TCS-TR-A-13-63

TCS Technical Report

Z-Skip-Links for Fast ZDD Traversal in Handling Large-Scale Sparse Datasets

by

Shin-ichi Minato

Division of Computer Science

Report Series A

April 25, 2013

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

Z-Skip-Links for Fast ZDD Traversal in Handling Large-Scale Sparse Datasets

Shin-ichi Minato[*]
Division of Computer Science
Hokkaido University
Sapporo 060-0814, Japan

April 25, 2013

(Abstract) ZDD (Zero-suppressed Binary Decision Diagram) is known as an efficient data structure for representing and manipulating large-scale sets of combinations. In this article, we propose a method of using *Z-Skip-Links* to accelerate ZDD traversals for manipulating large-scale sparse datasets. We discuss average case complexity analysis of our method, and present the optimal parameter setting. Our method can be easily implemented into the existing ZDD packages just by adding one link per ZDD node. Experimental results show that we obtained dozens of acceleration ratio for the instances of the large-scale sparse datasets including thousands of items.

1 Introduction

A *set of combinations* is one of the most fundamental model of discrete structure for solving combinatorial problems in various applications. Binary Decision Diagram (BDD) [2], a state-of-the-art data structure of Boolean function representation, is sometimes used for solving combinatorial problems because n-input Boolean functions have one-to-one correspondence to the sets of combinations considering n items. Zero-suppressed BDD (ZDD) [9] is a variant of BDD, customized for manipulating sets of combinations. ZDDs have been successfully applied not only for VLSI design but also for various real-life applications, such as data mining, system diagnosis, and network analysis.

Recently, processing of "Big Data" have attracted a great deal of attention, and we often deal with a large-scale sparse dataset, which has more than thousand or ten thousands of items as the columns of a dataset. If we represent such data using a ZDD, the height of ZDD grows as large as the number of items, and the depth

[*]He also works for ERATO MINATO Discrete Structure Manipulation System Project, Japan Science and Technology Agency.

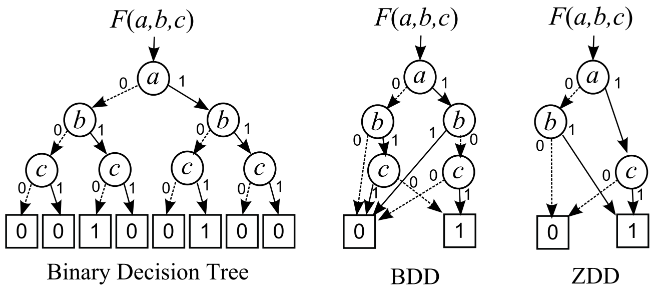


Figure 1: Binary Decision Tree, BDD and ZDD.

of recursive operations also becomes very large. Thus, ZDD-based manipulation is usually not very efficient for such large-scale sparse datasets.

In this paper, we propose an idea of attaching a “Z-Skip-Link” to each ZDD node for accelerating the traversal of ZDDs of large-scale sparse datasets. It consumes only a constant size of additional memory, and can easily be implemented into a conventional BDD/ZDD package. We also show the average-case computation time of traversing ZDDs, in order to evaluate the effect of Z-Skip-Links and thier optimal setting of the skip length. In the practical case of sparse datasets with thousands of items, our experiments show that the use of Z-Skip-Links makes the membership operations 10 to 30 times faster than using conventional ZDD operations.

In the rest of this paper, we first explain the basic properties of ZDDs and what is the problem in manipulating large-scale sparse datasets. We then present an idea of Z-Skip-Links and average-case complexity analysis. Finally we describe the algorithm implementation and the experimental results.

2 Preliminary – Basic Properties of ZDDs

A Binary Decision Diagram (BDD) [2] is a graph representation for a Boolean function. As illustrated in Fig. 1, it is derived by reducing a binary decision tree, which represents a decision making process that depends on some input variables. In this graph, we may find the following two types of decision nodes:

(a) *Redundant node*: A decision node whose two child nodes are identical.

(b) *Equivalent nodes*: Two or more decision nodes having the same variable and the same pair of child nodes.

If we find such types of nodes, we can reduce the graph without changing the semantics (in other words, we can compress the graph). If we fix the order of input variables and apply the two reduction rules as much as possible, then we obtain a canonical form for a given Boolean function [1]. Such a data structure is called an Ordered BDD (OBDD), but in this article we will just call it a BDD.

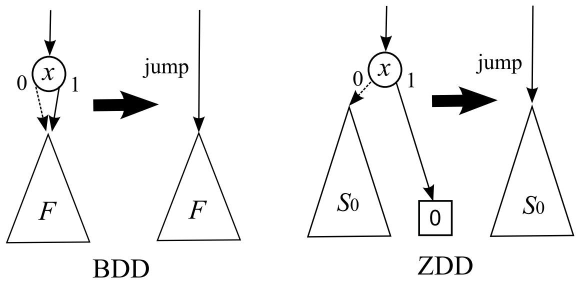


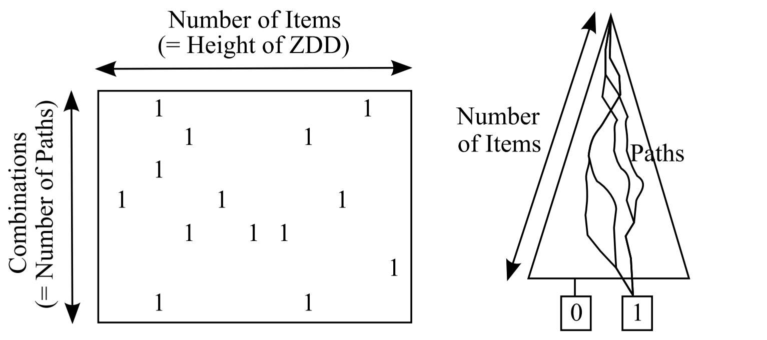
Figure 2: ZDD reduction rule.

The compression ratio of a BDD depends on the properties of Boolean function to be represented, but it can be 10 to 100 times more compact in some practical cases. In addition, we can systematically construct a BDD that is the result of a binary logic operation (i.e., AND or OR) for a given pair of BDDs. This algorithm is based on a recursive procedure with hash table techniques, and it is very efficient when the BDDs have a good compression ratio. The computation time is bounded by the product of the BDD sizes of the two operands, and in many practical cases, it is linearly bounded by the sum of input and output BDD sizes [12].

Zero-suppressed BDDs (ZDDs, or ZBDDs) [9] are a variant of BDDs, customized to manipulate sets of combinations. An example is shown in Fig.1. ZDDs are based on special reduction rules that differ from the ordinary ones. As shown in Fig. 2, we delete all nodes whose 1-edge directly points to the 0-terminal node, but do not delete the nodes which would be deleted in an ordinary BDD. Similarly to ordinary BDDs, ZDDs give compact canonical representations for sets of combinations. We can construct ZDDs by applying algebraic set operations such as union, intersection and difference, which correspond to logic operations in BDDs.

The zero-suppressing reduction rule is extremely effective if we are handling a set of sparse combinations. If the average appearance ratio of each item is 1%, ZDDs are possibly up to 100 times more compact than ordinary BDDs. Such situations often appear in real-life problems, for example, in a supermarket, the number of items in a customer's basket is usually much less than the number of all the items displayed.

Recently, ZDD has become more widely known, since D. E. Knuth intensively discussed ZDD-based algorithms in the latest volume of his famous series of books [6]. The original BDD was invented and developed for VLSI logic design, but ZDD is now recognized as the most important variant of BDD, and is widely used in various kinds of problems in computer science [3, 7, 8, 5].



(a) Large-scale sparse dataset. (b) ZDD

Figure 3: Large-scale sparse dataset and ZDD.

3 Problem for Handling Large-Scale Sparse Datasets

Recently, some kinds of Big Data are regarded as a large-scale sparse dataset, which is a set of many combinations each of which selects a few items out of thousands of ones, as illustrated in Fig. 3(a). There are so many practical examples, such as the basket data in a supermarket, the word-correlation dataset in a natural language, the key word lists in document database, grouping of Internet web pages, and SNPs (mutation points) of human gene data.

If we represent such a large-scale sparse dataset using a ZDD, each path from the root node to the 1-terminal node corresponds to one combination in the dataset, as shown in Fig. 3(b). Namely, the number of such paths in the ZDD equals the number of combinations in the dataset, and the 1-edges on each path represent the occurrence of items in a combination. Since the zero-suppression rule automatically deletes the ZDD nodes corresponding to the items not occurring in the combination, the ZDD size is bounded by the total number of item occurrences in the datasets. If there are many partially similar patterns of combinations, the paths in the ZDD are shared with each other, and in such cases very large compression ratio is obtained. For example, *LCM-ZDD* method [10] efficiently generates nearly a billion patterns of frequent itemsets by using only thousands of ZDD nodes.

Such data compression is one of the big advantage of using ZDDs, however, there is a hidden problem that the height of the ZDD must be n if the dataset contains n relevant items. If we represent a large-scale and sparse dataset using thousands of items, the ZDD becomes a very unbalanced form as shown in Fig. 4, such that we need more than thousand hops for 0-edge traversal while only a few hops needed for 1-edge side to reach a terminal node. Unfortunately, such type of datasets often appear in real-life applications. In general, ZDD-based operations require a computation time linear in the height of the ZDD. The membership testing operation is especially inefficient in this case because thousands of steps of ZDD traversal is needed to check the existence of the k-th item's decision node, even if the membership query has only a combination with a few items. It means

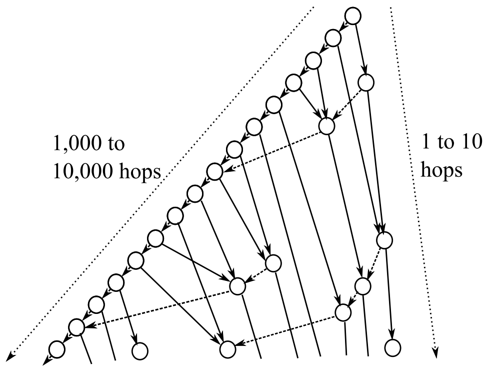


Figure 4: An example of very unbalanced ZDD.

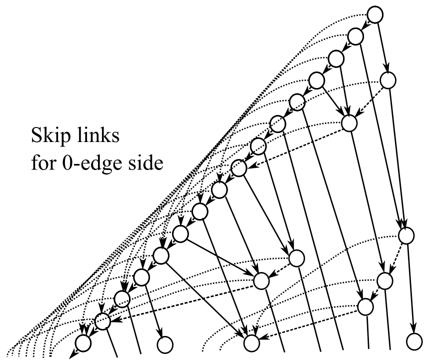


Figure 5: ZDD with Z-Skip-Links.

that the ZDD-based operation could be hundreds or more times slower than a naive data structure based on arrays and linked lists.

In this paper, we propose a practical method for addressing this problem for handling large-scale sparse datasets using ZDDs.

4 Z-Skip-Links

4.1 Basic Idea

Conventional ZDD operation linearly traverses the cascade of 0-edges, and it requires $O(n)$ steps in average to check the existence of a ZDD node with a given item where n is the height of the very unbalanced ZDD. If we prepare a table of pointers to the descendant ZDD nodes indexed by the item IDs, we can directly jump to the destination node in $O(1)$ steps. However, we have to prepare such a pointer table for each ZDD nodes as the start point, so additional memory requirement becomes $O(n)$ words per ZDD nodes, and the total memory requirement becomes $O(nG)$, where G is ZDD size. This is unacceptable memory increase for large n as hundreds or thousands.

Our basic idea is to attach only one pointer to each ZDD node, as illustrated in Fig. 5. The pointer, named *Z-Skip-Link*, indicates a descendant node reachable by a some length of cascade of 0-edges. When we want to search a ZDD node with the t-th level, we first refer the Z-Skip-Link at the root node and check the level of the pointed node s. If s is still higher than t, we execute the jump by the Z-Skip-Link and continue the search from there. If s is exceedingly lower than t, we cancel the jump and descend the 0-edges for one step, and then continue the search at the next node.

Our method is quite simple, but the reachability is clearly guaranteed as ordinary linear search. The search time can be reduced as much as the total jump

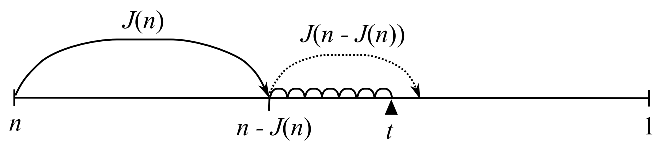


Figure 6: Basic Model with a Number Line.

length of Z-Skip-Links. The additional memory requirement is just a constant factor for the ZDD size. Since the Z-Skip-Links do not have any side-effect to the ZDD operations, we can easily implemented in a conventional BDD/ZDD package without any significant modification to the basic data structure and operation codes.

The key issue of this method is the design of the skip length of Z-Skip-Links. The longer skip length gets the more saving time, but too longer skip length may increase the probability of exceeding the target and the less chance of speed up. In this article, we discuss the average case analysis of time complexity and the optimal skip length for a given n.

4.2 Basic Model and Average Search Cost

For analyzing average search cost, we consider the model of a number line as shown in Fig. 6. There are n positions on the line from 1 to n, where the start position is n, and the target position is t. We define a *skip length function* $J(x)$ to represent the skip length at the current position x $(1 \leq x \leq n)$. The first skip length should be $J(n)$ and the second length becomes $J(n - J(n))$. We repeat the jumps until passing through the target t, and after that we execute a linear search from the last position. In this model, we do not know t beforehand, thus we assume that t has a discrete uniform distribution between 1 to n. Here, we define $C_1(n)$ as the average number of jumps, and $C_2(n)$ as the average number of moves in the linear search. Our objective is to design a good skip length function $J(x)$ to minimize the total average cost $C(n) = C_1(n) + C_2(n)$. The function $J(x)$ cannot depend on t but may depend on n.

At first, we assume the simple skip length function with a constant $\alpha \geq 1$:

$$J(x) \quad = \quad \left\lceil \frac{1}{\alpha} x \right\rceil . \tag{1}$$

For example, $\alpha = 4$ means that we skip 1/4 distance of the remaining search range. Then, we get the average search cost $C(n)$ as follows. (Detailed calculation is shown in Appendix.)

$$C(n) = \alpha + \frac{n}{4(\alpha - 2)} \tag{2}$$

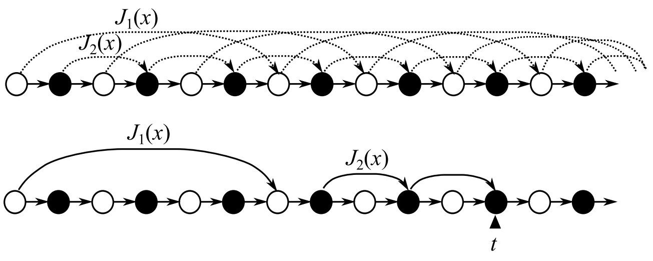


Figure 7: Two-Phase Model.

If we set $\alpha = 16$, we get $C(10000) \approx 182.6$, which is 27.4 times faster than ordinary linear search.

Let us consider the optimal α for a given n. The derivative of $C(n)$ with respect to α can be written as:

$$\frac{d}{d\alpha}C(n) = 1 - \frac{n}{4(\alpha-2)^2}$$

and it becomes zero, thus:

$$\alpha_{opt} \quad = \quad \frac{\sqrt{n}}{2} + 2, \quad C_{opt}(n) = \sqrt{n} + 2 \tag{3}$$

are obtained. In other words, we can accelerate up to $\frac{\sqrt{n}}{2}$ times in average from linear search. When $n = 10000$, we get $\alpha_{opt} = 52$ and $C_{opt}(10000) = 102$, about 50 times faster than linear search.

4.3 Two-Phase Model

The above discussion assumed a uniform skip length function, but we may improve the performance if we use different forms of skip length functions for different positions. Figure 7 illustrates our idea of the two-phase model, using the two types of skip length functions $J_1(x)$ and $J_2(x)$ for an even number x and an odd number x, respectively. $J_1(x)$ provides long jumps and $J_2(x)$ provides middle length jumps. We first repeat long jumps with $J_1(x)$ until passing through the target t, next we repeat middle jumps with $J_2(x)$, and after that we execute a linear search. We set that both $J_1(x)$ and $J_2(x)$ return an even number of skip length, and thus destination of J_1 is always a position of J_1, and J_2 also going to a J_2 position.

We also analyzed the average case search cost in this two-phase model. We assume the same skip length as $1/\alpha$ of remaining search range for both $J_1(x)$ and $J_2(x)$. Then $C(n)$ is described as follows. (Detailed calculation is shown in Appendix.)

$$C(n) = 2\alpha + \frac{n}{8(\alpha-2)^2} \tag{4}$$

If we set $\alpha = 16$, we get $C(10000) \approx 38.4$, which is about 130 times faster than ordinary linear search.

Similarly to the uniform model, we can calculate optimal α for a given n.

$$\alpha_{opt} = \frac{\sqrt[3]{n}}{2} + 2, \quad C_{opt}(n) = \frac{3}{2}\sqrt[3]{n} + 4 \tag{5}$$

When $n = 10000$, we get $\alpha_{opt} = 21.54$ and $C_{opt}(10000) \approx 36.3$, about 138 times faster than linear search.

In the above discussion, we ignored at most 2 steps which are required for changing J_1 and J_2 positions, so we must add a small constant factor for exact analysis. Anyway, we can significantly improve the performance only using two different skip length functions without any additional memory requirement.

4.4 *k*-Phase Model

Extending the two-phase model, we can consider k types of skip length functions, by classifying the position x into k groups by x modulo k. If we write $C^{(k)}(n)$ for the average search cost for k-phase model, it can be described as:

$$C^{(k)}(n) = k\alpha + \frac{n}{2^{k+1}(\alpha - 2)^k} \tag{6}$$

and optimal α is obtained as:

$$\alpha_{opt} = \frac{1}{2}n^{\frac{1}{k+1}} + 2, \quad C^{(k)}_{opt}(n) = \frac{k+1}{2}n^{\frac{1}{k+1}} + 2k \tag{7}$$

For example, when $k = 4$, we get

$$\alpha_{opt} = \frac{1}{2}\sqrt[5]{n} + 2, \quad C^{(4)}_{opt}(n) = \frac{5}{2}\sqrt[5]{n} + 8. \tag{8}$$

Next, let us consider the optimal k for given n. The detailed calculation is shown in Appendix and we can obtain the following result.

$$k_{opt} = \ln n - 1, \quad C^{(k_{opt})}_{opt}(n) = \frac{5}{2}\ln n - 2 \tag{9}$$

Consequently, the average search cost of the Z-Skip-Links is theoretically shown as $O(\log n)$, if we use optimal k.

In the case of $n = 65536$, we get $k_{opt} \approx 10$ and $C^{(10)}_{opt}(65536) \approx 25.6$. However, even for $k = 4$ we can achieve $C^{(4)}_{opt}(65536) \approx 30.97$, which is more than 1000 times faster than ordinary linear search. We can conclude that, for up to $n = 100000$, 4-phase model is sufficiently effective for practical applications.

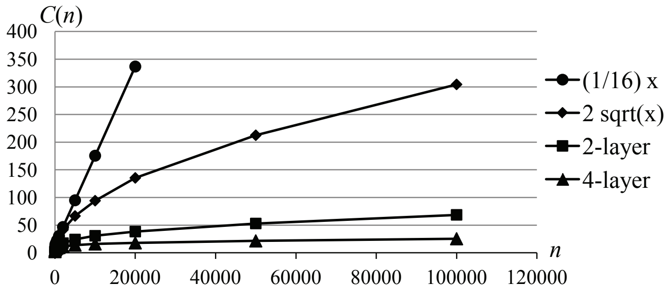


Figure 8: Computational Experiments.

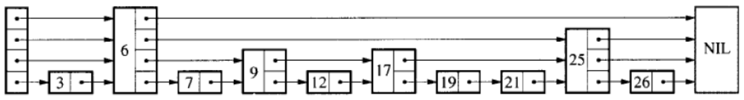


Figure 9: An Example of Skip List [11].

4.5 Computational Experiments

To confirm the above theoretical analysis, we conducted computational experiments. For a given n, we counted the number of moves to t, and computed average moves $C(n)$ for all t from 1 to n. In this experiments we tested four kinds of skip length functions as $\frac{1}{16}x$, $2\sqrt{x}$, the optimal 2-phase model, and the optimal 4-phase model. Figure 8 shows the results. We can observe the result very close to theoretical formulas up to $n = 100000$.

4.6 Related Work

Our method is related to *Skip List* proposed by Pugh [11] in 1990. Skip List is a quick access technique used for sorted linear linked list. It equips probabilistic distributed skip pointers as shown in Fig. 9. It is known that the average access time is $O(\log n)$. From this viewpoint, Z-Skip-Links can be regarded as a kind of Skip List technique deterministically attached to all ZDD nodes. However, our method is not limited a simple linear list but applicable to the complex ZDD structure including a large number of paths shared with each other.

5 Algorithm Implementation

5.1 Implementation to Conventional BDD/ZDD Package

Most of conventional BDD/ZDD packages use short int (16 bit integer) for item IDs, so we assumed up to $n = 65535$. For this range, we know that 4-phase

Table 1: Look-up Table for Skip Length Function.

range of x	$J_1(x)$	$J_2(x)$	$J_3(x)$	$J_4(x)$
$5 - 15$	4	–	–	–
$16 - 63$	8	4	–	–
$64 - 255$	64	16	8	4
$256 - 511$	128	32	8	4
$512 - 1023$	256	64	16	4
$1,024 - 2047$	512	128	32	8
$2,048 - 4095$	1,024	256	64	8
$4,096 - 8,191$	2,048	512	64	8
$8,192 - 32,767$	4,096	512	64	8
$32,768 - 65,535$	8,192	1,024	128	16

is enough effective, so we adopted the 4-phase model. It is time consuming to calculate $\sqrt[5]{n}$ many times, so we approximate it by table look-up. The table is shown in Table. 1

Also, Most of conventional BDD/ZDD Packages have an internal table called "operation cache," which stores the recent operations and their results by a hash-key with the pointers to the operand ZDDs and operation IDs. Our Z-Skip-Links can be easily implemented using the framework of the operation cache, and we needed only 100 lines of additional C++ codes for processing Z-Skip-Links.

Here we explain the algorithm of fast ZDD traversal using Z-Skip-Links, and the preprocessing algorithm for constructing Z-Skip-Links for all ZDD nodes.

5.2 ZDD Traversal Using Z-Skip-Links

Figure 10 shows pseudo codes of ZDescend(F, t). For given root node of ZDD F and the target item-ID t, this algorithm returns the pointer to a ZDD node which is the first meet such that the item ID is equal or lower than t when descending 0-edges starting from F. In this codes, cache("ZSkip at F") means to refer the operation cache. This algorithm enjoys the benefit of Z-Skip-Links, however, even if Z-Skip-Lists are not prepared yet, it returns correct result by linear search.

5.3 Construction of Z-Skip-Links

Figure 11 shows pseudo codes of SetZSkip(F). This algorithm constructs Z-Skip-Links for all internal nodes of ZDD F by using a recursive procedure. At first, we check the operation cache and if a Z-Skip-Link is already constructed, the procedure terminates. Otherwise, recursively call itself with the 0-child node to construct Z-Skip-Links for all descendant nodes. After that, compute the skip length and find the destination node by using ZDescend procedure, and then register it to the operation cache.

The number of recursive call of SetZSkip is bounded by the number of F's ZDD nodes if operation cache works well. SetZSkip also calls ZDescend procedure, so

```
ZDescend($F, t$)
{ while($F.top > t$)
  { $G \leftarrow$ cache("ZSkip at $F$") ;
    if ($G$ exists and $G.top \geq t$) $F \leftarrow G$ ;
    else $F \leftarrow$ (0-Child of $F$) ;
  }
  return $F$ ;
}
```

Figure 10: ZDD Traversal Using Z-Skip-Links.

```
SetZSkip($F$)
{ if ($F.top \leq 4$) return ;
  if (cache("ZSkip at $F$") exists) return ;
  SetZSkip(0-Child of $F$) ;
  $t \leftarrow (F.top - J(F.top))$ ; /* Destination from
$F$ */
  $G \leftarrow$ ZDescend($F, t$) ;
  cache("ZSkip at $F$") $\leftarrow G$ ;
  SetZSkip(1-Child of $F$) ;
}
```

Figure 11: Construction of Z-Skip-Links.

we need several steps to reach the destination node. In the range of $n \leq 65536$, ZDescend procedure requires about 20 to 30 steps in average, therefore, the total computation time will be several ten times of the number of ZDD nodes. This would be feasible overhead if we repeat the search process many times for a large-scale sparse dataset.

6 Experimental Results

We implemented the algorithms and conducted experiments for performance evaluation. The specification of our PC is as follows. Intel Core i7 2700K 3.5GHz, 32GB memory, OpenSuSE Linux 12.1 (64bit), GNU C++ 4.6.2. We used our own BDD/ZDD package written in C and C++.

6.1 Membership Testing for Random-Generated Sparse Datasets

We applied our method to the randomly generated large-scale sparse datasets. We generated a ZDD for the dataset $D_{(n,m)}$ including m combinations each of which randomly selecting two items out of n times. Next we also randomly generate a two item pattern p and evaluate the computation time for the membership testing if p is in $D_{(n,m)}$. We executed SetZSkip procedure for the ZDD as preprocessing and then repeat the membership testing 10000 times with different p's. We compared our method with conventional ZDD-based membership operation as $D_{(n,m)} \cap p_{(n)}$.

Table 2: Results for Random-Generated Sparse Datasets.

n	m	ZDD nodes	CPU time(sec) pre-prc.	search x10000	old x10000	accel. net	gross
1,000	10,000	10,011	0.004	0.012	0.158	13.2	9.9
2,000	20,000	20,181	0.005	0.013	0.390	30.0	21.7
5,000	50,000	50,683	0.007	0.018	1.025	56.9	41.0
10,000	100,000	101,521	0.026	0.021	2.055	97.9	43.7
20,000	200,000	203,272	0.058	0.029	4.499	155.1	51.7
50,000	500,000	508,335	0.170	0.028	14.393	514.0	72.7

Table 3: Results for Frequent Itemset Datasets.

θ	n	m	ZDD nodes	CPU time(sec) pre-prc.	search x10000	old x10000	accel. net	gross
100	932	11,928	2,298	0.001	0.009	0.094	10.4	9.4
50	1,597	70,713	6,059	0.001	0.011	0.169	15.4	14.1
20	2,434	63,4065	30,410	0.007	0.012	0.268	22.3	14.1
10	2,885	4,440,335	93,899	0.021	0.012	0.328	27.3	9.9
5	3,129	26,946,004	353,091	0.050	0.015	0.393	26.2	6.0

The experimental results are shown in Table 2. Here "pre-prc." means the time for preprocessing, "old" means conventional method. "accel." means the ratio of acceleration, "net" means ignoring preprocessing time, and "gross" means including preprocessing time. From the result, if we ignore the preprocessing time, the acceleration ratio reaches more than 500 times when $n = 50000$. Even if we including preprocessing time, more than 70 times faster than conventional method. The experimental results show that our method is effective when implementing on the practical BDD/ZDD package.

6.2 Membership Testing for Frequent Itemset Datasets

Next we applied our method to the frequent itemset data, which are dealt with in the basic problem of data mining. We used a dataset "BMS-WebView2" chosen from the KDD benchmark [4]. This data is known as the access log of web pages at an Internet shopping site, This dataset has 3340 items and 77512 transactions (combinations), but only 4.6 items appears in average per transaction, so it has very sparse combinations. We applied LCM-ZDD method[10] to this dataset and generate a ZDD D_θ including all frequent itemset patterns with a given minimum support θ. For example, D_5 includes 26946004 patterns and there are 3129 relevant items. The number of ZDD nodes are only 353091, so it has very good compression rate.

Then, we also randomly generate a two item pattern p, and similarly evaluate the computation time for the membership testing if p is in D_θ. Table 2 show that our method is 10 to 27 times faster than conventional method without considering preprocessing time, and 6 to 14 times faster with considering preprocessing time.

7 Summary

We proposed Z-Skip-Links to accelerate the traversal of ZDDs of large-scale sparse datasets. It consumes only a constant size of additional memory, and can easily be implemented into a conventional BDD/ZDD package. We analyzed the average-case computation time and have clarified the optimal settings. Our experiments show that the use of Z-Skip-Links makes the membership operations much faster than using conventional ZDD operations when handling large-scale sparse datasets. We expect that our method will widen the effective applications of ZDDs in the era of Big Data.

References

[1] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, 1978.

[2] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[3] Olivier Coudert. Solving graph optimization problems with ZBDDs. In *Proc. of ACM/IEEE European Design and Test Conference (ED&TC '97)*, pages 224–228, 1997.

[4] B. Goethals and M. J. Zaki. Frequent itemset mining dataset repository, 2003. Frequent Itemset Mining Implementations (FIMI'03), http://fimi.cs.helsinki.fi/.

[5] M. Ishihata, Y. Kameya, T. Sato, and S. Minato. Propositionalizing the EM algorithm by BDDs. In *Proc. of 18th International Conference on Inductive Logic Programming (ILP 2008)*, 9 2008.

[6] D. E. Knuth. *The Art of Computer Programming: Bitwise Tricks & Techniques; Binary Decision Diagrams*, volume 4, fascicle 1. Addison-Wesley, 2009.

[7] E. Loekit and J. Bailey. Fast mining of high dimensional expressive contrast patterns using zero-suppressed binary decision diagrams. In *Proc. of 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD2006)*, pages 307–316, 2006.

[8] S. Minato, K. Satoh, and T. Sato. Compiling bayesian networks by symbolic probability calculation based on zero-suppressed BDDs. In *Proc. of 20th*

International Joint Conference of Artificial Intelligence (IJCAI-2007), pages 2550–2555, 2007.

[9] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proc. of 30th ACM/IEEE Design Automation Conference (DAC'93)*, pages 272–277, 1993.

[10] Shin-ichi Minato, Takeaki Uno, and Hiroki Arimura. LCM over ZBDDs: Fast generation of very large-scale frequent itemsets using a compact graph-based representation. In *Proc. of 12th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2008), (LNAI 5012, Springer)*, pages 234–246, 2008.

[11] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Algorithms and Data Structures*, 33(6):668–676, 1990.

[12] Ryo Yoshinaka, Jun Kawahara, Shuhei Denzumi, Hiroki Arimura, and Shin-ichi Minato. Counterexamples to the long-standing conjecture on the complexity of bdd binary operations. *Information Processing Letters*, 112(16):636–640, 2012.

A Calculation in Average-Case Analysis

A.1 Calculation of $C(n)$ in Basic Model

As the target t is assumed discrete uniform distribution from 1 to n, the probability of passing through the target by a jump becomes $1/\alpha$ if x is enough large. Therefore, $C_1(n)$, the average number of jumps from n, can be described as follows.

$$\begin{aligned} C_1(n) &= 1 \cdot \frac{1}{\alpha} + 2 \cdot \frac{1}{\alpha}\left(1-\frac{1}{\alpha}\right) + 3 \cdot \frac{1}{\alpha}\left(1-\frac{1}{\alpha}\right)^2 + \dots \\ &= \sum_{i=1}^{\infty} i \cdot \frac{1}{\alpha}\left(1-\frac{1}{\alpha}\right)^{i-1} \\ &= \alpha \end{aligned} \tag{10}$$

Namely, if we use the skip length function $J(x) = x/\alpha$, $C_1(x)$ becomes constant value α for any large number n. When $\alpha = 4$, we need only 4 times of jumps in average to reach vicinity to the target even if $n = 10000$.

Next, let us consider $C_2(n)$. Here $R_1(n)$ denotes the average length of the last jump, then it is written as:

$$\begin{aligned} R_1(n) &= \frac{1}{\alpha} n \cdot \frac{1}{\alpha} + \frac{1}{\alpha}\left(1-\frac{1}{\alpha}\right) n \cdot \frac{1}{\alpha}\left(1-\frac{1}{\alpha}\right) + \frac{1}{\alpha}\left(1-\frac{1}{\alpha}\right)^2 n \cdot \frac{1}{\alpha}\left(1-\frac{1}{\alpha}\right)^2 + \dots \\ &= \sum_{i=1}^{\infty} n \left(\frac{1}{\alpha}\right)^2 \left(1-\frac{1}{\alpha}\right)^{2(i-1)} \\ &= \frac{n}{2(\alpha-2)} \end{aligned} \tag{11}$$

At the position of just before the last jump, the target t is located in the distance of 0 to $R_1(n) - 1$. As we assume t has uniform distribution, we get:

$$C_2(n) = \frac{R_1(n)}{2} = \frac{n}{4(\alpha-2)}$$

and thus,

$$C(n) = C_1(n) + C_2(n) = \alpha + \frac{n}{4(\alpha-2)} \tag{12}$$

is obtained.

A.2 Calculation of $C(n)$ in Two-Phase Model

Here we assume the same skip length as $1/\alpha$ of remaining search range for both $J_1(x)$ and $J_2(x)$. Then we can write as:

$$J_1(x) = \left\lceil \frac{1}{\alpha} x \right\rceil, \quad J_2(x) = \left\lceil \frac{1}{\alpha} R_1(x) \right\rceil \tag{13}$$

($R_1(n)$ is the average length of the last jump of J_1, and it becomes the initial search range of J_2.)

Let us consider $C_1(n), C_2(n)$, and $C_3(n)$ when using such $J_1(x)$ and $J_2(x)$. First, J_1's condition is completely same as the basic model,

$$C_1(n) = \alpha, \quad R_1(n) = \frac{n}{2(\alpha - 2)}$$

as well as equation 10, 11.

Next, we consider $C_2(n)$, however, it jumps $1/\alpha$ distance as well as J_1, we also get $C_2(n) = \alpha$. $R_2(n)$ is obtained by substituting $R_1(n)$ into n:

$$R_2(n) = \frac{R_1(n)}{2(\alpha - 2)} = \frac{n}{4(\alpha - 2)^2}$$

and thus,

$$C_3(n) = \frac{R_2(n)}{2} = \frac{n}{8(\alpha - 2)^2}$$

Consequently, the total cost is:

$$C(n) = C_1(n) + C_2(n) + C_3(n) = 2\alpha + \frac{n}{8(\alpha - 2)^2} \tag{14}$$

Let us calculate optimal α for a given n in a similar way as the basic model. The derivative of $C(n)$ with respect to α is written as:

$$\frac{d}{d\alpha}C(n) = 2 - \frac{n}{4(\alpha - 2)^3}$$

and it must be zero, then:

$$\alpha_{opt} = \frac{\sqrt[3]{n}}{2} + 2, \quad C_{opt}(n) = \frac{3}{2}\sqrt[3]{n} + 4 \tag{15}$$

are obtained.

A.3 Calculation of $C_{opt}^{(k_{opt})}(n)$ in k-Phase Model

The average cost in k-phase model is written as:

$$C^{(k)}(n) = k\alpha + \frac{n}{2^{k+1}(\alpha - 2)^k} \tag{16}$$

For calculating optimal α, the derivative of $C^{(k)}(n)$ with respect to α can be written as:

$$\frac{d}{d\alpha}C^{(k)}(n) = k - \frac{kn}{2^{k+1}(\alpha - 2)^{k+1}}$$

and it should be zero, then:

$$\alpha_{opt} = \frac{1}{2}n^{\frac{1}{k+1}} + 2, \quad C_{opt}^{(k)}(n) = \frac{k+1}{2}n^{\frac{1}{k+1}} + 2k \tag{17}$$

are obtained.

Next, let us consider the optimal k for given n. If we set $\frac{1}{k+1} = t$, then we can rewrite $C_{opt}^{(k)}(n)$ as:

$$C_{opt}^{(k)}(n) = \frac{n^t + 4}{2t} - 2.$$

Then calculate the derivative by t,

$$\frac{d}{dt}C_{opt}^{(k)}(n) = \frac{(t\ \ln n - 1)n^t}{t^2}$$

and it becomes zero when $t = \frac{1}{\ln n}$, thus,

$$k_{opt} = \ln n - 1, \quad C_{opt}^{(k_{opt})}(n) = \frac{5}{2}\ln n - 2 \tag{18}$$

are obtained.