# DualMiner: A Dual-Pruning Algorithm for Itemsets with Constraints

Cristian Bucila, Johannes Gehrke and Daniel Kifer
Dept. of Computer Science
Cornell University
{cristi,johannes,dkifer}@cs.cornell.edu

Walker White[*]
Department of Mathematics
University of Dallas
wmwhite@udallas.edu

## ABSTRACT

Constraint-based mining of itemsets for questions such as "find all frequent itemsets where the total price is at least \$50" has received much attention recently. Two classes of constraints, monotone and antimonotone, have been identified as very useful. There are 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 the first algorithm (called DualMiner) that uses both monotone and antimonotone constraints to prune its search space. 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.

## 1. INTRODUCTION

Mining frequent itemsets in the presence of constraints is an important data mining problem [5, 9, 10, 11, 12]. (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). In the remainder of this paper, we will 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. A predicate over a set of items is a condition that the set has to satisfy. 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 $X$. (In the remainder of this paper, we will use the terms predicate and constraint interchangeably.) We can now define the problem of constrained-based market basket analysis as follows:

*Given a set of predicates $P_1, P_2, \ldots, P_n$, find all sets in the powerset 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. [10, 9, 5, 11, 12] and there exist algorithms that are designed to take advantage of each class of constraints. However, the main deficiency in such algorithms is that they efficiently handle only one class of constraints. More recently, Raedt and Kramer [13] have generalized these algorithms to allow several types of constraints, but their generalization can handle only one type of constraint at a time. We present a new algorithm, DualMiner, which can simultaneously take advantage of *both* monotone and antimonotone predicates to efficiently mine constraint-based itemsets.

Previous work has shown that the search space of all itemsets forms a lattice. Actually, the search space of all itemsets forms a special type of lattice, namely an algebra, and we use algebras to succinctly represent the output in much the same way that maximal frequent itemsets can be used to succinctly represent the set of all frequent itemsets [4]. This representation adds to the efficiency of DualMiner and naturally leads to the following cost metrics as comparisons between different algorithms: number of nodes (in the search space) 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 calculated in constant time if the size of the set $X$ is bounded (which is generally true) 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, when using bitmaps as in the MAFIA algorithm [6], the cost also depends on the number of transactions in the database. 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

Let us first formally introduce monotone and antimonotone constraints. We begin by introducing the most popular antimonotone constraint, *support*. Consider a database that consists of a collection of nonempty subsets of $M$ called *transactions*. For a database $\mathcal{D}$ and a subset $X \subseteq M$, we define the function support$(X)$ to be the number of trans-

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

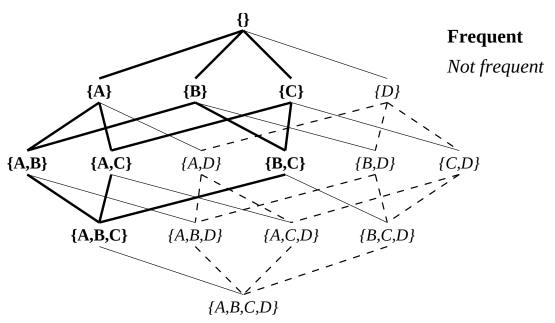

**Figure 1: Frequent Itemsets for $M = \{A, B, C, D\}$**

actions in $\mathcal{D}$ which contain the set $X$ (the "support" of $X$).[1] Note that we can choose the predicate $\text{support}(X) \geq c$ (for some constant $c$), and thus make the problem of frequent itemset mining a special case of the problem of constrained-based market analysis.

We can now introduce the notion of an antimonotone predicate such as $\text{support}(X) \geq c$.

*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 advantage of an antimonotone predicate $P$ is that if a set $X$ does not satisfy $P$, then no superset of $X$ can satisfy $P$. Whenever we find such a set $X$, we can prune away a substantial part of the search space. This pruning can result in substantial improvements in 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 the simply the unions of the smaller sets.

For example, consider the itemset lattice illustrated in Figure 1 (with the set of items $M = \{A, B, C, D\}$). The 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 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 also forms a *Boolean algebra* (which we will call an *algebra*). Like a lattice, an algebra has union and intersection, but it also has complementation. In the algebra of itemsets, the complement of a set is the result of 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 levelwise 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 $\text{support}(X) \leq c$ 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 appear naturally when we add constraints to frequent itemset analysis. 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 [11] have suggested several monotone approximations for the constraints $\text{avg}(X) \leq a$ and $\text{avg}(X) \geq b$; however, there are no known effective antimonotone approximations for these constraints.

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 [13], their algorithm performs a levelwise search with respect to the antimonotone predicate, followed by a levelwise 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 $Q$ simultaneosly 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 the algorithm DualMiner, an algorithm that can prune using *both* monotone and antimonotone constraints (Sections 2 and 3). This is the very first algorithm in the literature that can prune using both monotone and anti-monotone constraints.

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

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

- We analyze the complexity of our algorithm and compare it to other algorithms from the literature (Section 5).
- In a thorough experimental study, we show that DualMiner significantly outperforms previous work (Section 6).

## 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 [6]. Not only does this permit several optimizations in the algorithm, it provides a very concise representation of the output. For example, in Figure 1, the maximal frequent itemset $\{A, B, C\}$ uniquely defines the collection of all frequent itemsets in this algebra.

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

However, there is a suitable analogue of a maximal frequent itemset for this case. 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 $\overline{\phantom{x}}$. Given any collection of elements $\Gamma \subseteq 2^M$ that is closed under $\cap$ and $\cup$, we can define an algebra for $\Gamma$ using the following definitions.

1. $T = \bigcup_{X \in \Gamma} X$ is the top element of $\Gamma$.
2. $B = \bigcap_{X \in \Gamma} X$ is the bottom element of $\Gamma$.
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 structures 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, which in turn can each be compactly represented as a pair $(T, B)$ where $T$ is the top of the subalgebra and $B$ is the bottom. Clearly if both $P(T)$ and $Q(B)$ are true then any superset $X$ of $B$ which is also is a subset of $T$ satisfies $P(X) \wedge Q(X)$. 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 the top and bottom, our algorithm will simultaneously work 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.

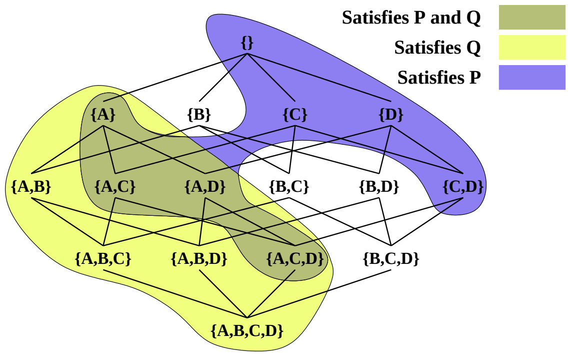

**Figure 2: Simultaneously Pruning with $P$ and $Q$**

Our algorithm is levelwise; at each level we will prune the algebra with $P$ on one end, and with $Q$ on the other end. 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 $P$. So 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 larger sets that we test with $Q$ will probably not be actual 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 $P$.

### 2.2 Example Run-through

Suppose the database consists of the following two transactions: $ABCD$ 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 $\geq 1$ and let $Q$ be the predicate total_price $> 50$. Figure 3 shows the evaluation tree of DualMiner. Each node $\tau$ in the tree has a state that can be represented as an ordered triple of the form $(X, Y, Z)$. We shall refer to the set $X$ as $\text{IN}(\tau)$, $Y$ as $\text{CHILD}(\tau)$ and $Z$ as $\text{OUT}(\tau)$. When it is unambiguous, we shall just refer to IN, CHILD, and OUT. Initially we are at the root node $\alpha$ with state $(\emptyset, ABCDE, \emptyset)$. 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

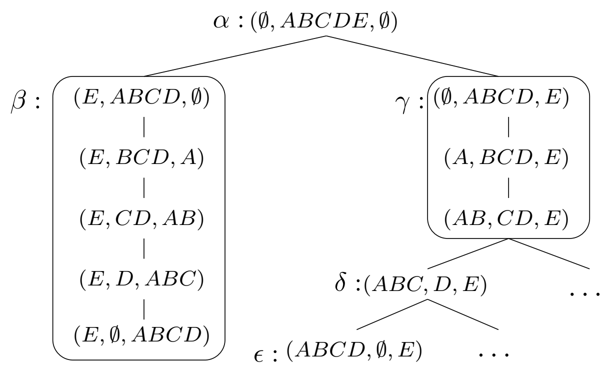
$\alpha : (\emptyset, ABCDE, \emptyset)$
$\beta :$ $(E, ABCD, \emptyset)$
$(E, BCD, A)$
$(E, CD, AB)$
$(E, D, ABC)$
$(E, \emptyset, ABCD)$
$\gamma :$ $(\emptyset, ABCD, E)$
$(A, BCD, E)$
$(AB, CD, E)$
$\delta : (ABC, D, E)$
$\epsilon : (ABCD, \emptyset, E)$
. . .
. . .

**Figure 3: Evaluation Tree**

total_price > 99 then clearly a set must contain $E$ or it will not satisfy $Q$). Thus we have a choice. We choose an element, such as $E$, from CHILD($\alpha$) and explore what happens when a set contains this element. This leads to the creation of node $\beta$ with state $(E, ABCD, \emptyset)$. Since no itemset contains both $E$ and $A$ (i.e. $EA$ does not satisfy $P$), we can remove $A$ from consideration and move it from CHILD($\beta$) to OUT($\beta$). Now $\beta$ has state $(E, BCD, A)$. The same is done with $B, C$ and $D$ and $\beta$ ends up in state $(E, \emptyset, ABCD)$. 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 found is the singleton $(E, E)$.

After examining sets that contain $E$ (in node $\beta$) we go to node $\gamma$ to explore sets that do not contain $E$. For this reason $\gamma$ has the initial state $(\emptyset, ABCD, E)$. 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 $\gamma$ changes to $(A, BCD, E)$. Another iteration of pruning show that any superset of $A$ not containing $E$ does not satisfy $Q$ unless it also contains $B$. The state of $\gamma$ becomes $(AB, CD, E)$. (Note that $\text{IN}(\gamma) = \{A, B\}$ satisfies both $P$ and $Q$ and none of $\gamma$'s ancestors have this property. This means that $AB$ is the minimal set of a subalgebra). At this point we cannot prune with any of our predicates (and so $\gamma$ has reached its final state). Once again we have to make a choice. A choice on $C$ leads to the node $\delta$ with state $(ABC, D, E)$ (here we examine what happens when a set contains $A, B$ and $C$ but not $E$). We cannot do any pruning at node $\delta$, so we make another choice and arrive at the node $\epsilon$ with state $(ABCD, \emptyset, E)$. We knew that $AB$ satisfied $Q$ and now we know $ABCD$ satisfies $P$ (otherwise $D$ would have been moved to OUT($\delta$)). From the monotone property of $Q$ and antimonotone property of $P$ we know that $(AB, ABCD)$ is an algebra that satisfies $P$ and $Q$ (all supersets of $AB$ that are subsets of $ABCD$ satisfy both $P$ and $Q$). In addition to this, knowing that $ABCD$ satisfies $P$ means that we do not have to examine the right children of nodes $\delta$ and $\epsilon$. It is clear that those nodes would only examine sets that belong to the algebra $(AB, ABCD)$. Thus the algorithm completes with the output $(E, E)$ and $(AB, ABCD)$. The sequence $\{\epsilon, \delta, \gamma\}$ is called a complete left chain due to its appearance in 3 (this idea will be formalized in Section 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 $\beta$ and $\gamma$ 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($\tau$). 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 $\tau$. If $\tau$ is the root, then $\tau_{\text{new\_in}}$ is defined to be $\emptyset$. Otherwise, if $\sigma$ is the parent of $\tau$, then $\tau_{\text{new\_in}}$ is generated from $\sigma$ and represents items that should be included in every element of SUBALG($\tau$) but are not necessarily contained in *every* element of SUBALG($\sigma$). In our example, $\gamma_{\text{new\_in}} = \{A, B\}$, $\delta_{\text{new\_in}} = \{C\}$.

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

3. IN($\tau$) is the minimal set of this subalgebra. Every element in SUBALG($\tau$) is a superset of IN($\tau$). IN($\tau$) 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 $\rho$ is either $\tau$ or an ancestor of $\tau$. In our example, $\text{IN}\,\delta = \{A, B, C\}$

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

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

Collectively, we refer to these objects as the *state* of $\tau$.

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. If a node is undetermined, then the state of $\tau$ may change. Because the state may change, we define the *$s^{th}$-iteration* of a node $\tau$ to be the $s^{\text{th}}$ state assigned to it during the algorithm. For convenience, we refer to this as $\tau^s$; hence the $s^{\text{th}}$ version of IN($\tau$) is IN($\tau^s$). If we simply write $\tau$, then we mean the most recent iteration of $\tau$ in our algorithm. Note that we do not consider $\tau^s$ to be a parent of $\tau^{s+1}$. Thus the node $\gamma$ 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 $\lambda$ be the initial node. $\mathrm{IN}(\lambda) = \mathrm{OUT}(\lambda) = \emptyset$. Note that by our definition of $\lambda$, SUBALG($\lambda$) is the algebra $2^M$. The algorithm EXPAND_NODE($\tau$) expands the tree $\mathcal{T}$ to break the algebra SUBALG($\tau$) into smaller, disjoint subalgebras that may be more easily searched. This is done by adding children to $\tau$ that specify further subalgebras of SUBALG($\tau$). It is important that the children of $\tau$ represent disjoint subalgebras of SUBALG($\tau$). 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($\tau$), and split SUBALG($\tau$) into those sets containing $x$ and those sets not containing $x$. The result is EXPAND_NODE, shown in Algorithm 2.

```
Algorithm 2 : EXPAND_NODE($\tau$)
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 above, the pruning strategy in PRUNE uses both ends of our algebra, evaluating both $P$ and $Q$ to generate successive states for $\tau$. When we prune with respect to $P$, we look at each item $x \in$ CHILD($\tau$). If $\mathrm{IN}(\tau) \cup \{x\}$ does not satisfy $P$, then the antimonotone property of $P$ implies that no superset of $\mathrm{IN}(\tau) \cup \{x\}$ satisfies $P$. This is equivalent to saying that no element of SUBALG($\tau$) containing $x$ satisfies $P$. Therefore, we can put $x$ into $\tau_{\text{new\_out}}$, further restricting SUBALG($\tau$). Putting this all together, we get MONO_PRUNE, which is shown in Algorithm 3.

An analogous result holds for pruning with respect to $Q$. 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.

```
Algorithm 3 : MONO_PRUNE($\tau^s$)
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($\tau^s$)
Require: $\overline{\mathrm{OUT}(\tau^s)}$ 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 $\overline{\mathrm{OUT}(\tau^s)} \cup \{x\}$ 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
```

Both of these algorithms assume that SUBALG($\tau$) is an interesting algebra. It is possible that, as ANTI_PRUNE puts elements into $\tau_{\text{new\_in}}$, IN($\tau$) 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($\tau$) by adding CHILD($\tau$) to $\tau_{\text{new\_out}}$. We represent this straightforward action as EMPTY_CHILD.

It is clear that these pruning algorithms affect each other. The output of MONO_PRUNE is determined by IN($\tau$), which is in turn modified by the algorithm ANTI_PRUNE. Similarly, MONO_PRUNE modifies OUT($\tau$), 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($\tau$) 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 chose only to do a fixed number of passes on each of the algorithms 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.

*Definition 4.* The *depth* of a node $\tau^s$ is an ordered pair $(p, s)$ where $p$ is the number of nodes (excluding $\tau$) whose descendants include $\tau$, and $s$ is the most recent iteration of $\tau$. 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. *An set $A$ satisfies $P \wedge Q$ if and only if $A$ is an element of a subalgebra returned by PRUNE at some point in the algorithm.*

**Algorithm 5** : PRUNE($\tau$)

```
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 $\tau^s_{\text{new\_out}} = \tau^{s-1}_{\text{new\_out}}$
16: if IN($\tau^s$) satisfies $Q$ and $\overline{\text{OUT}(\tau^s)}$ satisfies $P$ then
17:   return (IN($\tau^s$), $\overline{\text{OUT}(\tau^s)}$)
18: else
19:   return $\emptyset$
20: end if
```

Proof. We will prove the direction assuming that $A$ satisfies $P \wedge Q$; the other direction is clear. Define the predicate INCLUDED$_\tau$ as:

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

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

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

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

If we add a child to $\tau^s$ then let $x \in$ CHILD($\tau$) be the item defining the two children of $\tau$ in EXPAND_NODE. By symmetry, assume without loss of generality that $x \in A$, and let $\rho$ be the child of $\tau$ such that $x \in \rho_{\text{new\_in}}$. Then INCLUDED$_{\rho^0}(A)$ holds, which is also 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 $\sigma$ is the parent of $\tau$, then IN($\tau^s$) = IN($\sigma$). Therefore, there is nothing to be gained calling MONO_PRUNE on $\tau^s$ 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 $\tau^{s+2}$ that we run on $\tau^s$. Hence there is no point pruning $\tau^{s+2}$ 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.

A more subtle optimization is an analogue of the HUT strategy from the MAFIA algorithm [6]. 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). To generalize that concept, we introduce the following definition.

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

1. CHILD($\tau_0$) = $\emptyset$
2. $\tau_{k+1}$ is the parent of $\tau_k$ for all $k < n$.
3. $(\tau_k)_{\text{new\_out}} = \emptyset$ for all $k < n$.

Looking at Figure 3, we see that $\{\epsilon, \delta, \gamma\}$ forms a complete left chain. If we replace $(\tau_k)_{\text{new\_out}}$ with $(\tau_k)_{\text{new\_in}}$ in the previous definition, we get a *complete right chain*.

The antimonotone property of $P$ and monotone property of $Q$ imply the following.

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

This means that once we find a node $\tau$ such that CHILD($\tau$) = $\emptyset$ and $\tau_{\text{new\_out}} = \emptyset$, we need only find the least $\sigma \preceq \tau$ such that

1. There is a complete left chain from $\tau$ to $\sigma$.
2. IN($\sigma$) satisfies $Q$.

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

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

The traversal strategy also depends 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 $\sigma$), then the total cost of evaluating $P$ on $\mathrm{IN}(\tau)$ (for each node $\tau$ that we expand) is almost the same as the cost of evaluating $P$ on $\overline{\mathrm{OUT}(\sigma)}$. Therefore, there is no point in checking if $\overline{\mathrm{OUT}(\sigma)}$ 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 $\overline{\mathrm{OUT}(\sigma)}$ in the hope that this will result in fewer nodes being expanded.

Other MAFIA optimizations [6], 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.

## 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 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 ($\Omega$ will be used to denote an oracle/predicate which is either monotone or antimonotone):

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

Note that $\mathrm{B}^{-}(\Omega)$ 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 satisfy $\Omega$. This observation leads to the following proposition by Gunopulos et al. [7]:

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

In the sequel, quantities such as "$|\mathrm{B}(\Omega)|$ evaluations of $\Omega$" 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. $\square$

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 $\mathrm{B}^{-}(P)$ to prune the same region:

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

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.

### 5.1 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 $\mathrm{Th}(P) \cup \mathrm{B}^{-}(P)$. For example, given the set of items $\{A, B, C, D, E\}$, if the transaction $ABCD$ is frequent but $CE$ is not, it is possible that MAFIA test the support of the transactions $A$, then $AB$ then $ABC$, then $ABCD$ and $ABCDE$. However $ABCDE$ is not in $\mathrm{Th}(P)$, nor is it in $\mathrm{B}^{-}(P)$ since $CE$ 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 that 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 [6].

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 [12]: 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 [13]. 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 levelwise algorithm similar to Apriori). The borders are then refined using $c_2$ and a levelwise 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 levelwise 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.

### 5.2 DualMiner

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 [6].

The output representation with subalgebras is more compact than with just itemsets. However, DualMiner does not output the most compact representation. 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 $\mathit{sum}(\mathit{price}) > 100$, 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 $A_1A_2\ldots A_nB$ and $A_1A_2\ldots A_nC$. Let $P$ be $\mathit{frequency}(\mathit{itemset}) \geq \mathit{threshold}(> 0)$ and let $Q$ be $\sum_{i \in \mathit{itemset}} \mathit{price}(i) > n$. The most compact representation of the solution in this case is $(B, A_1A_2\ldots A_nB)$ and $(C, A_1A_2\ldots A_nC)$, 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. 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.

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 $sum(price) > qthreshold$ (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 4 and 5 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 $\mathrm{Th}(Q)$ when it is looking for the bottom of a subalgebra.

Figures 6 and 7 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 8 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 $\mathrm{Th}(\neg Q)$, which is a relatively small set).

## 7. RELATED WORK

Agrawal et al. first introduced the problem of mining frequent itemsets [1, 3, 2]. This work was later generalized to include constraints other than support in [10, 9]. These papers introduced the concepts of monotone and antimonotone constraints and introduced methods for using them to prune the search space. The classes of monotone and antimonotone constraints were further generalized by Pei et al. [12] and also studied by Pei and Han [11]. This problem was

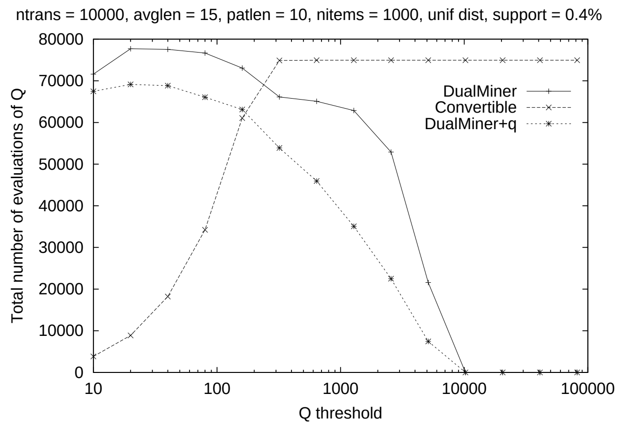

**Figure 4: Evaluations of $Q$ vs. Selectivity of $Q$**

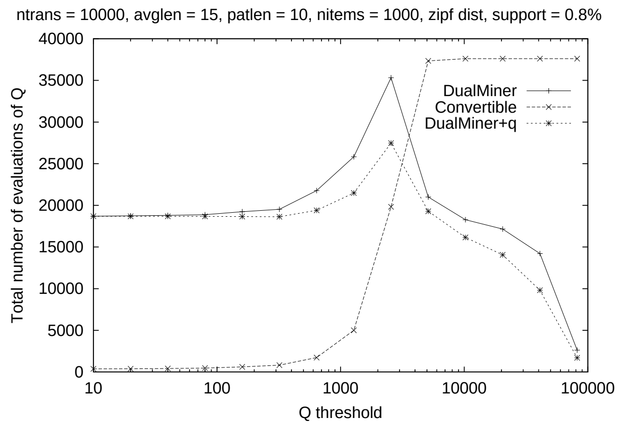

**Figure 5: Evaluations of $Q$ vs. Selectivity of $Q$**

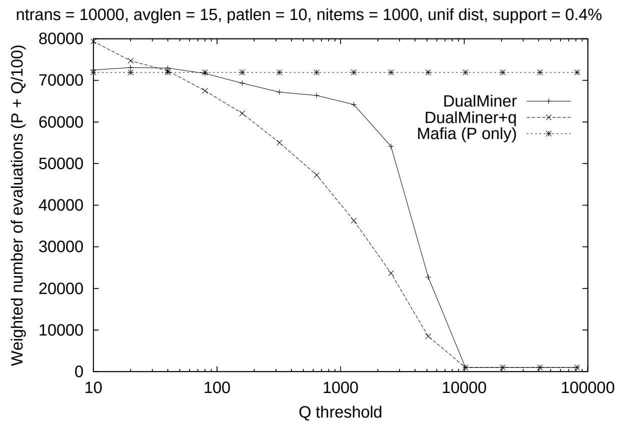

**Figure 6: Oracle Calls ($E_P + \frac{E_Q}{100}$) vs Selectivity of $Q$**

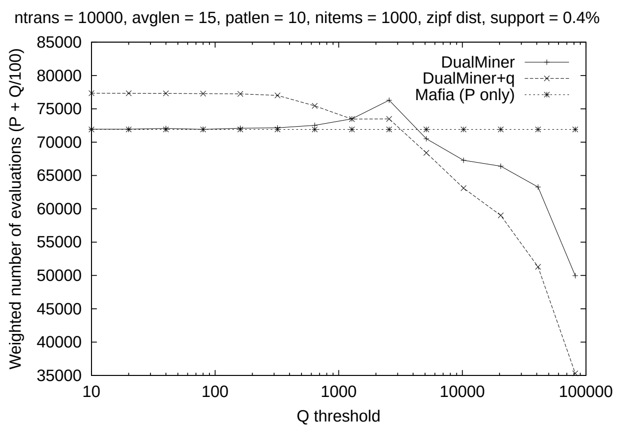

**Figure 7: Oracle Calls ($E_P + \frac{E_Q}{100}$) vs Selectivity of $Q$**

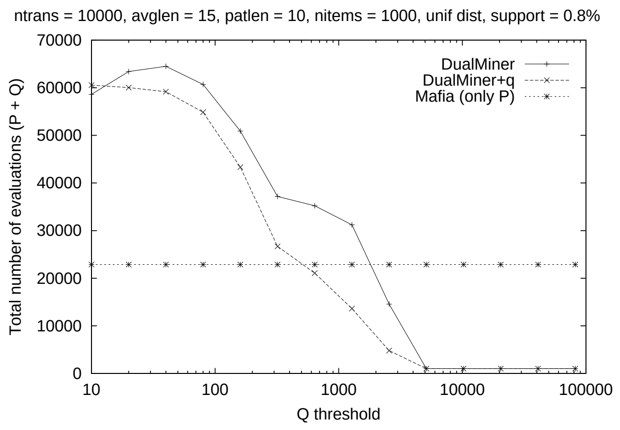

**Figure 8: Oracle Calls ($E_P + E_Q$) vs Selectivity of $Q$**

also given a theoretical treatment by Gunopulos, Khardon, Mannila and Toivonen [7]. However, none of previous work was able to prune using *both* monotone and antimonotone constraints simultaneously.

## 8. 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 [12], be incorporated as well? Can predicates such as $variance(X) \leq c$ (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.

## 9. 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 [8], 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] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *ICDE 2001*. IEEE Computer Society, 2001.

[7] D. Gunopulos, H. Mannila, R. Khardon, and H. Toivonen. Data mining, hypergraph transversals, and machine learning. In *Proc. PODS 1997*, pages 209–216, 1997.

[8] 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.

[9] R. T. Ng, L. V. S. Lakshmanan, J. Han, and T. Mah. Exploratory mining via constrained frequent set queries. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *SIGMOD 1999, Philadephia, Pennsylvania, USA*, pages 556–558. ACM Press, 1999.

[10] 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 [8], pages 13–24.

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

[12] 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.

[13] 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.