TCS-TR-A-13-64

TCS Technical Report

Efficient Computation of the Number of Paths in a Grid Graph with Minimal Perfect Hash Functions

by

Hiroaki Iwashita, Yoshio Nakazawa,
Jun Kawahara, Takeaki Uno, and Shin-ichi Minato

Division of Computer Science

Report Series A

April 26, 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 Computation of the Number of Paths in a Grid Graph with Minimal Perfect Hash Functions

Hiroaki Iwashita[*†‡] Yoshio Nakazawa[§] Jun Kawahara[¶] Takeaki Uno[‖]
Shin-ichi Minato[‡†]

April 26, 2013

Abstract

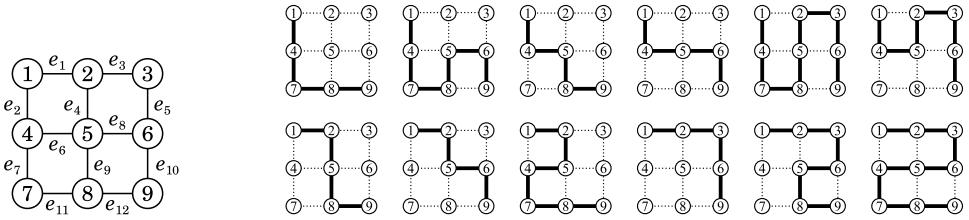
It is not easy to find the number of self-avoiding walks from $(0,0)$ to (n,n), because the number increases rapidly with the increase of n and mathematical formula for calculating it is not known. Our challenge is to develop advanced algorithmic techniques through efforts of finding the answer to the larger n. The idea of Knuth's algorithm is so effective that it can compute the answer to $n = 11$ in a second and $n = 21$ in a few days, even though it is designed for general graphs. In this paper, we specialize it in grid graphs and maximize space and time efficiency. Our program have successfully computed the answer to $n = 25$, which had not known before.

1 Introduction

A path (way to go from a vertex to another vertex without visiting any vertex twice) in a grid graph is called a self-avoiding walk (SAW), which is known to be introduced by the chemist Flory as a model of polymer chains [1]. In spite of its simple definition, many difficult mathematical problems are hidden behind the SAW [2, 3]. They include a problem of counting the number of paths connecting opposite corners of a $(n+1)\times(n+1)$ grid graph, which have become popular through a YouTube-animation [4]. The answer is known to grow as $\lambda^{n^2+o(n^2)}$ and $\lambda \simeq 1.7$ [5]. When $n = 10$, it is about 10^{24} and cannot be counted one by one in a realistic time even if we could find trillions of paths in a second.

We are studying advanced algorithmic techniques through efforts of finding the answer to the larger n. According to The On-Line Encyclopedia of Integer Sequences (OEIS) [6],

[*] 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.
[§] Amateur programmer.
[¶] Graduate School of Information Science, Nara Institute of Science and Technology, Ikoma, Nara, Japan.
[‖] Principles of Informatics Research Division, National Institute of Informatics, Chiyoda, Tokyo, Japan.



(a) $G_{3,3}$ (b) Paths between 1 and 9

Figure 1: Paths in a graph

Rosendale computed the answers up to $n = 11$ in 1981 and Knuth computed the answer to $n = 12$ in 1995. Bousquet-Mélou et al. presented the answers up tp $n = 19$ in their paper [5] in 2005. Their algorithm is based on the method of Conway et al. [7], which makes good use of the fact that the target is a grid graph. On the other hand, Knuth introduced an algorithm for general graphs called SIMPATH in 2008 [8][9, exercise 225 in 7.1.4], which constructs a zero-suppressed binary decision diagram (ZDD) [10] representing a set of all paths between two vertices in a graph. We have extended the answers up to $n = 21$ in 2012 by reimplementing SIMPATH to directly count the number of paths instead of building the ZDD structure [11]. It would be notable that the implementation is not dedicated to grid graphs but is made for general graphs.

We specialize it in grid graphs and maximize space and time efficiency. On the premise that the graph is a square grid, we can foretell the exact number of entries in the hash table filled in the algorithm. It allows us to use a simple array with a minimal perfect hash function instead of an ordinary hash table, which has a large impact on actual space and time efficiency. We improve it still more by additional techniques such as in-place update and parallel processing.

This paper is organized as follows. Section 2 introduces the original algorithm for general graphs. The basic idea specific to grid graphs is shown in Section 3. Detailed techniques to further improve space and time efficiency are described in Section 4. Section 5 shows experimental results and Section 6 mentions some additional ideas. Finally, conclusions are drawn in Section 7.

2 Original Algorithm for General Graphs

Input to the original algorithm [11] is an undirected graph $G = (V,E)$ and two vertices $s,t \in V$ 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 the number of all paths between s and t. For example, if the input is a 3×3 grid graph and two vertices at opposite corners, the output will be 12 as shown in Figure 1.

The algorithm is a kind of dynamic programming. A path can be represented by a subset of edges $E' \subseteq E$. The algorithm performs breadth-first search in the 2^E space that corresponds to all decision patterns on edge selections. Figure 2 illustrates the computation for $G_{3,3}$, where dotted and solid branches represent exclusion and inclusion of the

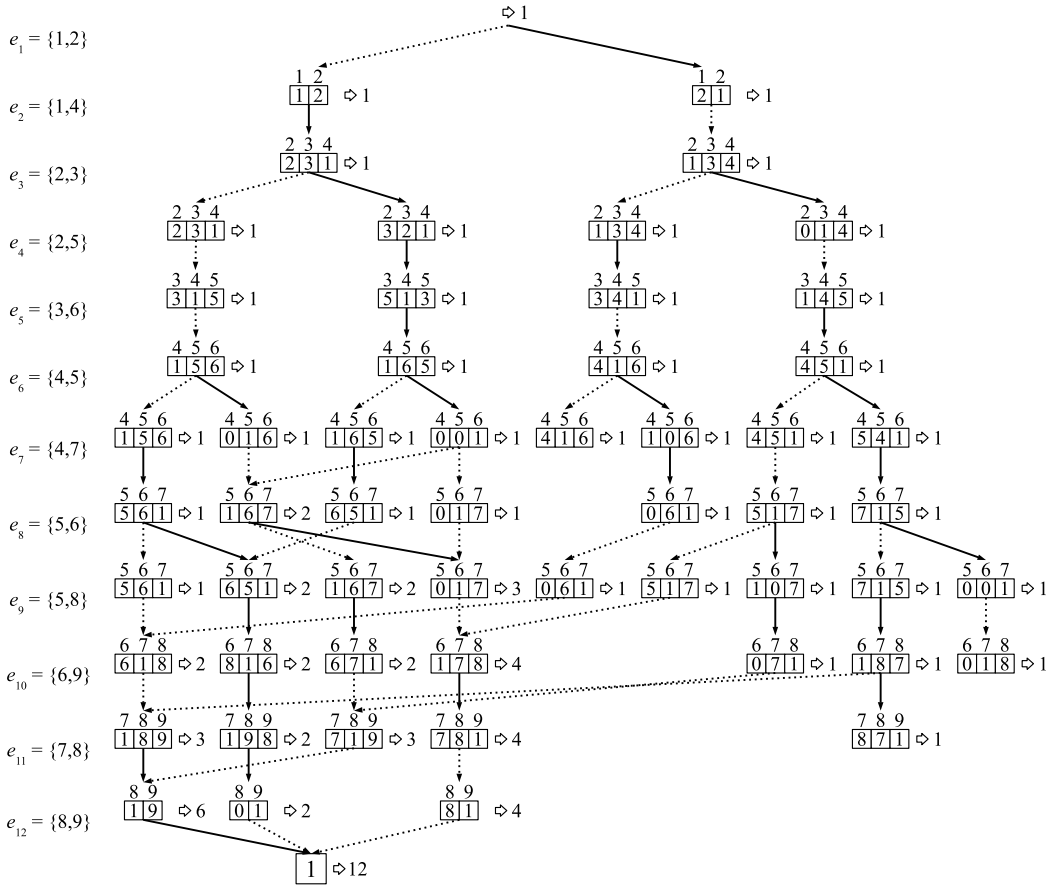


Figure 2: Computation for $G_{3,3}$

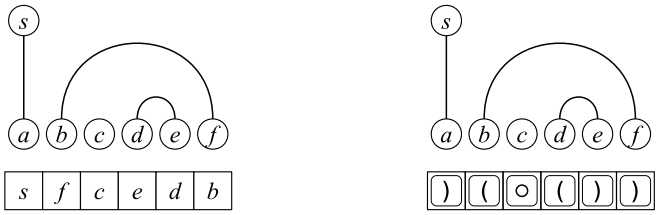
corresponding edge respectively. A branch is pruned when it makes an inconsistent situation with the final target, e.g. making a cycle or an unexpected endpoint.

Each node p at level i has a *frontier state* and an integer value. A frontier is a set of the vertices that are contiguous with both decided and undecided edges. The frontier state is represented by a map, $mate_p : V_i \to V_i \cup \{0\}$, where $V_i \subseteq V$ is the frontier at level i. It maintains information about path fragments formed by a set of selected edges as follows:

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

An entry for $mate_p[v]$ is created when vertex v is entering the frontier and an entry for $mate_p[v]$ is deleted when vertex v is leaving the frontier.

Since the mate table includes sufficient information of constraints for future edge selections, nodes with equivalent mate tables can be merged during the computation. The integer values of merged nodes are summed. The final result is the value totaled at the leaf node. This algorithm is implemented using a hash table of which keys are mate tables and values are integers.



(a) For general graphs (b) For plane graphs

Figure 3: Frontier state representation

3 Basic Idea for Grid Graphs

3.1 Frontier States

Let $v_{(i,j)}$ be the vertex at row i and column j of an $n \times n$ grid graph ($1 \leq i \leq n$, $1 \leq j \leq n$) and we compute the number of paths between $v_{(1,1)}$ and $v_{(n,n)}$. We visit the vertices in the order of $v_{(1,1)}, \ldots, v_{(1,n)}, v_{(2,1)}, \ldots, v_{(2,n)}, \ldots$ and make decisions on vertical line $\{v_{(i-1,j)}, v_{(i,j)}\}$ and horizontal line $\{v_{(i,j-1)}, v_{(i,j)}\}$. The frontier at the step visiting $v_{(i,j)}$ is $\{v_{(i,1)}, \ldots, v_{(i,j)}, v_{(i-1,j+1)}, \ldots, v_{(i-1,n)}\}$.

The original algorithm keeps vertex identifiers of path endpoints in $mate_p$ (Figure 3a), while we can compress the information efficiently on the premise that the graph is embedded in the plane (Figure 3b). Pairs of path endpoints always form nested structure on the frontier because no path fragments can intersect. We do not need to record all information of endpoint pairs but record only whether they are left or right endpoints. Let $s = c_1 c_2 \cdots c_n$ be a string representing a frontier state at a step visiting $v_{(i,j)}$ where $\Sigma = \{(,), ○, •\}$ and $c_k \in \Sigma$ is a character representing the state of vertex at the k-th column in the frontier:

$$c_k = \begin{cases} (& \text{if the } k\text{-th vertex is a left endpoint of a path fragment,} \\ ) & \text{if the } k\text{-th vertex is a right endpoint of a path fragment,} \\ • & \text{if } k = j \text{ and the } k\text{-th vertex is an intermediate point of a path fragment,} \\ ○ & \text{otherwise.} \end{cases}$$

We consider the endpoint connected to $v_{(1,1)}$ is always a right endpoint. • is a special character only for c_j, meaning that it is different from ○ as we cannot include the next horizontal line $\{v_{(i,j)}, v_{(i,j+1)}\}$.

Examples of state transitions are shown in Figure 4. We have no alternatives of vertical line selections in every step, because $\{v_{(i-1,j)}, v_{(i,j)}\}$ must be excluded if $v_{(i-1,j)}$ is not an endpoint and $\{v_{(i-1,j)}, v_{(i,j)}\}$ must be included if $v_{(i-1,j)}$ is an endpoint. Branches are always made by selection of horizontal lines. A frontier state changes only when horizontal line $\{v_{(i,j-1)}, v_{(i,j)}\}$ is included except for • to ○ changes. The state transition is summarized in Figure 5.

Since ( ) pairs are nested, the number of valid frontier states is closely related to Motzkin numbers [12]. Let $x_1 < x_2$ and $\mathscr{M}_{(x_1,y_1)\to(x_2,y_2)}$ be a set of routes from coordinates (x_1, y_1) to coordinates (x_2, y_2) on $x_2 - x_1$ steps of $(1,1)$, $(1,-1)$, or $(1,0)$ moves without

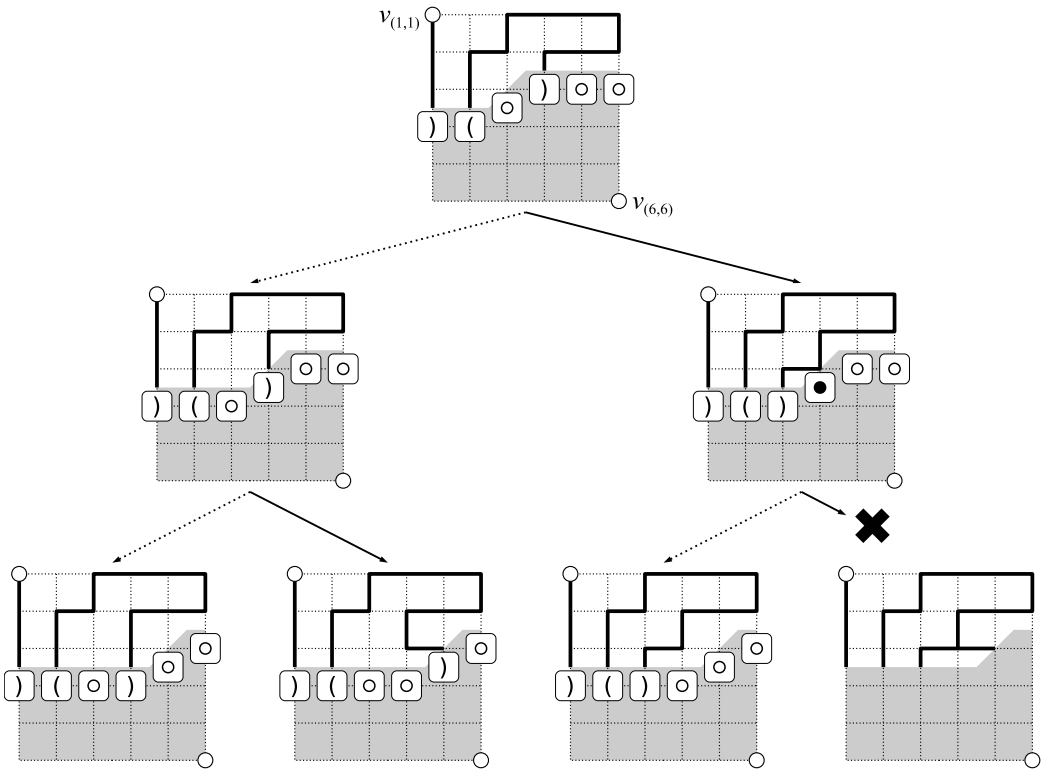
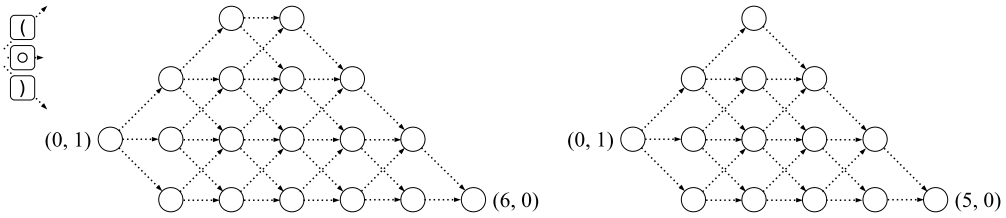


Figure 4: Examples of state transitions on a 6×6 grid graph

Current state	Next state	
	Line excluded	Line included
⋯ ○ ○ ⋯	⋯ ○ ○ ⋯	⋯ ( ) ⋯
⋯ ○) ⋯	⋯ ○) ⋯	⋯) ● ⋯
⋯ ○ (⋯	⋯ ○ (⋯	⋯ (● ⋯
⋯) ○ ⋯	⋯) ○ ⋯	⋯ ○) ⋯
⋯ (⋯)) ⋯	⋯ (⋯)) ⋯	⋯) ⋯ ○ ● ⋯
⋯) (⋯	⋯) (⋯	⋯ ○ ● ⋯
⋯ (○ ⋯	⋯ (○ ⋯	⋯ ○) ⋯
⋯ ((⋯) ⋯	⋯ ((⋯) ⋯	⋯ ○ ● ⋯ (⋯
⋯ ( ) ⋯	⋯ ( ) ⋯	N/A (making cycle)
⋯ ● ○ ⋯	⋯ ○ ○ ⋯	N/A (branching)
⋯ ●) ⋯	⋯ ○) ⋯	N/A (branching)
⋯ ● (⋯	⋯ ○ (⋯	N/A (branching)

Figure 5: State transition table



(a) Main states: $\mathscr{M}_{(0,1)\to(6,0)}$ (b) Blocked states: $\mathscr{M}_{(0,1)\to(5,0)}$

Figure 6: Possible frontier states for a 6×6 grid graph

visiting negative y-coordinates. The n-th Motzkin number is given by $M_n = \left|\mathscr{M}_{(0,0)\to(n,0)}\right|$. They are given by recurrence relation:

$$\begin{aligned} M_0 &= M_1 = 1 \ , \\ M_n &= \frac{3(n-1)M_{n-2} + (2n+1)M_{n-1}}{n+2} \ . \end{aligned} \tag{1}$$

We divide frontier states into two classes: *main states* and *blocked states*. A main state does not have [•] and a blocked state has [•] at the j-th position. Let $\Sigma' = \Sigma \setminus \{[\bullet]\}$, S_j be a set of frontier states after visiting j-th column, and $S' \subset S_j$ be a set of main states. S' does not include [•], i.e. $S' \subset \Sigma'^n$. It corresponds to $\mathscr{M}_{(0,1)\to(n,0)}$ (Figure 6a) where [(], [)], and [○] characters are associated with $(1,1)$, $(1,-1)$, and $(1,0)$ moves respectively. The number of main states is given by:

$$N_n \ = \ \left|\mathscr{M}_{(0,1)\to(n,0)}\right| \ = \ M_{n+1} - M_n \ . \tag{2}$$

Let $S''_j = S_j \setminus S'$ be a set of blocked states after visiting j-th column. As all blocked states have [•] at the j-th position, it can be ignored in enumerating S''_j. That corresponds to $\mathscr{M}_{(0,1)\to(n-1,0)}$ (Figure 6b) and the number of blocked states is given by:

$$N_{n-1} \ = \ \left|\mathscr{M}_{(0,1)\to(n-1,0)}\right| \ = \ M_n - M_{n-1} \ . \tag{3}$$

Therefore, the number of frontier states is:

$$N_n + N_{n-1} \ = \ M_{n+1} - M_{n-1} \ . \tag{4}$$

3.2 Basic Algorithm

Since the domain of frontier states has become clear, we can define a minimal perfect hash function $\varphi_j : S_j \to \{1, \ldots, N_n + N_{n-1}\}$ for each column position $1 \le j \le n$. It allows us to use a simple array instead of an ordinary hash table to keep intermediate results. A simple implementation is to define some lexicographical order on the frontier states, and to calculate the order of a given state string by scanning its characters one by one. An improved implementation is described later in Section 4.2.

Algorithm 1 shows the basic method for computing the number of paths in a grid graph. The *count* array keeps the integers that represent the numbers of cases for all frontier states at the current step. For example, if we apply this algorithm to a 24×24 grid graph using 56-byte integers, *count* requires $(M_{25} - M_{23}) \times 56$ bytes $=$ 413 gigabytes of memory. The *tmp* array is a temporary storage, which is the same size as *count*.

Algorithm 1 Computing the number of paths in a grid graph

1: $count[k] \leftarrow 0$ for all $1 \leq k \leq N_n + N_{n-1}$;
2: $count[\varphi_1(\overbrace{\boxed{)}\boxed{\circ}\boxed{\circ}\cdots\boxed{\circ}}^{n})] \leftarrow 1$;
3: **for** $i = 1$ **to** n **do**
4: **for** $j = 1$ **to** $n-1$ **do**
5: $tmp[k] \leftarrow 0$ for all $1 \leq k \leq N_n + N_{n-1}$;
6: **for all** $s \in S_j$ **do**
7: $t \leftarrow$ the next state of s when $\{v_{(i,j)}, v_{(i,j+1)}\}$ is excluded;
8: $tmp[\varphi_{j+1}(t)] \leftarrow tmp[\varphi_{j+1}(t)] + count[\varphi_j(s)]$;
9: $u \leftarrow$ the next state of s when $\{v_{(i,j)}, v_{(i,j+1)}\}$ is included;
10: **if** u is defined **then**
11: $tmp[\varphi_{j+1}(u)] \leftarrow tmp[\varphi_{j+1}(u)] + count[\varphi_j(s)]$;
12: **end if**
13: **end for**
14: $count[k] \leftarrow tmp[k]$ for all $1 \leq k \leq N_n + N_{n-1}$;
15: **end for**
16: $tmp[k] \leftarrow 0$ for all $1 \leq k \leq N_n + N_{n-1}$;
17: **for all** $s \in S_n$ **do**
18: $t \leftarrow$ the string made from s by replacing its $\boxed{\bullet}$ with $\boxed{\circ}$;
19: $tmp[\varphi_1(t)] \leftarrow tmp[\varphi_1(t)] + count[\varphi_n(s)]$;
20: **end for**
21: $count[k] \leftarrow tmp[k]$ for all $1 \leq k \leq N_n + N_{n-1}$;
22: **end for**
23: **return** $count[\varphi_1(\overbrace{\boxed{\circ}\boxed{\circ}\cdots\boxed{\circ}\boxed{)}}^{n})]$;

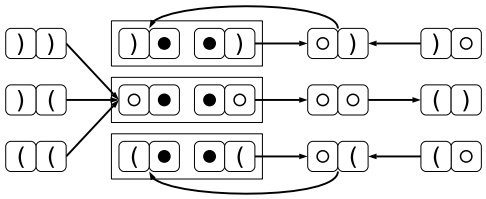


Figure 7: Data dependency

4 Techniques to Improve the Efficiency

4.1 In-place Update of the Array

If we eliminate the *tmp* array, the memory usage of Algorithm 1 can be reduced by half. We put it into practice by reordering the access to the *count* array elements. Let $1 \leq j \leq n-1$, $\alpha \in \Sigma'^{j-1}$, $\beta \in \Sigma'^{n-j-1}$, and $c \in \Sigma'$. We choose the minimal perfect hash functions that holds the following relations:

$$\begin{array}{ll} \varphi_j(s) \;=\; \varphi_{j+1}(s) & \text{for all } s \in S', \text{ and} \\ \varphi_j(\alpha[\bullet]c\beta) \;=\; \varphi_{j+1}(\alpha c[\bullet]\beta) & \text{for all } \alpha[\bullet]c\beta \in S''_j \text{ where } c \in \Sigma'. \end{array} \tag{5}$$

Hereafter, we write as $count(s)$ to denote $count[\varphi_j(s)]$ because position j is not important anymore.

Data dependency in the *count* array is illustrated in Figure 7. For instance, $[)][\circ]$ in Figure 7 represents a set of the frontier states that have $[)]$ at position j and $[\circ]$ at position $j+1$ when we are processing a horizontal line between columns j and $j+1$. A directed edge from $[\circ][\circ]$ to $[(][)]$ show that a value read from $count(\alpha[\circ][\circ]\beta)$ is added into $count(\alpha[(][)]\beta)$ for all $\alpha[\circ][\circ]\beta \in S'$. Since $count(\alpha[\circ][\circ]\beta)$ is used to calculate the new value of $count(\alpha[(][)]\beta)$, we update $count(\alpha[(][)]\beta)$ before adding the value of $count(\alpha[\bullet][\circ]\beta)$ into $count(\alpha[\circ][\circ]\beta)$. Storage for $count(\alpha[)][\bullet]\beta)$ and $count(\alpha[\bullet][)]\beta)$ is the same location. We update $count(\alpha[)][\bullet]\beta)$ and $count(\alpha[\circ][)]\beta)$ simultaneously because they depend on each other.

4.2 Fast Mapping from Frontier States to Serial Numbers

Hash functions are evaluated frequently and have a large impact to the total speedup. In this section, we describe a minimal perfect hash function for main states. One for blocked states can be made in the same way by ignoring $[\bullet]$ and considering that the string size is $n-1$. The hash function for all frontier states can be made with their combination.

The minimal perfect hash function for main states $\varphi' : S' \to \{1, \ldots, |S'|\}$ is implemented as a sum of two subfunctions:

$$\varphi'(s) = \varphi'_L(l) + \varphi'_R(r) \tag{6}$$

where l is the left half of string s and r is the right half of s. The sizes of s, l, and r are n, $m = \lfloor n/2 \rfloor$, and $n-m$, respectively. This composition works because the main states

correspond to $\mathscr{M}_{(0,1)\to(n,0)}$. The left half must be in $\mathscr{M}_{(0,1)\to(m,h)}$ and the right half must be in $\mathscr{M}_{(m,h)\to(n,0)}$, where $0 \le h \le m$. We allocate serial numbers to main states in order of h. Let $base_h + i$ denote the i-th serial number of the main states passing through (m,h). It is given by:

$$\begin{aligned} base_0 &= 0\,, \\ base_{h+1} &= base_h + \left|\mathscr{M}_{(0,1)\to(m,h)}\right| \cdot \left|\mathscr{M}_{(m,h)\to(n,0)}\right|\,. \end{aligned} \tag{7}$$

When $1 \le i_l \le \left|\mathscr{M}_{(0,1)\to(m,h)}\right|$ is a serial number of string l in $\mathscr{M}_{(0,1)\to(m,h)}$ and $1 \le i_r \le \left|\mathscr{M}_{(m,h)\to(n,0)}\right|$ is a serial number of string r in $\mathscr{M}_{(m,h)\to(n,0)}$, the subfunctions can be defined as:

$$\begin{aligned} \varphi'_L(l) &= base_h + (i_l - 1) \cdot \left|\mathscr{M}_{(m,h)\to(n,0)}\right|\,, \\ \varphi'_R(r) &= i_r\,. \end{aligned} \tag{8}$$

Main states are implemented as $2n$-bit codes under such 2-bit code assignments as $\boxed{\circ} = 00$, $\boxed{)} = 01$, and $\boxed{(} = 10$. We divide a $2n$-bit code into the left $2m$-bit subcode and the right $2(n-m)$-bit subcode. The two subfunctions can be implemented by the simple arrays that are indexed directly by the subcodes. The total size of the two arrays is about 2^{n+1} and is small enough in comparison with the size of the *count* array.

4.3 Fast Enumeration of All Main States

As shown in Section 4.1, we want to enumerate all frontier states in some specific order for in-place update of *count* entries. It is also acceptable to enumerate all main states in some specific order and visit each blocked state while visiting a main state related to it.

A simple implementation would be a single array of states sorted in the desired order. Although it is very fast and accepts any state order, it consumes memory as much as the *count* array. We introduce a method with much better memory efficiency for visiting main states in lexicographic order. We divide main states again into the left and right halves. A left state array *lstate* has ordered collection of $\langle l,h\rangle$ values where l is a left state subcode and l corresponds to $\mathscr{M}_{(0,1)\to(m,h)}$. A right state array $rstate_h$ $(0 \le h \le m)$ has ordered collection of right state subcodes in $\mathscr{M}_{(m,h)\to(n,0)}$. All main states can be visited by the following double loop:

for each $\langle l,h\rangle$ in *lstate* **do**
  for each r in $rstate_h$ **do**
    visit lr;
  end for
end for.

Running time of this double loop would be almost the same as the single array implementation because most of the time is consumed in the inner loop and it is just scanning an array in the same way as the single array implementation.

4.4 Shared Memory Parallel Processing

Let $s = c_1 c_2 \cdots c_n$ be a state string and the current step is for the horizontal line connecting columns j and $j+1$. As shown in Figure 5, the transition patterns of state strings have much locality: except for modification of c_j and c_{j+1}, at most one position can be changed from $\boxed{(}$ to $\boxed{)}$ or from $\boxed{)}$ to $\boxed{(}$.

We can partition the states into 2^{n-2} groups across which no transition occurs, based on an $(n-2)$-bit binary code "$g(c_1) \cdots g(c_{j-1})\, g(c_{j+2}) \cdots g(c_n)$" where $g : \Sigma' \to \{0,1\}$ is defined to be

$$g(c) = \begin{cases} 0 & \text{if } c = \boxed{\circ}, \\ 1 & \text{if } c = \boxed{(} \text{ or } c = \boxed{)}. \end{cases} \tag{9}$$

It is suitable for shared memory parallel processing since data update within a group is independent of other groups. We use the leftmost m-bit of the binary code for task allocation, where m is decided to make enough number of tasks and not to make each task too small. Actually, $m \simeq n/2$ is a convenient choice in combination with the technique described in Section 4.3.

5 Experimental Results

We have compared the original sequential program for general graphs [11] and three new programs for grid graphs:

GGCount-SB a sequential program based on the techniques in Section 3 and Section 4.1 using *count* array of 56-byte integers,

GGCount-SF an improved version of GGCount-SB with the techniques in Section 4.2 and Section 4.3, and

GGCount-PF a parallel version of GGCount-SF with the technique in Section 4.4, using 12 CPU cores.

All those programs are written in C++. Experiments were performed on 2.67GHz Intel Xeon E7-8837 CPUs with 1.5TB memory, running 64-bit SUSE Linux Enterprise Server 11. Results for $(n+1) \times (n+1)$ grid graphs are shown in Table 1 and Table 2. In comparison with the original implementation [11], space and time efficiency has been improved five times and ten times respectively in the sequential processing (GGCount-SF); time improvement of one another digit is achieved by the parallel processing using 12 CPU cores (GGCount-PF). GGCount-SF used slightly more memory than GGCount-SB, while it achieved three times speedup.

As the parallel program runs fast enough, it is worth exchanging space costs for time costs using the Chinese Remainder Theorem in the same way as [5]. Using a 64-bit modular arithmetic version of GGCount-PF, we have succeeded in finding the answer to $n = 25$ as well as confirming the past results for $n = 22, 23, 24$ by Spaans [6]. The numbers are shown in Table 3. For $n = 25$, we performed 9 runs with coprime moduli, of which each

Table 1: Memory usage (megabytes)

n	[11]	GGCount-SB	GGCount-SF	GGCount-PF
10	3	1	1	1
11	7	2	3	3
12	18	6	6	6
13	44	15	15	15
14	145	40	41	41
15	432	110	111	111
16	1290	304	306	306
17	3676	847	849	849
18	9993	2367	2376	2376
19	34298	6641	6663	6663
20	95329	18688	18729	18729
21	297260	52723	52791	52791
22	$\sim$700000	149108	149254	149254
23	$>$1500000	422634	422861	422861

Table 2: Computation time (seconds)

n	[11]	GGCount-SB	GGCount-SF	GGCount-PF
10	0.2	0.2	0.0	0.0
11	0.7	0.5	0.1	0.0
12	2.4	1.6	0.3	0.1
13	8.6	5.1	1.0	0.2
14	38.0	16.7	4.7	0.5
15	140.1	53.7	14.0	1.9
16	508.1	172.5	45.8	7.9
17	1763.7	554.1	146.4	20.9
18	6003.0	1755.0	459.3	67.6
19	17961.9	5687.2	1759.6	212.4
20	61570.0	18121.4	4616.1	652.0
21	208001.7	56263.6	17917.2	3244.3
22	$>$604800	178439.6	53671.1	5695.0
23	$>$604800	554159.8	170475.7	21313.0

run took 18 hours on 30 CPU cores and used 480 gigabytes of memory. We also confirmed a successful run for $n = 26$ on the same machine using 2 days and 1400 gigabytes; it will take three weeks to get the exact answer.

It was not difficult to modify Algorithm 1 to compute the number of cycles [13] instead of paths. The numbers of main and blocked states for cycle enumeration are M_n and M_{n-1} respectively; *count* array is initialized to be 1 for $\boxed{\circ}\boxed{\circ}\cdots\boxed{\circ}$; the answer is the total number of cycles found during the main loop. The answers to $(n+1)\times(n+1)$ grid graphs ($n \leq 25$) are shown in Table 4. For $n = 25$, we performed 9 runs with coprime moduli, of which each run took 9 hours on 30 CPU cores and used 260 gigabytes of memory.

Table 3: The number of paths between opposite corners of an $(n+1) \times (n+1)$ grid graph

n	#path
1	2
2	12
3	184
4	8512
5	1262816
6	575780564
7	789360053252
8	326659848698164 2
9	4104420870263249 6804
10	1568758030464750013214100
11	1824132915142480492414708852 36
12	6452803934327001896335718515848 2118
13	6945066476152136166427470154890735899 6488
14	2274497146768127396318264593279898633876133234 40
15	2266745568862672746374567396713098934866324885408319028
16	6874544560914993158763156313248923282458794596809945728541930 6
17	6344814611237963971310297540795524400449443986866480693646369387855336
18	1782112840842065129893384946652325275167838065704767655931452474605826692782532
19	1523344971704879993080742810319229690899454255323294555776029866737355060592877569255844
20	3962892199823037560207299517133362502106339705739463771515237113377010682364035706704472064940398
21	3137475105013710272042053813738221451310331219369872365306135199134643337938938579396557699224602131646386 8
22	7559702866673453396615191233152226193531037320724094811673914104795179257927436312349870388833176349872711714044397 92
23	5543542935523747700991431848906143793069037997096433133255695864648400840733488554456638692402087571124206008540851348293394572 0
24	1237171223120706475833874486267357083237304198901294353967872708048495169551593048564139455079215303719185802821251228092660030458138679109 4
25	8402974857881133471007083745436809127296054293775383549824742623937028497898215256929178577083970960121625602506027316549718402106494049978375604247408

Table 4: The number of cycles in an $(n+1) \times (n+1)$ grid graph

n	#path
1	1
2	13
3	213
4	9349
5	1222363
6	487150371
7	603841648931
8	2318527339461265
9	27359264067916806101
10	9888088110462835950680 99
11	1093313558101356299466983 61371
12	3695491796203988495338786833 4644457
13	3815770368857784530444553085 1242055267353
14	1202857893405338595584051242135928775169313717 15
15	1157093265735985301622023713535739570361128480949044165
16	3395384402338662155899230271369892203280641933074831282287 5435
17	3038442783379807358500340876029076156976090161888565971315511241074745
18	8289947809407545469351551937239994563974446123514580138331960668228829564586 45
19	6894541831135080568307432576946100739180275848844359458319323319418358102921546470515 79
20	1747591025316879549979786952655665410563226741505288698037078203555463416502484945892166703679171
21	1349875705622938962471028631466326112216678333347022847033029140084732167590718026243401514492209204343815 1
22	3176983247327488598959085410349551941279940986303248126366248262589972241503429080497950563995951809239550460924223 45
23	2278022392967057469122350787467196393954394523806192780344358815896687834120189809917442510966973738477298174934092272349740508 7
24	4976037374055300327445222400443546830498798429935446621053707681453137859077914419511395195468579271956320503531047087399178114550443364089
25	3310986696180301144320032978096652329596523503435882543114890295466038827631907053110552071003521069706315474677769157738758382711047460804662161917199

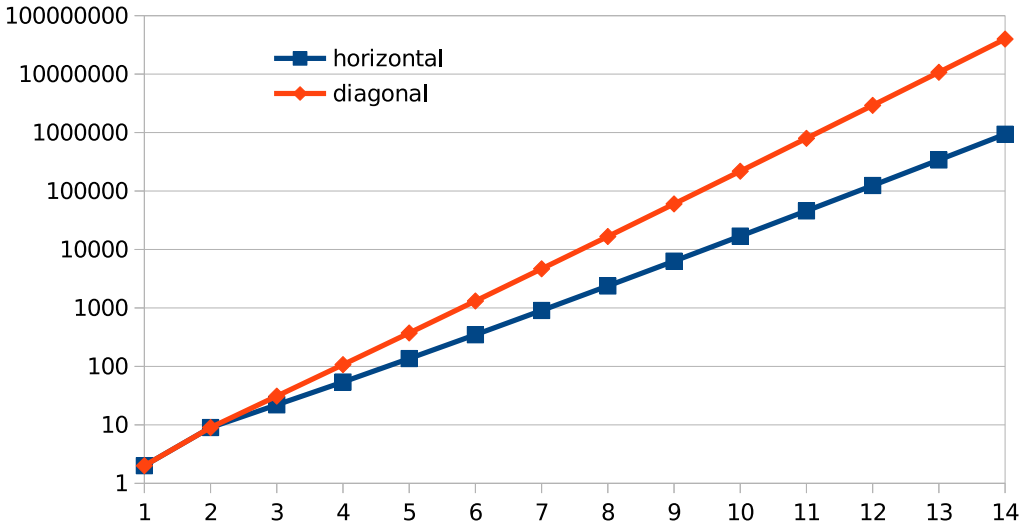


Figure 8: The number of frontier states against n

6 Additional Techniques

6.1 Using Line Symmetry

The problem is line symmetrical with respect to the line passing through $v_{(1,1)}$ and $v_{(n,n)}$; the number of paths starting from $\{v_{(1,1)}, v_{(1,2)}\}$ is exactly the same as the number of paths starting from $\{v_{(1,1)}, v_{(2,1)}\}$. It means that the answer can be computed by doubling the number of paths for one size. This method, however, would not contribute to much computational reduction because the two cases are merged soon in the breadth-first search algorithm.

One could make some algorithm utilizing line symmetry by introducing diagonal frontiers instead of horizontal frontiers. That approach seems difficult because it have to overcome faster growth of the number of frontier states than using horizontal frontiers. Figure 8 compares the peak numbers of frontier states appeared in the algorithm for general graphs [11].

6.2 Using Point Symmetry

If we use the point symmetry of the problem, Algorithm 1 can be stopped at the middle of computation. We can compute the answer by enumerating every pair of main states that matches. Figure 9 shows that paths are completed by $A =$ [(][(][)][)][○][)] and $B =$ [(][○][)][)][(][)] ($C =$ [(][○][(][)][)][)]) when B (C) is turned over and they are combined. The total number of paths found here is

$$count(A) \times (count(B) + count(C)) .$$

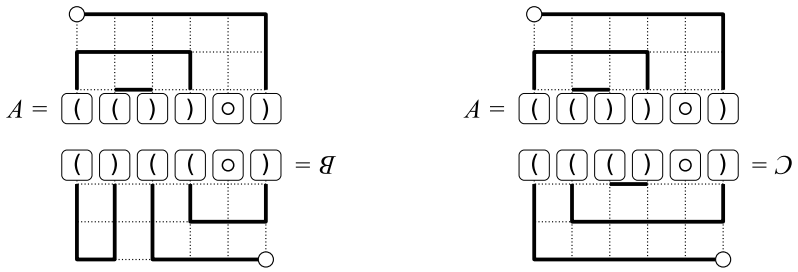


Figure 9: Example of matching

Numbers stored in *count* grows exponentially against the number of loop iterations in the algorithm. This technique reduces the memory usage by half because integer size can be halved when we only compute numbers for the upper half of the graph. As for the time cost, however, this technique might be at a disadvantage. Time growth of the matching process against n was larger than that of the basic algorithm in our preliminary experiments.

7 Conclusions

We have maximized space and time efficiency by focusing only on grid graphs. On the premise that the graph is a square grid, we were able to clarify the exact set of possible frontier states and to analyze their transition patterns. They allowed us to use simple arrays instead of dynamic hash tables and to integrate various techniques, such as in-place update and parallel processing, into the algorithm. Our algorithm for grid graphs have achieved double-digit performance improvement on the original algorithm for general graphs. It have extended the numbers recorded in OEIS [6][13] and have verified those past results correctly.

The YouTube-animation [4] demonstrated the importance of algorithmic techniques against combinational explosion. We were able to realize it through this challenge. We would like to thank all staff involved in the animation and its 1.3 million viewers who took interest in this issue, which was a major force to boost our study.

References

[1] Paul J. Flory. "The Configuration of Real Polymer Chains". In: *Journal of Chemical Physics* 17 (3 1949), pp. 303–310.

[2] N. Madras and G. Slade. *The Self-Avoiding Walk*. Birkhäuser, 1993. ISBN: 978-0-8176-3891-7.

[3] E. W. Weisstein. *Self-Avoiding Walk*. URL: `http://mathworld.wolfram.com/Self-AvoidingWalk.html`.

[4] MiraikanChannel. *Time with class! Let's count!* URL: `http://www.youtube.com/watch?v=Q4gTV4r0zRs`.

[5] 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.

[6] 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`.

[7] A. R. Conway, I. G. Enting, and A. J. Guttmann. "Algebraic Techniques for Enumerating Self-avoiding Walks on the Square Lattice". In: *Journal of Physics A: Mathematical and General* 26 (1993), pp. 1519–1534.

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

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

[10] 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.

[11] H.Iwashita, J. Kawahara, and S. Minato. *ZDD-Based Computation of the Number of Paths in a Graph*. Tech. rep. TCS-TR-A-12-60. Division of Computer Science, Graduate School of Information Science and Technology, Hokkaido University, 2012. URL: `http://www-alg.ist.hokudai.ac.jp/tra.html`.

[12] R. Donaghey and L. W. Shapiro. "Motzkin Numbers". In: *Journal of Combinatorial Theory, Seires A* 23.3 (1977), pp. 291–301.

[13] The On-Line Encyclopedia of Integer Sequences. *A140517 Number of cycles in an $n \times n$ grid.* URL: `http://oeis.org/A140517`.