Dynamic Miss-Counting Algorithms: Finding Implication and Similarity Rules with Confidence Pruning

Shinji Fujiwara
fujiwara@crl.hitachi.co.jp *

Jeffrey D. Ullman
ullman@db.stanford.edu †

Rajeev Motwani
rajeev@CS.Stanford.EDU ‡

Abstract

Dynamic Miss-Counting algorithms are proposed, which find all implication and similarity rules with confidence pruning but without support pruning. To handle data sets with a large number of columns, we propose dynamic pruning techniques that can be applied during data scanning. DMC counts the numbers of rows in which each pair of columns disagree instead of counting the number of hits. DMC deletes a candidate as soon as the number of misses exceeds the maximum number of misses allowed for that pair. We also propose several optimization techniques that reduce the required memory size significantly. We evaluated our algorithms by using 4 data sets, i.e., Web access logs, Web page-link graph, News documents, and a Dictionary. These data sets have between 74,000 and 700,000 items. Experiments show that DMC can find high-confidence rules for such a large data sets efficiently.

1. Introduction

Finding implication and similarity rules are two of the most interesting issues in the data mining area. Finding implication rules, also known as association-rule mining, was initially proposed by Agrawal, Imielinski, and Swami [1]. Finding similarity rules is also useful for various kinds of data-mining such as copy detection, clustering and collaborative filtering [5, 16, 9, 12, 11, 17].

Suppose that we have a set of transaction data D that has n transactions and m boolean attributes: $A_1, A_2, \ldots, A_m$. We say $A_i \Rightarrow A_j$ is an implication rule if the fraction of the transactions that contain both A_i and A_j among those transactions that contains A_i is more than a minimum *confidence*. We also say $A_i \simeq A_j$ is a similarity rule if the fraction of the transactions that contain both A_i and A_j among those transactions that contains either A_i or A_j is more than a minimum *similarity*. The goal is to identify all valid rules for a given data.

Most of algorithms proposed in the past few years are based on support pruning, which prunes the attributes that have low frequency [1, 2, 4]. This approach extracts association rules with high support (i.e., high frequency) efficiently. However they discard low-support items all the time.

Example 1.1 : Suppose that we want to extract similar pages in the Web by analyzing the page-link graph. In order to extract pages that have similar sets of page links, we transform a page-link graph to a binary matrix whose columns represent source pages and whose rows represent destination pages. If a page p_i has a link to p_j, then the column c_i for the row r_j is set to 1. Using a high-support threshold, we can only get similarity rules between pages that have many links, such as a directory page. □

We have focused on finding high-confidence implication and similarity rules without support pruning. We have done some work on this issue and earlier developed a family of algorithms such as *Min-Hash* and *Locality-Sensitive Hashing* schemes [7, 8, 10]. These algorithms use randomized techniques [13], and they can extract similar pairs very efficiently. However, these algorithms still have a chance of yielding false positives and false negatives.

In this paper we propose a family called *Dynamic Miss-Counting (DMC)* algorithms to avoid both false negatives and false positives. Let M be a boolean matrix that represents the data D. M has n rows and m columns. Each row represents a transaction in D, and in each row, a column c_i is set to '1' if the transaction has an attribute A_i.

The DMC algorithm uses a confidence pruning technique, rather than support pruning. The idea of pruning for high-confidence pairs is not new. Sergey Brin did some (unpublished) experiments, and the paper [3] likewise explored iterative methods for converging on the pairs of columns with highest correlation. These methods never look at all pairs of columns, and make many passes over the data. They

*Hitachi Ltd., Central Research Laboratory. This work was done while the author was on leave at the Department of Computer Science, Stanford University.

†Department of Computer Science, Stanford University. Supported in part by NSF Grant IIS-9811904.

‡Department of Computer Science, Stanford University. Supported in part by NSF Grant IIS-9811904.

are very expensive techniques useful for enormous data sets (e.g., Brin used them to look for correlations among the 100 million or so words that appear on the Web). Our methods are useful for sets with somewhat smaller numbers of columns, they extract all pairs of columns with a similarity or implication that is above a small threshold, and they use only two passes through the data and realistic amounts of main memory.

The key idea of the DMC algorithm is counting the numbers of rows in which each pair of columns disagree instead of counting the number of hits, and deleting a counter as soon as the number of misses exceeds the maximum number of misses allowed for that pair.

T_1: A_2, A_3
T_2: A_1, A_2, A_3
T_3: A_1
T_4: A_1, A_2
T_5: A_2, A_3

(a) Data *D*

$$\begin{array}{c|ccc|} & c_1 & c_2 & c_3 \\ \hline r_1 & 0 & 1 & 1 \\ r_2 & 1 & 1 & 1 \\ r_3 & 1 & 0 & 0 \\ r_4 & 1 & 1 & 0 \\ r_5 & 0 & 1 & 1 \\ \hline \end{array}$$

(b) Matrix *M*

Figure 1. Data format

Example 1.2 : Fig. 1 is an example of D and M. Suppose that we would like to extract 100%-confidence implication rules for this matrix M. In this sample case, no miss at all between two columns is allowed. When we read r_1, we have to keep two candidates: $c_2 \Rightarrow c_3$ and $c_3 \Rightarrow c_2$. Next, when we read r_2, we only have to add two more candidates: $c_1 \Rightarrow c_2$ and $c_1 \Rightarrow c_3$. We do not have to add candidates such as $c_2 \Rightarrow c_1$ or $c_3 \Rightarrow c_1$, since they have already had 1 miss at r_1. To detect the number of misses from one column to another that has already occurred, we maintain a counter for each column giving the number of 1's in that column seen so far. Since the c_2 counter is 1 at r_2, we find that the candidate pairs that are not in the candidate list for c_2 have already had 1 miss, which is too many in this simple example.

Furthermore, when we read r_3, we can immediately delete candidates $c_1 \Rightarrow c_2$ and $c_1 \Rightarrow c_3$, since these candidates miss at this row. After reading all rows in the matrix M, only one rule, $c_3 \Rightarrow c_2$, survives. □

Example 1.3 : Consider a column c_i that has 100 1's, and suppose we want to find implication rules with 85% or more confidence. In this case, the number of misses from c_i to any other column must not be more than 15. Therefore, we can delete a counter for candidate pairs $c_i \Rightarrow c_j$ as soon as the number of misses exceeds 15. Furthermore, we do not have to add a new counter for c_i after we have seen 16 rows in which c_i is set to 1, because a column c_j that has not yet appeared already has had 16 misses for the rule $c_i \Rightarrow c_j$. □

Note that this algorithm requires as many as m^2 counters in the worst case. However, in real data such as Web-page-link graphs, most pages are linked to ten or so pages, while the number of pages is in the millions. Therefore the number of counters for most pages will not approach, even remotely, the number of pages. This fact significantly reduces the memory requirements and computation cost.

In this paper we also propose several techniques such as row re-ordering, memory-explosion elimination, 100%-rule pruning, column-density pruning and maximum-hits pruning, each of which contribute to reducing the size of memory significantly.

We start to define our problem in Section 2. In Section 3 we present conventional data mining algorithms and our new algorithm. We have also applied many other optimization techniques to our DMC algorithm, which we mention in Section 4. We then present a variant algorithm for finding similarity rules in Section 5. In Section 6 we describe the data that we used to evaluate algorithms and show the experimental results of algorithms. We conclude in Section 7 by discussing some extensions of our work to apply our approach to extract more complicated rules among three or more attributes.

2 Problem statement

We view the data as a $0/1$ matrix M with n rows and m columns. Each column represents an "attribute," and each row represents a "transaction." Define S_i to be the set of rows that have a 1 in column c_i.

We define the confidence of the rule $c_i \Rightarrow c_j$ as

$$\textit{Conf}(c_i, c_j) = \frac{|S_i \cap S_j|}{|S_i|}$$

That is, the confidence of c_i and c_j is the fraction of rows, among those containing a 1 in c_i, that contain a 1 in both c_i and c_j. Note that if $|S_i| < |S_j|$, then $Conf(c_j, c_i) < Conf(c_i, c_j)$. Therefore, we consider only rules $c_i \Rightarrow c_j$ such that $|S_i| < |S_j|$ or ($|S_i| = |S_j|$ and $i < j$), and our goal is to extract all rules that have *minconf* or more confidence, where $0 < \textit{minconf} \leq 1$.

We also define the similarity of a column pair, $c_i \simeq c_j$ as

$$Sim(c_i, c_j) = \frac{|S_i \cap S_j|}{|S_i \cup S_j|}$$

That is, the similarity of c_i and c_j is the fraction of rows, among those containing a 1 in either c_i or c_j, that contain a

1 in both c_i and c_j. Note that this definition is symmetric with respect to c_i and c_j. Our other goal is to extract all combinations of column pairs that have *minsim* or more similarity, where $0 < minsim \leq 1$.

3 Algorithms

In this section we review conventional algorithms: *a-priori* and *Min-Hash*. Then we overview our new *Dynamic Miss-Counting* algorithm.

3.1 A-priori algorithm

A-priori [1, 2], which uses a support pruning technique to reduce the search space, is the most famous and one of the most effective algorithms for finding association rules. It prunes the candidate pairs if the frequency of each column by itself does not exceed the minimum support threshold.

However, there are some cases where support pruning does not work very well. For example, consider the data in Fig. 1, with minimum support, *minsup*, of 50% and minimum confidence, *minconf*, of 85%. Since all columns have 3 or more 1's in M, no candidate pairs can be pruned by a-priori, and it requires as much memory as $m(m-1)/2$ counters. To reduce the number of counters, the DHP [14] algorithm, which uses a hash-based technique to prune candidate pairs, was proposed. This algorithm works well to prune most of the useless counters in some cases, while it does not solve the problem in the next paragraph.

The most significant problem for these algorithms is that if many columns in the matrix remain after support pruning, they must use too many counters in main memory. For example, our Web-page-link data has about 700,000 columns, and even if we prune the pages that have less than 10 1's, there still remains 58,000 columns. Therefore about 1.7 billion counters would have to fit in main memory.

3.2 Min-Hash algorithm

[7, 8] proposed the *Min-hash* algorithm in order to find all similar pairs without support pruning. The basic idea in the *Min-hash* algorithm is to permute the rows randomly, and for each column c_i, to compute its hash value $h(c_i)$ to be the index of the first row under the permutation that has a 1 in that column. To avoid physically permuting rows, we instead give a random hash number to each row, and extract for each column the minimum hash value of any row where the column has a 1. Since for any column pair (c_i, c_j), $Prob[h(c_i) = h(c_j)] = Sim(c_i, c_j)$, we can estimate similarity by repeating the random min-hashing process k times. Note that we can generate all k min-hash values with a single data scan, as explained in [8].

This algorithm works very well to find almost all truly similar pairs. However, it can not find all pairs with 100% reliability, because there is a small probability of generating false negatives. Furthermore, generated candidates have to be verified to confirm that they actually satisfy the minimum similarity.

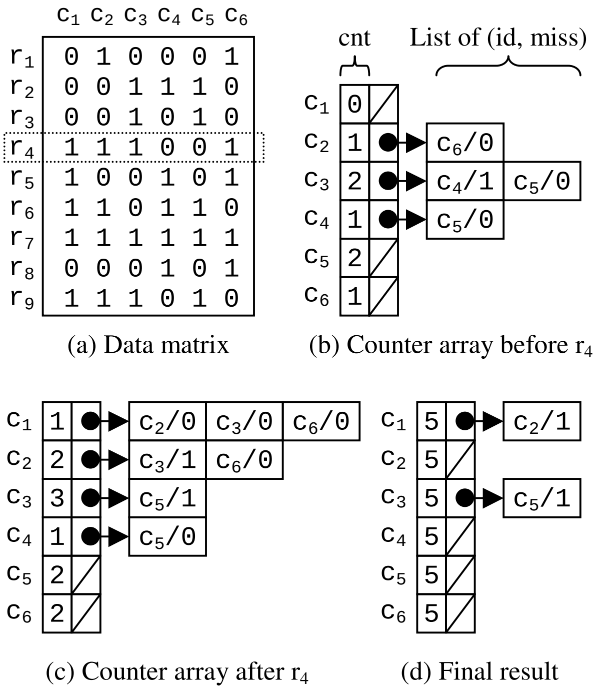


Figure 2. DMC Algorithm

3.3 Dynamic Miss-Counting algorithm

Before describing the DMC algorithm in detail, let us introduce an example that shows how our algorithm works.

Example 3.1 : Fig. 2 is example data that has 9 rows. Suppose that we want to extract 80%-or-more-confidence implication rules. Since each column contains five 1's, only one miss per column is allowed. Fig. 2(b) shows the data structure that:

1. Counts number of 1's that have already appeared in each column.

2. Keeps lists of candidates and miss counters for these candidates.

For instance, before processing r_4, 4 candidate rules are kept in the lists: $c_2 \Rightarrow c_6$, $c_3 \Rightarrow c_4$, $c_3 \Rightarrow c_5$, and $c_4 \Rightarrow c_5$. Recall that we only keep candidates such that $|S_i| < |S_j|$

or ($|S_i| = |S_j|$ and $i < j$). We do not have to keep an entry for c_6, because c_6 has the minimum number or 1's and the largest column ID number. Since $c_3 \Rightarrow c_4$ has already missed at r_3, its miss counter is equal to 1, other miss counters are 0.

At r_4, we process as follows:

- c_1: Since c_1 appears for the first time, every column in r_4 is set as a candidate with c_1. In this example, c_2, c_3, and c_6 are placed in the list for c_1.

- c_2: Though c_2 has already appeared in r_1, the c_2 counter is equal to or less than the maximum number of misses allowed, which is 1 in this example. Therefore we add to the list for c_2 every column in r_4 that has not previously appeared on the list for c_2, which is only c_3 in this case. Note that the new candidates have already missed as many times as the c_2 counter's value, which is 1 here. A candidate that has already been added to c_2's list, but that does not appear in this row, should have its miss counter increased (there is no such a column in this example).

- c_3: Since c_3 has already appeared in r_2 and r_3, a column that has not yet been added to the candidate list for c_3 has had 2 misses already. Therefore, we do not have to add new candidates for c_3. We only have to check the candidates in its candidate list, and delete a candidate whose miss counter exceeds the maximum misses. In this example, both c_4 and c_5 get misses, and only c_5 survives.

Fig. 2(c) shows the counter array after processing r_4. Fig. 2(d) is the final result after reading all rows, which shows $c_1 \Rightarrow c_2$ and $c_3 \Rightarrow c_5$ have at least 80% confidence. □

Algorithm 3.1 : DMC-base

1. Read M and count the number of 1's for each column, $ones(c_i)$.

2. Calculate the maximum number of misses for each column, $maxmis(c_i) = \lfloor (1 - \mathit{minconf}) \times ones(c_i) \rfloor$. Clear the counter and candidate list for each column, i.e., $cnt(c_i) = 0$ and $cand(c_i) = NULL$.

3. In the second scan, for each row r_i of M do:

   (a) For each column c_j that has 1 in a row r_i (we describe this condition in the form of $c_j \in r_i$, since a row consists of a set of columns), process as following:

   Case $cnt(c_j) = 0$: Create a candidate list for c_j by adding all columns $c_k \in r_i$ such that $ones(c_k) > ones(c_j)$ or ($ones(c_k) = ones(c_j)$ and $k > j$). Set all miss counters for candidates to 0.

   Case $cnt(c_j) \leq maxmis(c_j)$: Merge the candidate list with the column list in r_i. For all columns $c_k \in r_i \cup cand(c_j)$, if c_k exists only in r_i and c_k satisfies the same condition as the above case, add c_k into the candidate list, while initializing the miss counter to $cnt(c_j)$. If c_k exists only in the candidate list, increase the miss counter for c_k.

   Case $cnt(c_j) > maxmis(c_j)$: Merge the candidate list for c_j with the column $c_k \in r_i$. For all columns $c_k \in cand(c_j)$, if c_k does not exist in r_i, then increase the miss counter for c_k. If the counter exceeds $maxmis(c_j)$, then delete c_k from the candidate list for c_j.

   (b) After processing all c_j, increase the counter $cnt(c_j)$ for each column c_j in r_i and continue processing the next row. If $cnt(c_j)$ is equal to $ones(c_j)$, then output all rules $c_j \Rightarrow c_k$, where c_k is a column in $cand(c_j)$, and release the candidate list.

4 Memory optimization

The *DMC-base* algorithm reduces the memory requirement significantly, e.g. in the case of Web-page-link data with pruning of columns with fewer than 10 1's, *DMC-base* requires about 0.33GB memory, while *a-priori* requires 6.8GB memory. However, our algorithm still requires as much as m^2 main memory in the worst case. In this section we show some techniques that reduce the amount of main memory.

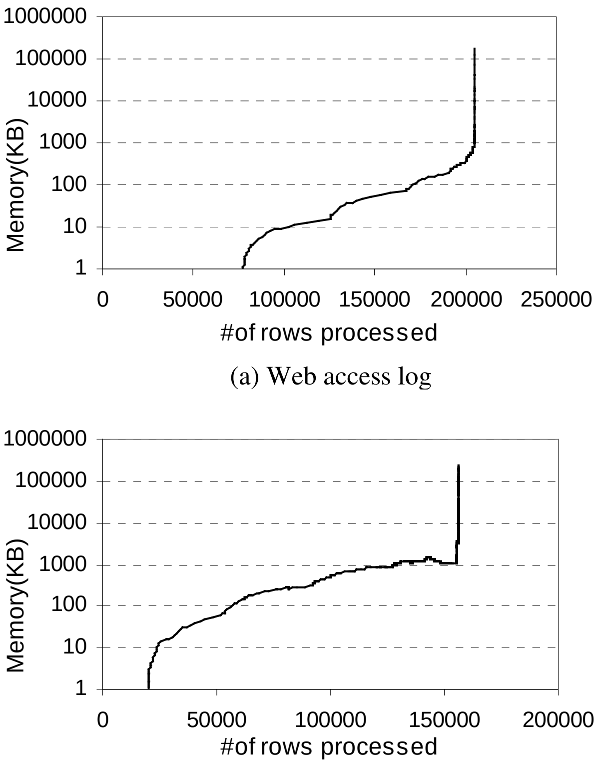
4.1 Row re-ordering

Suppose that r_7 is the first row in the matrix M in Fig. 2. We have to create all column pairs as candidates in this case. In general, the denser the rows that come first, the more memory is necessary for the *DMC-base* algorithm. Therefore we should read sparser rows first in order to reduce the memory size required.

In Example 3.1, the history of the total number of candidates in the candidate lists is $(1, 4, 4, 7, 9, 7, 7, 6, 2)$ if we read the matrix M in the original order; while it is $(1, 2, 3, 5, 6, 8, 5, 2, 2)$ if we read M in the sparsest-first order $(r_1, r_3, r_8, r_2, r_5, r_4, r_6, r_9, r_7)$. This technique works very well, especially if the row density of the matrix has a wide distribution. For example, the Web-access log has a few clients such as Web crawlers that access all pages on the site, while most clients access only a few pages.

This optimization reduces the memory size significantly, e.g. in the previous case of Web-page-link data with support pruning, we can reduce the memory size from 0.33GB to 0.033GB.

However, it is expensive to sort the original data by density. Instead of sorting, we make buckets corresponding to ranges of row density for each row. That is, we divide the original data according to the number of 1's in each row with ranges of $[2^i, 2^{i+1})$, when we scan the data the first time. Then, in the next scan, we read the lower density buckets first. Note that the number of buckets is no more than $\lceil log_2 m \rceil + 1$.



(a) Web access log

(b) Web page-link graph

Figure 3. Memory size for a counter array

4.2 Memory-explosion elimination

Scanning the densest rows last may cause memory explosion at the end of the algorithm. Fig. 3 shows the memory consumption for Web-access-log and Web-page-link data without support pruning, when extracting 100%-confidence rules. The required memory size explodes at the end of the processing, since both data sets have several rows with many 1's.

To avoid memory explosion, we switch the algorithm from the memory-consuming algorithm, *DMC-base*, to an algorithm that uses more time, but uses less space. We assume that there are few rows with many 1's. Therefore, when and if the memory explosion begins, we can read the rest of the rows and create bitmaps for each column in main memory. The low-memory algorithm, *DMC-bitmap*, consists of 2 phases. In the first phase, it cleans up the candidate list for those c_j such that $cnt(c_j) > maxmis(c_j)$, and in the second phase, it extracts implication rules, $c_j \Rightarrow c_k$, for those c_j such that $cnt(c_j) \leq maxmis(c_j)$.

Algorithm 4.1 : DMC-bitmap

1. When the memory needed for counters exceeds a threshold, and the bitmaps for the rest of the rows $r_i (i = t, t+1, \ldots, n)$ can fit in main memory, read the rest of the rows r_i from M and create $(n - t + 1)$ bits of a bitmap, $bm(c_j)$, for each column c_j such that $cnt(c_j) < ones(c_j)$ —that is, we do not have to create bitmaps for those columns that have no 1's in the rest of rows. In order for this method to work, there must be enough memory to maintain the counter array until such time as the bitmaps will fit in the same memory.

2. (Phase 1) For each column c_j that has a non-NULL candidate list and $cnt(c_j) > maxmis(c_j)$:

   - Count the number of misses for $c_j \Rightarrow c_k$ such that $c_k \in cand(c_j)$ by counting the number of 1's in $bm(c_j) \wedge \overline{bm(c_k)}$.
   - If the total number of misses is no more than $maxmis(c_j)$, then output $c_j \Rightarrow c_k$ as a result.
   - Free the candidate list for c_j.

3. (Phase 2) For each column c_j such that $cnt(c_j) \leq maxmis(c_j)$:

   - Initialize hit counter $hit(c_k)$ to 0. If $cand(c_j)$ is non-NULL, set $hit(c_k) = cnt(c_j) - mis(c_j, c_k)$ for each column $c_k \in cand(c_j)$, where $mis(c_j, c_k)$ is the number of misses of c_j against c_k.
   - Count the number of hits in the rest of rows, r_i $(i = t, t+1, \ldots, n)$, by increasing each hit counter $hit(c_k)$ such that $c_k \in r_i$, for each row r_i such that $r_i \in c_j$.
   - For each column c_k such that $ones(c_k) > ones(c_j)$ or $(ones(c_k) = ones(c_j)$ and $k > j)$, if $hit(c_k) \geq ones(c_j) - maxmis(c_j)$, then output $c_j \Rightarrow c_k$ as a result.

4.3 100%-rule pruning

Finding 100%-confidence rules is much easier than finding less-than-100%-confidence rules. We do not have to count the number of misses, since we can delete a candidate as soon as we find a miss for it. Therefore, we only have to keep the candidate ID lists in main memory. Furthermore, we can also simplify both the *DMC-base* and *DMC-bitmap* algorithms, since after finding the first 1 on a column c_j, there is no chance of adding any candidates into the candidate list, $cand(c_j)$.

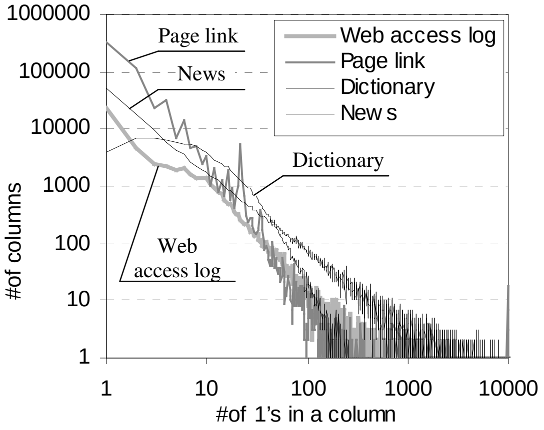


Figure 4. Column density distribution

Suppose that we want to extract 90% or more confidence rules. In such a case, a column that has fewer than 9 1's must have no miss. Therefore, if we can extract 100%-confidence rules first, then we only consider those columns that have 10 or more 1's. Fig. 4 shows the distribution of the number of 1's in each column of four real data sets. Since our data has many columns with few 1's, this optimization improves the performance.

4.4 DMC-imp algorithm

The final algorithm to extract implication rules is following.

Algorithm 4.2 : DMC-imp

1. Read M and count the number of 1's for each column, $ones(c_i)$. Partition the data into buckets B_i, $i = 0, \ldots, \lceil log_2 m \rceil$, according to the number of 1's for each row.
2. Extract 100%-confidence rules by using the simplified *DMC-base* and *DMC-bitmap* algorithms. Note we can skip Step 1 of *DMC-base* and we should scan the sparsest buckets first. In Phase 2 of *DMC-bitmap*, we count the number of misses between c_j and all c_k that have at least one of rows $r_t, r_{t+1}, \ldots, r_n$ in common with c_j.
3. Remove columns c_j such that $ones(c_j) \leq 1/(1 - minconf)$.
4. Extract less-than-100%-confidence rules by using the *DMC-base* and *DMC-bitmap* algorithms. Note that we can skip Step 1 of *DMC-base* and we should scan the sparsest buckets first.

In our implementation, we switch the algorithm from *DMC-base* to *DMC-bitmap* when the number of remaining rows becomes 64 or less, and the memory size for the counter array that keeps both miss counters and candidate lists for each column exceeds 50MB. Note that even if the memory size exceeds 50MB, we do not switch the algorithm if the number of remaining rows is more than 64, regardless of the number of columns.

5 Finding similarity rules

We can find similarity rules more efficiently than we can implication rules, since we can apply two more optimization techniques: *column-density pruning* and *maximum-hits pruning*.

5.1 Column-density pruning

If column c_i and c_j, such that $|S_i| \leq |S_j|$, have *minsim* or more similarity, the ratio of the number of 1's, $|S_i|/|S_j|$, must be *minsim* or more:

$$minsim \leq Sim(c_i, c_j) = \frac{|S_i \cap S_j|}{|S_i \cup S_j|} \leq \frac{|S_i|}{|S_j|} \leq 1$$

Therefore we can save the memory space for miss counters for those column pairs c_i and c_j such that $|S_i|/|S_j| <$ *minsim*, since we do not have to consider such pairs.

5.2 Maximum-hits pruning

Similarity is affected by the number of misses of both c_i against c_j and c_j against c_i, while confidence depends only on the number of misses of c_i against c_j, where $|S_i| \leq |S_j|$. Therefore, we can prune candidates earlier by considering the maximum number of hits, as follows:

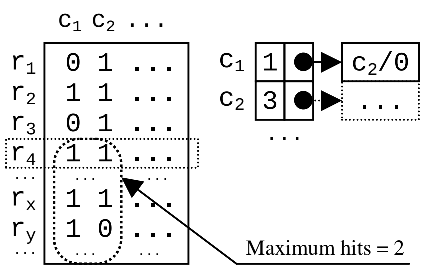


Figure 5. Maximum hits pruning

Example 5.1 : Consider a matrix M as Fig. 5. Since we do not count the number of misses between c_i and c_j such that $|S_i| > |S_j|$, we only maintain a miss counter from c_1 to c_2. If we want to extract 75%-or-more similar column pairs, one miss is allowed between c_1 and c_2. At r_2, we initialize the miss counter for (c_1, c_2) to 0. Then, at r_4 we check this candidate again. [1] We should delete the miss counter for (c_1, c_2), even though both c_1 and c_2 are 1 in r_4. Because $cnt(c_1) = 1$ and $cnt(c_2) = 3$ before reading r_4, the numbers of remaining 1's for each column are 3 and 2, respectively. Therefore we know that there are at most 2 rows with both c_1 and c_2 in the following rows. Since this pair has already had 1 hit in r_2, the maximum possible number of hits, $\widehat{hit}$, is at most 3, and the maximum possible similarity, $\widehat{Sim}(c_1, c_2)$ is $\widehat{hit}/(ones(c_1) + ones(c_2) - \widehat{hit}) = 0.5$. □

Let $rem(c_i)$ be the number of remaining 1's of c_i, i.e. $rem(c_i) = ones(c_i) - cnt(c_i)$. Then, the maximum number of hits with c_i and c_j such that $ones(c_i) \leq ones(c_j)$ is calculated as follows:

- If $rem(c_i) \leq rem(c_j)$, all remaining 1's of c_i can match 1's of c_j. Therefore, the maximum possible hits, $\widehat{hit}$, is $ones(c_i) - mis(c_i)$
- If $rem(c_i) > rem(c_j)$, then $rem(c_i) - rem(c_j)$ of the remaining 1's of c_i can not match 1's of c_j. Therefore, the maximum possible hits, $\widehat{hit}$, is $ones(c_i) - mis(c_i) - (rem(c_i) - rem(c_j))$.

Using $\widehat{hit}$, we can calculate the maximum possible similarity,

$$\widehat{Sim}(c_i, c_j) = \widehat{hit}/(ones(c_i) + ones(c_j) - \widehat{hit}),$$

which can be used for candidate pruning.

5.3 DMC-sim algorithm

The overall algorithm to find similar column pairs is following.

Algorithm 5.1 : DMC-sim

1. Same as Step 1 of *DMC-imp*.
2. Extract 100%-similar, i.e. identical, columns. We only have to keep candidates such that $ones(c_i) = ones(c_j)$ with no miss. To avoid memory explosion, we also apply a variant of the *DMC-bitmap* algorithm. The main points that we should modify it are following:
    - Compare those columns that have the same number of 1's, since we want to extract identical pairs.
    - Extract those column pairs c_i and c_j that have the same bitmap instead of counting the number of 1's in $bm(c_j) \wedge \overline{bm(c_k)}$.
3. Remove columns c_j such that $ones(c_j) \leq 1/(1 - \textit{minsim}) - 1$. Note that the cut-off threshold is not $1/(1 - \textit{minsim})$, since there might be less-than-100% similar pairs between the columns whose number of 1's are $\lceil 1/(1 - \textit{minsim}) - 1 \rceil$ and $\lceil 1/(1 - \textit{minsim}) \rceil$.
4. Extract less-than-100% similar columns by using a variant of *DMC-base* and *DMC-bitmap* algorithms. The modification points are following:
    - Skip Step 1 of *DMC-base* and scan the sparsest bucket first.
    - Keep candidates such that $ones(c_i)/ones(c_j)$ is *minsim* or more.
    - Discard candidates whenever the maximum possible similarity, $\widehat{Sim}(c_i, c_j)$, is less than *minsim.*

[1] Note that we could have deleted (c_1, c_2) at r_3, where c_2 has 1 but c_1 does not. However, in order to apply the maximum-hits pruning from a dense column c_d to sparse columns c_s (from c_2 to c_1, in this example), we have to check all candidate lists for sparse columns c_s, which increases the computation time significantly. Therefore we only apply the maximum-hits pruning from a sparse column to dense columns.

6 Performance Evaluation

We implemented our algorithms, *DMC-imp* and *DMC-sim*, on a Sun Ultra 2 (2×200MHz, 256MB memory) workstation. In this section, we describe the data sets that we used for our experiments. Then we show the experimental results of both algorithms. To compare DMC algorithms with *a-priori* and *Min-Hash* algorithms, we have done an experiment with applying both support pruning and confidence pruning. Note that support pruning can be applied to DMC and *Min-Hash* algorithms in the same manner as *a-priori*. Finally, we show sample rules extracted from news articles.

6.1 Data sets

Table 1 shows the size of the data sets we used in our experiments. The meaning of each data set is followings:

1. Web access log: This data set consists of the log entries from the sun Web server. The columns in this case are the URL's and the rows represent distinct client IP address that have accessed the server recently. An entry is set to 1 if there has been at least one hit for that URL from that particular client IP. The data set, *Wlog*, has more than 200,000 rows and 75,000 columns. We also prepared the data set *WlogP* by pruning those columns with 10-or-fewer 1's. The pruned data set *WlogP* has 13,000 columns.

2. Web-page-link graph: This data represents the URL link graph for the Stanford site, which has about 700,000 pages. Both columns and rows are the URL's. An entry is set to 1 if there is a link from the page p_i to the page p_j. The rows are p_i and the columns are p_j in the data *plinkF*, and the rows are p_j and the columns are p_i in the data *plinkT*. Extracting similar columns from *plinkF* means extracting pages that are referred to by similar sets of pages, while extracting from *plinkT* means extracting pages that have similar sets of links.

3. News documents: This data, *News*, consists of 84,000 Reuters news documents. Each row is a document, and each column is a word. *Stop words*, which appear among the documents very frequently, are pruned before processing, and 170,000 words remain. By using implication rules with low-support pruning, we can get sets of words each of which is related to a news topic. To compare the performance with other algorithms, we also prepare a smaller data set, *NewsP*, since other algorithms could not be applied to our data sets in a reasonable execution time on our machine due to its memory size.

4. Dictionary: This is an on-line version of the 1913 Webster's dictionary that is available through the Gutenberg Project [15]. Columns are *head words* (words being defined) and rows are *definition words* (words used in the definition). There are 96,000 head words and 45,000 definition words. We can get similar definition words, such as *'brother-in-law'* and *'sister-in-law'*, by extracting similarity rules.

Table 1. Real data sets

Data	Rows	Columns
Wlog	218,518	74,957
WlogP	203,185	13,087
pliknF	173,338	697,824
plinkT	695,280	688,747
News	84,672	170,372
NewsP	16,392	9,518
dicD	45,418	96,540

6.2 Experimental results

In this section we discuss our experimental results. We use 6 sets of the data – *Wlog*, *WlogP*, *linkF*, *linkT*, *News*, and *dicD* – for evaluating *DMC-imp* and *DMC-sim*. We also use one small data set *NewsP* for comparing the performance among the algorithms including *a-priori* and *Min-hash*. Fig. 6 shows the experimental results. The left graphs are for finding implication rules, and the right graphs are for finding similarity rules. Fig. 6(a) and (b) plot the execution time with a different threshold. Each execution can be finished in a reasonable time if we want to extract 85%-or-more rules. The execution time decreases lineally according to increasing the threshold, in general.

Fig. 6(c) and (d) shows the details of the execution time for *Wlog*. The execution time for pre-scanning is small compared to other execution times. The execution time for finding 100% rules is also small and constant for each threshold, while the execution time for finding less than 100% rules depends on the threshold.

The execution times of *DMC-imp* and *DMC-sim* for *plinkT* have a gap between 80% and 75% threshold. Fig. 6(e) and (f) show the detail of the execution time. In these cases, the low-memory algorithm, *DMC-bitmap*, jumps up from 22 to 398 seconds in the case of *DMC-imp* and from 27 to 399 seconds in the case of *DMC-sim*, respectively. The reason why this jump-up occurs is that larger rows that are handled in the *DMC-bitmap* includes many columns that have frequency 4. Since we can not prune columns with frequency 4 if the threshold is 75% or less, the execution time for *DMC-bitmap* was dramatically increased. If we could apply low-support pruning before executing algorithms, it would not occur.

Fig 6(g) and (h) show the maximum memory size for the counter array that keeps candidate IDs and their miss-counters. *DMC-sim* requires much less memory than *DMC-imp*, since *DMC-sim* can use the two additional pruning techniques that we mentioned in Sec. 5. Except *DMC-imp* for *News* with 75%-or-less threshold, all executions can fit in main memory. Since we apply the *DMC-bitmap* algorithms, the memory requirement does not explode even as the threshold decreases. For example, the memory requirement for *plinkT* does not jump up even if the threshold decreases from 80% down to 70%, while the execution time does.

In order to compare the *DMC* algorithms to *a-priori* and *Min-Hash*, we generated a pruned data set of news documents, *NewsP*. We created this data from 16,392 news documents, and pruned the columns with minimum support threshold 35 (0.2%) and maximum support threshold 3278 (20%). Since the number of remaining columns was 9518, the counters for all pairs could be fit in main memory (that required 172MB). This situation is best for a-priori, since the memory optimization techniques for DMC and *Min-Hash* will not improve their performance significantly.

Fig 6(i) and (j) show the execution times of these algorithms. The *K-Min* algorithm is a variant algorithm of *Min-Hash*, which can extract implication rules instead of similarity rules. However, it could not extract complete sets of true rules; therefore we plotted the execution time when the number of false negatives was less than 10%. The other algorithms, including *Min-Hash* for finding similarity rules,

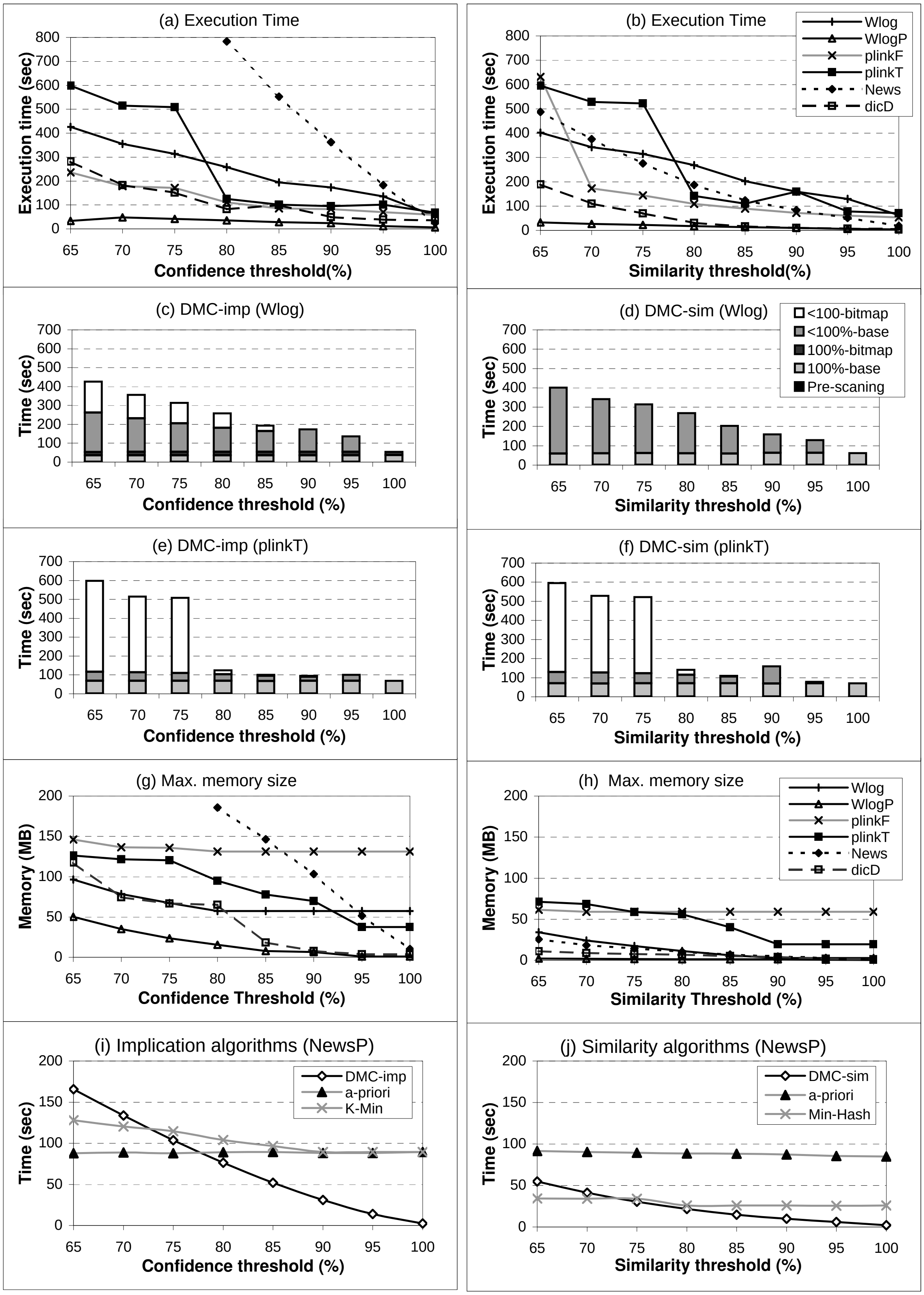


Figure 6. Experimental results

polgar -> international
polgar -> old
polgar -> judit
polgar -> champion
polgar -> youngest
polgar -> chess
polgar -> kasparov
polgar -> men
polgar -> highest
polgar -> top
polgar -> soviet
polgar -> players
polgar -> federation
polgar -> player
polgar -> ranked
polgar -> grandmaster
polgar -> garri

judit -> soviet
judit -> hungary

garri -> chess
garri -> kasparov
garri -> soviet
garri -> championship
garri -> champion

grandmaster -> soviet
grandmaster -> champion
grandmaster -> chess

kasparov -> chess
kasparov -> game
kasparov -> champion

Figure 7. Sample rules

could extract complete sets of true rules.

In this experiment, *a-priori* is best for finding implication rules with 75%-or-less confidence threshold, and *Min-Hash* is best for finding similarity rules with 70%-or-less similarity threshold, respectively. However, the DMC algorithms are best for finding both implication and similarity rules with high threshold.

6.3 Extracted rules

Text-mining by using extracted implication rules with low-support pruning is one of the interesting applications for our algorithms. Fig. 7 shows sample rules we extracted from *News* with an 85% confidence threshold and with a support pruning less than 5. These rules are extracted by selecting all rules related to keyword *Polgar* and its successors, recursively. This set of rules indicates information about Miss Judit Polgar, who is 12-years-old, has been ranked No. 1 in the women's world chess.

7 Conclusion and future works

We presented two new algorithms, *DMC-imp* and *DMC-sim*, for finding implication and similarity rules. Our algorithms do not use support pruning but use confidence or similarity pruning, which reduces the memory size significantly. We also proposed the other pruning techniques, *row re-ordering*, *100%-rule pruning*, *column-density pruning*, and *maximum-hits pruning*.

In order to evaluate the performance of the algorithms, we used 4 sets of data, Web-access logs, Web-page-link graph, news documents, and a dictionary. The algorithms have been implemented on a Sun Ultra 2 (2×200MHz, 256MB memory) workstation.

According to the experimental results, our algorithms can be executed in a reasonable time. The algorithms that proposed previously can not execute on our data sets, since the algorithms required more than 256MB memory. Therefore, we compared performance by using the *News* data sets by pruning by using support threshold 35 so that all counters for a-priori can fit in main memory. The comparison results shows that *DMC-imp* can execute 1.7 times faster than *a-priori*, and 1.9 times faster than *K-Min*, and that *DMC-sim* can execute 5.9 times faster than *a-priori*, and 1.7 times faster than *Min-Hash*, in case of an 85% threshold.

The followings are future research topics:

- Our algorithm can not extract rules among more than two columns, while *a-priori* can do so. However, by grouping similarity and implication rules as showed in Sec. 6.3, we can get useful groups of rules among more than two columns. This idea can be applied to other data sets, which generates more interesting rules.

- The memory requirement for *News* with less than 80% confidence threshold exceeds 256MB. To be scalable this algorithm, a parallel algorithm based on a divide-and-conquer technique, such as FDM [6] for *a-priori*, is necessary.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining Association Rules Between Sets of Items in Large Databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data,* 1993, pp. 207–216.

[2] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In *Proceedings of the 20th International Conference on Very Large Databases,* 1994.

[3] R. J. Bayardo Jr, R. Agrawal and D. Gunopulos. Constraint-Based Rule Mining in Large, Dense Databases. In *Proceedings of the 15th International Conference on Data Engineering,* 1999, pp. 188–197.

[4] S. Brin, R. Motwani, J.D. Ullman, and S. Tsur, Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the ACM SIGMOD Conference on Management of Data,* 1997, pp. 255–264.

[5] A. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences (SEQUENCES'97)*, 1998, pp. 21–29.

[6] D. W. Cheung, J. Han, V. T. Ng, et al. A Fast Distributed Algorithm for Mining Association Rules. In *Proceedings of Conference on Parallel and Distributed Information Systems,* 1996, pp. 31–42

[7] E. Cohen. Size-Estimation Framework with Applications to Transitive Closure and Reachability. *Journal of Computer and System Sciences* 55 (1997): 441–453.

[8] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, et al. Finding Interesting Associations without Support Pruning. In *Proceedings of the 16th International Conference on Data Engineering*, 2000.

[9] R.O. Duda and P.E. Hart. *Pattern Classification and Scene Analysis*. A Wiley-Interscience Publication, New York, 1973.

[10] A. Gionis, P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th International Conference on Very Large Databases,* 1999.

[11] D. Goldberg, D. Nichols, B.M. Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM* 55 (1991): 1–19.

[12] S. Guha, R. Rastogi, and K. Shim. CURE - An Efficient Clustering Algorithm for Large Databases In *Proceedings of the ACM-SIGMOD International Conference on Management of Data,* 1998, pp. 73–84.

[13] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[14] J.S. Park, M.S. Chen, and P.S. Yu. An effective hash-based algorithm for mining association rules. In *Proceedings of the ACM SIGMOD Conference on Management of Data,* 1995, pp. 175–186.

[15] Project Gutenberg. *http:..www.gutenberg.net*, 1999

[16] N. Shivakumar and H. Garcia-Molina. Building a Scalable and Accurate Copy Detection Mechanism. In *Proceedings of the 3rd International Conference on the Theory and Practice of Digital Libraries*, 1996.

[17] H.R. Varian and P. Resnick, Eds. CACM Special Issue on Recommender Systems. *Communications of the ACM* 40 (1997).