HASHMAX: A NEW METHOD FOR MINING MAXIMAL FREQUENT ITEMSETS

Natalia Vanetik and Ehud Gudes
Deutsche Telecom Laboratories and CS department, Ben Gurion University, Beer-Sheva, Israel

Keywords: Maximal frequent itemset mining.

Abstract: Mining maximal frequent itemsets is a fundamental problem in many data mining applications, especially in the case of dense data when the search space is exponential. We propose a top-down algorithm that employs hashing techniques, named HashMax, in order to generate maximal frequent itemsets efficiently. An empirical evaluation of our algorithm in comparison with the state-of-the-art maximal frequent itemset generation algorithm Genmax shows the advantage of HashMax in the case of dense datasets with a large amount of maximal frequent itemsets.

1 INTRODUCTION

The task of finding frequent itemsets in transactional databases is an essential problem and a first step in many data mining applications, such as finding association rules, relational data modeling etc. Because of the time and space complexity required to find all frequent itemsets in a database, sub-problems of finding only closed frequent itemsets (the itemsets that are not contained in a superset with the same support) or maximal frequent itemsets (the itemsets that are not a subset of other frequent itemsets) have been defined and studied. The set of maximal frequent itemsets is orders of magnitude smaller than the set of closed frequent itemsets, which itself is orders of magnitude smaller than the set of frequent itemsets. When the frequent patterns are long and their number is significant, sets of frequent itemsets and even closed frequent itemsets become very large and most traditional methods count too many itemsets to be feasible. For the case of dense datasets, both traditional frequent itemset search and closed itemset search become inefficient due to the very large number of patterns found. Therefore, several algorithms for maximal frequent itemset mining have been suggested. The Pincer Search algorithm (Lin and Kedem, 1998) uses both the top-down and bottom-up approaches to frequent itemset mining. The MaxEclat and MaxClique algorithms, proposed in (Zaki et al., 1997), identify maximal frequent itemsets by attempting to look ahead and identify long frequent itemsets in order to prune the search space efficiently. The MaxMiner algorithm for mining maximal frequent itemsets, presented in (Bayardo, 1998), is based on a breadth-first traversal approach. The DepthProject algorithm, presented in (Agarwal et al., 2000), finds long itemsets using a depth-first search of a lexicographic tree of itemsets, and uses a counting method based on transaction projections along its branches. The Mafia algorithm (Burdick et al., 2001) uses elaborate pruning strategies to get rid of non-maximal sets, namely subset inclusion, TID set inclusion and look-ahead pruning. The Fpgrowth algorithm of (Han et al., 2000) and (Hu et al., 2008) uses a pattern growth approach to frequent itemset and maximal frequent set generation, completely eliminating the need for candidate generation. The Genmax algorithm proposed in (Gouda and Zaki, 2005) finds all maximal frequent itemsets by efficiently enumerating the itemsets with the help of a backtracking search. Genmax is currently considered to be the state-of-the-art algorithm and has been shown to outperform the Mafia algorithm (Burdick et al., 2001) and the MaxMiner algorithm (Bayardo, 1998), which in turn outperforms the Pincer Search algorithm (Lin and Kedem, 1998).

In this paper, we propose a new HashMax algorithm that uses hashing and a top-down approach for maximal frequent itemset generation. The algorithm starts with a set of candidates, where each candidate corresponds to a (pruned) set of items in each transaction, and continues downwards until a maximal frequent itemset is discovered. Combined with efficient pruning and subset generation, this approach allows us to efficiently find maximal frequent item-

Vanetik N. and Gudes E..
HASHMAX: A NEW METHOD FOR MINING MAXIMAL FREQUENT ITEMSETS.
DOI: 10.5220/0003628101320137
In *Proceedings of the International Conference on Knowledge Discovery and Information Retrieval* (KDIR-2011), pages 132-137
ISBN: 978-989-8425-79-9

sets in dense datasets, beginning with the largest ones. The HashMax algorithm is designed specially for the dense dataset and low support case; however, on very sparse dataset our algorithm is also able finish quickly by using initial dataset pruning. We evaluated HashMax by comparing its performance to Genmax, a current state-of-the art algorithm for mining maximal frequent itemsets on several datasets of varying sizes and density. In the rest of the paper, we proceed in the following order: section 2 gives basic definitions and outlines the algorithm, section 3 describes the algorithm in detail, and section 4 deals with implementation issues and experimental evaluation.

2 FINDING MAXIMAL FREQUENT ITEMSETS

We are given a *transaction database* D of size $|D|$, whose tuples $t = (item_1, ..., item_{n_t})$ consist of items $item_1, ..., item_{n_t}$, and the number of items can vary from tuple to tuple. We are also given a user-defined real valued support threshold $S \in [0,1]$. An *itemset* is a set of items such that every item appears in some transaction of D. The *size* of an itemset is the number of items in it. The *support of an itemset* I is $\text{supp}(I) := \frac{\text{count}(I)}{|D|}$, where $\text{count}(I)$ denotes the number of transactions in D containing all the items of I. Itemset I is *frequent* if $\text{supp}(I) \geq S$. Itemset I is called *maximal frequent* if it is a frequent itemset and it is not contained (as a set) in any other frequent itemset.

The objective of a maximal frequent itemset mining algorithm is to find a collection F of all maximal frequent itemsets. F is a union of $F_1, ..., F_{max}$, where each F_i contains all maximal frequent itemsets of size i. Here and further, we denote by C_i a set of itemsets of size i that are thought to have a potential to be maximal frequent during the search performed by a maximal frequent itemset mining algorithm. The set C_i is called a *candidate set*. Each C_i is superset of F_i.

In this paper, we propose the HashMax algorithm for finding maximal frequent itemsets in a top-down fashion, while using hashing as a method for computing the itemsets' support. We start with maximal-sized itemsets and proceed to smaller itemsets. As a result, when an itemset is declared frequent, it is also maximal since smaller itemsets have not been generated yet. Special attention must be given to subsets of already discovered maximal frequent itemsets, as those subsets are frequent but not maximal. The algorithm uses a single database scan to build initial structures, and is iterative. Maximal itemsets of size $(max - i + 1)$ are built at iteration i, where max is the size of the largest maximal frequent itemset in the database. The main steps of our algorithm are:

1. **Initial Scan Phase.** scan the database and generate candidate set C_{max}, frequent set F_1 containing frequent itemsets of size 1, and coverage set F_{cover} containing a space-saving representation of F_2.
2. F_1 **Pruning.** remove from F_1 all items whose count is is $|D|$ as these items are members of every maximal frequent itemset and need not to be taken into consideration.
3. **Main Pruning Phase.** go over every candidate itemset $I \in C_{max}$ and delete from I all the items that do not appear in pruned F_1. Afterward, remove from F_{cover} all entries corresponding to items not in F_1.
4. **Generation Phase.** performed for every non-empty candidate set C_i. Go over each candidate itemset $I \in C_{max}$ add I to frequent set $F_{|I|}$ if $\text{supp}(I) \geq S$. Otherwise, if I is not contained in an already generated frequent itemset, add subsets of I of size $|I| - 1$ to candidate set $C_{|I|-1}$.
5. **Stop Criteria.** stop when current candidate itemset C_i is empty.

In order to generate candidate itemsets, we need three main data structures. First is the *candidate set hash table*, denoted by C_i. Each itemset I is stored at place $p(I)$ in C_i. A pass over the hash table C_i generates candidate itemsets for the table C_{i-1} and frequent itemsets for the set F_i. The second data structure we need is the *itemset* I itself, that must keep a list of its items and the *source list*, which is our third main data structure. The source list is a structure that is built gradually during as the algorithm is iterated. For an itemset $I \in C_{max}$, its source list $I.source$ contains the hash value $p(I)$ of I in the hash table C_{max} and the number of database tuples containing I. This value is computed during the initial database scan. For an itemset $I \in C_i$, $i < max$, $I.source$ is the union of source lists as sets for all candidate itemsets $J \in C_{max}$ containing I as a subset. The candidate list of I replaces the need to track transaction IDs by tracking bucket IDs in the hash table C_{max}, since the number of buckets is in general much smaller than the number of transactions.

3 THE HASHMAX ALGORITHM

3.1 Initial Scan and F_1 Creation and Pruning

During the initial scan, described in Algorithm 1, we build two sets: the first frequent set, denoted F_1, and the last candidate set in a form of a hash table, denoted

Algorithm 1: Initial scan phase.

```
Input: database $D$, support $S$
Output: frequent sets $F_1$ and $F_{\text{cover}}$, candidate set
    $C_{max}$, database size $|D|$
 1: $F_1 := \emptyset$, $C_{max} := \emptyset$, $|D| := 0$
 2: for all tuples $t = (t_1, ..., t_{n_t}) \in D$ do
 3:     sort $t$ so that $t_1 \leq ... \leq t_{n_t}$;
 4:     $p := hash(t_1, ..., t_{n_t})$;
 5:     $C_{max}[p].itemset := t$;
 6:     $C_{max}[p].count++$;
 7:     for $i = 1$ to $n_t$ do
 8:         $q := hash(t_i)$; $F_1[q].count++$;
 9:         for $j = i + 1$ to $n_t$ do
10:             $r := hash(t_j)$;
11:             $F_{\text{cover}}[q][r].count++$;
12:         end for
13:     end for
14:     $|D|++$;
15: end for
16: // $F_1$ pruning
17: for all $i$ s.t. $F_1[i].count < S|D|$
    or $F_1[i].count == |D|$ do
18:     $F_1[i] := \emptyset$;
19: end for
20: for all $i$ s.t. $F_1[i] == \emptyset$ do $F_{\text{cover}}[i][j] := \emptyset$;
```

Algorithm 2: Main pruning phase.

```
Input: frequent sets $F_1$ and $F_{\text{cover}}$,
    candidate set $C_{max}$,
    database size $|D|$,
    support $S$.
Output: maximal size $max$ of candidate itemsets,
    set of candidate itemsets $C_{max}$,
 1: $max := 0$;
 2: for all itemsets
    $t = (item_1, ..., item_{n_t}) \in C_{max}$ do
 3:     for $j = 0$ to $n_t$ do
 4:         let $p := hash(item_i)$;
 5:         if $F_1[p].count = \emptyset$ then
 6:             $t := t \setminus \{item_i\}$;
 7:         end if
 8:     end for
 9:     if $max < |t|$ then
10:         $max := |t|$;
11:     end if
12: end for
13: for all itemsets $t \in C_{max}$ do
14:     if $|t| < max$ then
15:         move $t$ to $C_{|t|}$;
16:     end if
17: end for
```

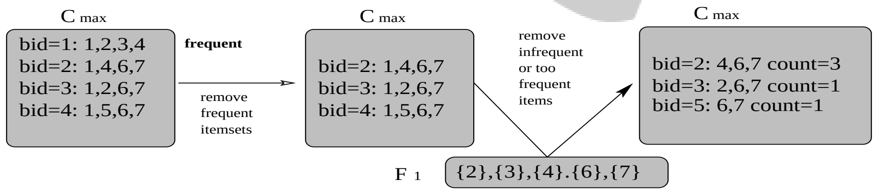

Figure 1: Pruning candidate set C_{max}.

C_{max}, where *max* is the size of the largest itemset in the scanned relations. We assume that a good hash function is chosen for this purpose and collision handling is a part of hash table implementation. F_1 contains all items in D with their respective counts, F_{cover} contains all pairs of items in D with their counts, and C_{max} contains all itemsets that appear in D as transactions. All itemset counts are assumed to be 0 at the start. Since the initial scan allows us to determine the database size $|D|$, a simple pass over F_1 and F_{cover} using values $|D|$ and S (user-defined support value) prunes out all non-frequent itemsets of size ≤ 2. During the pruning phase of Algorithm 2, we leave only frequent items in each itemset generated so far, i.e. items contained in F_1 in the candidate set C_{max}. Some of the candidate itemsets will shrink in size as a result of the pruning. Figure 1 shows the pruning process of a sample itemset C_{max}. The first pruning step is removing itemsets that are already frequent and reporting them to the user. In this example, itemset $1, 2, 3, 4$ is frequent. The second step of pruning consists of removing from every transaction 1-itemsets that do not appear in F_1. If a 1-itemset is too frequent ($\{1\}$ in our example), it is excluded from the mining process but later reported as part of every frequent itemset.

3.2 Candidate Generation

Generating itemsets for the next iteration is a task that needs to be approached with care. The purpose of the Subsets() function is not to generate an itemset with the same source list twice. Let $t = (item_1, ..., item_n)$ be a non-frequent candidate itemset. We observe the following.

- Every subset $t_{\neg i} := t \setminus item_i$ is a candidate itemset of size $|t| - 1$.

- A subset t' of $t_{\neg i}$ of size $|t|-2$ is also a subset of $t_{\neg j}$ if it does not contain $item_j$.

We introduce set parameter $t_{\neg i}.mandatory$ of the itemset $t_{\neg i}$, whose value determines which subsets of $t_{\neg i}$ are not generated by other subset $t_{\neg j}$. An order $t_{\neg 1} < ... < t_{\neg n}$ allows us to make sure that no subset of size $|t|-2$ is missed – we set $t_{\neg i}.mandatory = \{item_1,...,item_{i-1}\}$. Indeed, if an itemset $t' \subset t_{\neg i}$ has size $|t|-2$ and is not contained none of $t_{\neg 1},...,t_{\neg i-1}$ then it contains all the items $item_1,...,item_{i-1}$. Thus, no subset is missed and no subset is generated twice. The procedure for sub-itemset generation is shown in Algorithm 4.

3.3 Cleaning Itemsets

Once a candidate itemset t has been generated and $t.mandatory$ parameter has been set, we can use the frequent 2-itemset data stored in F_{cover} in order to reduce the size of t (*the cleaning process*). The items in $t.mandatory$ need to remain in t in order to allow further subset generation. Therefore, a pair $item_1, item_2 \in t.mandatory$ must be contained in a frequent 2-itemset in order for t to contain a maximal frequent itemset of size ≥ 3. Function Clean() in Algorithm 5 performs the task of removing all items from t that are not paired together with items in $t.mandatory$ in a frequent 2-itemset.

Algorithm 3: HashMax algorithm.

```
Input: candidate set $C_{max}$, support $S$, database size $|D|$, maximal itemset size $max$, $F_{\text{cover}}$
Output: maximal frequent sets $F_{max},...,F_3$
 1: $i := max$
 2: while $C_i$ is not empty and $i \geq 3$ do
 3:   for all itemsets $t = (item_1,...,item_i) \in C_i$ do
 4:     if $count(t) \geq S|D|$ then
 5:       $F_i := F_i \cup \{t\}$;
 6:     else
 7:       $subsets(t)$ :=Subsets($t$);
 8:       for all itemsets $I \in subsets(t)$ do
 9:         Clean($I$, $F_{\text{cover}}$);
10:         if $|I| > 2$ and $I \notin \cup F_i$ then
11:           $p := hash(I)$;
12:           add $I.sourcelist$ to $C_{|I|}[p].sourcelist$;
13:         end if
14:       end for
15:     end if
16:   end for
17:   $i--$;
18: end while
19: return $\cup F_i$;
```

Algorithm 4: Subsets.

```
Input: itemset $t = (item_1,...,item_n)$
Output: subsets of $t$ of size $n-1$ with $mandatory$ parameter set.
1: $subsets(t) := \emptyset$;
2: for $i = 1$ to $n$ do
3:   $t_{\neg i}.mandatory := \{item_j | j < i\}$;
4:   $subsets(t) := subsets(t) \cup \{t_{\neg i}\}$;
5: end for
6: return $subsets(t)$;
```

Algorithm 5: Clean().

```
Input: itemset $t = (item_1,...,item_n)$, $F_{\text{cover}}$, database size $|D|$, support $S$
Output: reduced itemset $t$
1: for all $item_1 \in t \setminus t.mandatory$, $item_2 \in t.mandatory$ do
2:   if $F_{\text{cover}}[hash(item_1)][hash(item_2)] = \emptyset$ then
3:     $t := t \setminus \{item_1\}$;
4:   end if
5: end for
6: return $t$;
```

Figure 2 illustrates the process of an itemset cleaning.

3.4 The Mining Phase

The mining phase of our algorithm consists of scanning the current candidate set C_i, extracting frequent itemset into set F_i and creating candidates for the set C_{i-1} from the non-frequent itemsets of C_i. Subsets of frequent itemsets are frequent but not maximal and therefore are not used for further mining.

Main cycle in lines $2-19$ of Algorithm 3 iterates over the candidate itemset size. At iteration i, lines $4-5$ of the algorithm generate frequent itemsets of size i. These itemsets are maximal because line 10 makes sure that no new candidate is contained in already generated frequent itemsets. Lines $7-8$ of Algorithm 3 generate candidate itemsets of size $i-1$ from a non-frequent itemset of size i. After cleaning the candidate in line 9, the newly generated candidate itemset is placed in the appropriate hash table representing the corresponding candidate set. Since a single candidate of size $i-1$ may arise as a subset of more than one non-frequent candidate itemset of size i, the source list of a new candidate must be the union of source lists of all candidate itemsets containing it as a subset (lines $12-13$). The time complexity of the problem of finding all maximal frequent itemsets is known to lie in Pspace and is well studied (see,

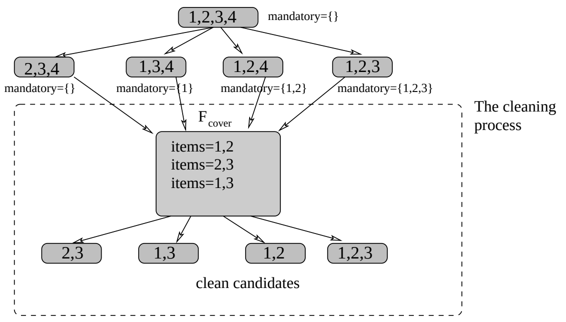

Figure 2: Cleaning itemsets with the help of F_{cover}.

e.g. (Yang, 2004)). The precise complexity of any algorithm that finds maximal frequent itemsets entirely depends on the distribution of the data in a specific dataset. The space complexity of HashMax is constant for each run and is at most $O(n^2)$, where n is the number of transactions. If a bitmap source list representation is used, the space complexity of HashMax algorithm decreases to $O(nlogn)$. At iteration k, the total space used by the algorithm is bounded by $\frac{n^2}{\mathrm{avg}(count(t)-1)}$ where the average is taken over all frequent k-itemsets t, and by $\frac{n\log n}{\mathrm{avg}(count(t)-1)}$ for the bitmap representation, where the average is taken over all frequent k-itemsets.

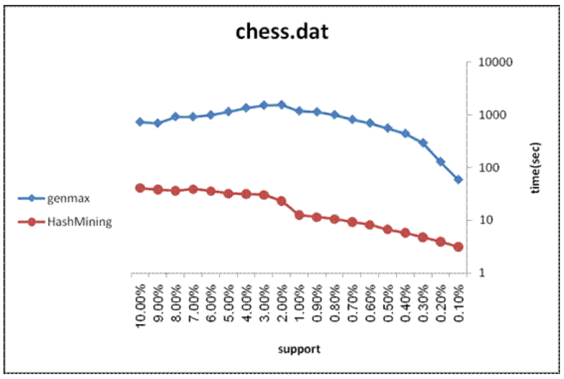

Figure 3: Comparison on the chess.dat database.

4 EXPERIMENTAL EVALUATION

The HashMax algorithm was implemented in Java and it was tested on a machine with Intel Xeon 2.60GHz CPU and 3Gb of main memory running Linux OS. We have compared HashMax to the Genmax algorithm ((Gouda and Zaki, 2005)), using efficient implementation available at (Genmax, 2011). In all charts, the X axis denotes support in % while the Y axis denotes the time that it took the algorithm to produce all maximal frequent itemsets for a given

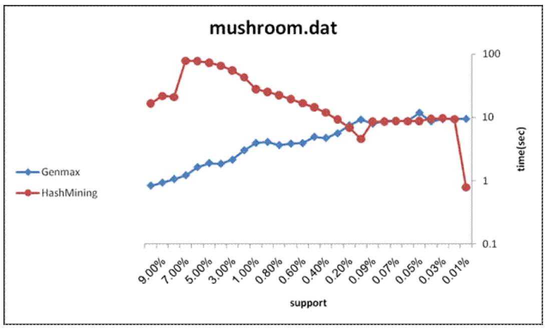

Figure 4: Comparison on the mushroom.dat database.

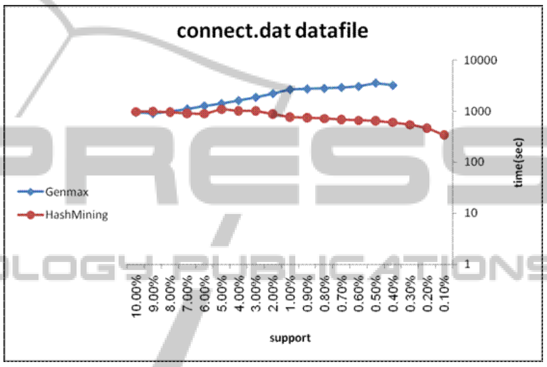

Figure 5: Comparison on the connect.dat database.

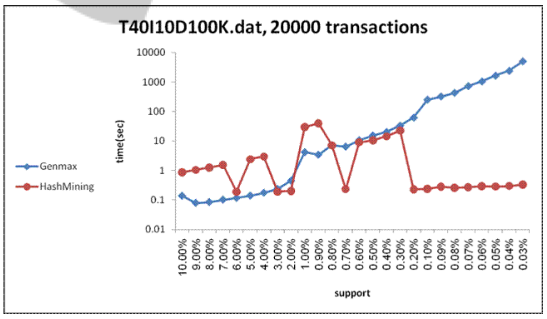

Figure 6: Comparison on a sparse database with 20000 transactions.

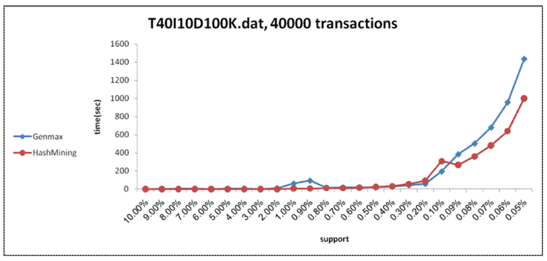

Figure 7: Comparison on a sparse database with 40000 transactions.

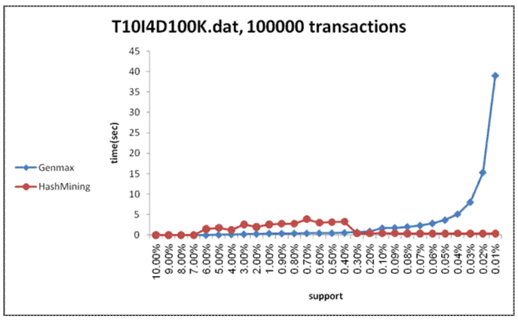

Figure 8: Comparison on a sparse database with 100000 transactions.

support value, in seconds. In Figures 3-6 we used log-scale for the *Y* axis in order to show the difference in small running times better.

We tested both algorithms on several datasets with different sizes and characteristics from the UCI machine learning repository (the datasets are available at (UCI, 2011)). The chess.dat file contains 3196 transactions and is a dense dataset (a *dense dataset* is a dataset that contains transactions with many common items). The number of maximal frequent itemsets in chess.dat varies from tens for very high support values to over ten thousand for lower support values. The mushroom.dat file contains 8124 transactions and is a relatively sparse dataset. It contains thousands of maximal frequent itemsets for low support values. The connect.dat file contains 67557 transactions and represents a dense dataset (for low support values, it contains up to 17000 maximal frequent itemsets). The number of items in each transaction is large and is constant for chess.dat, mushroom.dat and connect.dat. Datasets T10I4D100K.dat and T40I10D100K.dat have variable transaction size and are very sparse. These datasets contain no maximal frequent itemsets of size larger than 2 for higher support values and several thousands maximal frequent itemsets for very low support values. Figure 3 shows a comparison both algorithms on the chess.dat dataset. We see that due to the density of this dataset HashMax always shows substantially better times than Genmax. Figure 4 shows a comparison of both algorithms on the mushroom.dat dataset. Because this dataset is sparse, HashMax gains an advantage over Genmax for lower support values. Figure 5 shows a comparison both algorithms on the connect.dat dataset. As this dataset is quite dense, HashMax consistently shows better times than Genmax. Figures 6 and 7 show a comparison of the two algorithms on parts of T40I10D100K.dat of different sizes (20000 transactions and 40000 transactions respectively). Since the original dataset is very sparse, HashMax shows better results for lower support values. Figure 8 shows a comparison of the two algorithms on large (1000000 transactions) sparse dataset T10I4D100K.dat. The algorithms show similar times for medium support values, but HashMax times are much better for low support values. In conclusion, we have found that HashMax outperforms Genmax for dense datasets (i.e. when the total number of maximal frequent itemsets is significant) throughout and for low support values when tested on sparse datasets). For support values in the range of 0-0.1% the difference in running time was quite noticeable.

ACKNOWLEDGEMENTS

Authors thank the Lynn and William Fraenkel Center for Computer Science for partially supporting this work.

REFERENCES

Agarwal, R., Aggarwal, C., and Prasad, V. (2000). Depth first generation of long patterns. In *ACM SIGKDD Conf.*

Bayardo, R. J. (1998). Efficiently mining long patterns from databases. In *ACM SIGMOD Conf. on Management of Data*, pages 85–93.

Burdick, D., Calimlim, M., and Gehrke, J. (2001). Mafia, a maximal frequent itemset algorithm for transactional databases. In *IEEE Intl. Conf. on Data Engineering*, pages 443–452.

Genmax (2011). Genmax implementation. http://www.cs.rpi.edu/ zaki/www-new/pmwiki.php/Software.

Gouda, K. and Zaki, M. J. (2005). Genmax: An efficient algorithm for mining maximal frequent itemsets. *Data Mining and Knowledge Discovery 11(3)*, pages 223–242.

Han, J., Pei, J., and Yin, Y. (2000). Mining frequent patterns without candidate generation. In *ACM SIGMOD Conf. on Management of Data*, pages 1–12.

Hu, T., Sung, S. Y., Xiong, H., and Fu, Q. (2008). Discovery of maximum length frequent itemsets. In *Inf. Sci. 178(1)*, pages 69–87.

Lin, D.-I. and Kedem, Z. M. (1998). Pincer search: A new algorithm for discovering the maximum frequent set. In *EDBT*, pages 105–119.

UCI (2011). Uci machine learning data repository. http://archive.ics.uci.edu/ml/index.html.

Yang, G. (2004). The complexity of mining maximal frequent itemsets and maximal frequent patterns. In *KDD*, pages 344–353.

Zaki, M. J., Parthasarathy, S., Ogihara, M., and Li, W. (1997). New algorithms for fast discovery of association rules. In *Third Int 1 Conf. on Knowledge Discovery in Databases and Data Mining*, pages 283–286.