TCS-TR-A-13-62

TCS Technical Report

Fast Construction of ZDDs from Large-scale Hypergraphs

by

Takahisa Toda

Division of Computer Science

Report Series A

April 11, 2013

Hokkaido University
Graduate School of
Information Science and Technology

Email: toda@ist.hokudai.ac.jp

Phone: +81-011-706-7682
Fax: +81-011-706-7682

Fast Construction of ZDDs from Large-scale Hypergraphs

Takahisa Toda
JST ERATO Minato Project
Graduate School of Info. Sci. and Tech.
Hokkaido University
Sapporo 060-0814, Japan

April 11, 2013

(Abstract) We present an algorithm to compress hypergraphs into the data structure ZDDs and analyze the computational complexity. Since a ZDD provides an approach to solve large-scale problems that are difficult to compute in a reasonable amount of time and space, it is important to compress hypergraphs efficiently. Our algorithm uses multikey Quicksort given by Bentley and Sedgewick. By conducting experiments with various datasets, we show that our algorithm is significantly faster and requires smaller memory than an existing method.

1 Introduction

A *hypergraph* is a generalization of a graph. While a graph describes an adjacency relation for two vertices, a hypergraph describes that for any multiple vertices. Thus a hypergraph is defined to be a pair $(V, \mathcal{E})$ of a set V and a family $\mathcal{E}$ of subsets of V, where the sets in $\mathcal{E}$ are called *hyperedges*. Since a hypergraph can represent a wide variety of information, there are many applications in computer science. Recently datasets which are considered as hypergraphs have become widely available. Since such datasets tend to be very large in size, an efficient method to handle them has become increasingly important.

A *zero-suppressed binary decision diagram* (ZDD) is a compressed data structure for hypergraphs [9]. Although compression efficiency generally depends on hypergraphs, a ZDD is designed to be effective for *sparse* hypergraphs, that is, a hypergraph whose hyperedge sizes tend to be much smaller than the size of a ground set. Sparse hypergraphs often appear in real-life, e.g. as transaction databases. An advantage of a ZDD is not only compression efficiency. The greatest feature is an ability to manipulate hypergraphs without explicitly decompressing them. Many operations including intersection, union and difference can be efficiently performed

on ZDDs. As demonstrated in [18],[3] etc, a computation method to construct a desired hypergraph with ZDD operations allows us to solve problems that are difficult to compute in a reasonable amount of time and space in a usual method. It is thus considered as a practically efficient method to handle large-scale hypergraphs.

ZDDs and similar data structure BDDs have been applied to various combinatorial problems. Sekine et al. [17] presented a BDD-based algorithm to compute the Tutte polynomial of a graph. This algorithm has many applications because many counting problems in graph theory and related areas are reduced to this computation. For example, it was applied to the exact computation of network reliability [16]. Coudert [3] advocated a ZDD-based approach to solve graph optimization problems by showing algorithms for three primitive problems to which many graph optimization problems are reducible. Based on this approach, Knuth presented an algorithm to compute all minimal hitting sets, as an exercise in his famous book [8]. The minimal hitting set computation is known to be equivalent to the dualization of monotone Boolean functions, and there are many applications in computer science, especially in data mining, logic, and artificial intelligence (see for example [4] [5]). Recently, practically fast algorithms for this problem have been developed by many researchers (see [12]). Toda [18] presented a BDD and ZDD-based algorithm for the problem, and experimentally showed that this algorithm is highly competitive with existing algorithms.

Recently much attention has been paid to the analysis of large-scale databases. Minato and Arimura [10] proposed a ZDD-based approach for database analysis problems. Generating frequent itemsets from databases plays an important role in *frequent itemset mining* (see [7]). While many generation algorithms have been proposed, Minato et al. [11] presented a combination of one of the most efficient state-of-the-art algorithms, called LCM, and ZDDs. This method is called LCM over ZDDs.

In this paper, we present a fast algorithm to compress hypergraphs into ZDDs. A naive compression method constructs a ZDD by repeating the union operation provided in a usual ZDD library (see for example [10][14]). While it is easy to implement, unfortunately there are some drawbacks on performance. Yet another method is used in the LCM over ZDDs mentioned above. This method constructs a ZDD in a bottom up fashion while receiving sets successively generated by LCM. Our algorithm has the same basic principle to LCM over ZDDs, while we further improve it in cooperation with multikey Quicksort, which is a sort algorithm for strings given by Bentley and Sedgewick [1]. We analyze the computational complexity of our algorithm. Furthermore, by conducting experimental comparison with various datasets, we show that our algorithm is significantly faster and requires smaller memory than the naive method based on the union operation.

This paper is organized as follows. In Section 2 we introduce the data structure ZDDs. In Section 3 we present an algorithm and analyze the computational complexity. Section 4 provides experimental results.

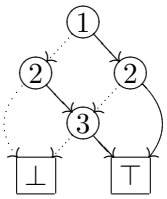
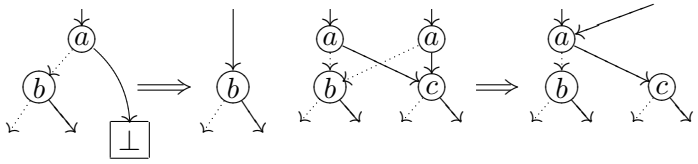


Figure 1: The ZDD for the set family $\{\{2,3\},\{1,3\},\{1,2\}\}$



(a) Node elimination

(b) Node sharing

Figure 2: Reduction rules on ZDDs

2 A Compressed Data Structure for Hypergraphs

Since we use a special data structure for hypergraphs, we provide necessary notions and results in this section. We identify hypergraphs with set families if the ground set is clear from the context.

2.1 Introduction to ZDDs

A *zero-suppressed binary decision diagram* (*ZDD*) is a graph-representation for hypergraphs. Figure 1 shows an example of ZDD. The node at the top is called the *root*. Each internal node has the three fields V, LO, HI. The field V holds an element in a ground set, where for simplicity we suppose that a ground set consists of positive numbers. The fields LO and HI point to other nodes, which are called LO and HI *children*, respectively. The arc to a LO child is called a LO *arc* and illustrated by a dashed arrow, while the arc to a HI child is called a HI *arc* and illustrated by a solid arrow. There are only two terminal nodes $\top$ and $\bot$.

For efficient compression, ZDDs satisfy the following two conditions. They must be *ordered*: if a node u points to an internal node v, then $\mathrm{V}(u) < \mathrm{V}(v)$. They must be *reduced*: the following two reduction operations can not be applied.

1. For each internal node u whose HI arc points to $\bot$, redirect all the incoming arcs of u to the LO child, and then eliminate u (Fig. 2(a)).
2. For any nodes u and v, if the subgraphs rooted by u and v are equivalent, then share the two subgraphs (Fig. 2(b)).

We can understand ZDDs as follows. Given a ZDD, each path from the root to $\top$ corresponds to a set in such a way that an element k is included in the set if a path contains the HI arc of a node with label k; otherwise, k is excluded. For example, in Fig. 1, the paths ① ⇢ ② → ③ → $\top$ and ① → ② → $\top$ correspond to $\{2,3\}$ and $\{1,2\}$, respectively. Note that although the node ③ does not appear in the latter path, the node elimination rule implies that the HI arc of ③ points to $\bot$, and thus 3 must be excluded.

It is known (see for example [9][8]) that for any ground set V, every hypergraph on V corresponds to a unique ZDD if the order of elements in V is fixed. ZDD nodes are maintained by a hash table, called a *uniquetable*, so that for a triple (k, l, h) of a node label and two ZDD nodes, there is a unique ZDD node p with $\mathrm{V}(p) = k$, $\mathrm{LO}(p) = l$, and $\mathrm{HI}(p) = h$. Given a triple (k, l, h), the function zdd_unique returns an associated node in the uniquetable if exists; otherwise, create a new node p such that $\mathrm{V}(p) = k$, $\mathrm{LO}(p) = l$, and $\mathrm{HI}(p) = h$; register p to the uniquetable and return p. A uniquetable guarantees that two nodes are different if and only if the subgraphs rooted by them represent different set families. Thus, for example, equivalence checking of set families can be done in constant time.

2.2 A Naive Construction Method

We describe a naive algorithm to construct the ZDD for a set family $\mathcal{E} := \{U_1, \ldots, U_m\}$ on a ground set V. For each set U_i, we construct the ZDD for the set family consisting only of U_i. We then add them by using the union operation. Here the union operation computes $Z(\mathcal{F} \cup \mathcal{G})$ for $Z(\mathcal{F})$ and $Z(\mathcal{G})$, where $Z(\mathcal{S})$ denotes the ZDD representing a set family $\mathcal{S}$. As argued in [18], this construction method requires $O(|\mathcal{E}| \cdot |V|)$ time.

Since the union operation is provided in a usual ZDD library, the naive method is easy to implement, however there are some drawbacks.

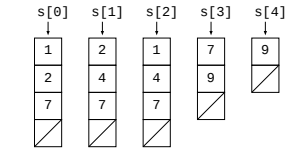
- A practical efficiency depends on the order to add sets.
- The worst-case time depends on the size of a ground set even if the size of a set tends to be much smaller than the size of a ground set.
- Many ZDD nodes created during the computation may not appear in an output ZDD.

3 Algorithm

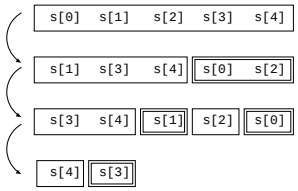
In this section, we first observe that ZDDs are isomorphic to processes of sorting hyperedges and then present our algorithm.

Figure 3 shows an example of such a correspondence. Suppose that we are given hypergraph $(V, \mathcal{E})$, where V consists of positive numbers. Each hyperedge $U \in \mathcal{E}$ is represented as the array of numbers in U, ordered in increasing order. The d-th entry of U always means the d-th largest number in U. For the sake of simplicity, we assume that every hyperedge contains $+\infty$. We sort all hyperedges in lexicographical order by recursively partitioning $\mathcal{E}$ into the equal part $\mathcal{E}_=$ and the greater part $\mathcal{E}_>$ for the minimum number v at position d of hyperedges. Note that the following procedure should be initially started with $d = 0$.

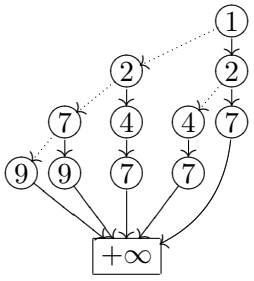
function SORT$(\mathcal{E}, d)$



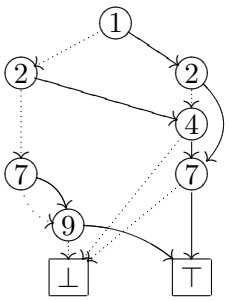
(a) The arrays



(b) The sorting process, ordered from right to left



(c) The binary tree representation



(d) The reduced ZDD

Figure 3: The process of sorting $s[0], \ldots, s[4]$ and the corresponding ZDD.

if $|\mathcal{E}| \leq 1$ **then**
  return;
end if
$v \leftarrow$ the minimum number among the d-th entries of sets in $\mathcal{E}$;
if $v = +\infty$ **then**
  return;
end if
$\mathcal{E}_{=} \leftarrow$ the set of hyperedges with the d-th entries equal to v;
$\mathcal{E}_{>} \leftarrow \mathcal{E} \setminus \mathcal{E}_{=}$;
sort$(\mathcal{E}_{=}, d+1)$; sort$(\mathcal{E}_{>}, d)$;
end function

The sorting process can then be represented as an ordered binary tree such that each node holds a partitioning number v; the left and the right children of a node correspond to $\mathcal{E}_{>}$ and $\mathcal{E}_{=}$, respectively. When we consider the left and the right children of a node as the LO and the HI children, this binary tree satisfies the order condition of ZDD nodes, and furthermore it is irreducible with respect to the node elimination rule, because $v < +\infty$ implies $\mathcal{E}_{=} \neq \emptyset$. Note that all recursive function calls exactly correspond to the nodes in this tree. By sharing equivalent subgraphs, we obtain the corresponding ZDD. It would be easy to observe that this correspondence is bijective.

This sort algorithm has a close relation to the ternary partitioning algorithm, called *multikey Quicksort*, given by Bentley and Sedgewick [1]. Their algorithm sorts strings in a similar way, but there are two different points. Firstly, a partitioning number v need not be the minimum number and there are many ways for the choice, from computing the true median to choosing a random value. Secondly,

it partitions $\mathcal{E}$ into the three parts $\mathcal{E}_<$, $\mathcal{E}_=$ and $\mathcal{E}_>$, where $\mathcal{E}_<$, $\mathcal{E}_=$ and $\mathcal{E}_>$ consists of hyperedges with the d-th entries smaller than, equal to and greater than v, respectively. An efficient partitioning method is important. The performance of multikey Quicksort is well-analyzed. We extract the following two theorems from [1], where c denotes a constant. A worst-case and an expected-time algorithms that establish $c = 3$ and $c = 3/2$ are given in [15] and [6], respectively.

Theorem 1. *If multikey Quicksort partitions around a median computed in cn comparisons, it sorts n k-vectors in at most $cn(\log n + k)$ scalar comparisons.*

Let H_n denote the harmonic numbers, defined by $H_n = \sum_{1 \le i \le n} 1/i$.

Theorem 2. *A multikey Quicksort that partitions around the median of $2t + 1$ randomly selected elements sorts n k-vectors in at most $2nH_n/(H_{2t+2} - H_{t+1}) + O(kn)$ expected scalar comparisons.*

When we sort sets with multikey Quicksort, the worst-case number of comparisons does not depend on the size of a ground set. Thus it would be effective if a sparse set family is given. However, since multikey Quicksort is based on a ternary partitioning method, it is isomorphic to ternary search trees and does not lead to ZDDs directly.

In view of the observation above, our idea is that we first sort sets with multikey Quicksort and then construct a ZDD, based on the binary partitioning method. Algorithm 1 shows the construction part. Since an input set family $\mathcal{E}$ is already sorted, finding the minimum number v and partitioning $\mathcal{E}$ into $\mathcal{E}_=$ and $\mathcal{E}_>$ can be done quickly: since the hyperedges with the d-th entries equal to v are all lined up from the beginning and we need not search all hyperedges.

Algorithm 1 Compute the ZDD for a set family $\mathcal{E}$.

```
function ZCOMP(E, d)
    if E = ∅ then
        return ⊥;
    end if
    v ← the minimum number among the d-th entries of sets in E;
    if v = +∞ then
        return ⊤;
    end if
    E_= ← the set of hyperedges with the d-th entries equal to v;
    E_> ← E \ E_=;
    hi ← zcomp(E_=, d + 1); lo ← zcomp(E_>, d);
    return zdd_unique(v, lo, hi);
end function
```

Theorem 3. *Suppose that an input set family $\mathcal{E}$ is given as an array of hyperedges sorted in lexicographical order. Algorithm 1 can be implemented to run in*

$O(N \log^2(\sum_{U\in\mathcal{E}} |U|/N))$ time, where N denotes the number of nodes in the binary tree that is obtained from an output ZDD by not sharing equivalent subgraphs. The required space is proportional to the size of an output ZDD.

Proof. Since $\mathcal{E}$ is already sorted, in order to partition $\mathcal{E}$ to $\mathcal{E}_=$ and $\mathcal{E}_>$, it is sufficient to find the last hypergraph whose d-th entries equals v. This can be efficiently done by skipping as many hypergraphs as possible. Consider for example the following procedure.

1. Let the current hypergraph be the first one.
2. Search only the 2^i-th hypergraphs from the current hypergraph for $i = 1, 2, \cdots$ while the d-th entries equal v.
3. If the last hypergraph is not found, then go to the 2nd step.
4. Return the last hypergraph.

Since the 2nd step is over in at most $\log |\mathcal{E}_=|$ steps, the overall procedure requires $O(\log^2 |\mathcal{E}_=|)$ time. Since the d-th entries equal to v are not examined in later recursive calls, the sum of all the sizes $|\mathcal{E}_=|$ equals the sum of all hyperedge sizes. From Jensen's inequality, it follows that the total time is $O(N \log^2(\sum_{U\in\mathcal{E}} |U|/N))$.

Since Algorithm 1 constructs an output ZDD in a bottom up fashion, all nodes requested by zdd_unique appear in the output ZDD. Note that since the maximum depth of a recursive function call corresponds to the maximum length of a path in an output ZDD, it is clear that the space required to keep track of the recursive function calls is dominated by an output ZDD size. ∎

4 Experiment

Implementation and Environment. We implemented our algorithm and the naive method presented in Section 2.2 in C. We furthermore implemented the combination of multikey Quicksort and the naive method, in which an input data file is sorted with multikey Quicksort and then the naive method is executed. In these programs, we used the BDD Package Sapporo-Edition-1.0 developed by Minato, in which ZDDs are available and various basic operations for ZDDs are provided. The (non-tuned) implementation of multikey Quicksort was obtained from [2]. All experiments were performed on a 2.67GHz Xeon®E7-8837 with 1.5TB RAM, running SUSE Linux Enterprise Server 11. We compiled our code with version 4.3.4 of the gcc compiler.

Problem Instances. The following instances were obtained from the Hypergraph Dualization Repository [13], where n denotes an instance parameter.

- BMS-WebView-2 (bms(n)): a hypergraph such that the size of a ground set is 3,341 and each hyperedge is the complement of a set of maximal frequent itemsets with support threshold n, of datasets "BMS-WebView2" taken from the Frequent Itemset Mining Dataset Repository.
- Uniform random (rand(n)): a hypergraph such that the size of a ground set is 50 and the number of hyperedges is 1,024,000 and each vertex is included in a hyperedge in the probability $n/10$, generated by Prof. Alain Bretto.

The following instances were obtained from the Frequent Itemset Mining Dataset Repository.

- T40I10D100K: this data set was generated using the generator from the IBM Almaden Quest research group.
- retail: this dataset contains the (anonymized) retail market basket data from an anonymous Belgian retail store, donated by Tom Brijs.
- accidents: this dataset contains (anonymized) traffic accident data, donated by Karolien Geurts.

We furthermore generated instances of the following type.

- Threshold function (TH(n)): a hypergraph whose hyperedges correspond to prime conjunctive normal forms of the threshold function with 30 variables that returns 1 if at least n variables are 1. The number of hyperedges is $\binom{30}{n-1}$.

All instances are given as data files with the following format: each row corresponds to a hyperedge, and entries are non-zero positive numbers (and less than or equal to the maximum number allowed in a BDD package); the entries in a row are sorted in increasing order and separated by a white space. Since some data sets obtained from the Frequent Itemset Mining Dataset Repository did not have a correct form, we formated them: in particular all entries are incremented by one so that they are positive.

Comparison of Algorithms We compared our algorithm zcomp, the naive method, and the combination of multikey Quicksort and the naive method. Note that zcomp means the combination of multikey Quicksort and Algorithm 1. In this experiment, the rows in data files were rearranged at random. The results are shown in Fig. 4 and Table 1.

5 Conclusion

We presented a new algorithm for constructing ZDDs for hypergraphs. This algorithm sorts hyperedges with multikey Quicksort and then construct a ZDD in

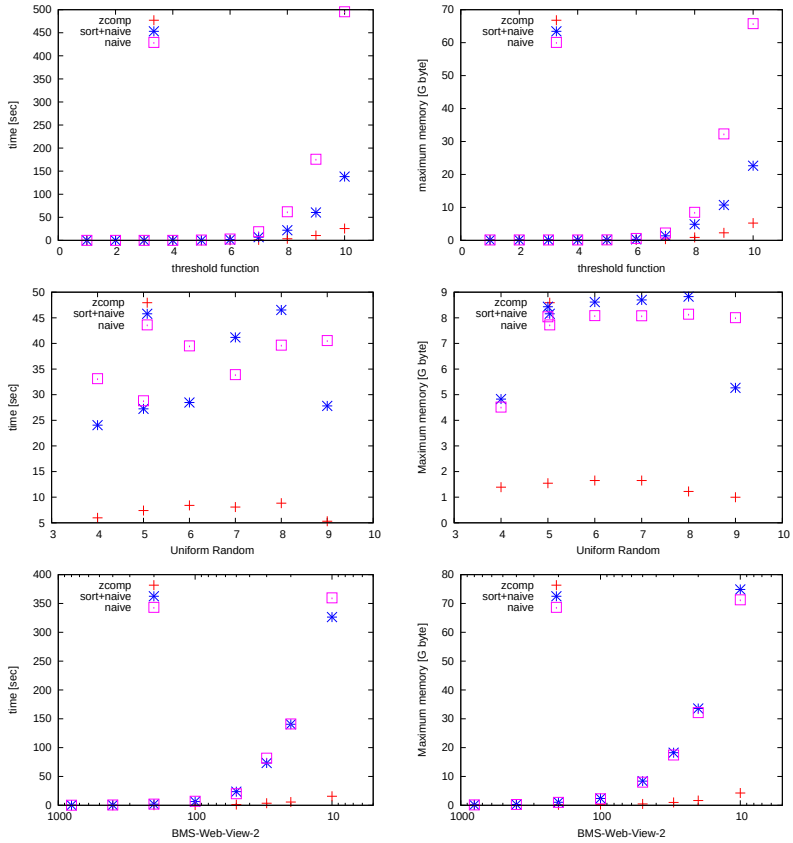


Figure 4: Comparison of running time and maximum memory usage, where the horizontal coordinate of a point represents an instance parameter n.

a bottom up fashion. We analyzed the computational complexity of the construction part. We furthermore conducted experiments with various datasets. The experiments showed that our algorithm is significantly faster and requires smaller memory than an existing method.

References

[1] J. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 360–369, New Orleans, Louisiana, USA, January 1997.

[2] J. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. `http://www.cs.princeton.edu/~rs/strings/`, 2013. accessed on 7 April.

[3] O. Coudert. Solving graph optimization problems with ZBDDs. In *the 1997 European Conference on Design and Test*, pages 224–228, Paris, France, March 1997.

Table 1: Comparison of running time and maximum memory usage

	time (sec)			memory (G byte)		
	naive	sort+naive	zcomp	naive	sort+naive	zcomp
T40I10D100K	7.32	7.53	2.53	2.00	2.07	0.63
retail	15.59	18.85	0.34	4.18	4.42	0.16
accidents	19.36	18.43	3.84	4.19	4.39	1.19

[4] T. Eiter and G. Gottlob. Hypergraph transversal computation and related problems in logic and AI. *LNAI*, 2424:549–564, 2002.

[5] T. Eiter, K. Makino, and G. Gottlob. Computational aspects of monotone dualization: A brief survey. *Discrete Applied Mathematics*, 156:2035–2049, 2008.

[6] R. Floyd and R. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18(3):165–172, 1975.

[7] B. Goethals. Survey on frequent pattern mining. `http://win.ua.ac.be/~adrem/bibrem/pubs/fpm_survey.pdf`, 2003.

[8] D. Knuth. *The Art of Computer Programming Volume 4a.* Addison-Wesley Professional, New Jersey, USA, 2011.

[9] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *30th ACM/IEEE Design Automation Conference (DAC-93)*, pages 272–277, Dallas, Texas, USA, Jun 1993.

[10] S. Minato and H. Arimura. Efficient method of combinatorial item set analysis based on Zero-Suppressed BDDs. In *IEEE/IEICE/IPSJ International Workshop on Challenges in Web Information Retrieval and Integration (WIRI-2005)*, pages 3–10, Tokyo, Japan, April 2005.

[11] S. Minato, T. Uno, and H. Arimura. LCM over ZBDDS: fast generation of very large-scale frequent itemsets using a compact graph-based representation. In *12th Pacific-Asia conference on Advances in knowledge discovery and data mining*, pages 234–246, Osaka, Japan, May 2008.

[12] K. Murakami and T. Uno. Efficient algorithms for dualizing large-scale hypergraphs. In *Proc. of the Meeting on Algorithm Engineering & Experiments (ALENEX)*, pages 1–13, New Orleans, Louisiana, USA, January 2013.

[13] K. Murakami and T. Uno. Hypergraph dualization repository. `http://research.nii.ac.jp/~uno/dualization.html`, 2013. accessed on 19 January.

[14] A. Salleb and C. Vrain. Estimation of the density of datasets with decision diagrams. In M.-S. Hacid, N. Murray, Z. Raś, and S. Tsumoto, editors, *Foundations of Intelligent Systems*, volume 3488 of *Lecture Notes in Computer Science*, pages 688–697. Springer Berlin Heidelberg, 2005.

[15] A. Schoenhage, M. Paterson, and N. Pippenger. Finding the median. *Journal of Computer and Systems Sciences*, 13:184–199, 1976.

[16] K. Sekine and H. Imai. A unified approach via BDD to the network reliability and path number. Technical Report TR-95-09, Department of Information Science, University of Tokyo, 1995.

[17] K. Sekine, H. Imai, and S. Tani. Computing the tutte polynomial of a graph of moderate size. In *6th International Symposium on Algorithms and Computations*, pages 224–233, Cairns, Australia, December 1995.

[18] T. Toda. Hypergraph transversal computation with binary decision diagrams. In *12th International Symposium on Experimental Algorithms*, Rome, Italy, June 2013. accepted.