𝕋ℂ𝕊-𝕋ℝ-𝔸-**12**-**60**

𝕋ℂ𝕊 Technical Report

ZDD-Based Computation of the Number of Paths in a Graph

by

Hiroaki Iwashita, Jun Kawahara, and
Shin-ichi Minato

Division of Computer Science

Report Series A

September 18, 2012

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

ZDD-Based Computation of the Number of Paths in a Graph

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

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

SHIN-ICHI MINATO*
Division of Computer Science
Graduate School of Info. Sci. and Tech.
Hokkaido University
Sapporo 060-0814, Japan

September 18, 2012

(Abstract)

Counting the number of paths in a graph, for example the number of nonintersecting (or self-avoiding) rook paths joining opposite corners of an $n \times n$ grid, is not an easy problem since no mathematical formula is found and the number becomes too large to enumerate one by one except for small graphs. We have implemented software based on the ZDD technique in Knuth's "The Art of Computer Programming" carefully so as to achieve better memory efficiency, and have succeeded in computing the exact numbers for some graphs that were not known until now.

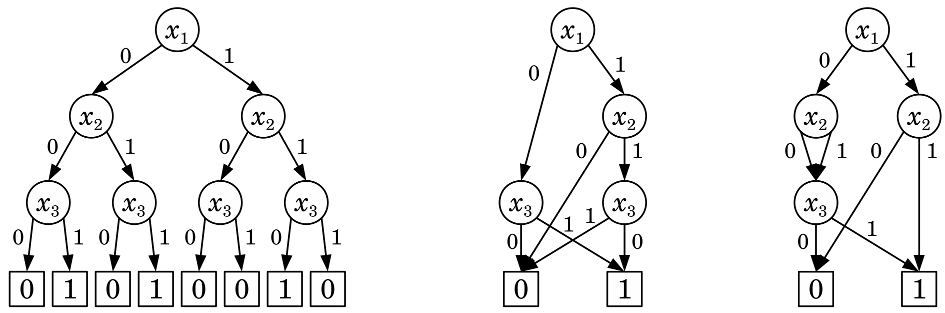
1 Introduction

Knuth introduced an interesting algorithm in his book [1, exercise 225 in 7.1.4], named SIMPATH, which constructs a zero-suppressed binary decision diagram (ZDD) [2] representing a set of all paths between two vertices in a graph. It is so efficient that a ZDD representing

$$227449714676812739631826459327989863387613323440 \simeq 2.27 \times 10^{47}$$

paths on a 15×15 grid graph can be constructed in a few minutes. To the best of our knowledge, SIMPATH is the most powerful and flexible among the algorithms that solves the same problem. ZDDs for cycles, Hamiltonian paths, and path matchings can also be

*He also works for JST ERATO Minato Project.



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

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

constructed in almost the same algorithms [1], and they have many real-world applications such as network reliability analysis [3], solving and generating puzzle instances [4], and finding configurations minimizing the loss of energy in the electric power network [5].

We have been developing a framework for applications of SIMPATH-like algorithms, which we call *frontier-based search* [6][7]. In this technical report, we demonstrate possibilities of the frontier-based search by an example application, which computes the number of paths in a graph efficiently. We have implemented software by modifying Knuth's algorithm carefully in order to improve memory efficiency without sacrificing execution time efficiency. In the experiments, we have succeeded in computing the exact numbers of nonintersecting (or self-avoiding) rook paths joining opposite corners of an $n \times n$ grid [8][9] up to $n = 21$, which had not been computed so far.

2 BDDs and ZDDs

Binary decision diagrams (*BDDs*) [10][11] and *zero-suppressed BDDs* (*ZDDs*) [2] are labeled directed acyclic graphs derived by reducing binary decision tree graphs, which represent decision making processes through binary input variables. As is 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*. 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. 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

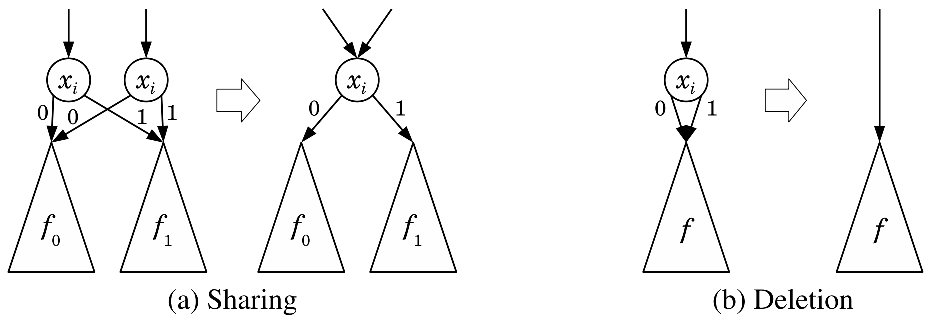


Figure 2: BDD reduction rules

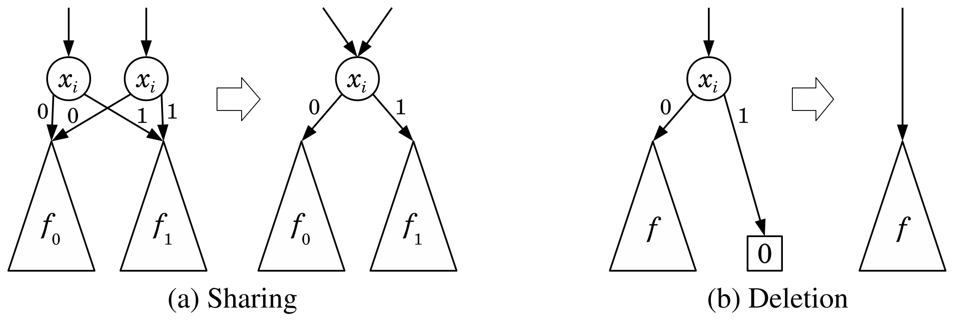
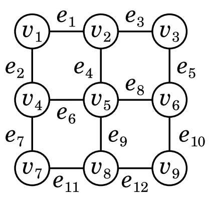


Figure 3: ZDD reduction rules

pointing to the 0-terminal node can be deleted in ZDDs (Figure 3b). The entire diagram can be reduced completely as BDD or ZDD by applying these rules from the bottom (index n) to the top (index 1) as follows:

```
Reduce($D$)
1: for $i = n$ to 1 do
2:   for each node $p$ at index $i$ in diagram $D$ do
3:     for each $c \in \{0,1\}$ do
4:       replace the $c$-child of $p$ with the reduced one;
5:     end for
6:   end for
7: end for
8: replace the root of $D$ with the reduced one.
```

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 $x_1, \dots, x_n$, 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.



(a) $G_{3,3}$

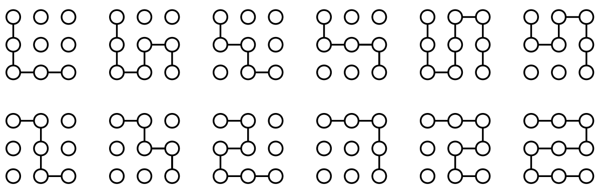

(b) Paths between v_1 and v_9

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

Graph enumeration and indexing problems are important applications of BDDs and ZDDs, which include enumeration of paths, cycles, cliques, spanning trees, cut sets, partitioning, and matching. They are tightly related to various real-life problems, such as geographic information systems, dependency analysis, and demarcation problems. Each enumeration problem is solved implicitly as a problem of constructing a monolithic BDD/ZDD representing a family of all instances, where each instance (path, cycle, etc.) is represented by a set of graph edges or vertices.

3 ZDD Construction Algorithm

The SIMPATH algorithm constructs a ZDD representing a set of paths (ways to go from a point to another point without visiting any point twice) in a graph [1, exercise 225 in 7.1.4][12]. 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 $V = \{v_1, \ldots, v_m\}$ is a set of vertices and $E = \{e_1, \ldots, e_n\}$ 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.

SIMPATH performs breadth-first search in the 2^E space to construct a directed acyclic graph (DAG), and then reduces the DAG as a ZDD. Figure 5 illustrates an example of the constructed DAG before reduction, where the 0-terminal node and edges to it are not drawn for visibility.

Each node of the DAG has the *configuration* $\langle i, mate\rangle$ that indicates node equality, which consists of a node index i and a table called *mate*. A configuration of node p with index i corresponds to a state after decisions on edges $e_1, \cdots, e_{i-1}$ are made, in which the set of selected edges $E_p \subseteq \{e_1, \cdots, e_{i-1}\}$ forms path fragments. The mate table on p represents a map from V_i to $V_i \cup \{0\}$, where $V_i \subseteq V$ is called *frontier* for index i:

$$mate_p[v] = \begin{cases} v & \text{if vertex } v \text{ is not a part of any path fragment,} \\ u & \text{if vertices } u \text{ and } v \text{ are endpoints of a path fragment,} \\ 0 & \text{if vertex } v \text{ is an intermediate point of a path fragment.} \end{cases}$$

As we are not interested in the vertex states that does not influence future edge selection for $e_i, \cdots, e_n$, the frontier is limited to a set of vertices contiguous with both decided and undecided edges. An entry for $mate[v]$ is created when vertex v is entering the frontier and an entry for $mate[v]$ is deleted when vertex v is leaving the frontier.

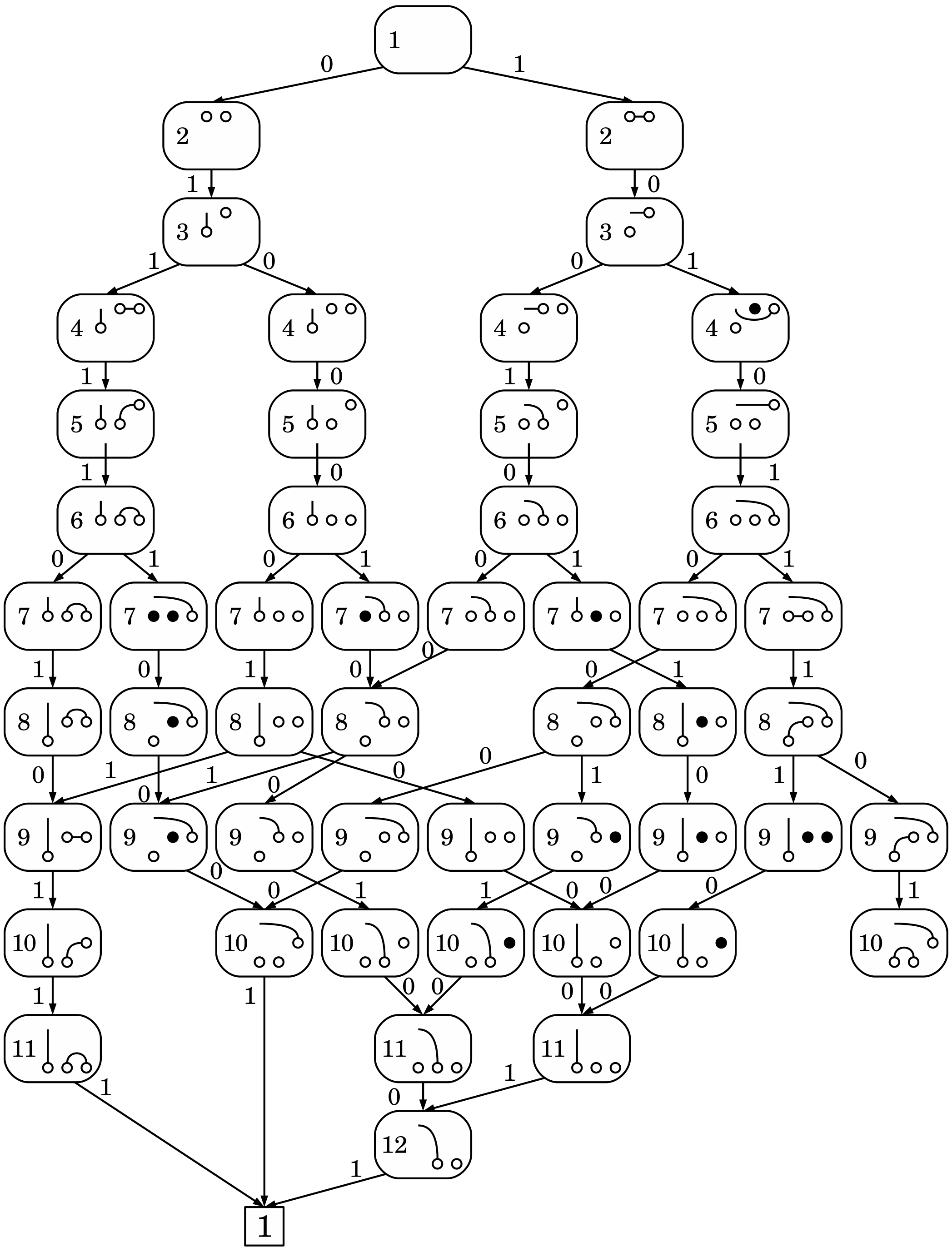


Figure 5: DAG for paths on $G_{3,3}$ constructed by SIMPATH

In Figure 5, the mate table 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 ($mate[v] = 0$).

For simplicity of the algorithm, the mate table is initially set to the value that has two entries $mate[v_1] = v_m$ and $mate[v_m] = v_1$ as if there were a built-in path between v_1 and v_m, and as if we were enumerating all the virtual cycles that include it. When a virtual cycle is formed and no other path fragment remains, the current set of selected edges is accepted and the DAG node is connected to the 1-terminal. It is rejected and the DAG node is connected to the 0-terminal 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.

We encapsulate the above details in SIMPATH by *recursive specification S* of SIMPATH's configurations, which are described by two functions:

- $S.getRoot()$ for getting the initial configuration of the root node,
- $S.getChild(\langle i, mate\rangle, c)$ for getting the configuration of c-child ($c \in \{0,1\}$) of the node that have configuration $\langle i, mate\rangle$.

Data structure for DAG node p has three fields: $p.state$ for the mate table, $p.child[0]$ for the 0-child pointer, and $p.child[1]$ for the 1-child pointer. Using these notations, the ZDD construction algorithm is summarized as follows:

```
Construct($S$)
 1: $P_i \leftarrow \emptyset$ for $i = 1, \ldots, n+1$;
 2: let $r$ be a new node;
 3: $\langle i, r.state\rangle \leftarrow S.getRoot()$;
 4: $P_i \leftarrow \{r\}$;
 5: for $i = 1$ to $n$ do
 6:     for each $p \in P_i$ do
 7:         for each $c \in \{0,1\}$ do
 8:             let $q$ be a new node;
 9:             $\langle j, q.state\rangle \leftarrow S.getChild(\langle i, p.state\rangle, c)$;
10:             if we already have node $q' \in P_j$ such that $q'.state = q.state$ then
11:                 delete $q$;
12:                 $p.child[c] \leftarrow q'$;
13:             else
14:                 $P_j \leftarrow P_j \cup \{q\}$;
15:                 $p.child[c] \leftarrow q$;
16:             end if
17:         end for
18:     end for
19: end for
20: reduce the DAG rooted by $r$ as a ZDD.
```

4 Computing the Number of Paths in a Graph

4.1 Computing the Number of Paths as 1-Paths in a ZDD

Once a ZDD for a set of paths in a graph is constructed, we can get the number of paths by computing the number of 1-paths in the ZDD. Here is a simple depth-first algorithm, which takes the root node of the ZDD as its argument:

```
CountZddOnePathsDF($p$)
 1: if $p$ is the 0-terminal return 0;
 2: if $p$ is the 1-terminal return 1;
 3: if $\langle p,k \rangle$ is found in the result table return $k$;
 4: $p_0 \leftarrow$ 0-child of $p$;
 5: $p_1 \leftarrow$ 1-child of $p$;
 6: $k \leftarrow$ CountZddOnePathsDF($p_0$) + CountZddOnePathsDF($p_1$);
 7: add $\langle p,k \rangle$ into the result table;
 8: return $k$.
```

This algorithm uses the result table for avoiding repeated computation on the same node. Useless table entries should be deleted on the fly in order to improve memory efficiency. It is important especially when we are dealing with very large ZDDs, because we need to compute "bignum" k's for all ZDD nodes that can occupy huge memory. For example, the number grows up to 48 digits in the computation for $G_{15,15}$, and up to 107 digits for $G_{22,22}$.

Here is another algorithm to get the number of 1-paths in ZDD P, which is based on breadth-first search:

```
CountZddOnePathsBF($P$)
 1: let $r$ be the root node of ZDD $P$;
 2: add $\langle r,1 \rangle$ into the result table;
 3: let $P_i$ be a set of nodes at index $i \in \{1,\dots,n+1\}$ in $P$;
 4: initialize the result table;
 5: for $i = 1$ to $n$ do
 6:    for each $p \in P_i$ do
 7:       find $\langle p,k \rangle$ in the result table and delete the entry;
 8:       for each $c \in \{0,1\}$ do
 9:          $q \leftarrow$ c-child of $p$;
10:          if $\langle q,l \rangle$ is found in the result table then
11:             replace the entry with $\langle q,l+k \rangle$;
12:          else
13:             add $\langle q,k \rangle$ into the result table;
14:          end if
15:       end for
16:    end for
17: end for
```

```
18: let $t_1 \in P_{n+1}$ be the 1-terminal node;
19: find $\langle t_1, k\rangle$ in the result table;
20: **return** $k$.
```

In the breadth-first algorithm, simple management of the result table realizes good memory efficiency; we can simply delete the table entry $\langle p,k\rangle$ for the current node p as soon as bignum k is propagated to its child nodes.

4.2 Computing the Number of Paths without Constructing ZDDs

In the previous section, we have described a two-step method to compute the number of paths in a graph, which constructs a ZDD as an intermediate result. Now we combine the two steps in order to further improve memory efficiency. We use a breadth-first algorithm and compute the number of paths from the top to the bottom. The key idea is to delete useless DAG nodes on the fly as well as bignums. In this algorithm, node p has two fields: $p.state$ for the mate table, and $p.value$ for the bignum that keeps an intermediate result instead of the result table. The algorithm is summarized as follows:

```
CountPathsInGraph($S$)
 1: $P_i \leftarrow \emptyset$ for $i = 1,\dots,n+1$;
 2: let $r$ be a new node;
 3: $\langle i, r.state\rangle \leftarrow S.getRoot()$;
 4: $r.value \leftarrow 1$;
 5: $P_i \leftarrow \{r\}$;
 6: for $i = 1$ to $n$ do
 7:    for each $p \in P_i$ do
 8:       for each $c \in \{0,1\}$ do
 9:          let $q$ be a new node;
10:          $\langle j, q.state\rangle \leftarrow S.getChild(\langle i, p.state\rangle, c)$;
11:          if we already have node $q' \in P_j$ such that $q'.state = q.state$ then
12:             delete $q$;
13:             $q'.value \leftarrow q'.value + p.value$;
14:          else
15:             $P_j \leftarrow P_j \cup \{q\}$;
16:             $q.value \leftarrow p.value$;
17:          end if
18:       end for
19:       delete $p$;
20:    end for
21: end for
22: let $t_1 \in P_{n+1}$ be the 1-terminal node;
23: return $t_1.value$;
```

Storage requirement for this algorithm is $O(am)$ where a is the memory usage for each node and m is the maximum number of the nodes that become alive during the computa-

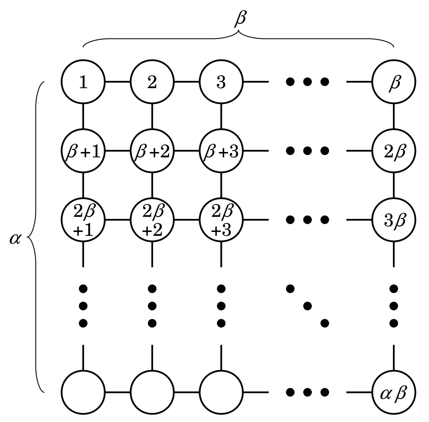


Figure 6: Vertex order of $G_{\alpha,\beta}$

tion. When S is a recursive specification of SIMPATH's configurations, $m \simeq \max_{1\le i\le n}|P_i|$ and it is much less than the number of nodes in the entire DAG structure ($\sum_{1\le i\le n}|P_i|$).

5 Experimental Results

We have implemented the two methods in C++; one is the two-step method that constructs a ZDD as an intermediate result, the other is the direct method that does not construct the entire DAG structure. They are single-threaded programs using large memory space. In early evaluation of them, we found that memory access has a significant impact on CPU time. Data storage for DAG nodes often becomes much larger than the last level cache memory of the CPU and the algorithm searches an equivalent node repeatedly among all of them, which implies repeated access to the main memory of the CPU. Fine tuning of memory access have made our software several times faster.

All experiments are performed on 2.67GHz Intel Xeon E7-8837 CPU with 1TB memory, running 64-bit SUSE Linux Enterprise Server 11. We experimented with square grid graphs $G_{\alpha,\beta}$ as benchmark examples. The vertex order is illustrated in Figure 6 and the edge order (ZDD variable order) is defined lexicographically with the vertex order: $\{v_i, v_j\} \le \{v_{i'}, v_{j'}\}$ if and only if $v_i < v_{i'}$ or $(v_i = v_{i'}$ and $v_j \le v_{j'})$ where $v_i \le v_j$ and $v_{i'} \le v_{j'}$. The terminal vertices of paths are v_1 and $v_{\alpha\beta}$ in all experiments.

Comparison of CPU time and memory usage between two methods is summarized in Table 1. We can confirm that the direct method is faster than the two-step method and requires orders of magnitude less memory. In both methods, CPU time and memory usage is tripled when the graph becomes the next larger size. It will take about 10 days and 700 gigabytes of memory to get the result for $G_{23,23}$ using the direct method.

Table 2 shows the exact number of paths in $G_{n+1,n+1}$ for $1 \le n \le 21$. It is consistent with "Number of nonintersecting (or self-avoiding) rook paths joining opposite corners of an $n \times n$ grid" listed in [8] except that it was not known for $n \ge 20$. Note that the prior art that computed the result for $n = 19$ is an algorithm designed for square grid graphs [9], while SIMPATH is an algorithm for arbitrary graphs.

Table 1: Comparison between two methods

		Two-step method		Direct method	
Graph	#path	Time (sec)	Mem (MB)	Time (sec)	Mem (MB)
$G_{10,10}$	4.10E+019	0.1	13	0.1	1
$G_{11,11}$	1.57E+024	0.4	41	0.3	3
$G_{12,12}$	1.82E+029	1.5	131	1.0	6
$G_{13,13}$	6.45E+034	4.9	422	3.2	13
$G_{14,14}$	6.95E+040	18.7	1354	11.7	37
$G_{15,15}$	2.27E+047	82.4	4320	52.4	111
$G_{16,16}$	2.27E+054	306.6	13705	206.0	351
$G_{17,17}$	6.87E+061	1032.4	43319	701.9	958
$G_{18,18}$	6.34E+069	3379.0	136473	2326.0	3018
$G_{19,19}$	1.78E+078	12639.4	428310	7607.1	7514
$G_{20,20}$	1.52E+087	>1TB		28279.2	27394
$G_{21,21}$	3.96E+096	>1TB		91944.1	74504
$G_{22,22}$	3.14E+106	>1TB		284117.0	216871

Table 2: The number of paths between opposite corners of $G_{n+1,n+1}$

n	#path
1	2
2	12
3	184
4	8512
5	1262816
6	575780564
7	789360053252
8	3266598486981642
9	41044208702632496804
10	1568758030464750013214100
11	182413291514248049241470885236
12	64528039343270018963357185158482118
13	69450664761521361664274701548907358996488
14	227449714676812739631826459327989863387613323440
15	2266745568862672746374567396713098934866324885408319028
16	68745445609149931587631563132489232824587945968099457285419306
17	6344814611237963971310297540795524400449443986866480693646369387855336
18	1782112840842065129893384946652325275167838065704767655931452474605826692782532
19	1523344971704879993080742810319229690899454255323294555776029866737355060592877569255844
20	3962892199823037560207299517133362502106339705739463771515237113377010682364035706704472064940398
21	31374751050137102720420538137382214513103312193698723653061351991346433379389385793965576992246021316463868

6 Conclusions

We have developed efficient software to compute the number of paths in a graph based on Knuth's SIMPATH algorithm, and have confirmed that it is a state-of-the-art technique.

We have experimented with grid graphs and have succeeded in computing the number of paths in larger graphs, which had not been computed so far. It shows a potential of the frontier-based search framework, since it was achieved by only a simple modification of the existing algorithm on the framework. The algorithm is abstracted using the recursive specification that encapsulates details of mate table operations in SIMPATH, which can be replaced for solving other graph enumeration problems, such as cycles and Hamiltonian paths.

References

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

[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] G. Hardy, C. Lucet, and N. Limnios. "K-Terminal Network Reliability Measures With Binary Decision Diagrams". In: *IEEE Transactions on Reliability* 56.3 (2007), pp. 506–515.

[4] 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/`.

[5] T. Inoue, K. Takano, T. Watanabe, J. Kawahara, R. Yoshinaka, A. Kishimoto, K. Tsuda, S. Minato, and Y. Hayashi. "Finding All Configurations Satisfying Operational Constraints in Delivery Networks by ZDDs (in Japanese)". In: *Proceedings of the Institute of Electrical Engineers of Japan National Conference.* 2012.

[6] J. Kawahara, T. Saitoh, and S. Minato. "New Enumeration Methods Using ZDD (in Japanese)". In: *The Journal of the Institute of Electronics, Information and Communication Engineers of Japan* 95.6 (2012), pp. 505–511.

[7] S. Takeuchi, J. Kawahara, A. Kishimoto, and S. Minato. *Shared-Memory Parallel Algorithms for Frontier-Based Search.* Tech. rep. TCS-TR-A-12-57. Division of Computer Science, Graduate School of Information Science and Technology, Hokkaido University, 2012. URL: `http://www-alg.ist.hokudai.ac.jp/tra.html`.

[8] The On-Line Encyclopedia of Integer Sequences. *A007764 Number of nonintersecting (or self-avoiding) rook paths joining opposite corners of an $n \times n$ grid.* URL: `http://oeis.org/A007764`.

[9] M. Bousquet-Mélou, A. J. Guttmann, and I. Jensen. “Self-avoiding walks crossing a square”. In: *Journal of Physics A: Mathematical and General* 38 (2005), pp. 9159–9181. DOI: `10.1088/0305-4470/38/42/001`.

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

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

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