Fast Algorithms for Finding Extremal Sets

Roberto J. Bayardo
bayardo@alum.mit.edu
Google, Inc.

Biswanath Panda
bpanda@google.com
Google, Inc.

Abstract

Identifying the *extremal* (minimal and maximal) sets from a collection of sets is an important subproblem in the areas of data-mining and satisfiability checking. For example, extremal set finding algorithms are used in the context of mining maximal frequent itemsets, and for simplifying large propositional satisfiability instances derived from real world tasks such as circuit routing and verification. In this paper, we describe two new algorithms for the task and detail their performance on real and synthetic data. Each algorithm leverages an entirely different principle – one primarily exploits set cardinality constraints, the other lexicographic constraints. Despite the inherent difficulty of this problem (the best known worst-case bounds are nearly quadratic), we show that both these algorithms provide excellent performance in practice, and can identify all extremal sets from multi-gigabyte itemset data using only a single processor core. Both algorithms are concise and can be implemented in no more than a few hundred lines of code. Our reference C++ implementations are open source and available for download.[1]

1 Introduction

The problem of finding *extremal sets* in a collection of sets is to identify all sets in the collection that are maximal or minimal with respect to set containment. Prior to their application in data-mining, algorithms for maximal set finding were motivated by work in the area of propositional logic [9]. Today, maximal set finding algorithms are regularly applied in practice in the process of simplifying large satisfiability problem instances from real world domains in order to solve them more efficiently [4]. In the area of data-mining, maximal set finding algorithms are used to produce condensed representations of all frequent itemsets [7], which are useful patterns for generating association rules. Maximal frequent itemsets can also serve as features for classification and clustering tasks [12].

Despite the broad applicability of extremal set finding algorithms and several theoretical insights into the problem [9] [8] [14] [15], algorithms used in practice are relatively primitive, and suffer from scalability issues when the number of sets and/or the average number of elements per set grows. For example, the maximal set finding approach used in the SateLite satisfiability instance simplifier employs a hashing optimization that limits sets to only a few (64 or less) items, and uses an index structure that doubles the amount of memory required. Maximal frequent itemset mining algorithms do not identify maximal sets within the original data, but instead within the set of frequent itemsets derived from the data. These sets, by virtue of applying a minimum support constraint, are in general much smaller in size and alphabet than sets from the original data. Still, early work on maximal frequent itemset mining [2] found that the time required to eliminate non-maximal itemsets from the final result was a large fraction of the entire mining time. More recent maximal frequent itemset mining methods integrate maximality enforcement during mining for better performance [5], and in fact any maximal frequent itemset miner can find maximal itemsets when configured to use a minimum support of 1. But, as we later demonstrate, this approach is too inefficient.

The poor performance of finding extremal itemsets in massive datasets is not entirely surprising given that it is easy to prove that the above-noted algorithms used in practice are quadratic in the worst case. The best known worst-case bounds for the problem are in fact not much better: $O(m^2/\log(m))$ where m is the total number of items in the dataset) [8] [15]. Obtaining efficient algorithms on large datasets therefore requires we exploit not only the best available algorithmic concepts, but also heuristics and principled implementation techniques. Such techniques include index based optimizations to alleviate primary bottlenecks, and structuring the program flow and data layout in order to maximize both spatial and temporal locality.

In this paper we specifically address the problem of finding extremal sets in large datasets with a variety of input characteristics. We describe two new algorithms for the problem that effectively leverage both theoretical insights and practical implementation techniques. The first algorithm exploits set cardinality constraints and itemset indexing techniques similar to those recently proposed for the problem of similarity joins [3] [13]. The second algorithm exploits the lexicographic property first noted in the early theoretical work of Pritchard

[1] http://code.google.com/p/google-extremal-sets/

[9], and a space-efficient indexing strategy for optimizing the most time-intensive seeks into the input data. Both algorithms exploit the frequency distribution of items, use only straightforward data structures, and can be applied to datasets that are too large to fit into primary memory. Experiments on real and synthetic datasets demonstrate their scalability. For example, using only a single processor core on a consumer desktop machine, we show that both algorithms require only a few minutes to identify the maximal sets of friends in an 8 gigabyte, 20 million node graph representing friendship relationships within a production social network.

2 Preliminaries

2.1 Terminology An *itemset* is a finite set of *items*, which may represent arbitrary entities such as grocery products purchased together, the friends of a person in a social network, the literals in a clause from propositional logic, or the terms that appear in a document. For most real applications, item frequencies have non-uniform distributions, with power law distributions being the most common. Almost all recent practical algorithms involving itemsets (be they for finding extremal sets, similarity joins, frequent itemset mining, an so on) exploit this non-uniform distribution for efficiency purposes. Our algorithms are no different; thus we assume up front that the *alphabet* of items from a given collection of itemsets has been ordered in increasing order of item frequency within the collection. For example, in the case of market basket analysis, the items which are purchased the least will appear earliest in this ordering. From this point on, we will treat itemsets as *ordered* sets, with items ordered according to this frequency based policy.

A *dataset* is a finite ordered multiset of itemsets, which we use to represent the itemset collection provided as input to the problem. As is the case with the items themselves, the ordering imposed upon itemsets is also critical for obtaining efficient algorithms. But unlike the item ordering which is exploited entirely for efficiency reasons, the itemset ordering is also generally exploited to guarantee correctness. Another difference in this case is the precise itemset ordering strategy to use is not as obvious. An algorithm of Pritchard [9], for example, exploits both itemset cardinality and lexicographic properties of itemsets. Pritchard did not evaluate the algorithm empirically, and in our experience, exploiting both properties together did not yield significant benefits in practice. Thus, we instead propose two separate algorithms, one that exploits a purely itemset cardinality based ordering and the other a purely lexicographic one.

An itemset S_1 is *maximal* (resp. *minimal*) among a collection of itemsets if there is no other itemset S_2 in the collection such that $S_1 \subset S_2$ (resp. $S_2 \subset S_1$). An itemset is *extremal* within a collection if it is either minimal or maximal. Algorithms for finding all maximal sets in a collection are in general easily modified to find all minimal itemsets (and vice versa) [8], so without loss of generality, from this point onward we focus on finding maximal sets only.

2.2 Notation Recall that the dataset accepted as input by each algorithm is an ordered multiset of itemsets D. We use standard index and range notation to denote specific items, elements, and subsets of the input data. Specifically:

- $D[i]$ denotes the i^{th} itemset from D according to the itemset ordering policy.
- $D[i][j]$ denotes the j^{th} item in itemset $D[i]$ according to the item ordering policy.
- $D[i : j]$ denotes the ordered multiset of itemsets $\{D[k] \mid k = i \ldots j\}$, in that order. Similarly, $D[i][j : k]$ denotes the itemset $\{D[i][l] \mid l = j \ldots k\}$.

In our algorithm descriptions, whenever we specify an iteration over all elements of an ordered set, we assume elements are produced in either the order explicitly stated or otherwise in the order implied by the ordering policy of the particular ordered set type.

3 Finding Extremal Sets Using Cardinality Constraints

The first algorithm we propose, like the subsumption detection scheme of SateLite [4], exploits the *cardinality constraint*, which is the simple and obvious fact that an itemset cannot properly contain (*subsume*) any itemset that is equal to or greater than it in size. While this seems like a relatively weak constraint on the surface, we shall demonstrate it can still be exploited much more extensively than in previous approaches. In order to better leverage the cardinality constraint, the itemset ordering required by this first algorithm is that itemsets appear in increasing cardinality. In other words, smallest itemsets always come before larger ones. The algorithm makes no additional requirements on the itemset ordering, though we note that lexicographically ordering the itemsets within in each block of equally sized itemsets supports further optimization in theory [8].

Another key to the algorithm is in leveraging an indexing strategy to narrow the space of itemsets a particular itemset must be compared against. The SateLite approach uses an indexing strategy whereby

an index (called an *occurs* list) is associated with each item and contains pointers to all itemsets that contain the item. Thus, if there are n itemsets and m items in all of D ($m = \sum_{i=1\ldots n} |D[i]|$), the index consists of exactly m pointers. The index is constructed up front before subsumption checking begins. In contrast, our approach constructs and leverages the index during the maximal set finding phase, which means the index only reaches its full size after performing significant work. Furthermore, for each itemset S, we index it in the occurs list of only its very first item $S[1]$. The total index size therefore consists of at most n pointers, which is much smaller than m, particularly when itemsets are large. Small indexes generally allow for better performance since there is less information to scan and better spatial locality.

Pseudocode for this algorithm, which we refer to as *AMS-Card*, appears in Algorithm 1. There are several important structures maintained by the algorithm:

- O is the *occurs* index, and consists of one ordered multiset denoted $O[i]$ for each item i. It is initially empty, but each itemset processed by the outer loop is eventually added to the end of $O[S[1]]$. In other words, we index each itemset by its least frequently occurring item shortly after it is first encountered. By virtue of the fact that itemsets are added to the index in the same order as they are encountered, we have the invariant that itemsets in each $O[i]$ appear in smallest to largest order.
- B is the current *block* of same-length itemsets that have been recently encountered and remain to be indexed. An invariant for B is that all itemsets within it are of the same cardinality.
- The value c is the cardinality of the itemsets currently being added to B.

Algorithm 1 GetMaximalItemsetsCard(Dataset D): Find all maximal sets from a dataset D by using the cardinality constraint.

Require: D ordered by increasing itemset cardinality
1: $B \leftarrow \emptyset$; $c \leftarrow 0$; $O \leftarrow \emptyset$
2: **for all** $i \in \{1, \ldots, n\}$ **do**
3: **for all** $j \in \{1, \ldots, |D[i]|\}$ **do**
4: **for all** $S \in O[D[i][j]]$ not marked as subsumed **do**
5: **if** $|S| > |D[i]| - j + 1$ **then**
6: break
7: **if** $D[i]$ properly subsumes S **then**
8: mark S as subsumed
9: **if** $|D[i]| > c$ **then**
10: Add each itemset $S \in B$ to the end of $O[S[1]]$
11: $B \leftarrow \emptyset$; $c \leftarrow |D[i]|$
12: $B \leftarrow B \cup \{D[i]\}$
13: **return** all elements of D not marked as subsumed

3.1 Description and Correctness Argument

The high level operation of the algorithm is straightforward, as are the arguments supporting its correctness. We scan the dataset in itemset order. For each itemset, we iterate over each its items. For each item, we scan over the list of itemsets within the occurs list for that item. For each such itemset, we do an explicit subsumption check between it and the current itemset. After processing all itemsets of a given size c, they are added to the occurs index as already described and the procedure continues. It is straightforward to see that if an itemset $D[i]$ properly subsumes an itemset S, then S will have been indexed by the time the loop on line 2 encounters $D[i]$. Furthermore, because by definition of subsumption S must contain every item in S', we are guaranteed to encounter S' when scanning at least one of the occurs lists while S is $D[i]$. It follows that the algorithm successfully marks each subsumed itemset, and only the maximal itemsets are returned.

Note there are a few more tricks employed by our version of this basic scheme. First, on line 5, note that we have an "early exit" condition for the inner loop that triggers when the set of items from the current itemset that remain to be tried ($D[i][j+1 : |D[i]|]$) is smaller than the current itemset S. This early exit is possible is because within the inner loop, any itemset that has not already been encountered does not contain any of the items in $D[i][1 : j-1]$. Therefore, for any itemset S not already found to be subsumed, if it is subsumed by $D[i]$, it must be entirely subsumed by the subset $D[i][j : |D[i]|]$. This inequality check simply enforces the tighter cardinality constraint that results from this fact, and the fact that all itemsets following S in the occurs list are at least as large as S. This fact can also be exploited by the explicit subsumption check performed on line 7. In our implementation, we explicitly check whether itemset S is subsumed by $D[i]$ by initiating a binary search in D[i] at item $D[i][j+1]$, since we know that $S[1] = D[i][j]$. We perform one binary search for each remaining item of S, returning as soon as we fail to find an item. Each binary search starts from the point at which previous search left off. As the difference in size between $D[i]$ and S grows, this binary searching is asymptotically faster than a linear scan based intersection.

3.2 Complexity and Performance

Algorithm 2 GetMaximalItemsetsLex(Dataset D): Find all maximal itemsets in D by using the lexicographic constraint.

Require: Itemsets in D are in lexicographic order
1: *;; Mark proper prefixes as subsumed*
2: $S \leftarrow D[n]$
3: **for all** $i \in \{n-1, \ldots, 1\}$ **do**
4: **if** $|D[i]| < |S|$ **and** $S[1 : |D[i]|] = D[i]$ **then**
5: *;; $D[i]$ is a proper prefix of S*
6: mark $D[i]$ as subsumed
7: **else**
8: $S \leftarrow D[i]$
9: *;; Mark remaining non-maximal itemsets as subsumed*
10: **for all** $i \in \{1, \ldots, n-1\}$ **do**
11: **if** $D[i]$ is not marked subsumed **then**
12: MarkSubsumed($D[i+1 : n]$, $D[i]$, 1, 0)
13: **return** all elements of D not marked as subsumed

3.2.1 Runtime It is straightforward to come up with problem instances where AMS-Card is quadratic in the number of itemsets, yet as we will show in the experimental results section, it still performs well on remarkably large datasets. Note that the algorithm uses tight loops, small index structures, and a single scan over the input data, all providing good locality properties. Most importantly, however, by positioning infrequent items up front in each itemset, the algorithm takes little time to quickly try the initial items from a given set, which lets it better exploit the cardinality constraint at line 5. Not entirely surprisingly, performance of the algorithm tends towards quadratic as the item frequency distribution becomes more uniform, as confirmed through our experiments on synthetic data.

3.2.2 Memory Usage The memory requirement of the algorithm is minimal: it is linear in the size of the input data with a small constant. We use only standard array data structures for representing the index and the itemsets. Each itemset is stored in memory exactly once, and a single pointer per itemset gets stored on the index. Because of the simple single-scan nature of the algorithm, it is straightforward to modify the algorithm to operate on datasets that are larger than available RAM by partitioning the dataset and performing one file-based dataset scan per partition [3].

4 Finding Extremal Sets Using Lexicographic Constraints

When itemsets are ordered *lexicographically*, an itemset S_1 precedes S_2 if and only if (1) S_1 is a proper prefix of S_2 ($\{1,2,3\}$ vs. $\{1,2,3,4\}$) or (2) for the smallest j for which $S_1[j] \neq S_2[j]$, item $S_1[j]$ precedes $S_2[j]$ in the item ordering ($\{1,2,4\}$ vs. $\{1,3,4\}$ where $j = 2$). We denote the fact that an item i_1 precedes another item i_2 according to the item ordering in the standard way: $i_1 < i_2$. Pritchard [9] proved that for an itemset S_1 to properly subsume another itemset S_2, then either S_2 is a proper prefix of S_1, or S_2 must lexicographically follow S_1. This section describes how we leverage this *lexicographic subsumption property* for finding maximal itemsets.

Subroutine 3 MarkSubsumed($D[b : e]$, S, j, d): Mark proper subsets of S within $D[b : e]$ as subsumed. The parameter j specifies we need only consider items $S[j, |S|]$, and d indicates the size of the prefix shared by all $I \in D[b : e]$ and subsumed by $S[1, j-1]$.

Require: $D[b : e]$ is ordered lexicographically
Require: $\forall S' \in D[b : e]$, $S'[1 : d] = D[b][1 : d]$
Require: $S[1, j-1] \supseteq D[b][1 : d]$
Require: $\forall S' \in D[b : e]$, $|S'| > d$
Require: $\forall S' \in D[b : e]$, $S'[k] \geq S[j]$ for all k s.t. $d < k \leq |S'|$.
1: **while** $b \leq e$ **do**
2: **if** $S[j] < D[b][d+1]$ **then**
3: $j \leftarrow$ NextItem(S, j, $D[b][d+1]$)
4: **if** j is null **then**
5: **return**
6: **if** $S[j] = D[b][d+1]$ **then**
7: $e' \leftarrow$ NextEndRange($D[b : e]$, $S[j]$, d)
8: **if** $|S| > d+1$ **then**
9: **while** $b \leq e'$ **and** $|D[b]| = d+1$ **do**
10: mark $D[b]$ as subsumed
11: $b \leftarrow b+1$
12: **if** $j+1 < |S|$ **and** $b \leq e'$ **then**
13: MarkSubsumed($D[b : e']$, S, $j+1$, $d+1$)
14: $b \leftarrow e'$
15: **else**
16: $b \leftarrow$ NextBeginRange($D[b : e]$, $S[j]$, d)

4.1 Description and Correctness Argument

The high level operation of our second approach for finding maximal itemsets, which we refer to as *AMS-Lex*, is formally described in Algorithm 2 and Subroutine 3. The top level function first performs a single scan of the data in reverse lexicographic order to remove all cases of subsumption due to prefix containment. Next, it performs a forward scan, and for each itemset it encounters, it invokes the MarkSubsumed subroutine to mark all proper subsets that follow it in the ordering as subsumed. If MarkSubsumed satisfies its specification, then the correctness of the top level procedure follows directly from the lexicographic subsumption property.

The MarkSubsumed subroutine is invoked recursively, and its input arguments must satisfy several preconditions. The first and most straightforward one is that the range of the input dataset it considers is itself in lexicographic order. More interestingly, it requires that all itemsets in the given dataset range share the same length-d prefix, and that all items in this prefix belong to the itemset $S[1 : j-1]$. Finally, it requires that all itemsets in the given range have at least one more item than those in the shared length d prefix, and that all of those extra items do not come before item $S[j]$ in the item ordering. Note that all of these preconditions trivially hold for the top level call made by AMS-Lex, since $S[1, 0]$ is empty, the length-0 prefix is empty, and the range consists of all itemsets following S in the lexicographic ordering.

Correctness of the MarkSubsumed procedure will be argued inductively. At any given point during the algorithm, note that with any narrowing of the input range, as long as the range is non-empty, the preconditions remain invariant. The procedure works by iteratively narrowing the front of the range until it is empty. Correctness follows as long as we show that the range is narrowed only to exclude portions of the range in which all properly subsumed itemsets are already known to have been identified.

Recall that according to the preconditions, $S[j] \leq D[b][d+1]$. By virtue of the lexicographic ordering, no subset containing $S[j]$ can subsume itemsets in $D[b : e]$ if $S[j]$ is strictly less than $D[b][d+1]$. The NextItem procedure therefore searches within S from item j onwards until it finds the first item $S[j']$ such that $j' \geq j$ and $S[j'] \geq S[j]$. The value of j is the updated to reflect this value. If there is no such item in S, then clearly S cannot subsume any itemsets in the range, and the procedure returns immediately (base case).

Consider now the case where NextItem returns a value j such that $S[j] = D[b][d+1]$ (line 6). In this case, it follows from our preconditions that any itemsets subsumed by $S[1, j]$ must reside at the very beginning of the range. We can therefore find these itemsets by simply advancing the beginning of the range b until they have all been identified and marked. Consider next the NextEndRange function. This function returns the largest value e' for which $D[e'][d+1] = S[j]$. If the preconditions were satisfied on input, it is straightforward to show that the preconditions hold for parameters $D[b : e'], S, j+1$, and $d+1$. We can therefore recursively invoke the procedure to mark all proper subsets of S in range $[b : e']$ (inductive step), and narrow the current range in which to continue searching to $[e' : e]$.

The only remaining case to consider is when $S[j] > D[b][d+1]$ after invoking NextItem (line 16). In this case, we have that item $D[b][d+1]$ is not contained in S, and hence $D[b]$ cannot be subsumed by S. Furthermore any itemset following D[b] that shares its $d+1$-length prefix also cannot be subsumed by S. The NextBeginRange function identifies the earliest point b' at which $D[b'][d+1]$ differs from $D[b][d+1]$, and the current range is safely narrowed to this new begin point.

4.2 Complexity and Performance

4.2.1 Runtime Performance of the algorithm in practice once again depends critically on the item ordering, and also the implementations of the low level searching routines NextItem, NextEndRange, and NextBeginRange. Note that each time these methods are invoked, we restrict the search to a particular range of the input data and always try to narrow the searchable range as the algorithm progresses. Each comparison can be implemented by comparing only a single item from each itemset with the item provided as an argument, so comparison operations are extremely fast. Because the infrequent items appear early in the itemsets, the ranges typically become narrow very early on in the search. The standard implementation technique for searches along ordered data is binary search since it is logarithmic in the number of elements in the worst case. Unfortunately even binary search can be relatively slow in practice when the range to search is large due to poor spatial locality. In our implementation of NextEndRange and NextBeginRange, we therefore leverage an index structure for the case where $d = 0$. When $d = 0$, the initial range consists of the entire dataset following itemset S, which can be huge. The index structure we use simply maps each item i to the first itemset in D that starts with item i. When $d = 0$, NextBeginRange can therefore find the appropriate starting point b without any searching, as can NextEndRange assuming for a given item i we know the item j that follows it in the ordering. We therefore resort to binary search only when $d = 1$ or greater. And again, these ranges are typically dramatically smaller than the initial range due to the frequency based item ordering, which results in fast binary search.

Because MarkSubsumed monotonically decreases the size of the range it is searching in, it is easy to see that it is linear in the number of input itemsets. Combined with the fact that it is invoked for each itemset in D, we easily derive a quadratic worst case asymptotic runtime bound in the number of itemsets. If we quantify runtime in terms of the total number of items m in D instead of the number of itemsets, then it is possible to show that runtime is $O(m^2/log(m))$ using

a straightforward adaption of the Pritchard's analysis [8].

4.2.2 Memory Usage AMS-Lex has memory usage characteristics similar to AMS-Card. The prefix removal step requires only the current itemset and the potentially prefix-subsuming itemset remain in main memory. The second scan requires the entire dataset range being considered by RemoveSubsumes to be memory resident. However, it is once again possible to break the input dataset into chunks and perform multiple scans should the dataset exceed available memory. The index structure we use for range narrowing in AMS-Lex is extremely compact, and requires space that scales only with the size of the item alphabet.

5 Experiments

5.1 Algorithms We implemented both AMS-Card and AMS-Lex in C++ for empirical evaluation. The implementations we tested are the very same ones available in our open source release of the code. These implementations are in C++ and consist of only of a few hundred lines of code each. Both implementations support specifying a limited memory buffer size in which to operate in order to avoid thrashing. Should the dataset size exceed the available buffer space, the algorithms appropriately partition the input data, and thus can efficiently handle even datasets that are larger than available memory.

We also implemented an algorithm we call *AMS-SAT* that identifies all maximal itemsets using the strategy employed by SateLite [4]. Recall that the SateLite approach indexes every item of every itemset in the input. This algorithm is also provided in our open source release. Unlike AMS-Card and AMS-Lex, this algorithm does not require any specific sort order of the input. However, as we later demonstrate, the itemset ordering can have a big impact on runtime performance.

All runtimes we report are wall-clock times, and we do not include runtime required to prepare the data (e.g. sorting into the required itemset order) since this overhead was not a large part of the overall runtime. The machine we used had a recent Intel Xeon 2.5 GHz multi-core processor, though our implementations take advantage of only a single core.

5.2 Datasets

5.2.1 Real Data The datasets we used that were derived from real data are described below.

- DBLP: this dataset is derived from publicly available data, and was prepared as described in previous work on similarity joins [1] [13]. It consists of almost 1 million records, and has 14 items per itemset on average. It is 50 megabytes in size.
- WebDocs: this dataset is publicly available from the FIMI itemset mining benchmark repository, and consists of 1.7 million itemsets representing the content of pages encountered during a web crawl.[2] Total file size is 1.2 gigabytes. The average number of items per itemset is 265, with the largest itemset containing 71,472 items.
- PubMed: this dataset is publicly available from the Irvine Machine Learning Repository, and consists of 8 million itemsets each representing the significant terms that appear in a single PubMed abstract. The number of items per itemset is 90 on average, and the total file size is 2 gigabytes.
- Graph: this dataset is proprietary, and represents relationships within a 20 million person social network. Each itemset represents a node and contains the other nodes it connects to. On average there are 100 items per itemset, though the length distribution was very skewed, and many have size up to 1000. The total file size is 7.4 gigabytes. To prevent thrashing we limited the buffer size to 4 gigabytes, which resulted in each of our algorithms having to perform one extra pass over the input data after partitioning the data into two pieces.

5.2.2 Synthetic Data We found that each of the datasets derived from real data had skewed, power-law like item frequency distributions. In order to test how the algorithms might perform given a uniform item frequency distribution, we also implemented a synthetic dataset generator. The synthetic dataset generator also allows us to see how the algorithms perform with more control over input data characteristics, such as the ratio of maximal to non-maximal sets, and with much larger itemsets than we happened to find in our real data.

The synthetic data generator requires the following parameters:

- A value $amax$ defining an alphabet $\{0 \ldots amax\}$ from which items are drawn uniformly when generating an itemset.
- A value $maxsize$ that defines a range $\{1 \ldots maxsize\}$ from which the size of an itemset is drawn uniformly before selecting its items.
- A value n saying how many *potential* maximal itemsets to generate.

[2] http://fimi.cs.helsinki.fi/data/webdocs.pdf

- A value s saying how many subsets per *potential* maximal itemset to generate.

We generate n *potential* maximal itemsets using the distributions defined by $amax$ and $maxsize$. For each itemset S of these n itemsets, we generate s random subsets. A random subset is generated by first selecting a size uniformly from the range $[1, |S| - 1]$, and then drawing that many items uniformly randomly from S. We use random permutations to ensure true uniform selection of all items. The n itemsets generated using the distributions defined by $amax$ and $maxsize$ need not necessarily be maximal, and hence we call them *potential* maximal itemsets. The subsets of a large *potential* maximal itemset can subsume a smaller *potential* maximal itemset.

For these experiments, except when explicitly noted, we always run AMS-SAT on the lexicographically sorted input data, which always resulted in better performance over cardinality sorted input – typically by at least a factor of 2.

5.3 Results on Real Data The results on the four datasets derived from real data appear in the Table 1. While all algorithms trivially handled the DBLP dataset, differences in performance were more dramatic on the larger datasets. In general AMS-Lex comfortably outperformed all others, except on the Graph dataset where AMS-Card was the fastest by a slight margin. We believe AMS-Card was superior on this dataset because it exhibited the most variance in itemset cardinality.

On our machine which had 8GB of data, AMS-SAT could not handle the graph dataset without thrashing, so we report runtimes only for the other datasets. Because the algorithm performs roughly the same number of subsumption checks regardless of the itemset ordering, we originally assumed the itemset ordering would not significantly impact performance. It turned out however that the itemset ordering did have a big effect, with the lexicographic sort order providing significantly better performance. The reason for this improvement is improved spatial locality. Recall that for each itemset, AMS-SAT scans the index list associated with its least frequent item in order to identify candidates. Itemsets which share their least frequent items appear together in the lexicographic ordering. Regardless, AMS-Lex always substantially outperformed it. However, AMS-SAT on the lexicographically ordered data outperformed AMS-Card on the PubMed data. The PubMed data had low variance in the number of items per itemset, hence the better locality allowed by the lexicographic ordering was more beneficial than the more powerful cardinality filtering allowed by AMS-Card.

5.4 Results on Synthetic Data

5.4.1 Experiment 1 For our first experiment on synthetic data, we wanted to measure performance as the number of total itemsets increases while maximum itemset size and the number of maximal itemsets is held. The parameters provided to the synthetic data generator were as follows:

- $amax = 1e6$
- $maxsize = 1000$
- $n = 10000$
- s is varied so that the resulting dataset contains 500k to 2.0 million itemsets.

Results for this experiment appear in Figure 1. Recall that we use a uniform item frequency distribution when generating the synthetic data. Even though the size of itemsets varies in this synthetic data more than the first three datasets derived from real data, the performance of AMS-Card still suffers because of the uniform item distribution. In fact, AMS-Card appears to be exhibiting quadratic runtime scaling. AMS-SAT also exhibits quadratic scaling similar to AMS-Card, but is always slower. While AMS-SAT has the benefit of better spatial locality due to the lexicographic itemset order, it must perform many more subsumption checks than AMS-Card, which exploits the cardinality constraint more aggressively. Figure 1 does not report the running time for AMS-SAT on the 2 million point dataset because its index exhausted main memory, leading to page faults and a very large running time. While AMS-Lex also exploits skewed item frequencies, because it recursively narrows the range that must be searched, the impact of the uniform distribution is far less dramatic.

5.4.2 Experiment 2 In our second synthetic data experiment, we scaled the number of *potential* maximal itemsets, while keeping other parameters constant. Specifically, the parameter settings we used for the synthetic data generator were as follows:

- $amax = 1e6$
- $maxsize = 1000$
- n was varied
- s is set to keep the dataset size at 1.0 million itemsets

Results for this experiment appear in Figure 2. Here the results are somewhat surprising. While the performance of AMS-Lex remains mostly constant as the

Table 1: Runtime performance in seconds of finding all maximal sets on real datasets. Note that we provide runtimes for AMS-SAT on both lexicographic and cardinality sorted inputs.

Dataset	AMS-Card	AMS-Lex	AMS-SAT(L)	AMS-SAT(C)
DBLP	2	1	3	4
PubMed	487	163	384	1217
WebDocs	81	54	115	316
Graph	217	240	abort	abort

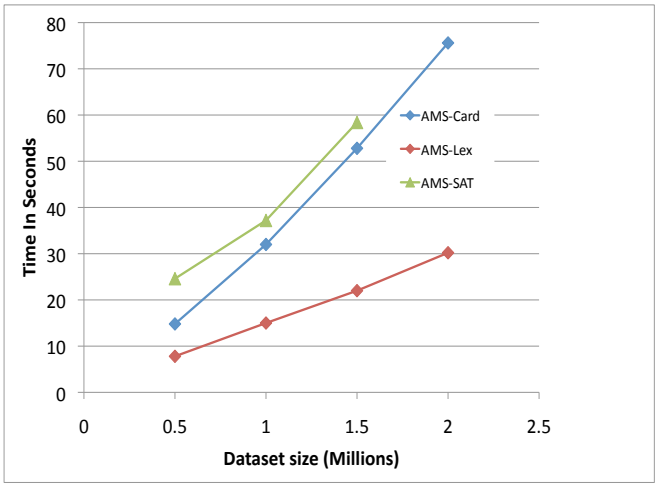

Figure 1: Runtime on synthetic data when the size of the dataset is varied while holding all other data generation parameters constant.

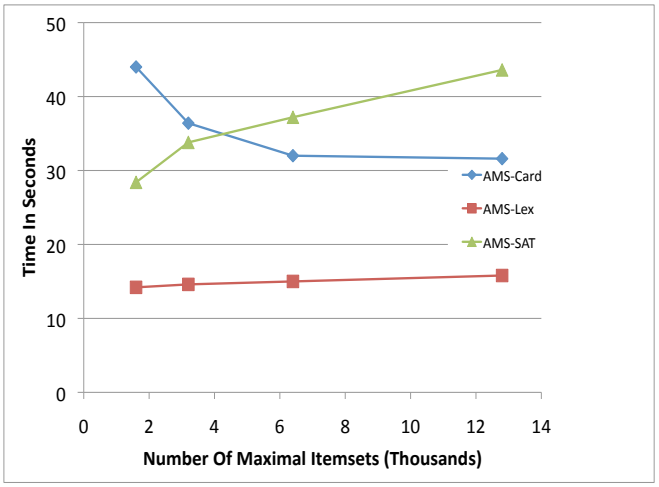

Figure 2: Runtime on synthetic data when the number of maximal itemsets varied.

number of maximal itemsets is increased, AMS-Card becomes progressively faster, and AMS-SAT becomes slower. At this point we can only speculate that the reason may have something to do with the order in which the algorithms scan the itemsets. It is natural to expect that as the number of maximal itemsets increases, running time should increase since lesser number of sets are marked as subsumed, and hence continue to be checked for subsumption. AMS-Card however scans the dataset from smallest to largest itemset cardinality. When there are few maximal itemsets, they are more likely to be the largest sets in the collection and appear last in the dataset. AMS-Card would therefore not encounter them until very late in its execution, and as a result, it would spend significant effort trying to determine subsumption relationships between non-maximal itemsets. To confirm this suspicion we ran AMS-SAT on the cardinality sorted input, and indeed its performance scaled more similarly to that of AMS-Card.

5.4.3 Experiment 3 For our third and final experiment on synthetic data, we explored the effect of varying itemset size while keeping the number of items and number of maximal itemsets constant. Data generation parameters were as follows:

- $amax = 1e6$
- $maxsize$ was varied from 250 to 2000
- $n = 10000$
- s is set to keep the dataset size at 1.0 million itemsets

Results appear in Figure 3. Again AMS-Lex outperforms AMS-Card, though in both cases the algorithms appear to scale linearly with increasing itemset size. As expected, the differences between AMS-SAT and AMS-Card are less significant when the average size of an itemset is smaller. Figure 3 does not include the running time for AMS-SAT when the max itemset size is 2000, since at this point its performance degraded dramatically due to page faults.

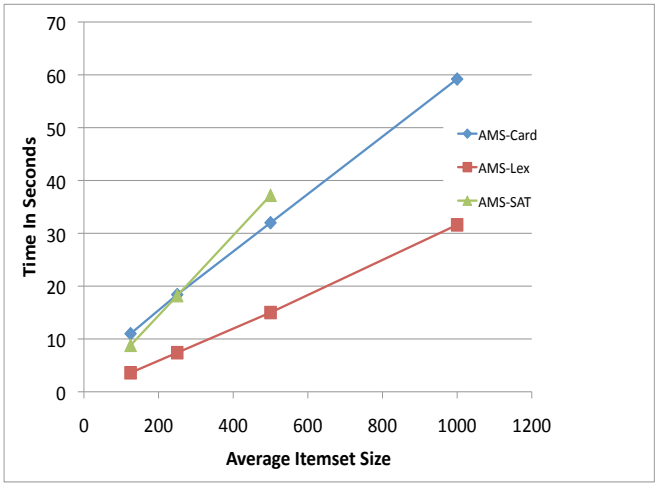

Figure 3: Runtime on synthetic data when the maximum itemset size is varied.

5.5 Comparison with Maximal Frequent Itemset Mining Any algorithm for mining maximal *frequent* itemsets can also produce all maximal itemsets if its minimum support constraint is set to 1. This approach, however, is unlikely to be practical except on small datasets since most such algorithms scale poorly at low values of support. We confirmed this fact by running GenMax[5], a state of the art maximal frequent itemset miner, on the real datasets used in our evaluation. GenMax exhausted available memory and aborted before completion on all datasets except for DBLP. On DBLP, it required 67 seconds of runtime, compared to only 1 second for AMS-Lex.

6 Related Work

As already noted, the works of Pritchard [8] [9] and Yellin [14] [15] were first to demonstrate the first subquadratic bound for the extremal-sets finding problem. These works present only algorithm descriptions primarily aimed at demonstrating asymptotic time and space bounds, and do not evaluate their approaches empirically. Our work borrows some concepts from this work such as the lexicographic ordering property, but leverages them in algorithms intended to be simple to implement and fast in practice. We combined these concepts with appropriate indexing techniques to relieve the primary bottlenecks, and heuristics to exploit skewed frequency distributions. Our implementations were carefully structured so that operations required at the innermost loops exhibited good locality and low overhead. Some of these techniques were inspired by recent work in similarity joins [3] [13] but were specifically tailored to the extremal sets problem in this work.

The only publicly available implementation of an approach to finding extremal sets we are aware of is the naive approach used by SateLite [4]. This approach uses aggressive indexing of all items, resulting in far more memory usage than our algorithms, and runtime that is impractical on datasets as large as the ones from our evaluation. SateLite does employ one optimization we did not consider that can speed up a pairwise subsumption test when the two sets are both small. The idea is to store a 64-bit bloom filter per itemset with bits set according to the modulo-64 hashes of the itemset's items. The bloom filters of two itemsets can be quickly XORd to determine subsumption with no false negatives but potential false positives. Should this fast test return true, a full pairwise subsumption check can then be performed. Unfortunately, this approach is not useful for datasets with more than 50 or so items per itemset on average, since the probability of false positives approaches 100%. For bloom filters to provide any runtime improvements, the false positive rate must be low enough such that the benefit of skipped subsumption checks exceeds the extra resources required to store and process them. While a small number of items per itemset on average is common in CNF satisfiability instances, it is not common in data from data-mining domains such as market basket analysis, document analysis, and graph analysis.

Algorithms for finding maximal frequent itemsets [2] [5] [6] must enforce maximality constraints in a manner similar to extremal set finding algorithms. However, frequent itemsets must satisfy a minimum support constraint which typically implies the itemsets necessarily have a much smaller alphabet and size than those from the original data. As a result, these approaches are not suitable for extremal set finding in general.

Other work in extremal set finding includes Shen's [11] which proposes data structures and algorithms for efficiently maintaining extremal sets under the presence of dataset updates such as itemset insertions.

7 Conclusion and Future Work

We have presented two new algorithms for efficiently identifying all extremal sets within a given dataset. Our algorithms are fast, simple to implement, and exhibit excellent performance on even very large (multi-gigabyte) datasets. These algorithms leverage index structures to reduce algorithm bottlenecks, and item frequency distributions to heuristically minimize the search space and improve locality.

Our algorithms employ relatively simple "single item" indexing strategies in order to improve performance. We also suspect more sophisticated index structures that consider larger itemset prefixes

could yield additional performance improvements, particularly when the item frequency distribution is less skewed. Indeed, Xiao et al. [13] have demonstrated that more aggressive prefix indexing can be effective for similarity joins.

While we demonstrated that both algorithms perform well, the second algorithm, which leverages the lexicographic constraint, appears to perform best in most circumstances. Only on the largest dataset we looked at, Graph, which also happened to have the most skewed itemset size distribution, was the cardinality based approach superior. Because the two algorithms leverage entirely different properties, a third algorithm that leverages both might achieve superior performance to either one. Unfortunately our initial attempts at combining the approaches have yet to provide significant performance improvements beyond what we have reported here. Nevertheless, we suspect opportunities remain in combining the techniques.

Lastly, we have primarily focused in this work on applying these algorithms to typical data-mining oriented datasets. It is likely however that these techniques would improve the performance of propositional satisfiability simplification methods as well.

Acknowledgment

We thank Mohammed Zaki for his feedback on this work and his assistance with GenMax.

References

[1] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set similarity joins. In *Proc. of the 32nd Intl Conf. on Very Large Data Bases,* 918–929, 2006.

[2] R. J. Bayardo. Efficiently mining long patterns from databases. In *Proc. of the 1998 ACM-SIGMOD Int'l Conf. on Management of Data,* 85–93, 1998.

[3] R. J. Bayardo, Y. Ma, R. Srikant. Scaling up all-pairs similarity search. In *Proc. of the 16th Int'l Conf. on World Wide Web*, 131–140, 2007.

[4] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proc. of the Eighth Int'l Conf. on Theory and Applications of Satisfiability Testing,* 2005.

[5] K. Gouda, M. J. Zaki: GenMax: An Efficient Algorithm for Mining Maximal Frequent Itemsets. *Data Mining and Knowledge Discovery,* 11(3), 223–242, 2005.

[6] G. Grahne and J. Zhu. High performance mining of maximal frequent itemsets. In *The 6th SIAM International Workshop on High Performance Data Mining,* 35–143, 2003.

[7] T. Mielikäinen. Transaction databases, frequent itemsets, and their condensed representations. In *Fourth International Workshop on Knowledge Discovery in Inductive Databases, Lecture Notes in Computer Science,* 3933, 139–164, 2006.

[8] P. Pritchard. An old sub-quadratic algorithm for finding extremal sets. *Information Processing Letters,* 62(6), 329–334, 1997.

[9] P. Pritchard. Opportunistic algorithms for eliminating supersets. *Acta Informatica,* 29, 733–754, 1991.

[10] P. Pritchard. On computing the subset graph of a collection of sets. *Journal of Algorithms,* 33(2), 182–203, 1999.

[11] H. Shen. Fully dynamic algorithms for maintaining extremal sets in a family of sets. *International Journal of Computer Mathematics,* 69(3-4), 203–215, 1998.

[12] Z. Wang, H. Fan, and K. Ramamohanarao. Exploiting maximal emerging patterns for classification. in *Proc. of the 17th Australian Joint Conference on Artificial Intelligence, Lecture Notes in Computer Science,* 3339, 1062–1068, 2004.

[13] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *Proc. of the 17th Int'l World Wide Web Conference,* 131–140, 2008.

[14] D. M. Yellin. Algorithms for subset testing and finding maximal sets. In *Proc. of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms,* 386–392, 1992.

[15] D. M. Yellin and C. S. Jutla. Finding extremal sets in less than quadratic time. In *Information Processing Letters,* 48(1), 29–34, 1993.