Frequent Closed Item Set Mining Based on Zero-suppressed BDDs

Shin-ichi Minato
Graduate School of Information Science and Technology,
Hokkaido University, Sapporo, 060-0814 Japan.
minato@ist.hokudai.ac.jp

Hiroki Arimura
(affiliation as previous author)
arim@ist.hokudai.ac.jp

Keywords: data mining, item set, BDD, ZBDD, closed pattern

Summary

Frequent item set mining is one of the fundamental techniques for knowledge discovery and data mining. In the last decade, a number of efficient algorithms for frequent item set mining have been presented, but most of them focused on just enumerating the item set patterns which satisfy the given conditions, and it was a different matter how to store and index the result of patterns for efficient data analysis. Recently, we proposed a fast algorithm of extracting all frequent item set patterns from transaction databases and simultaneously indexing the result of huge patterns using Zero-suppressed BDDs (ZBDDs). That method, ZBDD-growth, is not only enumerating/listing the patterns efficiently, but also indexing the output data compactly on the memory to be analyzed with various algebraic operations. In this paper, we present a variation of ZBDD-growth algorithm to generate frequent closed item sets. This is a quite simple modification of ZBDD-growth, and additional computation cost is relatively small compared with the original algorithm for generating all patterns. Our method can conveniently be utilized in the environment of ZBDD-based pattern indexing.

1. Introduction

Frequent item set mining is one of the fundamental techniques for knowledge discovery and data mining. Since the introduction by Agrawal et al.[Agrawal 93], the frequent item set mining and association rule analysis have been received much attention from many researchers, and a number of papers have been published about the new algorithms or improvements for solving such mining problems[Goethals 03a, Han 04, Zaki 00]. However, most of such item set mining algorithms focused on just enumerating or listing the item set patterns which satisfy the given conditions and it was a different matter how to store and index the result of patterns for efficient data analysis.

Recently, we proposed a fast algorithm[Minato 06] of extracting all frequent item set patterns from transaction databases, and simultaneously indexing the result of huge patterns on the computer memory using Zero-suppressed BDDs. That method, called ZBDD-growth, does not only enumerate/list the patterns efficiently, but also indexes the output data compactly on the memory. After mining, the result of patterns can efficiently be analyzed by using algebraic operations.

The key of the method is to use BDD-based data structure for representing sets of patterns. BDDs[Bryant 86] are graph-based representation of Boolean functions, now widely used in VLSI logic design and verification area. For the data mining applications, it is important to use Zero-suppressed BDDs (ZBDDs)[Minato 93], a special type of BDDs, which are suitable for handling large-scale sets of combinations. Using ZBDDs, we can implicitly enumerate combinatorial item set data and efficiently compute set operations over the ZBDDs.

In this paper, we present an interesting variation of ZBDD-growth algorithm to generate frequent closed item sets. Closed item sets are the subset of item set patterns each of which is the unique representative for a group of sub-patterns relevant to the same set of transaction records. Our method is a quite simple modification of ZBDD-growth. We inserted several operations in the recursive procedure of ZBDD-
growth, to filter the closed patterns from all frequent patterns. The experimental result shows that the additional computation cost is relatively small compared with the original algorithm for generating all patterns. Our method can conveniently be utilized in the environment of ZBDD-based data mining and knowledge indexing.

2. ZBDD-based item set representation

As the preliminary section, we describe the methods for efficiently indexing item set data based on Zero-suppressed BDDs.

2.1 Combinatorial item set and ZBDDs

A combinatorial item set consists of the elements each of which is a combination of a number of items. There are $2^n$ combinations chosen from $n$ items, so we have $2^{2^n}$ variations of combinatorial item sets. For example, for a domain of five items $a, b, c, d,$ and $e,$ we can show examples of combinatorial item sets as: \{$ab, e\}, \{abc, cde, bd, acde, e\}, \{1, ad\}, 0.$ Here “1” denotes a combination of null items, and “0” means an empty set. Combinatorial item sets are one of the basic data structure for various problems in computer science, including data mining.

A combinatorial item set can be mapped into Boolean space of $n$ input variables. For example, Figure 1 shows a truth table of Boolean function: $F = (a b c) \lor (\overline{b} c),$ but also represents a combinatorial item set $S = \{ab, ac, c\}.$ Using BDDs for the corresponding Boolean functions, we can implicitly represent and manipulate combinatorial item set. In addition, we can enjoy more efficient manipulation using “Zero-suppressed BDDs” (ZBDD)[Minato 93], which are special type of BDDs optimized for handling combinatorial item sets. An example of ZBDD is shown in Figure 2.

The detailed techniques of ZBDD manipulation are described in the articles[Minato 93]. A typical ZBDD package supports cofactoring operations to traverse 0-edge or 1-edge, and binary operations between two combinatorial item sets, such as union, intersection, and difference. Our ZBDD package generates new ZBDD nodes in the main memory, as the results of these algebraic operations. The computation time for each operation is almost linear to the number of ZBDD nodes related to the operation. We can also delete a ZBDD which has become useless, and such garbage ZBDD nodes are efficiently collected to be used for another new ZBDD.
2-2 Tuple-Histograms and ZBDD vectors

A Tuple-histogram is the table for counting the number of appearance of each tuple in the given database. An example of tuple-histogram is shown in Figure 3. This is just a compressed table of the database to combine the same tuples appearing more than once into one line with the frequency.

Our item set mining algorithm manipulates ZBDD-based tuple-histogram representation as the internal data structure. Here we describe how to represent tuple-histograms using ZBDDs. Since ZBDDs are representation of sets of combinations, a simple ZBDD distinguishes only existence of each tuple in the database. In order to represent the numbers of tuple’s appearances, we decompose the number into m-digits of ZBDD vector \( \{ F_0, F_1, \ldots, F_{m-1} \} \) to represent integers up to \((2^m - 1)\), as shown in Figure 4. Namely, we encode the appearance numbers into binary digital code, as \( F_0 \) represents a set of tuples appearing odd times (LSB = 1), \( F_1 \) represents a set of tuples whose appearance number’s second lowest bit is 1, and similar way we define the set of each digit up to \( F_{m-1} \).

In the example of Figure 4, The tuple frequencies are decomposed as: \( F_0 = \{ abc, ab, c \} \), \( F_1 = \{ ab, bc \} \), \( F_2 = \{ abc \} \), and then each digit can be represented by a simple ZBDD. The three ZBDDs shares their sub-graphs each other.

Now we explain the procedure for constructing a ZBDD-based tuple-histogram from original database. We read a tuple data one by one from the database, and accumulate the single tuple data to the histogram. More concretely, we generate a ZBDD of \( T \) for a single tuple picked up from the database, and accumulate it to the ZBDD vector. The ZBDD of \( T \) can be obtained by starting from “1” (a null-combination), and applying “Change” operations several times to join the items in the tuple. Next, we compare \( T \) and \( F_0 \), and if they have no common parts, we just add \( T \) to \( F_0 \). If \( F_0 \) already contains \( T \), we eliminate \( T \) from \( F_0 \) and carry up \( T \) to \( F_1 \). This ripple carry procedure continues until \( T \) and \( F_k \) have no common part. After finishing accumulations for all data records, the tuple-histogram is completed.

Using the notation \( F_.add(T) \) for addition of a tuple \( T \) to the ZBDD vector \( F \), we describe the procedure of generating tuple-histogram \( H \) for given database \( D \).

\[
H = 0 \\
\forall T \in D \text{ do } H = H.add(T) \\
\text{return } H
\]

When we construct a ZBDD vector of tuple-histogram, the number of ZBDD nodes in each digit is bounded by total appearance of items in all tuples. If there are many partially similar tuples in the database, the sub-graphs of ZBDDs are shared very well, and compact representation is obtained. The bit-width of ZBDD vector is bounded by \( \log S_{\max} \), where \( S_{\max} \) is the appearance of most frequent items.

Once we have generated a ZBDD vector for the tuple-histogram, various operations can be executed efficiently. Here are the instances of operations used in our pattern mining algorithm.

- \( H.factor0(v) \): Extracts sub-histogram of tuples without item \( v \).
- \( H.factor1(v) \): Extracts sub-histogram of tuples including item \( v \) and then delete \( v \) from the tuple combinations. (also considered as the quotient of \( H/v \))
- \( v \cdot H \): Attaches an item \( v \) on each tuple combinations in the histogram \( F \).
- \( H_1 + H_2 \): Generates a new tuple-histogram with sum of the frequencies of corresponding tuples.
- \( H\text{.tuplecount} \): The number of tuples appearing at least once.

These operations can be composed as a sequence of ZBDD operations. The result is also compactly represented by a ZBDD vector. The computation time is bounded by roughly linear to total ZBDD sizes.

3. ZBDD-growth Algorithm

Recently, we developed a ZBDD-based algorithm [Minato 06], ZBDD-growth, to generate “all” frequent item set patterns. Here we describe this algorithm as the basis of our method for “closed” item set mining.

ZBDD-growth is based on a recursive depth-first search over the ZBDD-based tuple-histogram representation. The basic algorithm is shown in Figure 5, where \( H \) is a ZBDD for the tuple-histogram of the given database, and \( \alpha \) is a given number of minimum support threshold.

In this algorithm, we choose an item \( v \) used in \( H \), and compute the two sub-histograms \( H_1 \) and \( H_0 \). (Namely, \( H = (v \cdot H_1) \cup H_0 \)) Since we always choose \( v \) at the highest position in the ZBDD vector, \( H_1 \) and \( H_0 \) can be obtained just by referring the 1-edge and 0-edge of the highest ZBDD-node, and the computation time for factoring each digit of ZBDD is a constant.
ZBDDgrowth$(H, \alpha)$
{
    if$(H$ has only one item $v$)
    \hspace{1em} if$(v$ appears more than $\alpha$)
    \hspace{2em} return $v$
    \hspace{1em} else return \texttt{"0"};
    F ← Cache$(H)$;
    if$(F$ exists) return $F$;
    $v$ ← $H$.top; /* Top item in $H$ */
    $H_1$ ← $H$.factor1$(v)$;
    $H_0$ ← $H$.factor0$(v)$;
    $F_1$ ← ZBDDgrowth$(H_1, \alpha)$;
    $F_0$ ← ZBDDgrowth$(H_0 + H_1, \alpha)$;
    $F$ ← $(v \cdot F_1) \cup F_0$;
    Cache$(H)$ ← $F$;
    return $F$;
}

Fig. 5 ZBDD-growth algorithm.

to the ZBDD size.

The number of appearance of $v$ in $H$ can be calculated by seeing the total number of records in $H_1$. This computation is done in a linear time to the total ZBDD size of $H_1$. The heaviest operation in ZBDD-growth algorithm is to generate ZBDD for $(H_0 + H_1)$. This requires union and intersection operations for each corresponding digit of $H_0$ and $H_1$, and the result of ZBDDs sometimes grows large depending on the data property. Anyway, the total computation time is linearly bounded by the size of ZBDDs related to the operations.

The algorithm consists of the two recursive calls, one of which computes the subset of patterns including $v$, and the other computes the patterns excluding $v$. The two subsets of patterns can be obtained as a pair of pointers to ZBDDs, and then the final result of ZBDD is computed. This procedure may require an exponential number of recursive calls, however, we prepare a hash-based cache, Cache$(H)$, to store the result of each recursive call. Each entry in the cache is formed as pair $(H, F)$, where $H$ is the pointer to the ZBDD vector for a given tuple-histogram, and $F$ is the pointer to the result of ZBDD. On each recursive call, we check the cache to see whether the same histogram $H$ has already appeared, and if so, we can avoid duplicate processing and return the pointer to $F$ directly. By using this technique, the computation time becomes almost linear to the final (may be largest) ZBDD size.

In our implementation, we use some simple techniques to prune the search space. For example, if $H_1$ and $H_0$ are equivalent, we may skip to compute $F_0$. For another case, we can stop the recursive calls if total frequencies in $H$ is no more than $\alpha$. There are some other elaborate pruning techniques, but they needs additional computation cost for checking the conditions, so sometimes effective but not always.

4. Frequent closed item set mining

In frequent item set mining, we sometimes faced with the problem that a huge number of frequent patterns are extracted and hard to find useful information. Closed item set mining is one of the techniques to filter important subset of patterns. In this section, we present a variation of ZBDD-growth algorithm to generate frequent closed item sets.

4.1 Closed item sets

Closed item sets are the subset of item set patterns each of which is the unique representative for a group of sub-patterns relevant to the same set of tuples. For more clear definition, we first define the common item set $Com(S_T)$ for the given set of tuples $S_T$, such that $Com(S_T)$ is the set of items commonly included in every tuple $T \in S_T$. Next, we define occurrence $Occ(D, X)$ for the given database $D$ and item set $X$, such that $Occ(D, X)$ is the subset of tuples in $D$, each of which includes $X$. Using these notations, if an item set $X$ satisfies $Com(Occ(D, X)) = X$, we call $X$ a closed item set in $D$.

For example, let us consider the database $D$ as shown in Figure 3. Here, all item set patterns with threshold $\alpha = 1$ is: $\{abc, ab, ac, a, bc, b, c\}$, but closed item sets are: $\{abc, ab, bc, b, c\}$. In this example, \textquote{ac} is eliminated from a closed pattern because $Occ(D, \{ac\}) = Occ(D, \{abc\})$.

In recent years, many researchers discuss the efficient algorithms for closed item set mining. One of the remarkable result is LCM algorithm[Uno 03] presented by Uno et. al. LCM is a depth-first search algorithm to extract closed item sets. It features that the computation time is bounded by linear to the output data length. Our ZBDD-based algorithm is also based on a depth-first search manner, so, it has similar properties as LCM. The major difference is in the data structure of output data. Our method generates ZBDDs for the set of closed patterns, ready to go for more flexible analysis using ZBDD operations.

312
sub-histograms \( H \) can use a ZBDD-based operation, called permit operation by Okuno et al.\[Okuno 98\].\(^{1}\) For checking the condition for closed patterns, we can use a ZBDD-based operation, called permit operation by Okuno et al.\[Okuno 98\].\(^{1}\) For checking the condition for closed patterns, we can use a ZBDD-based operation, called permit operation by Okuno et al.\[Okuno 98\].\(^{1}\)

\[ P\text{.permit}(Q) \]

\[
\begin{align*}
\text{if}(P = \text{"0"} \text{ or } Q = \text{"0"}) & \text{ return } \text{"0"} ; \\
\text{if}(P = Q) & \text{ return } P ; \\
\text{if}(P = \text{"1"}) & \text{ return } \text{"1"} ; \\
\text{if}(Q = \text{"1"}) & \\
\text{if}(P \text{ include } \text{"1"}) & \text{ return } \text{"1"} ; \\
\text{else} & \text{ return } \text{"0"} ; \\
R & \leftarrow \text{Cache}(P, Q) ; \\
\text{if}(R \text{ exists}) & \text{ return } R ; \\
v & \leftarrow \text{TopItem}(P, Q) ; /* \text{Top item in } P, Q */ \\
(P_0, P_1) & \leftarrow \text{factors of } P \text{ by } v ; \\
(Q_0, Q_1) & \leftarrow \text{factors of } Q \text{ by } v ; \\
R & \leftarrow (v \cdot P_1\text{.permit}(Q_1)) \\
& \quad \cup (P_0, \text{permit}(Q_0 \cup Q_1)) ; \\
\text{Cache}(P, Q) & \leftarrow R ; \\
\text{return } R ;
\end{align*}
\]

Fig. 6 Permit operation.

4.2 Eliminating non-closed patterns

Our method is a quite simple modification of ZBDD-growth shown in Figure 5. We inserted several operations in the recursive procedure of ZBDD-growth, to filter the closed patterns from all frequent patterns. The ZBDD-growth algorithm is starting from the given tuple-histogram \( H \), and computes the two sub-histograms \( H_1 \) and \( H_0 \), such that \( H = (v \cdot H_1) \cup H_0 \). Then ZBDD-growth\((H_1)\) and ZBDD-growth\((H_1 + H_0)\) is recursively executed.

Here, we consider the way to eliminate non-closed patterns in this algorithm. We call the new algorithm ZBDD-growth\(C(H)\). It is obvious that \((v \cdot \text{ZBDD-growth}(H_1))\) generates (a part of) closed patterns for \( H \) each of which includes \( v \), because the occurrence of any closed pattern with \( v \) is limited in \((v \cdot H_1)\), thus we may search only for \( H_1 \). Next, we consider the second recursive call ZBDD-growth\(C(H_1 + H_0)\) to generate the closed patterns without \( v \). Important point is that some of patterns generated by ZBDD-growth\(C(H_1 + H_0)\) may have the same occurrence as one of the patterns with \( v \) already found in \( H_1 \). The condition of such duplicate pattern is that it appears only in \( H_1 \) but irrelevant to \( H_0 \). In other words, we eliminate the patterns from ZBDD-growth\(C(H_1 + H_0)\) such that the patterns are already found in ZBDD-growth\(C(H_1)\) but not included in any tuples in \( H_0 \).

For checking the condition for closed patterns, we can use a ZBDD-based operation, called permit operation by Okuno et al.\[Okuno 98\].\(^{1}\) For checking the condition for closed patterns, we can use a ZBDD-based operation, called permit operation by Okuno et al.\[Okuno 98\].\(^{1}\)

\[ \text{ZBDD-growth}\(C(H, \alpha)\) \]

\[
\begin{align*}
\text{if}(H \text{ has only one item } v) & \\
\text{if}(v \text{ appears more than } \alpha) & \text{ return } v ; \\
\text{else return } \text{"0"} ; \\
F & \leftarrow \text{Cache}(H) ; \\
\text{if}(F \text{ exists}) & \text{ return } F ; \\
v & \leftarrow H\text{.top} ; /* \text{Top item in } H */ \\
H_1 & \leftarrow H\text{.factor}\(1(v)\) ; \\
H_0 & \leftarrow H\text{.factor}\(0(v)\) ; \\
F_1 & \leftarrow \text{ZBDD-growth}\(C(H_1, \alpha)\) ; \\
F_0 & \leftarrow \text{ZBDD-growth}\(C(H_0 + H_1, \alpha)\) ; \\
F & \leftarrow (v \cdot F_1) \cup (F_0 - (F_1 - F_1\text{.permit}(H_0))) ; \\
\text{Cache}(H) & \leftarrow F ; \\
\text{return } F ;
\end{align*}
\]

Fig. 7 ZBDD-growth\(C\) algorithm.

5. Experimental Results

Here we show the experimental results to evaluate our new method. We used a Pentium-4 PC, 800MHz, 1.5GB of main memory, with SuSE Linux 9. We can deal with up to 40,000,000 nodes of ZBDDs in this machine.

Table 1 shows the time and space for generating ZBDD vectors of tuple-histograms for the FIMI2003 benchmark databases[Goethals 03b]. This table shows the computation time and space for providing input data for ZBDD-growth algorithm. In this table, \#T shows the number of tuples, \textit{total}[T] is the total of tuple sizes (total appearances of items), and \(|\text{ZBDD}|\) is the number of ZBDD nodes for the tuple-histograms. We can see that tuple-histograms can be constructed

\(^{1}\) Permit operation is basically the same as \textit{SubSet} operation.
Table 1: Generation of tuple-histograms [Minato 06].

<table>
<thead>
<tr>
<th>Data name</th>
<th>#I</th>
<th>total[T]</th>
<th>ZBDD Vector</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T10I4D100K</td>
<td>100,000</td>
<td>1,010,228</td>
<td>552,429</td>
<td>55.6</td>
</tr>
<tr>
<td>T40I10D100K</td>
<td>100,000</td>
<td>3,960,507</td>
<td>1,750,883</td>
<td>167.10</td>
</tr>
<tr>
<td>chess</td>
<td>3,196</td>
<td>118,252</td>
<td>40,028</td>
<td>0.98</td>
</tr>
<tr>
<td>mushroom</td>
<td>8,124</td>
<td>186,852</td>
<td>8,006</td>
<td>1.09</td>
</tr>
<tr>
<td>pumsb</td>
<td>49,046</td>
<td>3,629,404</td>
<td>1,324,502</td>
<td>125.17</td>
</tr>
<tr>
<td>chess-star</td>
<td>515,597</td>
<td>3,367,020</td>
<td>1,350,970</td>
<td>1,317.07</td>
</tr>
<tr>
<td>BMS-POS</td>
<td>59,602</td>
<td>149,639</td>
<td>46,148</td>
<td>19.55</td>
</tr>
<tr>
<td>BMS-WebView-1</td>
<td>77,512</td>
<td>358,278</td>
<td>198,471</td>
<td>184.83</td>
</tr>
<tr>
<td>accidents</td>
<td>340,183</td>
<td>11,500,870</td>
<td>3,877,333</td>
<td>128.47</td>
</tr>
</tbody>
</table>

Table 2: Result of the original ZBDD-growth [Minato 06].

<table>
<thead>
<tr>
<th>Data name:</th>
<th>#Frequent patterns</th>
<th>(output) ZBDD</th>
<th>Time(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mushroom: 5,000</td>
<td>123,277</td>
<td>1.06</td>
<td></td>
</tr>
<tr>
<td>T10I4D100K: 5,000</td>
<td>385</td>
<td>5.81</td>
<td></td>
</tr>
<tr>
<td>BMS-WebView-1: 1,000</td>
<td>359,067</td>
<td>128.47</td>
<td></td>
</tr>
</tbody>
</table>

For all instances in a feasible time and space. The ZBDD sizes are almost same or less than total[T].

After generating ZBDD vectors for the tuple-histograms, we applied ZBDD-growth algorithm to generate frequent patterns. Table 2 shows the results of the original ZBDD-growth algorithm [Minato 06] for the selected benchmark examples, "mushroom," "T10I4D100K," and "BMS-WebView-1." The execution time does not include the time for generating the initial ZBDD vectors for tuple-histograms.

The results shows that the ZBDD size is exponentially smaller than the number of patterns for "mushroom." This is a significant effect of using the ZBDD data structure. On the other hand, no remarkable reduction is seen in "T10I4D100K," "T10I4D100K" is known as an artificial database, consists of randomly generated combinations, so there are almost no relationship between the tuples. In such cases, ZBDD nodes cannot be shared well, and only the overhead factor is revealed. For the third example, "BMS-WebView-1," the ZBDD size is almost linear to the number of patterns when the output size is small, however, an exponential factor of reduction is observed for the cases of generating huge patterns.

Next, we show the experimental results of frequent closed pattern mining using ZBDD-growthC algorithm. In Table 3, we show the results for the same examples as used in the experiment of the original ZBDD-growth. The last column Time_{closed}/Time_{all} shows the ratio of computation time between the ZBDD-growthC and the original ZBDD-growth algorithm. We can observe that the computation time is almost the same order as the original algorithms for "mushroom" and "BMS-WebView-1," but some additional factor is observed for "T10I4D100K." Anyway, filtering closed item sets has been regarded as not an easy task. We can say that the ZBDD-growthC algorithm can generate closed item sets with a relatively small additional cost from the original ZBDD-growth.

Finally, we compared our results with a state-of-the-art implementation of the LCM algorithm [Uno 03] on the same PC. The results are shown in the rightmost column of Table 3. We can observe that the LCM-based program is more than a hundred times faster than our ZBDD-based program. The LCM algorithm features that the computation time is lin-
early bounded by the total size of closed item sets. In addition, the implementation is highly optimized for enumerating the number of closed item sets without printing out. On the other hand, ZBDD-based method is especially effective when a huge number of item sets produced and they are compactly represented by small size of ZBDDs. In general, closed item sets are already reduced representation of all item sets, and in such cases, ZBDD-based compression is not very effective.

If our final goal is only enumerating closed item sets, the LCM algorithm would be much better. However, our ZBDD-based method does not only enumerate but also constructs efficient data structures for indexing the result of item sets, and they can be used for various data analysis with ZBDD-based algebraic set operations. Here we show several examples of useful post-processing.

(Extracting all non-closed patterns): After executing ZBDD-growth and -growthC, we can easily obtain a set of non-closed patterns by applying a difference operation between the two ZBDDs of all item sets and closed item sets.

(Filtering closed patterns with sub-patterns): By using “factor1(ν)” operation, we can filter the subset of closed patterns including an item ν. Repeating this operations, “key-word filtering” of closed patterns can be executed.

(Filtering closed patterns by a permissible set): When a set of permissible patterns are given as a ZBDD, we can filter the closed patterns each of which satisfies the constraint given by the ZBDD. For example, we first generate a ZBDD representing a set of patterns each of which contains exactly three items. Next we generate all closed patterns by ZBDD-growthC. Then, an intersection operation between the two ZBDDs extracts all closed patterns each of which consists of three items.

### 6. Conclusion

In this paper, we presented a variation of ZBDD-growth algorithm to generate frequent closed item sets. Our method is a quite simple modification of ZBDD-growth. We inserted several operations in the recursive procedure of ZBDD-growth, to filter the closed patterns from all frequent patterns. The experimental result shows that the additional computation cost is relatively small compared with the original algorithm for generating all patterns.

In order to just enumerate the closed patterns, our method is not faster than LCM algorithm, which is an existing state-of-the-art method. The main reason is that the closed patterns are already a kind of reduced information of the frequent patterns. In such cases, the ZBDD-based data compressing power is not very effective, and only the overhead factor reveals.

However, our method is still useful because it does not only enumerate but also constructs an efficient indexing data structure for all set of closed patterns, and we can apply various data analysis tasks using the ZBDD-based algebraic operations. In addition, it
will be an interesting future work to deeply combine the techniques of ZBDD manipulation and the LCM algorithm.

Acknowledgments
This research was partially supported by Grant-in-Aid for Specially Promoted Research on “Semi-Structured Data Mining,” 17002008, Ministry of Education, Culture, Sports, Science and Technology of Japan.

References


Received August 15, 2006.

Author’s Profile

Minato, Shin-ichi (Member)
He received the B.E., M.E., and D.E. degrees in Information Science from Kyotou University in 1988, 1990, and 1995, respectively. From 1990 to 2004, he was a researcher of NTT Laboratories, and he was a Visiting Scholar at Stanford University in 1997. Since 2004, he has been an associate professor of Hokkaido University. His research interests include data structures and algorithms for manipulating large-scale logic data. He published “Binary Decision Diagrams and Applications for VLSI CAD” (Kluwer, 1995). He is a member of IEEE, IEICE, and IPSJ.

Arimura, Hiroki (Member)
He received the B.S. degree in 1988 in Physics, the M.S. and the Dr.Sc. degrees in 1990 and 1994 in Information Systems from Kyushu University. From 1990 to 1996, he was at the Department of Artificial Intelligence in Kyushu Institute of Technology, and from 1996 to 2004, he was at the Department of Informatics in Kyushu University. Since 2006, he has been a professor of the Graduate School of Information Science and Technology Hokkaido University, Sapporo, Japan. His research interests include data mining, computational learning theory, information retrieval, and algorithm design. He is a member of JSAI, IPSJ, and ACM.