TCS-TR-A-**06-17**

TCS Technical Report

Generating Frequent Closed Item Sets Based on Zero-suppressed BDDs

by

Shin-ichi Minato

Division of Computer Science

Report Series A

July 3, 2006

Hokkaido University
Graduate School of
Information Science and Technology

Email: minato@ist.hokudai.ac.jp

Phone: +81-011-706-7682

Fax: +81-011-706-7682

Generating Frequent Closed Item Sets Based on Zero-suppressed BDDs

Shin-ichi Minato
Division of Computer Science, Hokkaido University
North 14, West 9, Sapporo, 060-0814 Japan

July 3, 2006

(Abstract) 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.[1], the frequent item set mining and association rule analysis have been received much attentions from many researchers, and a number of papers have been published about the new algorithms or improvements for solving such mining problems[4, 6, 11]. 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[8] 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[2] 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)[7], 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 re-

a	b	c	F	
0	0	0	0	As a Boolean function:
1	0	0	0	F(a,b,c) = (a b ~c) V (~b c)
0	1	0	0	As a combinatorial item set:
1	1	0	1	→ ab
0	0	1	1	→ c
1	0	1	1	→ ac
0	1	1	0	
1	1	1	0	

Figure 1: A Boolean function and a combinatorial item set.

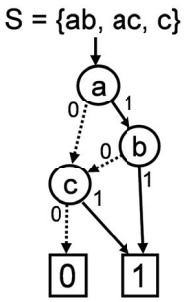

Figure 2: An example of ZBDD.

Record ID	Tuple
1	a b c
2	a b
3	a b c
4	b c
5	a b
6	a b c
7	c
8	a b c
9	a b c
10	a b
11	b c

Original database

Tuple	Freq.
a b c	5
a b	3
b c	2
c	1

Tuple-histogram

Figure 3: Example of tuple-histogram.

sult 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:
$\{ab, e\}$, $\{abc, cde, bd, acde, e\}$, $\{1, cd\}$, 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, Fig. 1 shows a truth table of Boolean function: $F = (a\ b\ \overline{c}) \vee (\overline{b}\ c)$, 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)[7], which are special type of BDDs optimized for handling combinatorial item sets. An example of ZBDD is shown in Fig. 2.

The detailed techniques of ZBDD manipulation are described in the articles[7]. 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. The computation time for each operation is almost linear to the number of ZBDD nodes related to the operation.

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 Fig. 3. This is just a compressed table of the

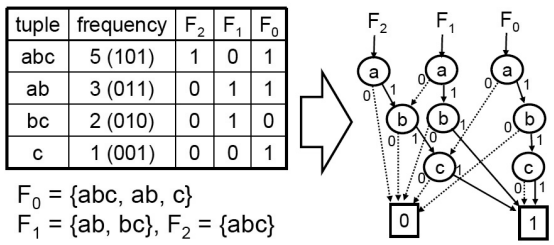

Figure 4: ZBDD vector for tuple-histogram.

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 Fig. 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 Fig. 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 are shared 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.

$H = \mathbf{0}$
forall $T \in D$ **do**
  $H = H$.add(T)
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.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.

```
ZBDDgrowth(H, α)
{
  if(H has only one item v)
    if(v appears more than α )
      return v ;
    else return "0" ;
  F ← Cache(H) ;
  if(F exists) return F ;
  v ← H.top ; /* Top item in H */
  H1 ← H.factor1(v) ;
  H0 ← H.factor0(v) ;
  F1 ←ZBDDgrowth(H1, α) ;
  F0 ←ZBDDgrowth(H0 + H1, α) ;
  F ← (v · F1) ∪ F0 ;
  Cache(H) ← F ;
  return F ;
}
```

Figure 5: ZBDD-growth algorithm.

3 ZBDD-growth Algorithm

Recently, we developed a ZBDD-based algorithm[8], 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 Fig. 5.

In this algorithm, we choose an item v used in the tuple-histogram H, and compute the two sub-histograms H_1 and H_0. (Namely, $H = (v \cdot H_1) \cup H_0$.) As v is the top item 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, so the computation time is constant for each digit of ZBDD.

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 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 total ZBDD sizes.

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 α. 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 *occurence* $Occ(D, X)$ 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 is a closed item set in D.

For example, let us consider the database D as shown in Fig. 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, "ac" is eliminated from a closed pattern because $Occ(D, \text{“}ac\text{”}) = Occ(D, \text{“}abc\text{”})$.

In recent years, many researchers discuss the efficient algorithms for closed item set mining. One of the remarkable result is *LCM* algorithm[10] 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.

4.2 Eliminating non-closed patterns

Our method is a quite simple modification of ZBDD-growth shown in Fig. 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 staring from the given tuple-histogram H, and compute 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-growthC(H). It is obvious that ($v\cdot$ ZBDD-growthC(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-growthC($H_1 + H_0$) to generate the closed patterns without v. Important point is that some of patterns generated by ZBDD-growthC($H_1 + H_0$) may have the same occurrence as one of the pattern 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-growthC($H_1 + H_0$) such that the patterns are already found in ZBDD-growthC(H_1) but not included in any tuples in H_0.

```
P.permit(Q)
{
  if(P ="0" or Q ="0") return "0" ;
  if(P = Q) return F ;
  if(P ="1") return "1" ;
  if(Q ="1")
    if(P include "1" ) return "1" ;
    else return "0" ;
  R ← Cache(P, Q) ;
  if(R exists) return R ;
  v ←TopItem(P, Q) ; /* Top item in P, Q */
  (P0, P1) ←factors of P by v ;
  (Q0, Q1) ←factors of Q by v ;
  R ← (v · P1.permit(Q1))
       ∪ (P0.permit(Q0 ∪ Q1)) ;
  Cache(P, Q) ← R ;
  return R ;
}
```

Figure 6: Permit operation.

For checking the condition for closed patterns, we can use a ZBDD-based operation, called *permit* operation by Okuno et al.[9].[1] P.permit(Q) returns a set of combinations in P each of which is a subset of some combinations in Q. For example, when $P = \{ab, abc, bcd\}$ and $Q = \{abc, bc\}$, then P.permit(Q) returns $\{ab, abc\}$. The permit operation is efficiently implemented as a recursive procedure of ZBDD manipulation, as shown in Fig. 6. The computation time of permit operation is almost linear to the ZBDD size.

Finally, we describe the ZBDD-growthC algorithm using the permit operation, as shown in Fig. 7. The difference from the original algorithm is only one line, written in the frame box.

5 Experimental Results

Here we show the experimental results to evaluate our new method. We used a Pentium-4 PC,

[1] Permit operation is basically same as *SubSet* operation by Coudert et al.[3], defined for ordinary BDDs.

Table 1: Generation of tuple-histograms[8].

Data name	$\#T$	$total\|T\|$	|ZBDD Vector|	Time(s)
T10I4D100K	100,000	1,010,228	552,429	43.2
T40I10D100K	100,000	3,960,507	3,396,395	150.2
chess	3,196	118,252	40,028	1.4
connect	67,557	2,904,951	309,075	58.1
mushroom	8,124	186,852	8,006	1.2
pumsb	49,046	3,629,404	1,750,883	188.5
pumsb_star	49,046	2,475,947	1,324,502	123.6
BMS-POS	515,597	3,367,020	1,350,970	895.0
BMS-WebView-1	59,602	149,639	46,148	18.3
BMS-WebView-2	77,512	358,278	198,471	138.0
accidents	340,183	11,500,870	3,877,333	107.0

Table 2: Result of the original ZBDD-growth[8].

Data name: Min. freq. α	#Frequent patterns	(output) |ZBDD|	Time(sec)
mushroom: 5,000	41	11	1.2
1,000	123,277	1,417	3.7
200	18,094,821	12,340	9.7
50	198,169,865	36,652	10.2
16	1,176,182,553	53,804	7.7
4	3,786,792,695	59,970	4.3
1	5,574,930,437	40,557	1.8
T10I4D100K: 5,000	10	10	81.3
1,000	385	382	135.5
200	13,255	4,288	279.4
50	53,385	20,364	408.7
16	175,915	89,423	543.3
4	3,159,067	1,108,723	646.0
BMS-WebView1: 1,000	31	31	27.8
200	372	309	31.3
50	8,191	3,753	49.0
40	48,543	12,176	46.6
36	461,521	34,790	102.4
35	1,177,607	47,457	111.4
34	4,849,465	64,601	120.8
33	69,417,073	80,604	130.0
32	1,531,980,297	97,692	133.7
31	8,796,564,756,112	117,101	138.1
30	35,349,566,550,691	152,431	143.9

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[5]. 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, $total|T|$ is the total of tuple sizes (total appearances of items), and |ZBDD| is the number of ZBDD nodes for the tuple-histograms. We can see that tuple-histograms can be constructed 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 show the results of the original ZBDD-growth algorithm[8] for the selected benchmark examples, "mushroom," "T10I4D100K," and "BMS-WebView-1." The execution time includes 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,

```
ZBDDgrowthC(H, α)
{
  if(H has only one item v)
    if(v appears more than α )
      return v ;
    else return "0" ;
  F ← Cache(H) ;
  if(F exists) return F ;
  v ← H.top ; /* Top item in H */
  H1 ← H.factor1(v) ;
  H0 ← H.factor0(v) ;
  F1 ←ZBDDgrowthC(H1, α) ;
  F0 ←ZBDDgrowthC(H0 + H1, α) ;
  F ← (v · F1) ∪
      (F0 − (F1 − F1.permit(H0))) ;
  Cache(H) ← F ;
  return F ;
}
```

Figure 7: ZBDD-growthC algorithm.

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 a 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.

6 Conclusion

In this paper, we presented an interesting 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.

A ZBDD can be regarded as a compressed trie for representing a set of patterns. ZBDD-based method will be useful as a fundamental technique for database analysis and knowledge indexing, and will be utilized for various data mining applications.

References

[1] R. Agrawal, T. Imielinski, and A. N. Swami, Mining Association rules between sets of items in large databases, In P. Buneman and S. Jajodia, edtors, *Proc. of the 1993 ACM SIGMOD International Conference on Management of Data*, Vol. 22(2) of SIGMOD Record, pp. 207–216, ACM Press, 1993.

[2] Bryant, R. E., Graph-based algorithms for Boolean function manipulation, IEEE Trans. Comput., C-35, 8 (1986), 677–691.

[3] O. Coudert, J. C. Madre, H. Fraisse, A new viewpoint on two-level logic minimization, *in Proc. of 30th ACM/IEEE Design Automation Conference*, pp. 625-630, 1993.

[4] B. Goethals, "Survey on Frequent Pattern Mining", Manuscript, 2003. `http://www.cs.helsinki.fi/u/goethals/publications/survey.ps`

[5] B. Goethals, M. Javeed Zaki (Eds.), Frequent Itemset Mining Dataset Repository, Frequent Itemset Mining Implementations (FIMI'03), 2003.
`http://fimi.cs.helsinki.fi/data/`

Table 3: Results of ZBDD-based closed pattern mining.

Data name: Min. freq. α	#Freq. closed patterns	(output) |ZBDD|	ZBDD-growthC Time(s)	$Time_{(closed)}$ $/Time_{(all)}$
mushroom:				
5,000	16	16	1.2	1.00
1,000	3,427	1,660	3.8	1.02
200	26,968	9,826	9.9	1.02
50	68,468	19,054	13.0	1.27
16	124,411	24,841	13.3	1.73
4	203,882	26,325	13.2	3.06
1	238,709	20,392	12.9	7.19
T10I4D100K:				
5,000	10	10	104.8	1.29
1,000	385	382	208.1	1.54
200	13,108	4,312	2713.6	9.71
50	46,993	20,581	4600.1	11.25
16	142,520	89,185	5798.5	10.67
4	1,023,614	691,154	18573.0	28.75
BMS-WebView-1:				
1,000	31	31	30.1	1.08
200	372	309	36.8	1.18
50	7,811	3,796	71.9	1.47
40	29,489	11,748	111.4	2.39
36	64,762	25,117	153.7	1.50
35	76,260	30,011	169.2	1.52
34	87,982	35,392	186.5	1.54
33	99,696	40,915	207.7	1.60
32	110,800	46,424	221.7	1.66
31	120,190	51,369	247.7	1.79
30	127,131	55,407	271.5	1.89

[6] J. Han, J. Pei, Y. Yin, R. Mao, Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach, Data Mining and Knowledge Discovery, 8(1), 53–87, 2004.

[7] S. Minato: Zero-suppressed BDDs for set manipulation in combinatorial problems, In Proc. 30th ACM/IEEE Design Automation Conf. (DAC-93), (1993), 272–277.

[8] S. Minato, H. Arimura: ZBDD-growth: An Efficient Method for Frequent Pattern Mining and Knowledge Indexing, *Hokkaido University, Division of Computer Science, TCS Technical Reports*, TCS-TR-A-06-12, Apr. 2006.
`http://www-alg.ist.hokudai.ac.jp/tra.html`

[9] H. Okuno, S. Minato, and H. Isozaki: On the Properties of Combination Set Operations, Information Procssing Letters, Elsevier, 66 (1998), pp. 195-199, 1998.

[10] T. Uno, Y. Uchida, T. Asai, and H. Arimura: "LCM: An Efficient Algorithm for Enumerating Frequent Closed Item Sets," Proc. *Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, Mohammed J. Zaki and Bart Goethals (eds.), 2003.
`http://fimi.cs.helsinki.fi/fimi03/`

[11] M. J. Zaki, Scalable Algorithms for Association Mining, IEEE Trans. Knowl. Data Eng. 12(2), 372–390, 2000.