TCS-TR-A-13-69

TCS Technical Report

Efficient Top-Down ZDD Construction Techniques Using Recursive Specifications

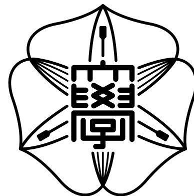
by

Hiroaki Iwashita and Shin-ichi Minato

Division of Computer Science

Report Series A

December 17, 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

Efficient Top-Down ZDD Construction Techniques Using Recursive Specifications

Hiroaki Iwashita*†‡ Shin-ichi Minato‡†

December 17, 2013

Abstract

Many important problems of graph enumeration and indexing can be solved by frontier-based methods, which construct ZDDs in a breadth-first manner from the top to the bottom. We present new techniques toward an efficient framework to deal with the frontier-based methods. Ad hoc parts of the algorithm are encapsulated using the recursive specifications that represent properties to be compiled into a ZDD. In this framework, we can apply the ZDD node deletion rule on the fly, while conventional methods does not take it into account. Operations on the recursive specifications, which allow us to combine multiple properties without constructing ZDD structure for each property, are also introduced. These techniques are applicable to existing frontier-based methods and accelerates even Knuth's sophisticated path enumeration algorithm doubly.

1 Introduction

Ordered binary decision diagrams (BDDs) and zero-suppressed BDDs (ZDDs) are important data structures for representing Boolean functions and families of sets on computers [1][2][3]. They have originally become popular in problems of computer-aided design for digital systems (CAD), such as logic synthesis and verification. Their range of applications are still expanding.

There are many open source and in-house BDD packages which have been used in such traditional applications as CAD problems. They are general-purpose packages for manipulating a collection of BDDs [4][5], allowing us to create primitive BDDs (variables and constants) and to construct complex BDDs by applying operations repeatedly to existing ones. They usually traverse given BDDs in a depth-first manner and construct the resulting BDD in a bottom-up way. Breadth-first BDD manipulation algorithms have

*iwashita@erato.ist.hokudai.ac.jp

†ERATO MINATO Discrete Structure Manipulation System Project, Japan Science and Technology Agency, Sapporo, Japan.

‡Graduate School of Information Science and Technology, Hokkaido University, Sapporo, Japan.

been also proposed [6][7][8][9]. They are designed to have the same functionality with ordinary BDD packages and to accelerate them by increasing memory access locality.

Graph enumeration and indexing problems are also important applications of BDDs and ZDDs, which include enumeration/indexing of paths, cycles, connected components, trees, forests, cut sets, partitioning, cliques, colorings, tilings, and matching. They are tightly related to various real-life problems, such as geographic information systems, dependency analysis, and demarcation problems. Each problem is solved implicitly by construction of a monolithic ZDD representing a family of all instances, where each instance (path, cycle, etc.) is represented by a set of graph edges or vertices. Some of them can be constructed efficiently by conventional bottom-up ZDD operations; others are covered by *frontier-based methods*, which construct result ZDDs directly from the root to the terminal nodes.

Coudert introduced a ZDD based framework for solving graph and set related optimization problems [10]. It includes a bottom-up construction algorithm of the ZDD that represents all maximal cliques of a given graph. Sekine et al. proposed top-down BDD construction algorithms for computing Tutte polynomial and all spanning trees of a given graph [11]. They also showed that the BDD for all spanning trees can be used to obtain a BDD for all forests and a BDD for all paths between two vertices [12]. Knuth introduced an algorithm to construct a ZDD representing all paths between two vertices in a top-down way [3, exercise 225 in 7.1.4]. His algorithm is so efficient that a ZDD representing

$$227449714676812739631826459327989863387613323440$$

paths on a 15×15 grid graph is constructed within a few minutes. Cycles, Hamiltonian paths, and path matching of a given graph can also be computed by similar algorithms [3][13].

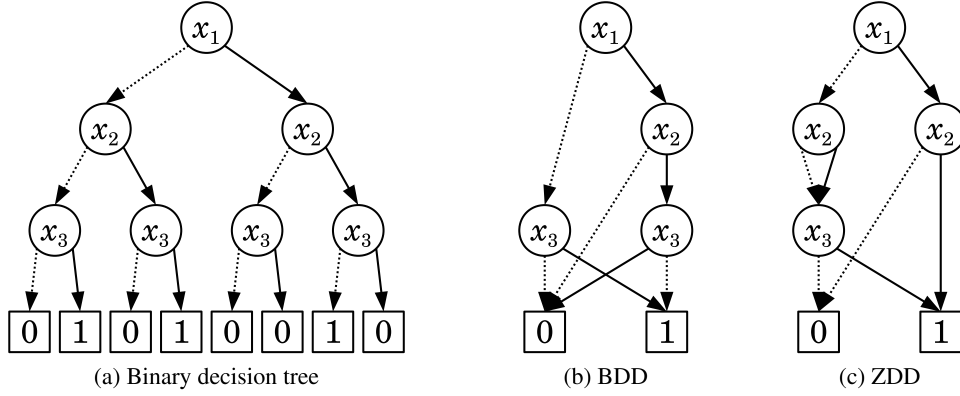
We present new techniques toward an efficient ZDD framework to deal with frontier-based methods. Our approach applies the ZDD node deletion rule on the fly, while conventional methods does not take it into account. We also introduce top-down ZDD construction algorithms for a combination of multiple properties. They do not construct intermediate ZDDs for all the properties, which may blow up and become bottleneck in conventional methods.

This paper is organized as follows. Basics of BDDs and ZDDs, including top-down construction methods and Knuth's algorithm, are reviewed in Section 2. Our top-down construction framework is introduced in Section 3, which contains a new method to speedup the construction. Section 4 describes the operations and techniques on our framework that can be chosen for more efficient computation. Experimental results in Section 5 shows practicality of our framework and techniques. We then conclude the paper in Section 6.

2 Preliminaries

2.1 BDDs and ZDDs

Binary decision diagrams (*BDDs*) [14][1] and *zero-suppressed BDDs* (*ZDDs*) [2] are labeled directed acyclic graphs derived by reducing binary decision tree graphs, which rep-



(a) Binary decision tree (b) BDD (c) ZDD

Figure 1: Diagrams for $f(x_1,x_2,x_3) = x_1x_2\overline{x_3} + \overline{x_1}x_3$

resent decision making processes through binary input variables. As illustrated in Figure 1, there are two kinds of terminal nodes, *0-terminal* and *1-terminal*, which represent the output binary value. Every nonterminal node is labeled by an input variable and has two outgoing edges, namely *0-edge* and *1-edge*, which are drawn as dotted and solid arrows respectively. The 0-edge (1-edge) points to the node called *0-child* (*1-child*), which represent a state after the decision that 0 (1) is assigned to the variable. When the root node of a BDD/ZDD for Boolean function f is labeled by variable x, its 0-child and 1-child represent $f_{x=0}$ and $f_{x=1}$ respectively; it corresponds to the Shannon expansion: $f = \bar{x} f_{x=0} + x f_{x=1}$. In this paper, we also write it as $f = (\bar{x}\ ?\ f_{x=0} : f_{x=1})$, implying structure of the diagram.

We only deal with ordered BDDs/ZDDs in this paper, where input variables are indexed as $x_1,\ldots,x_n$ according to their total order. The index of the input variable of a nonterminal node is just called the index of the node, and the index of a terminal node is assumed to be $n+1$ for convenience. The index of any node is properly smaller than that of its children.

Figure 2 and Figure 3 show the reduction rules of BDDs and ZDDs respectively. Equivalent nodes, which have the same indices and the same 0- and 1-child nodes, can be shared both in BDDs and in ZDDs (Figure 2a and Figure 3a). A node with edges to the same destination can be deleted in BDDs (Figure 2b). In contrast, a node with a 1-edge directly pointing to the 0-terminal node can be deleted in ZDDs (Figure 3b). If x has a smaller index than the top variable of f, $f_{x=0} = f_{x=1} = f$ in BDDs while $f_{x=0} = f$ and $f_{x=1} = 0$ in ZDDs. An entire BDD/ZDD can be reduced completely by applying the reduction rules from the bottom (index n) to the top (index 1) as follows:

```
REDUCE(f)
1: for i = n to 1 do
2:   for all node p at index i in the diagram rooted by f do
3:     for all b ∈ {0,1} do
4:       apply reduction rules to the b-child of p;
5:     end for
6:   end for
```

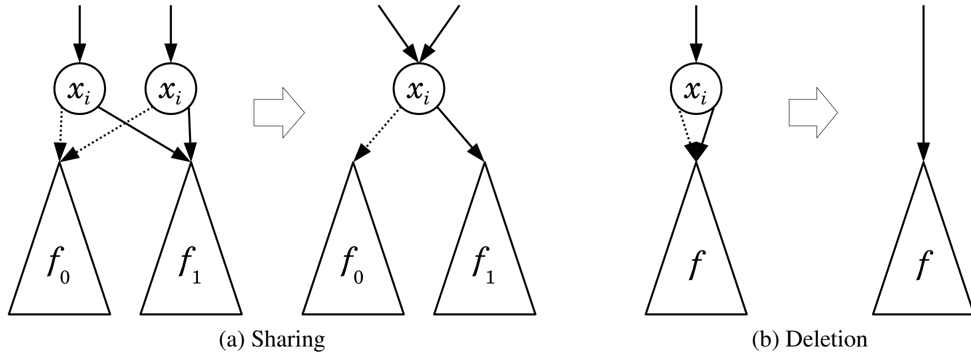


Figure 2: BDD reduction rules

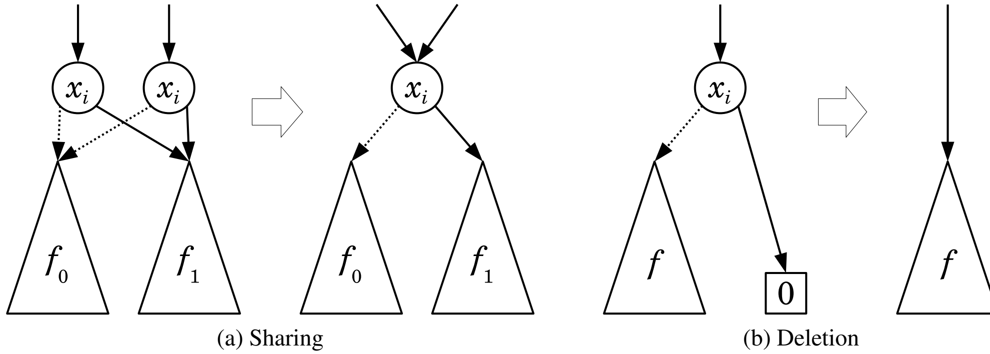


Figure 3: ZDD reduction rules

7: **end for**
8: **return** reduced node for f.

BDDs and ZDDs are efficient data structures for representing not only Boolean functions but also families of sets. A set of n items can be represented by input variables $x_1, \ldots, x_n$, where $x_i \in \{0,1\}$ indicates if the i-th item is contained in the set. The diagrams in Figure 1 can be considered as $\{\{x_1,x_2\},\{x_2,x_3\},\{x_3\}\}$ in that sense. Paths from the root to the 1-terminal in BDDs and ZDDs, called *1-paths*, correspond to item sets included in the family. ZDDs have the interesting property that every 1-path represents an individual set, while a 1-path may represent multiple sets in BDDs, because of the difference of their node deletion rules. ZDDs are especially suitable for representing families of sparse item sets. If the average appearance rate of each item is 1%, ZDDs are possibly up to 100 times more compact than BDDs. Such situations often appear in real-life problems.

2.2 Operations on BDDs/ZDDs

We can build up complex BDDs/ZDDs for various functions and sets by combinations of their rich algebraic operations such as Boolean operations and family algebra [3]. They use divide-and-conquer scheme based on the Shannon expansion, which is accelerated

by the memo cache that avoids recomputation of the same subproblems. The following algorithm outlines a typical depth-first implementation of binary operations:

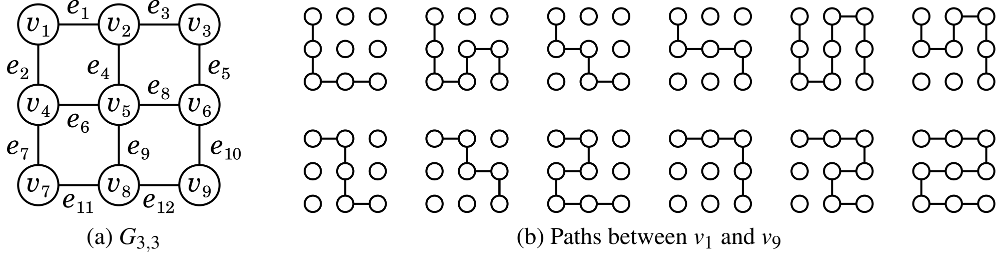
DF_BINARYOPERATION($\diamond, f, g$)
1: **if** $f \diamond g$ has a terminal value, **return** it;
2: **if** $f \diamond g = h$ is in the memo cache, **return** h;
3: $x \leftarrow$ the top variable of f and g;
4: $h_0 \leftarrow$ DF_BINARYOPERATION$(op, f|_{x=0}, g|_{x=0})$;
5: $h_1 \leftarrow$ DF_BINARYOPERATION$(op, f|_{x=1}, g|_{x=1})$;
6: $h \leftarrow (\overline{x}\ ?\ h_0 : h_1)$;
7: apply reduction rules to h;
8: put $f \diamond g = h$ into the memo cache;
9: **return** h.

This algorithm constructs a reduced diagram recursively from the bottom to the top.

Binary operations on BDDs/ZDDs can be implemented also in a breadth-first manner. We create a new node immediately after dividing the problem; the new node is incomplete as its descendants are not determined yet. The algorithm is outlined as follows:

BF_BINARYOPERATION($\diamond, f, g$)
1: let i_0 be the top index of f and g;
2: create a new node h and label it as $\langle i_0, f, g\rangle$;
3: **for** $i = i_0$ to n **do**
4:     **for all** node r labeled $\langle i, p, q\rangle$ **do**
5:         **for all** $b \in \{0, 1\}$ **do**
6:             $p' \leftarrow p|_{x_i=b}$; $q' \leftarrow q|_{x_i=b}$;
7:             **if** $p' \diamond q'$ has a terminal value **then**
8:                 set it to the b-child of r;
9:             **else**
10:                 $i' \leftarrow$ the top index of p' and q';
11:                 find or create node r' labeled $\langle i', p', q'\rangle$;
12:                 set r' to the b-child of r;
13:             **end if**
14:         **end for**
15:     **end for**
16: **end for**
17: **return** REDUCE(h).

This algorithm constructs a diagram from the top to the bottom. Incomplete nodes are labeled by its index and two operands of the subproblems, in order to share nodes for the same subproblems. The operand information of each node can be removed when its child nodes are fixed. Since the top-down phase does not fully reduce the diagram, the reduction algorithm is applied as a post-process.



(a) $G_{3,3}$ (b) Paths between v_1 and v_9

Figure 4: Path enumeration on $G_{3,3}$

2.3 Top-Down Construction

Single-pass BDD/ZDD construction from the root to the terminals, which we call *top-down construction* in this paper, is another way to build a complex BDD/ZDD structure. It is known that some important graph problems can be solved efficiently using such methods [11][12][3].

Node sharing must be performed on the fly during top-down construction in order to avoid explosion of the diagram. Multiple nonterminal nodes with the same index can be shared if and only if they take the same output values for all combinations of the rest of input values. Since this condition is not always easy to be determined on the fly, it is checked in a false-negative way by comparing the labels generated at some reasonable cost. They are so designed that multiple nonterminal nodes are equivalent if their labels are equivalent; the converse is not necessarily true because unshared nodes can be left for the final reduction phase.

Knuth introduced an interesting algorithm in his book, named SIMPATH, which constructs a ZDD representing a set of paths (ways to go from a point to another point without visiting any point twice) in an undirected graph [3][15]. For example, a 3×3 grid graph ($G_{3,3}$) in Figure 4a has 12 paths between v_1 and v_9 as shown in Figure 4b. The input to the algorithm is an undirected graph $G = (V,E)$ where $V = \{v_1,\dots,v_m\}$ is a set of vertices and $E = \{e_1,\dots,e_n\}$ is a set of edges. The output is a ZDD representing all the set of edges that form paths between v_1 and v_m.

In the SIMPATH algorithm, edge selections from $E = \{e_1,\dots,e_n\}$ are decided one by one in the order of indices. At each step of the algorithm, a set of selected edges represents path fragments and each vertex has one of the three states:

- not included in any path fragment,
- an endpoint of a path fragment,
- an intermediate point of a path fragment.

The label for a nonterminal node is defined to be $\langle i, mate \rangle$ where $1 \le i \le n$ and *mate* is a

partial map from V to $V \cup \{0\}$:

$$mate[v] = \begin{cases} v & \text{if vertex } v \text{ is untouched so far,} \\ u & \text{if vertices } u \text{ and } v \text{ are endpoints,} \\ 0 & \text{if vertex } v \text{ is an intermediate point.} \end{cases}$$

For simplicity of the algorithm, *mate* is maintained as if there were a built-in path between v_1 and v_m and we were enumerating all the virtual cycles that include it. The current set of selected edges is accepted when:

- a virtual cycle is formed and no other path fragment remains,

and it is rejected when:

- a virtual cycle is formed and some other path fragment remains, or
- an edge to an intermediate point is added, or
- the final chance to attach an edge to some endpoint is not taken.

In order to check the above conditions, we need *mate* entries only for *frontier*, which is a set of vertices contiguous with both decided and undecided edges. When vertex v is entering the frontier, a table entry $mate[v] = v$ is created except for $mate[v_1] = v_m$ and $mate[v_m] = v_1$. The entry for $mate[v]$ is deleted after v has left the frontier.

Figure 5 illustrates the result of SIMPATH for $G_{3,3}$, where the 0-terminal node is omitted and *mate* is drawn graphically on each node. Circles and lines represent vertices in the frontier and path fragments among them respectively. An isolated open circle represents a vertex not included in any path fragments ($mate[v] = v$). An isolated filled circle represents an intermediate point of some path fragment ($mate[v] = 0$). Note that the ZDD node deletion rule is only used at edges to the 1-terminal node.

3 Top-Down ZDD Construction Framework

In this section, we introduce a framework of implementing top-down construction algorithms for ZDDs. The basic idea is to define a common interface to ad hoc parts of the algorithms. Although we describe techniques for ZDDs hereafter, many of them are also applicable to BDDs.

3.1 Recursive Specifications

We define that a *configuration* is a node label used in a top-down construction algorithm, composed of a pair $\langle i, s \rangle$ of node index i ($1 \leq i \leq n$) and other information s. We assume that $\langle n+1, 0 \rangle$ and $\langle n+1, 1 \rangle$ are pre-defined configurations for the 0- and 1-terminals respectively.

A *recursive specification* of a ZDD is a definition of the following pair of functions:

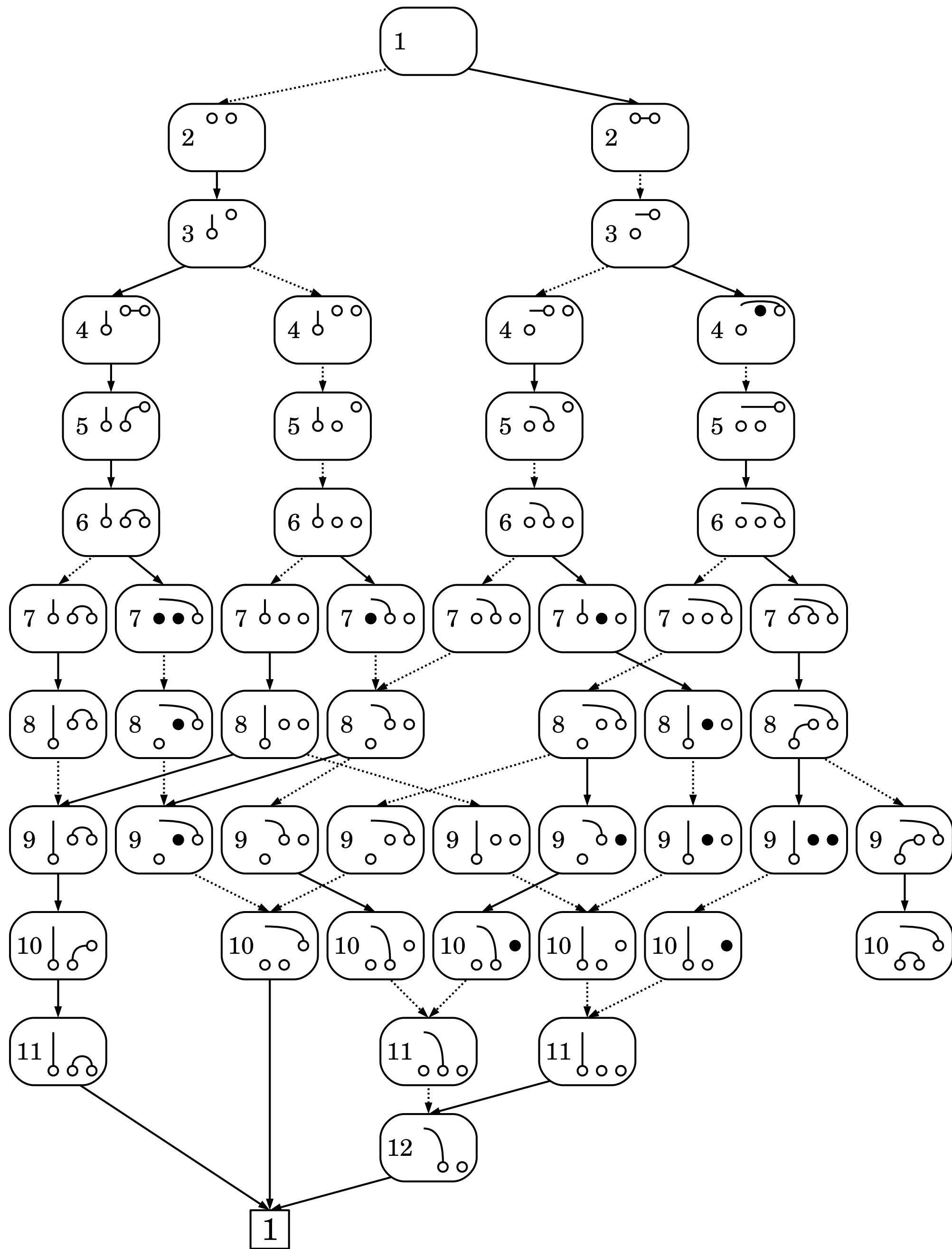


Figure 5: ZDD structure constructed by SIMPATH

- $\textsc{Root}()$ takes no argument and returns a *root configuration*, or a configuration of the root node;
- $\textsc{Child}(\langle i,s\rangle, b)$ takes configuration $\langle i,s\rangle$ of a node and branch $b \in \{0,1\}$ as arguments, and returns a new configuration for the b-child of the node.

A recursive specification can be viewed as a blueprint of a ZDD since it implicitly represents a unique diagram structure in a compact form. Many interesting top-down ZDD construction algorithms, including SIMPATH, can be adapted in this framework.

For example, let us consider a ZDD representing a family of all combinations of k items out of n items where node index i corresponds to the i-th item for $1 \le i \le n$. We define a set of configurations for nonterminal nodes as

$$\{\, \langle i,s\rangle \mid 1 \le i \le n,\ 0 \le s \le k \,\}$$

where i is the item index for the next decision and s is the number of items included before that node. The current item set is rejected immediately when $s > k$, and accepted when $s = k$ and no more undecided item remains. Its recursive specification $Comb_{n,k}$ is defined as follows:

$Comb_{n,k}.\textsc{Root}()$
1: **return** $\langle 1,0\rangle$;

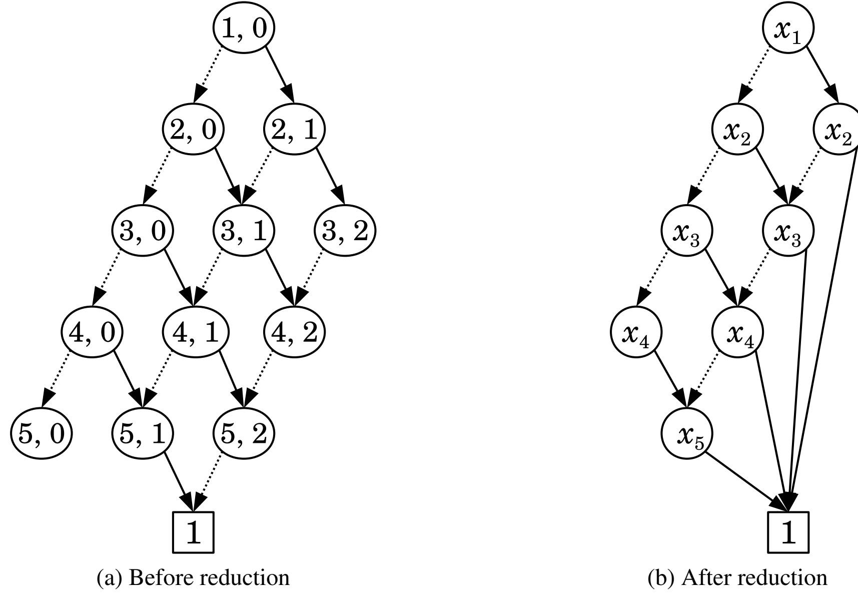
$Comb_{n,k}.\textsc{Child}(\langle i,s\rangle, b)$
1: $s \leftarrow s+b$;
2: **if** $i=n$ **and** $s=k$, **return** $\langle n+1,1\rangle$; // $\boxed{1}$
3: **if** $i=n$ **or** $s>k$, **return** $\langle n+1,0\rangle$; // $\boxed{0}$
4: **return** $\langle i+1,s\rangle$.

Figure 6 shows the ZDD specified by $Comb_{5,2}$ before and after reduction. In this case, it is not very difficult to define the recursive specification that directly represents the reduced ZDD structure:

$Comb'_{n,k}.\textsc{Root}()$
1: **return** $\langle 1,0\rangle$;

$Comb'_{n,k}.\textsc{Child}(\langle i,s\rangle, b)$
1: $s \leftarrow s+b$;
2: **if** $s=k$, **return** $\langle n+1,1\rangle$; // $\boxed{1}$
3: **if** $s+n-i<k$, **return** $\langle n+1,0\rangle$; // $\boxed{0}$
4: **return** $\langle i+1,s\rangle$.

The current item set is rejected as soon as we find that the remaining items are too few to make the k-combination. It is accepted as soon as $s=k$ is satisfied without taking all the remaining items.



(a) Before reduction (b) After reduction

Figure 6: ZDD structure for $Comb_{5,2}$

3.2 General Top-Down ZDD Construction Algorithm

Provided that the recursive specifications are given, the top-down ZDD construction can be processed by a common algorithm shown below; let S be a recursive specification and n be the number of its input variables:

CONSTRUCT(S)

1: $\langle i_0, s_0 \rangle \leftarrow S$.ROOT();
2: create a new node r and label it as $\langle i_0, s_0 \rangle$;
3: **for** $i = i_0$ to n **do**
4: **for all** node p labeled $\langle i, s \rangle$ for some s **do**
5: **for all** $b \in \{0, 1\}$ **do**
6: $\langle i', s' \rangle \leftarrow S$.CHILD($\langle i, s \rangle, b$);
7: **if** $\langle i', s' \rangle$ corresponds to a terminal node **then**
8: set it to the b-child of p;
9: **else**
10: find or create node p' labeled $\langle i', s' \rangle$;
11: set p' to the b-child of p;
12: **end if**
13: **end for**
14: **end for**
15: **end for**
16: **return** REDUCE(r).

This algorithm searches all reachable configurations of S from the root to the terminals in a breadth-first manner. Hash tables can be used to ensure one-to-one correspondence

between configurations and ZDD nodes. Their entries should be disposed properly because memory size for a configuration might be much larger than that for a ZDD node. Assuming that the hash table operations and evaluations of the recursive specification are constant time operations, this algorithm runs in linear time against the number of reachable configurations.

3.3 Parallelizing the Construction Algorithm

We can parallelize the loop at lines 4–14 of CONSTRUCT based on the fact that the tasks are independent except for the hash table operations at line 10. One can use thread-safe hash table for this purpose; or can divide the hash table into the multiple ones that manage disjoint subsets of possible configurations. The following algorithm makes use of the latter idea:

```
PARCONSTRUCT(S)
 1: ⟨i_0, s_0⟩ ← S.ROOT();
 2: let d be a dummy node;
 3: insert ⟨s_0, d, 0⟩ to bucket[i_0][1];
 4: for i = i_0 to n do
 5:   for all k ∈ {1,...,m} do     // in parallel
 6:     for all ⟨s, p̂, b̂⟩ ∈ bucket[i][k] do
 7:       find or create node p labeled ⟨i, s⟩;
 8:       set p to the b̂-child of p̂;
 9:       if p is newly created then
10:         for all b ∈ {0,1} do
11:           ⟨i', s'⟩ ← S.CHILD(⟨i, s⟩, b);
12:           if ⟨i', s'⟩ corresponds to a terminal node then
13:             set it to the b-child of p;
14:           else
15:             k' ← bucket number for ⟨i', s'⟩;
16:             insert ⟨s', p, b⟩ to bucket[i'][k'];
17:           end if
18:         end for
19:       end if
20:     end for
21:   end for
22: end for
23: let r be the 0-child of d;
24: return REDUCE(r).
```

In the above algorithm, configurations are grouped into m buckets; $bucket[i][k]$ works as a task queue for node index $i \in \{1,\dots,n\}$ and bucket number $k \in \{1,\dots,m\}$. Since tasks for the same configurations are always stored in the same bucket, different buckets can be processed in parallel without caring about thread-safeness of the hash tables. The number

of buckets m should be larger enough than the number of parallel threads for better load balancing, and should be smaller enough than the average number of nodes for each index for less overhead costs. The reduction algorithm in 2.1 also can be parallelized in similar ways.

4 Operations on Recursive Specifications

4.1 Lookahead

In general, any reduced/unreduced ZDD can be represented by a recursive specification; the index might be increased by more than one in the CHILD function when we take the zero-suppress rule aggressively into account. It improves the performance of ZDD construction, while it may worsen simplicity of the recursive specification. It would be pleased if an optimized specification can be generated automatically from an easy-to-understand description for humans.

The *lookahead* operation wraps a given recursive specification and makes the one that represents a smaller and logically equivalent ZDD. It skips redundant configurations of the original specification in terms of the zero-suppress rule.

LOOKAHEAD(S).ROOT()
1: **return** S.ROOT();

LOOKAHEAD(S).CHILD($\langle i,s\rangle, b$)
1: $\langle i',s'\rangle \leftarrow S$.CHILD($\langle i,s\rangle, b$);
2: **while** $i' \leq n$ **and** S.CHILD($\langle i',s'\rangle, 1) = \langle n+1, 0\rangle$ **do**
3: $\quad\langle i',s'\rangle \leftarrow S$.CHILD($\langle i',s'\rangle, 0$);
4: **end while**
5: **return** $\langle i',s'\rangle$.

Figure 7 shows the result of SIMPATH for $G_{3,3}$ combined with the lookahead. In comparison with the original result (Figure 5), the number of nonterminal nodes is reduced from 52 to 29. This example also shows that the lookahead operation do not always remove all redundant nodes, because they do not care the node sharing and do not backtrack for the node deletion.

4.2 Composition

Let us suppose that there are two properties represented by recursive specifications and we want to compute the ZDD that represents the intersection of the properties. It is achieved easily by constructing two ZDDs from the specifications and by applying the intersection operation on ZDDs.

In this section, we present an alternative to this approach, in which we first composite the two specifications and then construct a ZDD. It has the advantage of robustness when an intermediate ZDD may blow up while the final ZDD should be compact.

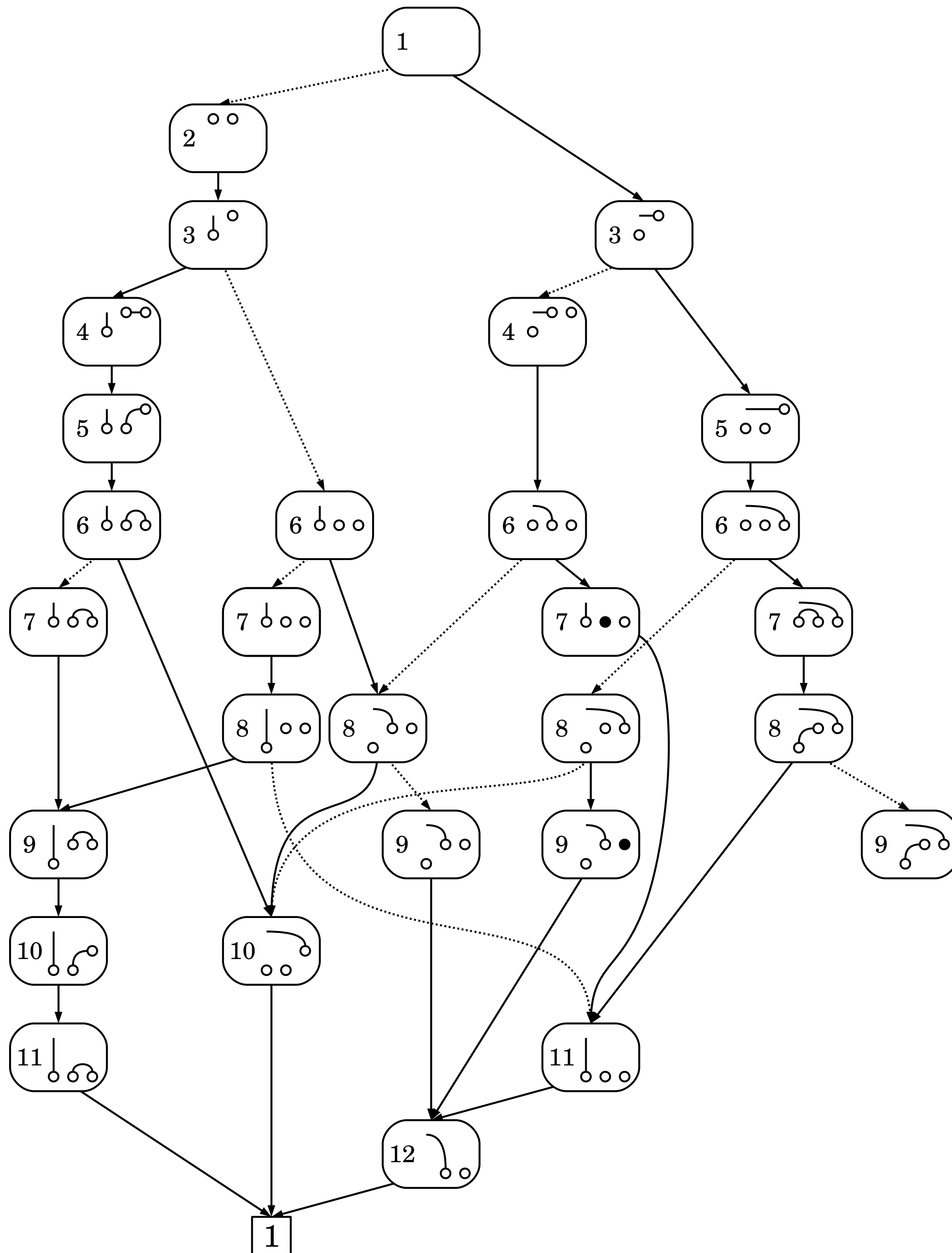


Figure 7: ZDD constructed by SIMPATH with the lookahead

Let S and T be recursive specifications and "$\diamond$" be some binary operator such as "$\vee$" or "$\wedge$". Here we consider a binary operation on S and T, namely $S \diamond T$, such that

$$\textsc{Construct}(S \diamond T) \;=\; \textsc{Construct}(S) \diamond \textsc{Construct}(T)\,.$$

It can be defined as follows:

$(S \diamond T).\textsc{Root}()$

1: $\langle i,s\rangle \leftarrow S.\textsc{Root}();$

2: $\langle j,t\rangle \leftarrow T.\textsc{Root}();$

3: **return** $\begin{cases} \langle n+1, s \diamond t\rangle & \text{if } i = j = n+1, \\ \langle \min(i,j), \langle i,s,j,t\rangle\rangle & \text{otherwise.} \end{cases}$

$(S \diamond T).\textsc{Child}(\langle k, \langle i,s,j,t\rangle\rangle, b)$ // $k = \min(i,j)$

1: $\langle i,s\rangle \leftarrow \begin{cases} \langle i,s\rangle & \text{if } k < i \text{ and } b = 0, \\ \langle n+1, 0\rangle & \text{if } k < i \text{ and } b = 1, \\ S.\textsc{Child}(\langle i,s\rangle, b) & \text{otherwise;} \end{cases}$

2: $\langle j,t\rangle \leftarrow \begin{cases} \langle j,t\rangle & \text{if } k < j \text{ and } b = 0, \\ \langle n+1, 0\rangle & \text{if } k < j \text{ and } b = 1, \\ S.\textsc{Child}(\langle j,t\rangle, b) & \text{otherwise;} \end{cases}$

3: **return** $\begin{cases} \langle n+1, s \diamond t\rangle & \text{if } i = j = n+1, \\ \langle \min(i,j), \langle i,s,j,t\rangle\rangle & \text{otherwise.} \end{cases}$

In case the operation is set intersection, or logical AND when the ZDDs are interpreted as Boolean functions, we can optimize it by taking more advantage of the zero-suppress rule:

$(S \cap T).\textsc{Root}()$

1: **return** $(S \cap T).\textsc{Sync}(S.\textsc{Root}(), T.\textsc{Root}()).$

$(S \cap T).\textsc{Child}(\langle i, \langle s,t\rangle\rangle, b)$

1: **return** $(S \cap T).\textsc{Sync}(S.\textsc{Child}(\langle i,s\rangle, b), T.\textsc{Child}(\langle i,t\rangle, b)).$

$(S \cap T).\textsc{Sync}(\langle i,s\rangle, \langle j,t\rangle)$

1: **while** $i \neq j$ **do**

2: **if** $i < j$, $\langle i,s\rangle \leftarrow S.\textsc{Child}(\langle i,s\rangle, 0);$

3: **if** $j < i$, $\langle j,t\rangle \leftarrow T.\textsc{Child}(\langle j,t\rangle, 0);$

4: **end while**

5: **if** $i = n+1$, **return** $\langle n+1, s \wedge t\rangle;$

6: **return** $\langle i, \langle s,t\rangle\rangle.$

Lines 1–4 of the SYNC subroutine skips the nodes that would have 1-edges to the 0-terminal. It can be decided easily by checking if configurations derived from S and T have different index numbers. We can go down through 0-edges until the indices are synchronized. It is an interesting property of the combination of intersection operation and zero-suppress rule.

4.3 Wrapping and subsetting

We can wrap an existing ZDD structure in a recursive specification. The wrapper of ZDD f is given as follows:

$\textsc{Wrap}(f).\textsc{Root}()$
1: $i \leftarrow$ the top index of f;
2: **return** $\langle i, f \rangle$;

$\textsc{Wrap}(f).\textsc{Child}(\langle i, f \rangle, b)$
1: $f' \leftarrow$ the b-child of f;
2: $i' \leftarrow$ the top index of f';
3: **return** $\langle i', f' \rangle$.

The same ZDD structure as f can be derived from $\textsc{Wrap}(f)$, that is:

$$\textsc{Construct}(\textsc{Wrap}(f)) = f \,.$$

The wrapping technique extends the usefulness of the operations on recursive specifications. Let us suppose the situation where we have some ZDD f and want to restrict it by a property represented by specification S. It can be computed as usual by the intersection operation on ZDDs:

$$f \cap \textsc{Construct}(S) \,.$$

Using the wrapping technique, we can also compute it as the intersection on specifications:

$$\textsc{Construct}(\textsc{Wrap}(f) \cap S) \,.$$

We call it *subsetting* technique on top-down ZDD construction, which is an alternative that is worth trying when Construct(S) becomes the bottleneck in the usual method.

5 Experimental Results

Our top-down ZDD construction framework is implemented in C++. We measured single-threaded performance of the algorithms on 2.67GHz Intel Xeon E7-8837 CPU with 1.5TB memory running 64-bit SUSE Linux Enterprise Server 11.

5.1 Path Enumeration

First, we evaluated the efficiency of our framework in comparison with the original SIMPATH implementation [15], and measured improvements achieved by the lookahead and subsetting techniques.

We experimented with graph examples listed in Table 1, where m is the number of vertices, n is the number of edges, and #path is the number of paths to be enumerated (paths between v_1 and v_m). We used complete graphs (K_m), triangular grid graphs ($T_{\alpha,\beta}$),

Table 1: Characteristics of graph examples

Graph	m	n	#path	SIMPATH (sec.) Main	Reduce
K_{17}	17	136	3.55×10^{12}	16.45	18.27
K_{18}	18	153	5.69×10^{13}	57.45	59.70
K_{19}	19	171	9.67×10^{14}	183.98	194.77
K_{20}	20	190	1.74×10^{16}	641.52	662.34
K_{21}	21	210	3.31×10^{17}	2106.68	2344.67
K_{22}	22	231	6.61×10^{18}	7201.05	7939.68
$T_{11,11}$	121	320	4.35×10^{39}	5.58	7.20
$T_{12,12}$	144	385	6.81×10^{47}	27.39	30.48
$T_{13,13}$	169	456	6.33×10^{56}	115.19	124.89
$T_{14,14}$	196	533	3.50×10^{66}	504.95	522.68
$T_{15,15}$	225	616	1.15×10^{77}	2168.47	2260.38
$T_{16,16}$	256	705	2.24×10^{88}	8868.41	9218.46
$G_{13,13}$	169	312	6.45×10^{34}	12.42	14.94
$G_{14,14}$	196	364	6.95×10^{40}	46.23	50.40
$G_{15,15}$	225	420	2.27×10^{47}	152.77	161.76
$G_{16,16}$	256	480	2.27×10^{54}	503.63	534.77
$G_{17,17}$	289	544	6.87×10^{61}	1644.86	1826.86
$G_{18,18}$	324	612	6.34×10^{69}	5598.11	5912.85
$H_{22,23}$	506	726	2.20×10^{61}	10.91	11.90
$H_{24,25}$	600	864	4.90×10^{73}	37.77	39.70
$H_{26,27}$	702	1014	1.50×10^{87}	136.18	129.32
$H_{28,29}$	812	1176	6.28×10^{101}	452.25	434.77
$H_{30,31}$	930	1350	3.61×10^{117}	1531.62	1486.30
$H_{32,33}$	1056	1536	2.85×10^{134}	4935.35	4864.38

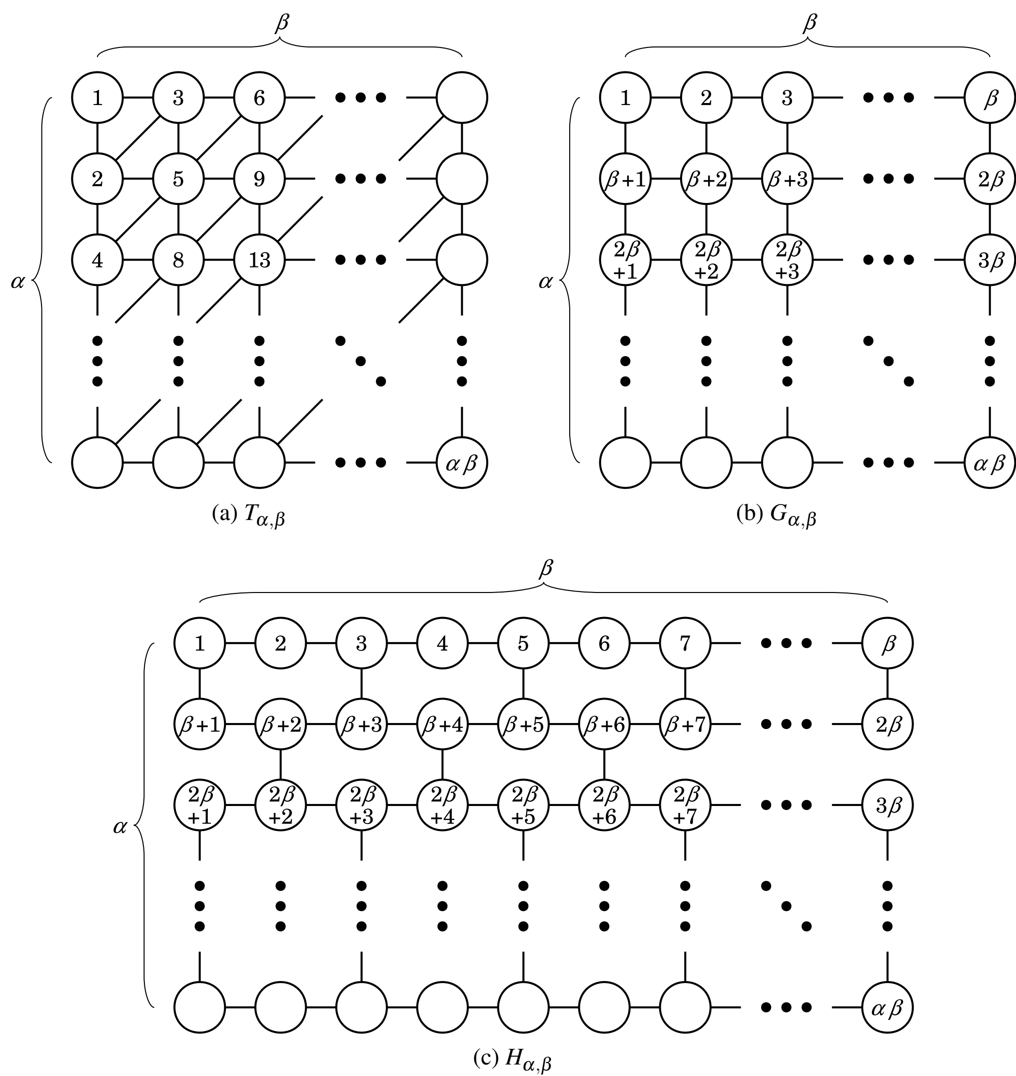


Figure 8: Grid graphs and their vertex order

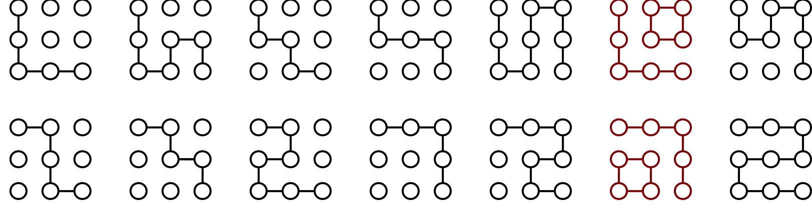


Figure 9: Sets of graph edges represented by $Degree(G_{3,3}, 1, 9)$

square grid graphs ($G_{\alpha,\beta}$), and hexagonal grid graphs ($H_{\alpha,\beta}$) as benchmark examples. The grid graphs and their vertex ordering rules are shown in Figure 8. The edge order (ZDD variable order) is defined lexicographically with the vertex order: $\{v_i, v_j\} \leq \{v_{i'}, v_{j'}\}$ if and only if $v_i < v_{i'}$ or $(v_i = v_{i'}$ and $v_j \leq v_{j'})$ where $v_i \leq v_j$ and $v_{i'} \leq v_{j'}$. We have tested some simple vertex ordering rules and have chosen the one that makes final ZDDs compact.

Table 1 also includes CPU time for the original SIMPATH implementation,[1] which is composed of the main program and the ZDD reduction program. The columns "Main" and "Reduce" show their CPU time in seconds. Note that they hand over an unreduced ZDD via text file, while our implementation performs both ZDD construction and reduction on memory.

We wrote the recursive specification $Path(G, v_1, v_m)$ that corresponds to the SIMPATH algorithm. The parameter $G = (V, E)$ is a target graph where $V = \{v_1, \ldots, v_m\}$ is a set of vertices and $E = \{e_1, \ldots, e_n\}$ is a set of edges. We also wrote the recursive specification $Degree(G, v_1, v_m)$ that represents constraints on vertex degrees (number of edges incident to a vertex). If $E' \subseteq E$ forms a path between vertices v_1 and v_m, vertices in graph $G' = (V, E')$ must have a degree of 0 or 2 except that v_1 and v_m must have a degree of 1. That condition is necessary but not sufficient for the set of edges to form a path. For example, Figure 9 shows the 14 instances represented by $Degree(G_{3,3}, 1, 9)$, which include the 2 instances that do not actually form paths.

We compared a basic one-pass method (1P), that with lookahead (1P+L), a two-pass subsetting method (2P), and that with lookahead (2P+L). The four methods are defined as follows:

1P $f \leftarrow \text{CONSTRUCT}(Path(G, v_1, v_m));$

1P+L $f \leftarrow \text{CONSTRUCT}(\text{LOOKAHEAD}(Path(G, v_1, v_m)));$

2P $g \leftarrow \text{CONSTRUCT}(Degree(G, v_1, v_m));$
$f \leftarrow \text{CONSTRUCT}(\text{WRAP}(g) \cap Path(G, v_1, v_m));$

2P+L $g \leftarrow \text{CONSTRUCT}(\text{LOOKAHEAD}(Degree(G, v_1, v_m)));$
$f \leftarrow \text{CONSTRUCT}(\text{WRAP}(g) \cap \text{LOOKAHEAD}(Path(G, v_1, v_m))).$

Table 2 shows CPU time of the four methods. In comparison with the original SIMPATH implementation (Table 1), our implementation 1P looks reasonably fast. It shows

Table 2: CPU time for path enumeration (sec.)

Graph	1P	1P+L	2P	2P+L
K_{17}	18.68	13.86	11.06	11.09
K_{18}	62.50	47.69	33.10	33.93
K_{19}	210.82	169.54	103.84	109.43
K_{20}	776.18	586.84	318.73	322.96
K_{21}	2573.39	2083.32	1096.70	1024.71
K_{22}	9683.19	7068.47	3839.62	3856.21
$T_{11,11}$	5.32	3.44	4.66	4.86
$T_{12,12}$	26.78	17.11	21.04	21.27
$T_{13,13}$	115.23	78.26	84.69	85.66
$T_{14,14}$	555.83	340.44	372.78	334.79
$T_{15,15}$	2121.35	1610.42	1355.35	1357.32
$T_{16,16}$	11185.38	6587.02	5499.10	5644.78
$G_{13,13}$	10.53	6.53	6.13	5.71
$G_{14,14}$	41.09	25.31	20.08	18.76
$G_{15,15}$	159.97	91.54	64.80	60.34
$G_{16,16}$	501.28	320.46	211.42	193.67
$G_{17,17}$	1693.46	1067.76	681.77	636.42
$G_{18,18}$	6398.75	3664.18	2296.25	2274.09
$H_{22,23}$	6.07	3.63	3.94	3.47
$H_{24,25}$	32.98	18.29	16.37	14.34
$H_{26,27}$	124.76	70.37	55.55	47.81
$H_{28,29}$	454.18	268.33	183.16	165.88
$H_{30,31}$	1542.16	953.21	610.25	571.67
$H_{32,33}$	5680.29	3275.21	2078.65	1934.34

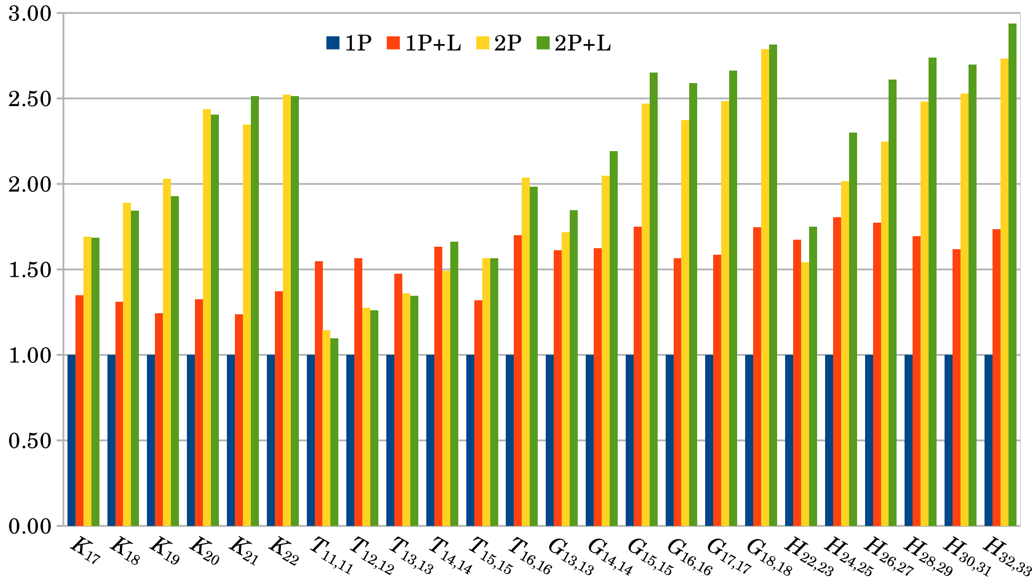


Figure 10: Speed ratio for path enumeration

that there is no noticeable overhead in our top-down construction framework. The lookahead and subsetting techniques accelerated top-down ZDD construction by a factor of 1.1 to 2.9. Figure 10 summarizes computation speed of methods 1P+L, 2P, and 2P+L relative to the basic method 1P. While the fastest method is dependent on the example, we can read that the subsetting technique is relatively effective in large examples.

Construction of ZDD g dominates CPU time in 2P and 2P+L because g is much smaller than f. Table 3 compares ZDD size (the number of nonterminal nodes) of g and f. The larger the graph is, the larger the size difference between g and f becomes. "Peak size" is the ZDD size just before the reduction phase, which indicates the number of iterations run in both construction and reduction phases. The lookahead and subsetting techniques have effect to reduce 40 to 50 percent of the peak size. It is interesting that methods 1P+L and 2P does not show much difference in the peak size of f, even though the node deletion rule is not checked explicitly in 2P. It means that the zero-suppress information was effectively inherited from g in the subsetting method.

The total memory usage in megabytes is shown in Table 4. It is confirmed that the lookahead and subsetting techniques are also effective to reduce memory usage.

5.2 Numberlink and Slitherlink Puzzles

Secondly, we have improved the ZDD-based solvers of Numberlink and Slitherlink introduced in [13] using the lookahead and subsetting techniques. Examples of Numberlink and Slitherlink are shown in Figure 11 and Figure 12 respectively. They are logic puzzles that involve finding the paths or the cycle that satisfy given local and global properties

[1] We have slightly modified the programs in order to process larger input graphs on a 64-bit machine.

Table 3: ZDD size in path enumeration

	g			f				
	Peak size		Final size	Peak size				Final size
Graph	2P	2P+L		1P	1P+L	2P	2P+L	
K_{17}	3,383,182	1,446,371	1,415,798	31,687,586	16,776,941	17,913,664	16,685,440	15,469,186
K_{18}	7,803,921	3,331,561	3,266,139	98,595,128	52,894,116	56,302,697	52,644,447	48,935,273
K_{19}	17,916,021	7,638,950	7,498,891	309,329,033	168,011,957	178,326,054	167,320,943	155,922,881
K_{20}	40,960,634	17,445,357	17,144,913	978,702,177	537,803,391	569,319,129	535,863,645	500,559,700
K_{21}	93,303,788	39,699,692	39,055,059	3,122,714,316	1,734,809,580	1,832,020,786	1,729,287,871	1,619,050,484
K_{22}	211,843,795	90,058,256	88,674,234	10,047,097,379	5,639,089,096	5,941,791,248	5,623,154,813	5,276,150,643
$T_{11,11}$	3,454,359	1,678,273	1,534,383	13,365,043	6,909,231	7,648,691	6,909,231	6,432,417
$T_{12,12}$	11,445,656	5,558,016	5,075,298	53,816,252	27,977,245	30,916,232	27,977,245	26,076,799
$T_{13,13}$	37,584,679	18,243,715	16,642,341	215,875,876	112,786,898	124,439,778	112,786,898	105,238,888
$T_{14,14}$	122,497,140	59,440,240	54,176,364	863,508,297	453,159,395	499,288,002	453,159,395	423,254,393
$T_{15,15}$	396,720,695	192,448,119	175,277,115	3,446,706,536	1,816,007,604	1,998,415,097	1,816,007,604	1,697,726,218
$T_{16,16}$	1,277,849,872	619,726,690	564,075,414	13,735,340,349	7,262,868,868	7,983,662,545	7,262,868,868	6,795,583,172
$G_{13,13}$	1,952,762	983,037	971,773	26,894,640	15,032,057	16,178,631	15,032,057	13,803,430
$G_{14,14}$	4,599,802	2,314,237	2,289,661	86,698,791	48,641,299	52,307,691	48,641,299	44,871,856
$G_{15,15}$	10,702,842	5,382,141	5,328,893	277,581,568	156,253,978	167,908,407	156,253,978	144,759,636
$G_{16,16}$	24,641,530	12,386,301	12,271,613	883,640,711	498,888,415	535,749,877	498,888,415	464,004,180
$G_{17,17}$	56,213,498	28,246,013	28,000,253	2,799,256,918	1,584,605,112	1,700,699,101	1,584,605,112	1,479,128,501
$G_{18,18}$	127,205,370	63,897,597	63,373,309	8,830,604,856	5,010,748,938	5,375,051,545	5,010,748,938	4,692,765,814
$H_{22,23}$	1,895,414	1,004,539	897,019	20,985,221	12,431,317	13,117,268	12,431,317	10,686,910
$H_{24,25}$	4,548,598	2,410,491	2,151,419	68,690,969	40,853,448	43,081,787	40,853,448	35,229,328
$H_{26,27}$	10,751,990	5,697,531	5,083,131	222,730,862	132,929,717	140,105,957	132,929,717	114,956,610
$H_{28,29}$	25,092,086	13,295,611	11,857,915	716,615,275	429,006,718	451,953,721	429,006,718	371,973,561
$H_{30,31}$	57,917,430	30,687,227	27,361,275	2,290,741,210	1,375,126,756	1,448,070,796	1,375,126,756	1,195,179,926
$H_{32,33}$	132,415,478	70,156,283	62,537,723	7,282,606,658	4,382,454,784	4,613,178,936	4,382,454,784	3,817,373,513

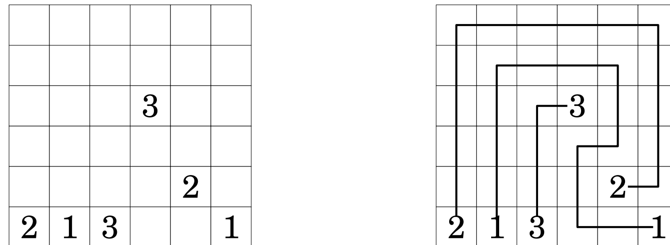


Figure 11: Numberlink problem and its solution

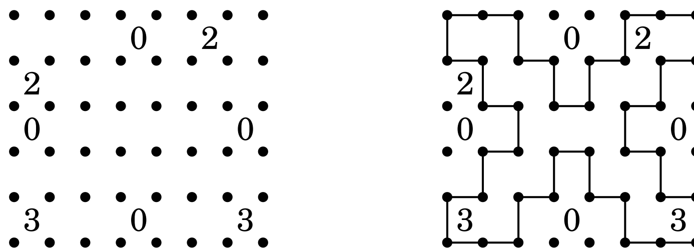


Figure 12: Slitherlink problem and its solution

Table 4: Memory usage for path enumeration (MB)

Graph	1P	1P+L	2P	2P+L
K_{17}	891	806	986	951
K_{18}	2,701	2,717	3,447	3,187
K_{19}	8,204	8,293	10,831	10,250
K_{20}	26,173	26,275	33,392	32,031
K_{21}	82,395	84,297	112,346	108,139
K_{22}	254,359	272,999	367,226	357,666
$T_{11,11}$	327	194	249	231
$T_{12,12}$	1,273	713	905	820
$T_{13,13}$	5,059	2,770	3,616	3,121
$T_{14,14}$	20,075	10,855	13,202	12,086
$T_{15,15}$	79,987	43,862	53,101	49,570
$T_{16,16}$	318,664	184,504	224,682	205,607
$G_{13,13}$	786	402	412	388
$G_{14,14}$	2,277	1,212	1,287	1,202
$G_{15,15}$	6,532	3,758	4,079	3,802
$G_{16,16}$	20,721	11,927	13,017	12,125
$G_{17,17}$	65,952	37,686	41,235	38,419
$G_{18,18}$	206,430	118,760	129,699	120,904
$H_{22,23}$	505	322	333	319
$H_{24,25}$	1,652	990	1,045	986
$H_{26,27}$	5,464	3,155	3,318	3,151
$H_{28,29}$	16,639	10,422	10,640	10,081
$H_{30,31}$	53,028	32,016	33,912	32,215
$H_{32,33}$	168,351	101,819	107,812	102,273

Table 5: Characteristics of Numberlink problems

Name	Graph	m	n	Description
BN64	$G_{10,10}$	100	180	64th problem in [17]
BN79	$G_{10,10}$	100	180	79th problem in [17]
BN85	$G_{20,15}$	300	565	85th problem in [17]
BN99	$G_{20,15}$	300	565	99th problem in [17]
C108	$G_{36,20}$	720	1384	Vol. 108 in [18]

[16]. These experiments show case studies of designing efficient ZDD construction procedures on our framework.

5.2.1 Numberlink Solver

Numberlink is played on a grid with the following rules.

1. Connect pairs of the same hint numbers with a continuous line.
2. Lines go through the center of the cells, horizontally, vertically, or changing direction, and never twice through the same cell.
3. Lines cannot cross, branch off, or go through the cells with hint numbers.
4. Lines must cover all the cells.

It can be viewed as a problem of finding a path matching on grid graph G under hint h where each cell corresponds to a vertex of G. It can be solved by an algorithm derived from SIMPATH [13]. The output is the ZDD that represents the set of all solutions, which must be a singleton if the problem is designed correctly.

We have experimented on Numberlink problems listed in Table 5. In the same way as experiments in the previous section, we wrote the main algorithm as recursive specification $\mathit{Numlin}(G,h)$, which gives the exact solutions, and also wrote the constraints on vertex degrees (1 for vertices with hint numbers and 2 for others) as another recursive specification $\mathit{Degree}(G,h)$, which gives a superset of the solutions. We compared a basic one-pass method (1P), that with lookahead (1P+L), a two-pass subsetting method (2P), and that with lookahead (2P+L). The four methods are defined as follows:

1P $f \leftarrow \textsc{Construct}(\mathit{Numlin}(G,h))$;

1P+L $f \leftarrow \textsc{Construct}(\textsc{Lookahead}(\mathit{Numlin}(G,h)))$;

2P $g \leftarrow \textsc{Construct}(\mathit{Degree}(G,h))$;
$f \leftarrow \textsc{Construct}(\textsc{Wrap}(g) \cap \mathit{Numlin}(G,h))$;

2P+L $g \leftarrow \textsc{Construct}(\textsc{Lookahead}(\mathit{Degree}(G,h)))$;
$f \leftarrow \textsc{Construct}(\textsc{Wrap}(g) \cap \textsc{Lookahead}(\mathit{Numlin}(G,h)))$.

Table 6: CPU time for solving Numberlink puzzles (sec.)

Name	1P	1P+L	2P	2P+L
BN64	0.02	0.01	0.02	0.02
BN79	0.03	0.02	0.03	0.03
BN85	72.69	42.19	32.93	27.14
BN99	79.95	42.47	38.72	28.97
C108	10092.48	5546.26	4653.71	3309.26

Table 7: Memory usage for solving Numberlink puzzles (MB)

Name	1P	1P+L	2P	2P+L
BN64	4	3	10	9
BN79	9	6	20	14
BN85	8,010	4,741	6,118	4,561
BN99	8,907	5,073	6,965	5,013
C108	817,969	467,968	644,967	474,646

In the methods 2P and 2P+L, g is used as a search space for the solutions.

Table 6 and Table 7 show CPU time in seconds and memory usage in megabytes respectively. The lookahead and subsetting techniques were effective for the Numberlink solvers to reduce CPU time and memory usage. Method 2P+L was the fastest and was about three times as fast as the basic method 1L.

5.2.2 Slitherlink Solver

Slitherlink is played on a grid of dots with the following rules.

1. Connect adjacent dots with vertical or horizontal lines.
2. A single loop is formed with no crossing or branch.
3. Each hint cell indicates the number of lines surrounding it, while empty cells may be surrounded by any number of lines.

It can be viewed as a problem of finding a cycle on grid graph G under hint h where each dot corresponds to a vertex of G. It can be solved by an algorithm also derived from SIMPATH [13]. The output is the ZDD that represents the set of all solutions, which must be a singleton if the problem is designed correctly.

We have experimented on Slitherlink problems listed in Table 8. We made three recursive specifications: $\mathit{Cycle}(G)$ for enumerating all cycles in G, $\mathit{Hint}(G,h)$ for the constraints defined by the hints, and $\mathit{Degree}(G)$ for the constraints on vertex degrees (0 or 2 for all vertices). Intersection of $\mathit{Cycle}(G)$ and $\mathit{Hint}(G,h)$ gives the solution, while $\mathit{Degree}(G)$ is expected to be an extra guide to get the solution. ZDD for $A_{\mathit{Cycle}}(G)$ could not be

Table 8: Characteristics of Slitherlink problems

Name	Graph	m	n	Description
BS68	$G_{25,15}$	375	710	68th problem in [19]
BS77	$G_{25,15}$	375	710	77th problem in [19]
BS89	$G_{37,21}$	777	1496	89th problem in [19]
BS96	$G_{37,21}$	777	1496	96th problem in [19]
S10	$G_{37,21}$	777	1496	10th problem in [20]
C95	$G_{46,32}$	1472	2866	Vol. 95 in [18]
C103	$G_{46,32}$	1472	2866	Vol. 103 in [18]
C113	$G_{46,32}$	1472	2866	Vol. 113 in [18]

constructed because G is fairly large in the Slitherlink problems. We compared a basic one-pass method (1P), that with lookahead (1P+L), a two-pass subsetting method (2P), that with lookahead (2P+L), a three-pass subsetting method (3P), and that with lookahead (3P+L), using $\mathit{Hint}(G,h)$ as the start points of the subsetting methods. The six methods are defined as follows:

1P $f \leftarrow \textsc{Construct}(\mathit{Hint}(G,h) \cap \mathit{Cycle}(G));$

1P+L $f \leftarrow \textsc{Construct}(\textsc{Lookahead}(\mathit{Hint}(G,h) \cap \mathit{Cycle}(G)));$

2P $g \leftarrow \textsc{Construct}(\mathit{Hint}(G,h));$
$f \leftarrow \textsc{Construct}(\textsc{Wrap}(g) \cap \mathit{Cycle}(G));$

2P+L $g \leftarrow \textsc{Construct}(\textsc{Lookahead}(\mathit{Hint}(G,h)));$
$f \leftarrow \textsc{Construct}(\textsc{Wrap}(g) \cap \textsc{Lookahead}(\mathit{Cycle}(G)));$

3P $g \leftarrow \textsc{Construct}(\mathit{Hint}(G,h));$
$g' \leftarrow \textsc{Construct}(\textsc{Wrap}(g) \cap \mathit{Degree}(G));$
$f \leftarrow \textsc{Construct}(\textsc{Wrap}(g') \cap \mathit{Cycle}(G));$

3P+L $g \leftarrow \textsc{Construct}(\textsc{Lookahead}(\mathit{Hint}(G,h)));$
$g' \leftarrow \textsc{Construct}(\textsc{Wrap}(g) \cap \textsc{Lookahead}(\mathit{Degree}(G)));$
$f \leftarrow \textsc{Construct}(\textsc{Wrap}(g') \cap \textsc{Lookahead}(\mathit{Cycle}(G)));$

Table 9 and Table 10 describe CPU time in seconds and memory usage in megabytes respectively. Method 2P+L was the fastest for most examples, while 3P+L was very efficient for C95. The results show that the lookahead and subsetting techniques were effective also in Slitherlink solvers and that it is not easy to find the best subsetting strategy before trial.

6 Conclusion

First, we proposed an efficient top-down ZDD construction framework for solving graph enumeration and indexing problems, in which the property is specified as a recursive spec-

Table 9: CPU time for solving Slitherlink puzzles (sec.)

Name	1P	1P+L	2P	2P+L	3P	3P+L
BS68	0.02	0.02	0.02	0.02	0.02	0.02
BS77	0.04	0.04	0.02	0.02	0.02	0.01
BS89	1.92	1.11	0.85	0.52	1.79	1.16
BS96	6.60	3.43	2.34	1.43	12.21	7.59
S10	25.92	12.75	7.94	4.19	10.15	6.06
C95	12095.40	5209.64	2394.51	1284.40	1304.61	740.11
C103	6456.14	2400.81	1434.15	808.77	7830.46	4385.51
C113	713.80	341.19	285.49	155.98	393.05	243.67

Table 10: Memory usage for solving Slitherlink puzzles (MB)

Name	1P	1P+L	2P	2P+L	3P	3P+L
BS68	7	7	17	12	19	14
BS77	12	8	24	17	18	13
BS89	340	173	430	301	689	540
BS96	817	426	761	552	2,517	1,570
S10	2,247	1,113	1,382	936	2,025	1,273
C95	624,616	318,325	211,630	124,468	149,335	89,381
C103	364,887	178,929	139,410	84,036	739,841	441,777
C113	44,478	21,776	32,714	19,231	56,104	33,072

ification.

The lookahead technique applies the zero-suppress rule on the fly, while conventional methods does not take it into account during the top-down construction phase. Binary operations on recursive specifications, which allow us to combine multiple properties without constructing ZDD structure for each property, can be a strong alternative to the conventional procedures that use operations on ZDDs. An improved algorithm for set intersection on recursive specifications also reduces CPU time and memory usage for constructing ZDDs by skipping redundant states in the top-down construction phase. The subsetting technique enables us to use an existing ZDD for restricting search space of top-down ZDD construction.

The experimental results confirmed that our techniques actually improve time and space for the top-down ZDD construction algorithms. The lookahead technique easily fits for any application and produces good results in most cases. We can improve it further by adding the subsetting technique especially for large examples, though it is not always easy to find the best strategy of subsetting.

References

[1] R. E. Bryant. “Graph-based Algorithms for Boolean Function Manipulation”. In: *IEEE Transactions on Computers* C-35.8 (1986), pp. 677–691.

[2] S. Minato. “Zero-suppressed BDDs for Set Manipulation in Combinatorial Problems”. In: *Proceedings of the 30th ACM/IEEE Design Automation Conference*. 1993, pp. 272–277.

[3] D. E. Knuth. *The Art of Computer Programming, Volume 4A, Combinatorial Algorithms, Part 1*. 1st. Addison-Wesley Professional, 2011. ISBN: 0321751043.

[4] K. S. Brace, R. L. Rudell, and R. E. Bryant. “Efficient Implementation of a BDD Package”. In: *Proceedings of the 27th ACM/IEEE Design Automation Conference*. 1990, pp. 40–45.

[5] S. Minato, N. Ishiura, and S. Yajima. “Shared Binary Decision Diagram with Attribute Edges for Efficient Boolean Function Manipulation”. In: *Proceedings of the 27th ACM/IEEE Design Automation Conference*. 1990, pp. 52–57.

[6] H. Ochi, K. Yasuoka, and S. Yajima. “Breadth-First Manipulation of Very Large Binary-Decision Diagrams”. In: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. 1993, pp. 48–55.

[7] R. Ashar and M. Cheong. “Efficient Breadth-First Manipulation of Binary Decision Diagrams”. In: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. 1994, pp. 622–627.

[8] J. V. Sanghavi, R. K. Ranjan, R. K. Brayton, and A. Sangiovanni-Vincentelli. “High Performance BDD Package by Exploiting Memory Hierarchy”. In: *Proceedings of the 33rd Design Automation Conference*. 1996, pp. 635–640.

[9] Y. Chen, B. Yang, and R. E. Bryant. *Breadth-First with Depth-First BDD Construction: A Hybrid Approach*. Tech. rep. CMU-CS-97-120. School of Computer Science, Carnegie Mellon University, 1997.

[10] O. Coudert. “Solving Graph Optimization Problems with ZBDDs”. In: *Proceedings of the ACM/IEEE European Design and Test Conference*. 1997, pp. 224–228.

[11] K. Sekine, H. Imai, and S. Tani. “Computing the Tutte Polynomial of a Graph of Moderate Size”. In: *Proceedings of the 6th International Symposium on Algorithms and Computation (ISAAC)*. 1995, pp. 224–233.

[12] K. Sekine and H. Imai. “Counting the Number of Paths in a Graph via BDDs”. In: *IEICE Transactions on Fundamentals*. Vol. E80-A. 4. 1997, pp. 682–688.

[13] R. Yoshinaka, T. Saitoh, J. Kawahara, K. Tsuruma, H. Iwashita, and S. Minato. “Finding All Solutions and Instances of Numberlink and Slitherlink by ZDDs”. In: *Algorithms* 5.2 (2012), pp. 176–213. ISSN: 1999-4893. DOI: 10.3390/a5020176. URL: http://www.mdpi.com/1999-4893/5/2/176/.

[14] S. B. Akers. “Binary Decision Diagrams”. In: *IEEE Transactions on Computers* C-27.6 (1978), pp. 509–516.

[15] D. E. Knuth. *Don Knuth’s Home Page*. URL: http://www-cs-staff.stanford.edu/~uno/.

[16] Nikoli. *WEB Nikoli*. URL: http://www.nikoli.co.jp/.

[17] Nikoli. *Numberlink 1*. Nikoli, Tokyo, Japan, 1989. ISBN: 4890720103.

[18] Nikoli. *Nikoli.com Puzzle Championship*. URL: http://www.nikoli.com/en/event/puzzle_hayatoki.html.

[19] Nikoli. *Slitherlink 1*. Nikoli, Tokyo, Japan, 1992. ISBN: 4890720243.

[20] Nikoli. *Sample Problems of Slitherlink Puzzle*. URL: http://www.nikoli.com/en/puzzles/slitherlink/.