tests as soon as an element is not placed in IN(τ) (i.e. it doesn't satisfy the "if" condition on line 4 in ANTI_PRUNE). Since no element with a smaller value could be placed in IN(τ) during this evaluation of ANTI_PRUNE, unnecessary work is avoided. A control heuristic could decide to alternate evaluations of ANTI_PRUNE and MONO_PRUNE until no changes are possible, *unless* $|\,\mathrm{CHILD}(\tau)|$ is larger than a specific threshold (in which case ANTI_PRUNE is not evaluated at all). Note that if Q were an upper bound on support rather than a lower bound on sum, the stop heuristics in conjunction with the order heuristics would not be as effective; a new set of heuristics would be needed.

With all this leeway, DualMiner is a highly customizable algorithmic framework. Its efficiency can be improved by utilizing domain-specific knowledge and properties of the desired constraints.

5. Complexity

We can consider the antimonotone predicate P and monotone predicate Q to be oracles whose answers satisfy the antimonotone (resp. monotone) constraints. Thus we identify the predicate P with the oracle that evaluates P on a set of items (and similarly for Q). The underlying dataset $\mathcal{D}$ and the set of all items $\mathcal{I}$ are assumed to be arbitrary but fixed and so need not be mentioned explicitly. Given these preliminaries, we can define the following sets (Ω will be used to denote an oracle/predicate which is either monotone or antimonotone):

$$\begin{aligned}
\text{Theory of } \Omega: \quad & \mathrm{Th}(\Omega) = \{S \mid \Omega(S)\} \\
\text{Border of } P: \quad & \mathrm{B}(P) = \{S \mid \forall T \subset S : P(T) \\
& \qquad \wedge \forall W \supset S : \neg P(W)\} \\
\text{Border of } Q: \quad & \mathrm{B}(Q) = \{S \mid \forall T \supset S : Q(T) \\
& \qquad \wedge \forall W \subset S : \neg Q(W)\} \\
\text{Positive Border of } \Omega: \quad & \mathrm{B}^{+}(\Omega) = \mathrm{B}(\Omega) \cap \mathrm{Th}(\Omega) \\
\text{Negative Border of } \Omega: \quad & \mathrm{B}^{-}(\Omega) = \mathrm{B}(\Omega) \cap \mathrm{Th}(\neg\Omega)
\end{aligned}$$

Note that the last two equations imply that the border of P is divided into the positive border (whose sets satisfy P) and the negative border (whose sets do not satisfy P). Also note that $\mathrm{B}^{-}(\Omega)$ is equivalent to

$\mathrm{B}(\Omega) - \mathrm{B}^{+}(\Omega)$ and is used for pruning, while $\mathrm{B}^{+}(\Omega)$ is used to verify that a set of items satisfies Ω. This observation leads to the following proposition by Gunopulos et al. [10]:

Proposition 3. *Given a monotone or antimonotone predicate* Ω*, computing* $\mathrm{Th}(\Omega)$ *requires at least* $|\,\mathrm{B}(\Omega)|$ *calls to the oracle* Ω*.*

In the sequel, quantities such as "$|\,\mathrm{B}(\Omega)|$ evaluations of Ω" will be written as "$|\,\mathrm{B}(\Omega)|_{\Omega}$ evaluations." Analogously, to compute all sets of items that satisfy P and Q (i.e. $\mathrm{Th}(P) \cap \mathrm{Th}(Q)$ - which we shall refer to as $\mathrm{Th}(P \wedge Q)$) we have the following bounds:

Proposition 4. *Computing* $\mathrm{Th}(P \wedge Q)$ *using the oracle model requires at least* $|\,\mathrm{Th}(Q) \cap B(P)|_P + |\,\mathrm{Th}(P) \cap B(Q)|_Q$ *oracle calls.*

Proof. The search space consists of four regions: $\mathrm{Th}(P) \cap \mathrm{Th}(Q)$, $\mathrm{Th}(P) \cap \mathrm{Th}(\neg Q)$, $\mathrm{Th}(\neg P) \cap \mathrm{Th}(Q)$ and $\mathrm{Th}(\neg P) \cap \mathrm{Th}(\neg Q)$. The first region cannot be described more succinctly than by the set of tops $(\mathrm{Th}(Q) \cap \mathrm{B}^{+}(P))$ and by the set of bottoms $(\mathrm{Th}(P) \cap \mathrm{B}^{+}(Q))$ of maximal subalgebras. The second and third regions are areas where only one predicate can be used to prune the search space. Thus knowing $\mathrm{Th}(P) \cap \mathrm{B}^{-}(Q)$ is necessary to prune part of the second region, while $\mathrm{B}^{-}(Q)$ is sufficient to prune all of it (neither bounds are very tight for reasons discussed later). Similar statements hold for the third region. Thus we need at least

$$\begin{aligned}&|\,\mathrm{Th}(Q) \cap \mathrm{B}^{+}(P)|_P + |\,\mathrm{Th}(Q) \cap \mathrm{B}^{+}(P)|_Q \\ &+|\,\mathrm{Th}(Q) \cap \mathrm{B}^{-}(P)|_P + |\,\mathrm{Th}(P) \cap \mathrm{B}^{-}(Q)|_Q \\ &= |\,\mathrm{Th}(Q) \cap B(P)|_P + |\,\mathrm{Th}(P) \cap B(Q)|_Q\end{aligned}$$

oracle calls. □

The presence of the fourth region $(\mathrm{Th}(\neg P) \cap \mathrm{Th}(\neg Q))$ shows why the bounds are not very tight. This region can be pruned using $\mathrm{B}^{-}(P) \cap \mathrm{Th}(\neg Q)$ or $\mathrm{B}^{-}(Q) \cap \mathrm{Th}(\neg P)$ or by using various sets from each of those borders. The best choice relies heavily on the relative costs and cost structures of P and Q as well as the structure of the areas they prune. For example, pruning some areas with P may require many sets from $\mathrm{B}^{-}(P) \cap \mathrm{Th}(\neg Q)$ while pruning the same area with Q may require less sets from $\mathrm{B}^{-}(Q) \cap \mathrm{Th}(\neg P)$. There may also be a lot of redundancy due to heavy overlapping of regions pruned by various sets in $\mathrm{B}^{-}(P) \cap \mathrm{Th}(\neg Q)$.

For the regions where only one predicate can be used to prune (such as $\mathrm{Th}(Q) \cap \mathrm{Th}(\neg P)$) we have similar issues. This region can be pruned by evaluating P on the $\mathrm{B}^{+}(\neg P \wedge Q)$ (i.e. the minimal sets that satisfy Q but not P). This set includes the necessary $\mathrm{Th}(Q) \cap \mathrm{B}^{-}(P)$ (as required by the previous proposition). However, we can also use the following subset of $\mathrm{B}^{\text{-}}(P)$ to prune the same region:

$$\mathcal{M} = \{S | S \in \mathrm{B}^{\text{-}}(P) \ \wedge \ \exists T \in \mathrm{Th}(\neg P) \cap \mathrm{Th}(Q) \ \wedge \ T \supseteq S\}$$

There are situations where either collection has a smaller cardinality, however $\mathcal{M}$ has the added benefit of pruning some of $\mathrm{Th}(\neg Q) \cap \mathrm{Th}(\neg P)$ as well.

Existing Algorithms

Running times for algorithms that mine constrained itemsets tend to be highly correlated with the number of predicate evaluations that are required (since the predicates are evaluated for each candidate set that is generated). Thus we use the number of evaluations of P and Q as the cost metric for analyzing DualMiner and various alternative algorithms.

The Apriori algorithm for computing $\mathrm{Th}(P)$ requires exactly $|\,\mathrm{Th}(P) \cup B^{-}(P)|_P$ oracle queries (since the collection of non-frequent candidate sets it generates is precisely the negative border). Similarly, its dual algorithm for calculating $\mathrm{Th}(Q)$ requires exactly $|\,\mathrm{Th}(Q) \cup B^{-}(Q)|_Q$ oracle queries.

MAFIA, in contrast to Apriori, uses a depth-first traversal strategy. Because of this, MAFIA cannot use a candidate generation algorithm as good as Apriori. In fact, it is possible that MAFIA tests the support of sets outside $\mathrm{Th}(P) \cup \mathrm{B}^{\text{-}}(P)$. For example, given the set of items $\{A, B, C, D, E\}$, if the transaction $\{A, B, C, D\}$ is frequent but $\{C, E\}$ is not, it is possible that MAFIA test the support of the transactions $\{A\}$, then $\{A, B\}$ then $\{A, B, C\}$, then $\{A, B, C, D\}$ and $\{A, B, C, D, E\}$. However $\{A, B, C, D, E\}$ is not in $\mathrm{Th}(P)$, nor is it in $\mathrm{B}^{\text{-}}(P)$ since $\{C, E\}$ is not frequent. If a set S is frequent, or has a frequent subset of size $|S| - 1$, then it is possible that MAFIA will test its support and so MAFIA will use at most $(N + 1)\,\mathrm{Th}(P)$ oracle calls (where N is the number of items). This is a very loose upper bound since MAFIA looks for maximal frequent itemsets and has various optimizations that reduce the number of sets in $\mathrm{Th}(P)$ whose support needs to be checked (such as the HUT strategy which can avoid testing the exponentially many subsets of a maximal frequent itemset). Let S be the collection of all sets in $\mathrm{Th}(P)$ for which MAFIA evaluates P. Then MAFIA will evaluate P for at most $N|S|$ sets outside $\mathrm{Th}(P)$. Thus MAFIA uses heuristics to reduce the size of S. It also has smaller memory requirements than Apriori (due to its depth-first rather than breadth-first traversal of the search space) and avoids Apriori's expensive candidate generation algorithm. In addition, there is experimental evidence to show that it is more efficient than Apriori [8].

Computing $\mathrm{Th}(P \wedge Q)$ can be done naively by separately running Apriori or MAFIA (using P), the corresponding dual algorithm (using Q) and then intersecting the results. This algorithm, which will be referred to as INTERSECT, clearly has a worst case bound $|\,\mathrm{Th}(P) \cup B^-(P)|_P + |\,\mathrm{Th}(Q) \cup B^-(Q)|_Q$ oracle queries (when using Apriori).

Another naive approach, POSTPROCESS, computes $\mathrm{Th}(P)$ and then post-processes the output by evaluating Q for each element of the output. This can be done using $|\,\mathrm{Th}(P) \cup B^-(P)|_P + |\,\mathrm{Th}(P)|_Q$ oracle evaluations. POSTPROCESS does not leverage the monotone properties of Q, and so can be improved as in [20]: for each $x \in \mathrm{Th}(P)$ that is found, evaluate $Q(x)$ unless we know $Q(t)$ is true for some $t \subset x$. This algorithm (when using Apriori), which we will refer to as CONVERTIBLE, evaluates Q on every set in $\mathrm{Th}(P) \cap (\mathrm{Th}(\neg Q) \cup \mathrm{B}^+(Q))$ and thus uses $|\,\mathrm{Th}(P) \cup B^-(P)|_P + |\,\mathrm{Th}(P) \cap \mathrm{Th}(\neg Q)|_Q + |\,\mathrm{Th}(P) \cap \mathrm{B}^+(Q)|_Q$ predicate evaluations. Assuming P is more selective than Q (as is usually true when P contains a support constraint), the CONVERTIBLE algorithm will tend to dominate both POSTPROCESS and INTERSECTION. It will also be very efficient when Q is not very selective since then $|\,\mathrm{Th}(\neg Q)|$ is small.

One more alternative to DualMiner is an enhanced version of Mellish's algorithm [22]. This algorithm outputs two sets S and G (the collection of tops of all maximal subalgebras and the collection of all bottoms, respectively). While this representation is very compact, considerable work needs to be done to output $\mathrm{Th}(P \wedge Q)$ from its result. The algorithm takes as input a predicate of the form $c_1 \wedge c_2 \wedge \cdots \wedge c_n$ where each c_i is either a monotone or antimonotone predicate. MELLISH+ computes S and G for c_1 (using a top-down or bottom-up level-wise algorithm similar to Apriori). The borders are then refined using c_2 and a level-wise algorithm that starts at the appropriate border S or G (depending on the type of the constraint c_2). This process is then repeated for the predicates $c_3, \ldots, c_n$. This algorithm is very likely to waste computation because it does not combine all the monotone predicates into one (more selective) monotone predicate (through the use of conjunctions) and similarly for antimonotone predicates. The result is that predicate evaluations occur for sets that could have already been pruned. For this reason the running time is heavily influenced by the order in which the predicates are presented to the algorithm. To avoid this, we can merge all the antimonotone predicates into one (and similarly for the monotone ones) and then use the antimonotone predicate first (since it is likely to be more selective). Assuming P has a support constraint (and is thus probably more selective than Q) it makes sense to run the bottom-up level-wise algorithm for P first. If top-down algorithm is then run for Q, the resulting complexity is similar to CONVERTIBLE. If the bottom-up version is used for Q then we can get the following upper bounds: $|\,\mathrm{Th}(P) \cup \mathrm{B}^{-}(P)|_P + |\,\mathrm{Th}(P) \cap \mathrm{Th}(Q)|_Q + |\,\mathrm{Th}(P) \cap \mathrm{B}^{-}(Q)|_Q$ evaluations.

The main distinction between DualMiner and its competitors is that DualMiner interleaves the pruning of P and Q. The other algorithms can be logically separated into two phases (even though they may not be implemented that way): finding $\mathrm{Th}(P)$ and then refining the result to compute $\mathrm{Th}(P) \cap \mathrm{Th}(Q)$. This is not very efficient for selective Q since its pruning power is not used in the first phase.

As the results for MAFIA indicate, it is possible that DualMiner (using depth-first traversal) can make an oracle call for a set outside $\mathrm{Th}(P) \cup \mathrm{B}^{-}(P)$ and $\mathrm{Th}(Q) \cup \mathrm{B}^{-}(Q)$. However, the same thing can happen for a breadth-first traversal strategy. The reason for this is that pruning with Q can eliminate some frequent itemsets from consideration. This prevents the use of the Apriori candidate generation algorithm: for a given set S, we may not be able to verify (without additional evaluations of P) that all of its subsets of size $|S| - 1$ are frequent. Thus using a depth-first or breadth-first strategy, DualMiner will never make more than $(N+1)|\,\mathrm{Th}(P)|_P + (N+1)|\,\mathrm{Th}(Q)|_Q + |\,\mathrm{Th}(P) \cap \mathrm{Th}(\neg Q)|_Q$ oracle calls. The last term in the sum is the result of DualMiner testing for the bottom of a subalgebra. This is an overly pessimistic upper bound since it does not take into account the savings we get from pruning $\mathrm{Th}(P)$ with Q and vice-versa. Also, optimizations can be used to reduce the number of evaluations of P and Q. For example, for depth-first strategy, we can use the same optimizations that MAFIA uses [8].

6. The Subalgebra Fragmentation Problem

The same thing that lets DualMiner beat its complexity estimates can also cause a disadvantage. The performance of DualMiner may be adversely affected by a fragmentation problem due to our use of subalgebras. While this output representation is more compact than just outputting itemsets, DualMiner does not output the minimal representation in terms of subalgebras. This happens because every time a choice on an item is made, the search space is split in two halves, and if a subalgebra exists that covers parts of both halves then that subalgebra would also be split. It turns out that a subalgebra can be split arbitrarily many times. This fragmentation problem can be addressed either by trying to merge the resulting subalgebras or by using some strategies in choosing the splitting item to minimize the fragmentation problem (for example if Q is "total_price$(X) > 100$," we may choose an item with large price - this resulted in an algorithm that will be referred to as DualMiner+q in the experiments). The second approach is more promising, even though we cannot eliminate the problem entirely.

An example that illustrates the fragmentation problem is the following. Suppose the items are $A_1, A_2, \ldots, A_n, B, C$, with prices $1, 1, \ldots, 1$, $n+1, n+1$, the transaction database contains transactions $\{A_1, A_2, \ldots, A_n, B\}$ and $\{A_1, A_2, \ldots, A_n, C\}$. Let P be "support(X) $\geq$ threshold (> 0)" and let Q be "total_price(X) $> n$." The most compact representation of the solution in this case is $(\{B\}, \{A_1, A_2, \ldots, A_n, B\})$ and $(\{C\}, \{A_1, A_2, \ldots, A_n, C\})$, which happens if the algorithm first selects B, then C (or first C then B). If the first choices are made on some A_i's (as is very likely for DualMiner without using some clever strategies) then these two subalgebras can become very fragmented.

6.1. Splitting and Merging Subalgebras

A subalgebra can be split into two or more disjoint subalgebras. We will consider the first case, the other cases are just generalizations of the first one, that is the resulting two subalgebras are further split in two and so on. We first need to establish the possible cases for splitting a subalgebra or the reverse operation, that is merging two subalgebras into one.

Given that the subalgebras produced by DualMiner are complete, that is have the form $(B, T) = \{X \mid B \subseteq X \subseteq T\}$, the following lemma gives us the information required for splitting and merging complete subalgebras.

Lemma 5. *Let $G = (B, T)$, $G_1 = (B_1, T_1)$, $G_2 = (B_2, T_2)$ be complete subalgebras. Then $G = G_1 \cup G_2$, $G_1 \cap G_2 = \emptyset$ if and only if, for some $E \in T \setminus B$, either of the following hold:*

1. $B_1 = B$, $B_2 = B \cup \{E\}$, $T_1 = T \setminus \{E\}$, $T_2 = T$;
2. $B_1 = B \cup \{E\}$, $B_2 = B$, $T_1 = T$, $T_2 = T \setminus \{E\}$.

Proof. First we prove the reverse direction. Suppose (1) holds. It is clear that $G_1 \subset G$ and $G_2 \subset G$, so $G_1 \cup G_2 \subseteq G$. To show that $G \subseteq G_1 \cup G_2$, take $X \in G$. We have either $E \in X$, or $E \notin X$. If $E \in X$, then $B \cup \{E\} \subseteq X \subseteq T$, so $X \in G_2$. If $E \notin X$, then $B \subseteq X \subseteq T \setminus \{E\}$, so $X \in G_1$. Therefore $G \subseteq G_1 \cup G_2$, then $G = G_1 \cup G_2$. It is clear that $G_1 \cap G_2 = \emptyset$ because every set in G_1 does not contain E and every set in G_2 contains E. The proof for when (2) holds is similar.

Now suppose $G = G_1 \cup G_2$, $G_1 \cap G_2 = \emptyset$. Because $B \in G$, we have that $B \in G_1$ or $B \in G_2$. Suppose $B \in G_1$. Then we can show that (1) holds. In the case when $B \in G_2$, we can show that (2) holds using similar arguments. $B \in G_1$ implies that $B_1 \subseteq B$. $B_1 \in G_1$ implies that

$B_1 \in G_1 \cup G_2 = G$, then $B \subseteq B_1$. Therefore $B_1 = B$. $T \in G$ implies that $T \in G_1$ or $T \in G_2$. But $T \notin G_1$ otherwise $G = G_1$ and $G_2 = \emptyset$ which contradicts lemma's assumptions. So $T \in G_2$. Then $T \subseteq T_2$. Similar to $B \subseteq B_1$ we can show that $T_2 \subseteq T$. Therefore $T_2 = T$.

We also have $B \subset B_2$ and $T_1 \subset T$. We can not have equality because in that case $G_1 \cap G_2 \neq \emptyset$ would contradict the hypothesis. Assume $B_2 = B \cup \{E_1, E_2, \ldots, E_k\}$, for some $k \geq 1$ and $E_1, E_2, \ldots, E_k \in T \setminus B$. If $k \geq 2$, then $\forall i \leq k, B \cup \{E_i\} \notin G_2$, since $B \cup \{E_i\} \subset B \cup \{E_1, E_2, \ldots, E_k\}$, the latter being the bottom element of G_2. Since $B \cup \{E_i\} \in G$, it follows that $B \cup \{E_i\} \in G \setminus G_2 = G_1$ for any $i \leq k$. As G_1 is a complete subalgebra, it must contain $B \cup \{E_1, E_2, \ldots, E_k\} = B_2$. This contradicts the fact that $G_1 \cap G_2 = \emptyset$. Therefore it must be the case that $k = 1$. So $B_2 = B \cup \{E\}$, considering $E = E_1$. Similarly, we can show that $T_1 = T \setminus \{E'\}$. It is easy to see that $E = E'$, otherwise we could get $B_2 = B \cup \{E\} \subseteq T \setminus \{E'\}$ then $G_1 \cap G_2 \neq \emptyset$, again a contradiction. $\square$

The above lemma says that a subalgebra can be split into two subalgebras, one of them having all sets that contain a chosen element and the other one having all sets that do not contain the same chosen element. It also tells that two subalgebras can be merged into one subalgebra only when all the sets of one subalgebra can be obtained from the sets of the other subalgebra by adding or removing a single element.

Thus, merging two subalgebras reduces to checking for these cases among the subalgebras in the output. One solution would be to post-process the result, but there is a better solution, that is merging subalgebras during the execution of the algorithm.

6.2. Adjusting the Algorithm

In order to do the merging of subalgebras during the normal execution of the algorithm, we need to add a new field to the nodes of our search tree. Thus, in addition to IN, CHILD and OUT, a node will have a field $SUBALGS$ that would contain some subalgebras of algebra $(\mathrm{IN}, \overline{\mathrm{OUT}})$. The result of the algorithm will be the union of all the sets SUBALG for all the expanded nodes of the search tree.

We only need to modify Algorithm 1 (DUAL_MINER) and Algorithm 2 (EXPAND_NODE(τ)). In DUAL_MINER we do not need lines 2 and 6, and in line 9 we replace $\mathcal{R}$ with the union of the fields $SUBALGS$ described above. EXPAND_NODE needs not return a value anymore, since it would not be used in DUAL_MINER. In EXPAND_-NODE we add G to $SUBALGS(\tau)$ and then call MERGE(G, τ).

The new algorithm would insure that there is no more merging possible. However, the output is not necessarily optimal, as can be seen

Algorithm 7 : MERGE(G, τ)

```
Require: $G = (B, T)$ is a good subalgebra, $\tau$ is a node.
 1: for all ancestors $\sigma$ of $\tau$ (taken from the closest to $\tau$ to the farthest,
    and including $\tau$) do
 2:     if any of the children of $\sigma$ is undetermined then
 3:         continue
 4:     end if
 5:     let $\rho$ be the child of $\sigma$ that is an ancestor of $\tau$ and let $\eta$ be the
        other child.
 6:     if $\rho^0_{new\_in} = \{E\}$ then
 7:         let $G' = (B \setminus \{E\}, T \setminus \{E\})$
 8:     else
 9:         let $G' = (B \cup \{E\}, T \cup \{E\})$
10:     end if
11:     look for subalgebra $G'$ in the tree rooted at $\eta$ (This can be done
        efficiently by following only one path in the tree.)
12:     while $G'$ was found do
13:         remove $G$ and $G'$ from their location, let $G = G \cup G'$ and add
            $G$ to $SUBALGS(\sigma)$
14:         look for a subalgebra $G'$ in $SUBALGS(\sigma)$ that can be merged
            to $G$.
15:     end while
16: end for
```

from the following example. Consider two transactions $\{A, B, C\}$ and $\{B, C, D, E\}$ where item B has a high price (e.g. 100) while the other items have low prices (e.g. 1). Furthermore, let P be "support$(X) \geq 0.5$" and let Q be "total_price$(X) \geq 50$." Then the solution would be $(\{B\}, \{A, B, C\}), (\{B\}, \{B, C, D, E\})$. However, these two subalgebras are overlapping; that is they both contain subalgebra$(\{B\}, \{B, C\})$, so they can not be output by DUAL_MINER. We could have either $(\{A, B\}, \{A, B, C\})$ and $(\{B\}, \{B, C, D, E\})$ or $(\{B\}, \{A, B, C\}), (\{B, D\}, \{B, C, D\})$, and $(\{B, E\}, \{B, C, D, E\})$. While the former representation is desirable (especially when we replace $\{E\}$ with $\{E, F, G, \dots\}$, as then the second subalgebra can be split into more than two subalgebras), the latter is also possible. We may want to first split $(\{B\}, \{A, B, C\})$ into $(\{A, B\}, \{A, B, C\})$ and $(\{B\}, \{B, C\})$ and then merge the second subalgebra into $(\{B, D\}, \{B, C, D\})$ and $(\{B, E\}, \{B, C, D, E\})$ to form $(\{B\}, \{B, C, D, E\})$, but that requires much more work as we would need to split every subalgebra we find at the beginning, then to decide which representation is better. If we allowed overlapping subalgebras we would still have to break a subalgebra in order to be able to merge its components with other subalgebras. But breaking subalgebras is not advisable, because the complexity of the problem would increase too much.

The following theorem establishes the correctness of the modified algorithm.

Theorem 6. *At the end of the execution of the algorithm, there are no subalgebras that could be merged.*

Proof. We prove the theorem by contradiction. Suppose there are two subalgebras that could have been merged. They are either at the same node in the tree or not. In the case they are located at the same node in the tree, we have a contradiction with lines 13-14 of algorithm MERGE. So, they must be located at different nodes in the tree. Now, if they are located at nodes that are descendants of a node on its two different branches, they should have been merged at lines 5-14. The only case that is left is that one subalgebra $G_1 = (B_1, T_1)$ is located at a node σ_1 that is a descendant of the node σ_2 where the other subalgebra $G_2 = (B_2, T_2)$ is located. Let $B = B_1 \cap B_2$. Then, there is some E such that either $E \in B_1$, but $E \notin B_2$, or $E \notin T_1$, but $E \in T_2$. E is in fact the element that is contained in σ^0_{2,new_in} or in σ^0_{2,new_out} respectively. But this case is not among the cases of Lemma 5. This completes the proof. □

7. Experimental Results

While DualMiner interleaves the pruning of P and Q, the other algorithms (POSTPROCESS, MELLISH+, INTERSECTION, CONVERTIBLE) essentially calculate $\mathrm{Th}(P)$ and then refine the result (any algorithm that does this will be called a “2-phase” algorithm). If Q has little selectivity, then CONVERTIBLE is expected to be the best algorithm, since it examines $\mathrm{Th}(\neg Q)$ (which should be a small set). However, if Q is selective then it is possible that 2-phase algorithms waste too much time finding $\mathrm{Th}(P)$. Our experiments show that in this case DualMiner beats even a “super” 2-phase algorithm (where the second phase is computed at no cost). To make this evaluation, we assume that P is more expensive than Q to evaluate. If P is a support constraint and Q is a constraint on the sum of prices, it is reasonable to expect that P is about N times more expensive to compute than Q (where N is the number of transactions in the database). We use a more conservative estimate and say that P is only 100 times as expensive as Q ($P = 100Q$). For the first phase of the 2-phase algorithms, we used a MAFIA implementation since it turns out to be more efficient than Apriori. We used the IBM data generator to create the transaction files. Prices were selected using either uniform or Zipf distributions.

The IBM synthetic data generator [24] is given a set of parameters, of which the following are of direct interest to us: *ntrans, avglen, patlen, nitems, npats.* The number of transactions is given by *ntrans,* the length of a transaction is determined by a *Poisson* distribution with parameter *avglen.* The length of a transaction may be zero, thus the number of transactions generated may be less than *ntrans.* The number of patterns is given by *npats* and the average size of a pattern is given by *patlen.* Finally, the number of distinct items in all the transactions is given by *nitems.* The values used for these parameters are displayed on the top of the graphs of this section. The value for *npats* was the default one, that is 10000. The other two parameters on the graphs are *unif,* which denotes the distribution of the prices, and *support,* which gives the minimum support for which an itemset is considered frequent.

We also compare the evaluations of Q for DualMiner (using depth-first traversal) and an implementation of CONVERTIBLE that uses a MAFIA-style traversal for the first phase. The predicate Q was of the form "total_price(x) $>$ qthreshold" (the value of "qthreshold" is taken from the x-axis), and P was a support constraint. Here we also show results for DualMiner+q, an optimization which chooses the most expensive item to split on.

Figures 6 and 7 show the number of evaluations of Q for CONVERTIBLE, DualMiner and DualMiner+q vs. the selectivity of Q, using a uniform and a Zipf price distribution, respectively. The y-axis is the number of evaluations of Q and the x-axis is the threshold that the sum of prices in an itemset must exceed. As expected, CONVERTIBLE makes much less oracle calls when Q is not selective, but DualMiner does better as the selectivity of Q increases. There is a sharp spike in the graph for the Zipf distribution. At this point DualMiner needs more evaluations of Q even though it is more selective. This could be a characteristic of the distribution and the fact that DualMiner may evaluate Q outside Th(Q) when it is looking for the bottom of a subalgebra.

Figures 8 and 9 compare total oracle queries for DualMiner and the first phase of the 2-phase algorithms (using uniform and Zipf price distributions, respectively). Here P is 100 times as expensive as Q. The y-axis is the weighted sum $E_P + \frac{E_Q}{100}$ (where E_P is the number of evaluations of P, and similarly for Q) and the x-axis is the same as before.

Figure 10 compares total oracle queries for DualMiner and the first phase of the 2-phase algorithms. The y-axis is the weighted sum $E_P + E_Q$ (P and Q are weighted equally) and the x-axis is the same as before.

DualMiner does very well when Q is selective and inexpensive; it is also competitive when Q is just as expensive as P and is also selective (keeping in mind that all 2-phase algorithms need to do an extra refinement step with Q). When Q is expensive and not very selective, DualMiner performs too many evaluations of Q (in this circumstance, CONVERTIBLE would be the algorithm of choice, since it does not waste time pruning with an ineffective Q and only looks at Th($\neg Q$), which is a relatively small set).

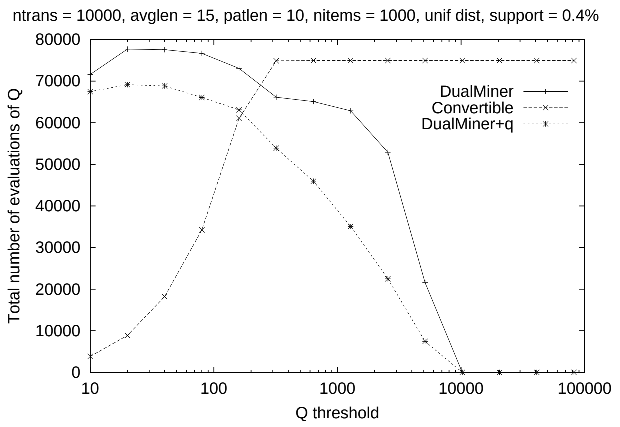

Figure 6. Evaluations of Q vs. Selectivity of Q

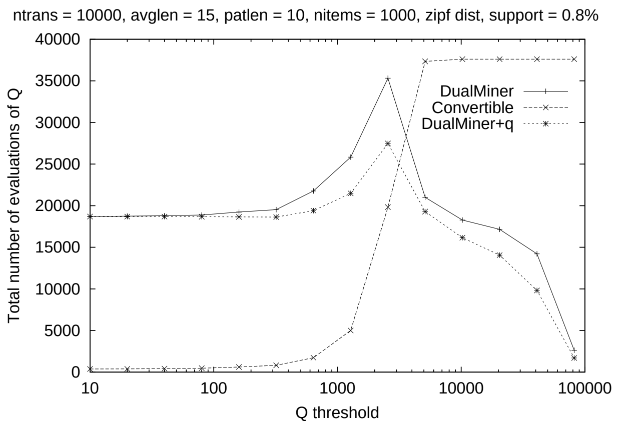

Figure 7. Evaluations of Q vs. Selectivity of Q

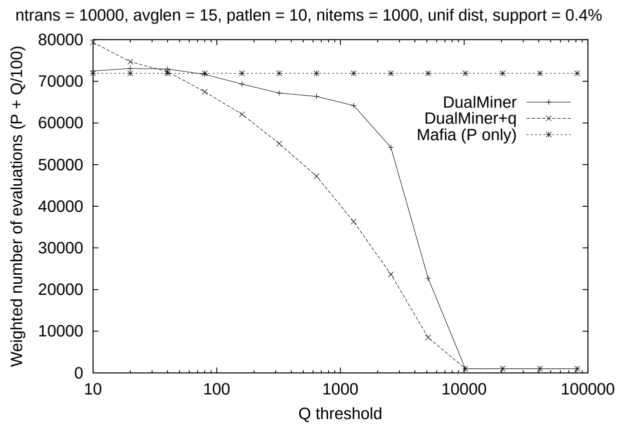

Figure 8. Oracle Calls ($E_P + \frac{E_Q}{100}$) vs Selectivity of Q

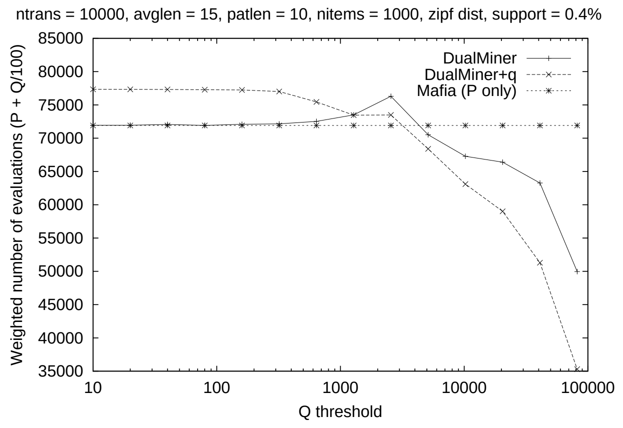

Figure 9. Oracle Calls ($E_P + \frac{E_Q}{100}$) vs Selectivity of Q

8. Approximations

While monotone and antimonotone predicates behave nicely, not all constraints have the form $P \wedge Q$. We may have a support constraint together with an unclassified constraint like "avg$(X) < c$." In addition, a monotone or antimonotone predicate may be too expensive to evaluate many times. If we can approximate these constraints with either monotone or antimonotone predicates (that aren't too expensive), then we can use them in DualMiner in the place of the original constraints. An approximation is necessarily a weaker constraint and might not prune all possible itemsets. As a result, DualMiner will return a superset of the correct result. The output should be filtered by a post-processing step that checks the actual constraint on the itemsets that were produced.

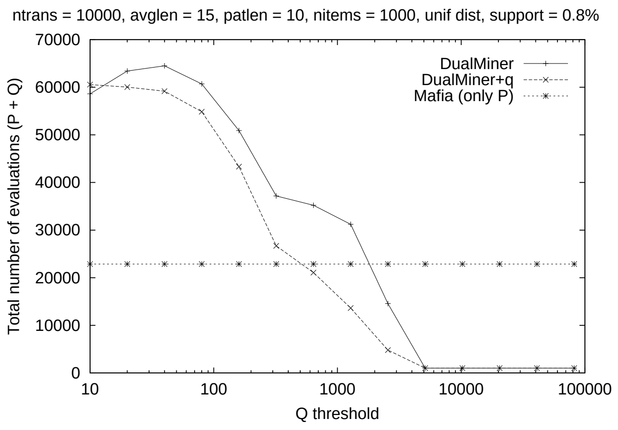

Figure 10. Oracle Calls ($E_P + E_Q$) vs Selectivity of Q

Definition 6. Let Q be a predicate on the algebra $\mathcal{A}$. A predicate Q^+ is a *positive approximation* of Q if for any $X \in \mathcal{A}$, $Q^+(X) \Rightarrow Q(X)$. Similarly, Q^- is a *negative approximation* of Q if $\neg Q^-(X) \Rightarrow \neg Q(X)$.

Note that our basic algorithm is still correct if we replace Q in ANTI_PRUNE with a monotone, negative approximation Q^-, provided that we post-process the final output with Q. However, if we wish to use the Dual HUT strategy outlined in Section 4, we need a monotone, positive approximation of Q as well (in order to use the complete right chain). Similarly, we could replace P in MONO_PRUNE with negative and positive approximations that are antimonotone.

Many constraints have the form "$f(X) \geq c$" for some aggregate function f. This happens to include constraints of the form "$f(X) \leq c$" since it is equivalent to "$-f(X) \geq -c$." A common example of such a constraint is "avg$(X) \leq c$."

Han et. al. [12] have implicitly studied approximations for these types of constraints. They suggested looking at the k smallest elements (min$_k$). Given a subalgebra with least element B (such that $|B| \geq k$) and greatest element T, if "avg(min$_k(T)) \leq c$" is false, (i.e. the average of the k smallest values exceeds c) then "avg$(X) \leq c$" is false for any X in that subalgebra. Thus "avg(min$_k(T)) \leq c$" is a negative approximation of "avg$(X) \leq c$."

For technical reasons, $k \leq |B|$ because "avg(min$_k(T)) \leq c$" is not a negative approximation for for sets of size less than k. To see why, let $k = |T|$. Then avg(min$_k(T)$) = avg(T). However, we can't infer much about the rest of the sets in the algebra (subsets of T) even if we know "avg$(T) \geq c$" or "avg$(T) \leq c$." When we use this approximation to evaluate the top of the subalgebra (as is the case when we prune with Q) it acts like a monotone function since it prunes away subsets.

Note that we do not have to use the same approximation for a constraint throughout the algorithm; we can use a different approximation each time that we call ANTI_PRUNE. This is especially useful when combined with the observation that we do not really need monotone or antimonotone approximations when τ is not the root node of our tree; we only need for our approximations to be monotone or antimonotone on SUBALG(τ).

Definition 7. Let $\mathcal{A}$ be a subalgebra of 2^M. A constraint Q is $\mathcal{A}$-monotone if it is monotone when restricted to the sets in $\mathcal{A}$. Similarly, a constraint P is $\mathcal{A}$-antimonotone if it is antimonotone when restricted to sets in $\mathcal{A}$.

Thus "$\text{avg}(\min_k(T)) \leq c$" is $\mathcal{A}$-monotone for any subalgebra $\mathcal{A}$ whose itemsets have cardinality at least k. Therefore, we can use this approximation in DualMiner by selecting, at each node τ, the approximation "$\text{avg}(\min_k(T_\tau)) \leq c$" where $k = |\,\text{IN}(\tau)|$ and T_τ is the top of SUBALG(τ).

8.1. Mean-Like Aggregate Functions

We can generalize the idea above to a larger class of aggregate functions.

Definition 8. An aggregate function f *weakly respects union* if for any set A, singleton $\{x\}$ and number c

1. if $f(A), f(\{x\}) \leq c$, then $f(A \cup \{x\}) \leq c$
2. if $f(A), f(\{x\}) \geq c$, then $f(A \cup \{x\}) \geq c$

An aggregate function f is *mean-like* if, in addition to weakly preserving union, for any set A, and elements $f(x) \leq f(y)$, $f(A \cup \{x\}) \leq f(A \cup \{y\})$.

Note that this definition implies that the domain of f is composed of sets, not multisets. This is not true of average. However, in practice, all of our items have unique identifiers and we are taking the average of the values of the items, not the items themselves. In this case, our aggregate function is actually

$$f(A) = \frac{1}{|A|} \sum_{x \in A} g(x)$$

where g is the function that maps the unique identifier of an item to its value. If we consider it in this context, where the ordering on the identifiers is induced by $f \circ g$, then average satisfies this definition. So does median and any moment aggregate $\text{avg}(X^k)$. It is important to note that $\min_k(X)$ means the set of the k least elements of X *under this ordering.* A similar thing is true for $\max_k(X)$.

Proposition 7. *Let f be any aggregate function on $\mathcal{A}$ that is mean-like. Furthermore suppose that every element in $\mathcal{A}$ has cardinality at least k. Then "$f(\min_k(X)) \geq c$" is an $\mathcal{A}$-antimonotone, positive approximation of "$f(X) \geq c$" and "$f(\max_k(X)) \geq c$" is an $\mathcal{A}$-monotone, negative approximation of "$f(X) \geq c$." Similarly, "$f(\max_k(X)) \leq c$" and "$f(\min_k(X)) \leq c$" are $\mathcal{A}$-antimonotone, positive and $\mathcal{A}$-monotone, negative approximations of "$f(X) \leq c$," respectively.*

Proof. We will only show that "$f(\max_k(X)) \geq c$" is an $\mathcal{A}$-monotone, negative approximation of "$f(X) \geq c$." The proofs of the other claims are analogous. First, we need to prove that it is $\mathcal{A}$-monotone. It is enough to show that, for any set X and singleton $\{x\}$, $f(\max_k(X \cup \{x\})) \geq f(\max_k(X))$; the rest then follows by induction.

There is nothing to prove unless $f(x) > f(y)$ for some $y \in \max_k(X)$. Choose y from $\max_k(X)$ such that $f(y)$ is smallest. Let $Z = \max_k(X) \setminus \{y\}$. Then, since f is mean-like,

$$f(\max_k(X \cup \{x\})) = f(Z \cup \{x\}) \geq f(Z \cup \{y\}) = f(\max_k(X))$$

To show that it is a negative approximation, we need only show that $f(X) \leq f(\max_k(X))$ for any set X. Let $Z = X \setminus \max_k(X)$. Let $d = f(\max_k(X))$. Since f weakly preserves union, induction on k shows that there must be some element $y \in \max_k(X)$ such that $f(y) \leq d$. This implies that $f(x) \leq d$ for all $x \in Z$. Another induction on the elements of Z therefore shows that

$$f(X) = f(Z \cup \max_k(X)) \leq d = f(\max_k(X))$$

Thus if $f(\max_k(X)) \geq c$ is false, so is $f(X) \geq c$. □

Finally, there are several statistical aggregates, such as variance, that are not mean-like. However, variance is an algebraic combination of aggregate functions that are.

Theorem 8. *Let $p(x_0, \ldots, x_n)$ be a multinomial. Also, let $f_0, \ldots, f_n$ be nonnegative, mean-like aggregate functions. Then for any subalgebra $\mathcal{A}$ containing elements with cardinality at least k, we can use $\min_k$ and $\max_k$ to define $\mathcal{A}$-antimonotone, positive and $\mathcal{A}$-monotone, negative approximations of the constraint "$p(f_0(X), \ldots, f_n(X)) \geq c$."*

Proof. Fix a subalgebra $\mathcal{A}$ containing elements with cardinality at least k. Let N be the number of terms in p. We can express p as

$$p(f_0(X), \ldots, f_n(X)) = \sum_{j=1}^{N} a_j \prod_{\ell=0}^{n} [f_\ell(X)]^{m_{(j,\ell)}}$$

where the $m_{(j,\ell)}$ are nonnegative integers. We construct the $\mathcal{A}$-antimonotone, positive approximation g as follows:

$$g \;=\; \sum_{j:\, a_j \geq 0} a_j \prod_{\ell=0}^{n} \left[f_\ell(\min_k(X)) \right]^{m_{(j,\ell)}} + \sum_{j:\, a_j < 0} a_j \prod_{\ell=0}^{n} \left[f_\ell(\max_k(X)) \right]^{m_{(j,\ell)}}$$

Clearly "$g(X) \geq c$" is a positive approximation of "$p(f_0(X), \ldots, f_n(X)) \geq c$," and since it isdecreasing it is $\mathcal{A}$-antimonotone as well. A similar argument gives us an $\mathcal{A}$-monotone, negative approximation h of this constraint as well. □

An immediate application of this theorem is that "$\text{avg}(\min_k(X^2)) - \text{avg}(\max_k(X))^2 \geq c$" and "$\text{avg}(\max_k(X^2)) - \text{avg}(\min_k(X))^2 \geq c$" are $\mathcal{A}$-antimonotone, positive and $\mathcal{A}$-monotone, negative approximations, respectively, for "$\text{var}(X) \geq c$" when all items have nonnegative values (this a trivial restriction because variance is translation invariant). However, of these two, only the negative approximation is useful, as the positive approximation almost always returns a negative value.

8.2. Optimization Considerations

If approximations are used, then a new type of optimization heuristic may be useful.

Function Choice Heuristic When using approximations, we are not limited to just one approximation. As previously discussed, at each node (or even at every pruning test) we can choose a new heuristic. For example, given the constraint "$\text{avg}(X) \geq c$," we can use the monotone, negative approximation "$\text{avg}(\max_k(X)) \geq c$" and we can keep changing k at every node so that $k = |\,\text{IN}(X)|$.

It is important to note that the presence of function choice heuristics adds complications. If Q is a monotone function, calling ANTI_PRUNE twice (or more times) consecutively will have the exact same result as calling it once. If one uses function choice heuristics in such a way that the approximation depends on $|\,\text{IN}(X)|$, consecutive calls to ANTI_PRUNE are likely to use two different approximations and produce different results than a single call to ANTI_PRUNE. It is important to realize this when designing the control heuristic.

9. Related Work

Agrawal et al. first introduced the problem of mining frequent itemsets as a first step in mining association rules [1]. They also considered item constraints such as an item must or must not be contained in an association rule. Agrawal and Srikant introduced the Apriori algorithm and some variations of it [3, 2]. Srikant et al. generalized this mining problem to item constraints over taxonomies[23]. Other types of constraints were introduced later by Ng et al. [17, 16]. These papers introduced the concepts of antimonotone and succinct constraints and presented methods for using them to prune the search space. These classes of constraints were also studied in the case of 2-variable constraints [14] and along with monotone constraints were further generalized and studied by Pei et al. [20, 18]. Boulicant and Jeudy present algorithms for mining frequent itemsets with both antimonotone and non antimonotone constraints [6, 7]. However they assume that the minimal itemsets satisfying the monotone constraint are easy to compute, also the minimum size of such itemsets is one and there is no gap in the sizes of itemsets that satisfy all the constraints, assumptions that frequently do not hold. This problem was also given a theoretical treatment by Gunopoulos et al. [10]. Some recent papers study the problem in the context of multi attribute data of high dimensionality [21] or take another approach to the problem, such as not pushing the constraints deeply into the mining process, but enforcing the constraints in a final phase [13]. Other papers present specializations of previous algorithms, based on FP-trees [15] or based on projected databases [19].

10. Conclusions and Future Work

It is clear from our experiments that taking advantage of the structures of *both* monotone and antimonotone predicates yields good results. In the case of constrained frequent itemsets, the use of a monotone predicate to remove from consideration what can be considered “uninteresting” itemsets is beneficial.

The success of DualMiner in exploiting the structures of two classes of constraints leads to several interesting areas of future work. Can other similar classes of constraints, such as convertible constraints [20], be incorporated as well? Can predicates such as “$\text{var}(X) \leq c$” (which do not seem to have “nice” structures) also be used efficiently? We are also interested in what kinds of implications this has for mining constrained sequential patterns.

Acknowledgments. This work was sponsored by NSF grants IIS-0121175 and IIS-0084762, by the Cornell Intelligent Information Systems Institute, and by generous gifts from Microsoft and Intel. We thank the anonymous reviewers for helpful comments.

References

1. R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 207–216. ACM Press, 1993.
2. R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast Discovery of Association Rules. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, chapter 12, pages 307–328. AAAI/MIT Press, 1996.
3. R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 487–499. Morgan Kaufmann, 1994.
4. R. J. Bayardo. Efficiently mining long patterns from databases. In Haas and Tiwary [11], pages 85–93.
5. R. J. Bayardo, R. Agrawal, and D. Gunopulos. Constraint-based rule mining in large, dense databases. *Data Mining and Knowledge Discovery*, 4(2/3):217–240, 2000.
6. J. Boulicaut and B. Jeudy. Using constraints during set mining: Should we prune or not, 2000.
7. J.-F. Boulicaut and B. Jeudy. Mining free itemsets under constraints. In *International Database Engineering and Application Symposium*, pages 322–329, 2001.
8. D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *ICDE 2001*. IEEE Computer Society, 2001.
9. A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors. *SIGMOD 1999, Philadephia, Pennsylvania, USA*. ACM Press, 1999.
10. D. Gunopulos, H. Mannila, R. Khardon, and H. Toivonen. Data mining, hypergraph transversals, and machine learning. In *Proc. PODS 1997*, pages 209–216, 1997.
11. L. M. Haas and A. Tiwary, editors. *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*. ACM Press, 1998.
12. J. Han, J. Pei, G. Dong, and K. Wang. Efficient computation of iceberg cubes with complex measures. In *SIGMOD Conference*, 2001.
13. J. Hipp and U. Guntzer. Is pushing constraints deeply into the mining algorithms really what we want? *SIGKDD Explorations*, 4(1):50–55, 2002.
14. L. V. S. Lakshmanan, R. T. Ng, J. Han, and A. Pang. Optimization of constrained frequent set queries with 2-variable constraints. In Delis et al. [9], pages 157–168.
15. C. K.-S. Leung, L. V. Lakshmanan, and R. T. Ng. Exploiting succinct constraints using fp-trees. *SIGKDD Explorations*, 4(1):31–39, 2002.

16. R. T. Ng, L. V. S. Lakshmanan, J. Han, and T. Mah. Exploratory mining via constrained frequent set queries. In Delis et al. [9], pages 556–558.
17. R. T. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained association rules. In Haas and Tiwary [11], pages 13–24.
18. J. Pei and J. Han. Can we push more constraints into frequent pattern mining? In *ACM SIGKDD Conference*, pages 350–354, 2000.
19. J. Pei and J. Han. Constrained frequent pattern mining: A pattern-growth view. *SIGKDD Explorations*, 4(1):31–39, 2002.
20. J. Pei, J. Han, and L. V. S. Lakshmanan. Mining frequent item sets with convertible constraints. In *ICDE 2001*, pages 433–442. IEEE Computer Society, 2001.
21. C.-S. Perng, H. Wang, S. Ma, and J. L. Hellerstein. Discovery in multi-attribute data with user-defined constraints. *SIGKDD Explorations*, 4(1):56–64, 2002.
22. L. D. Raedt and S. Kramer. The levelwise version space algorithm and its application to molecular fragment finding. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 853–862, August 2001.
23. R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In D. Heckerman, H. Mannila, D. Pregibon, and R. Uthurusamy, editors, *Proc. 3rd Int. Conf. Knowledge Discovery and Data Mining, KDD*, pages 67–73. AAAI Press, 14–17 1997.
24. IBM data generator. http://www.almaden.ibm.com/cs/quest/syndata.html.