Vol.2013-AL-143 No.8
2013/3/1

最小完全ハッシュ関数を用いたグリッドグラフ上の効率的なパス数え上げ

岩下 洋哲[1,2,a)] 中澤 吉男[3] 川原 純[4] 宇野 毅明[5] 湊 真一[2,1]

概要：正方形を縦横それぞれ n 分割してできる $(n+1)\times(n+1)$ グリッドグラフにおいて、対角の 2 頂点を結ぶパスの数は n に対して急激に増大する。例えば $n=10$ に対しては 10^{24} もの数となり、もはや一つずつ列挙するようなことはできない大きさである。これまでに Knuth のアルゴリズムに基づく方法で $n=21$ までのパス数が計算されているが、我々はグリッドグラフの性質を利用して計算速度と使用メモリを大幅に改善し、$n=23$ までの計算に成功した。

Iwashita Hiroaki[1,2,a)] Nakazawa Yoshio[3] Kawahara Jun[4] Uno Takeaki[5] Minato Shin-ichi[2,1]

1. はじめに

グラフにおいて同じ点を 2 回以上通らないパスは self-avoiding walk（SAW）とも呼ばれ、化学者 Flory がポリマー分子鎖構造のモデルとして導入したものとして知られている [1]。その単純な定義にも関わらず、SAW には多くの難解な数学問題が隠されている [2], [3]。$(n+1)\times(n+1)$ グリッドグラフの対角頂点を結ぶパスの数を求める問題は Youtube 動画 [4] のヒットによって「おねえさんの問題」として注目されることになったが、これも単なる公式や漸化式で求める方法が知られていない難解な SAW 問題の一つである。

The On-Line Encyclopedia of Integer Sequences[5] によれば、1981 年に Rosendale が $n=11$ までの解を求め、1995 年に Knuth が $n=12$ に対する解を求めた。2005 年には、Bousquet-Mélou らが論文 [6] の中で $n=19$ まで求めた結果を示している。彼らのアルゴリズムは Conway らによる transfer matrix 法 [7] に基づいており、対象がグリッドグラフであることを最大限に利用している。一方 Knuth は 2011 年発行の著書「The Art of Computer Programming, Volume 4A」7.1.4 節の演習問題 255 で、任意のグラフの 2 点を結ぶ全てのパスを圧縮表現した ZDD（ゼロサプレス型二分決定グラフ）[8] を構築する新しいアルゴリズム Simpath を示した [9], [10]。2012 年に発表された現在の記録 $n=21$ は、Simpath アルゴリズムを ZDD 構造ではなくパス数の合計を求めるように変更することによって得られた結果である [11]。

$n=19$ を計算した方法 [6] がグリッドグラフ専用のものであったのに対して、$n=21$ を計算した方法 [11] が任意のグラフに対して適用可能なものであることは注目に値するだろう。これは、Simpath に基づく方法の効率の良さを示しているとともに、グリッドグラフに特化すればさらなる改善が期待できるということでもある。

我々は、Simpath の幅優先探索における状態および状態遷移の管理に transfer matrix 法と同様な考え方を導入した。これによって計算途中に現れる状態数（幅優先探索における「幅」）を事前に正確に数え上げておくことが可能になり、出現する全ての状態に無駄なく通し番号のインデックスを与えることに成功した。従来は状態と途中結果（多倍長整数）の対応を動的なハッシュテーブルに保持しなければならなかったが、これによって、同じ情報を（多倍長整数を要素とした）固定長の配列に収められるようになった。さらに、状態からインデックスへの高速な変換手法や作業

1 科学技術振興機構 ERATO 湊離散構造処理系プロジェクト
ERATO MINATO Discrete Structure Manipulation System Project, Japan Science and Technology Agency, Sapporo 060-0814, Japan.
2 北海道大学大学院 情報科学研究科
Graduate School of Information Science and Technology, Hokkaido University, Sapporo 060-0814, Japan.
3 アマチュアプログラマー
Amateur programmer.
4 奈良先端科学技術大学院大学 情報科学研究科
Graduate School of Information Science, Nara Institute of Science and Technology, Ikoma, Nara 630-0192, Japan.
5 国立情報学研究所 情報学プリンシプル研究系
Principles of Informatics Research Division, National Institute of Informatics, Chiyoda, Tokyo 101-8430, Japan.
a) iwashita@erato.ist.hokudai.ac.jp

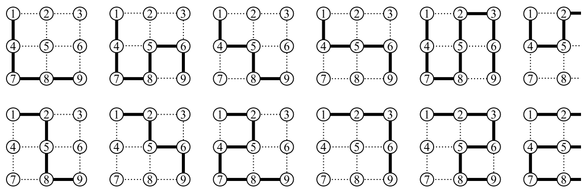


図 1　3×3 グリッドグラフの頂点 1, 9 を結ぶパス

用のデータ領域を最小限に抑えるアルゴリズムの工夫などにより、[11] に対する大幅な高速化とメモリ使用量削減を達成した。

2. 汎用的なパス数え上げ手法

Knuth の Simpath アルゴリズム [9], [10] は、頂点の集合 $V = \{v_1, \ldots, v_m\}$ と辺の集合 $E = \{e_1, \ldots, e_n\}$ で定められるグラフ $G = (V, E)$ が与えられると、v_1 と v_m を（同じ頂点を 2 度通らず）結ぶ全てのパスを列挙する。例えば図 1 に示すように、3×3 グリッドグラフで左上と右下の頂点を結ぶパスは、全部で 12 通りある。

パスは辺の部分集合として表現することができる。Simpath アルゴリズムでは、それぞれの辺をパスの構成要素に含めるか含めないかの選択に対応した 2^E の空間を幅優先で探索する。その際、最終的にパスにならないケースを早く検出して探索の枝刈りをするだけでなく、そこから先の辺の選び方に関する制約が全く同じケースを併合しながら探索を進めることが高速化のポイントとなっている。

探索の結果は木ではなく DAG（無閉路有向グラフ）となる。DAG の各ノードには、ノードのインデックスに加えて mate と呼ばれる配列が対応付けられる。ノード p がインデックス i と配列 $mate_p$ を持つとき、$mate_p$ はそれまでに選択された辺の集合 $E_p \subseteq \{e_1, \cdots, e_{i-1}\}$ そのものではなく、E_p を適切に抽象化した状態を表現している。処理済みの辺 $e_1, \cdots, e_{i-1}$ と未処理の辺 $e_i, \cdots, e_m$ の両方に接している頂点の集合 V_i をフロンティアと呼ぶ。$mate_p$ はフロンティアを構成する頂点の間の接続関係を示しており、V_i から $V_i \cup \{0\}$ への次のような写像を保持する。

$$mate_p[v] = \begin{cases} v & \text{if } v \text{ はどのパスにも含まれない} \\ w & \text{if } v, w \text{ はパスの両端になっている} \\ 0 & \text{if } v \text{ はあるパスの通過点になっている} \end{cases}$$

mate 配列の計算により DAG が構築される様子を図 2 に示す。見やすさのため、$mate_p[v] = v$ のエントリは空欄で示した。実線は注目している辺を含める場合、点線は含めない場合の mate 配列の変化を示している。

グラフ上のパスは、構築された DAG における根から終端節点までのパスと 1 対 1 に対応している。DAG におけるパスの数は、幅優先探索によって容易に計算することができる。[11] には、DAG 構築と同時にパス数を計算することによってより少ないメモリ使用量で結果を得るアルゴリズムが示されている。

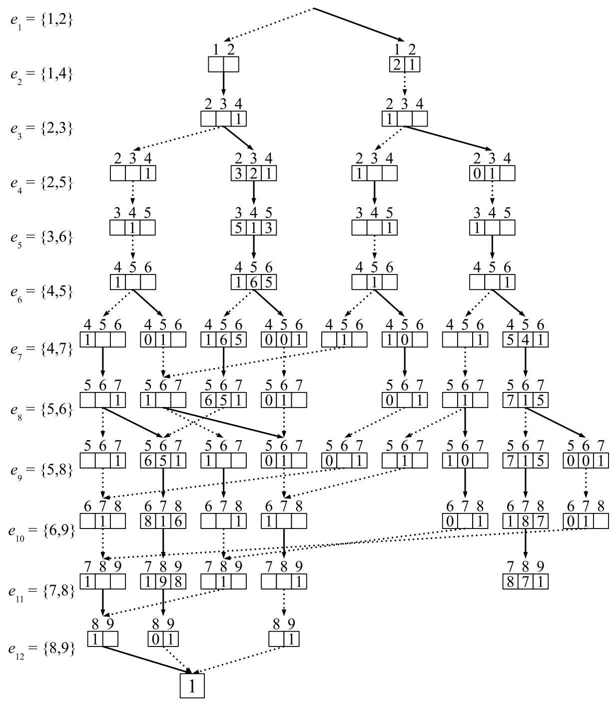


図 2　3×3 グリッドグラフの頂点 1, 9 を結ぶパスを列挙した様子

3. グリッドグラフ上のパス数え上げ手法

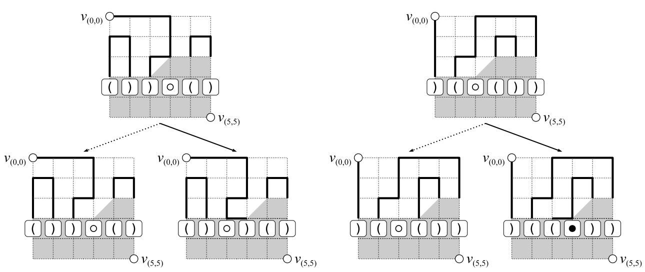
3.1 フロンティア状態

$n \times n$ グリッドグラフにおいて i 行 j 列の位置にある頂点を $v_{(i,j)}$ とする（$1 \leq i \leq n$, $1 \leq j \leq n$）。頂点を $v_{(1,1)}, \ldots, v_{(1,n)}, v_{(2,1)}, \ldots, v_{(2,n)}, \ldots$ の順に訪れ、$v_{(i,j)}$ を訪れるとき縦線 $\{v_{(i-1,j)}, v_{(i,j)}\}$ と横線 $\{v_{(i,j-1)}, v_{(i,j)}\}$ に関する選択処理を行うものとする。このとき、$V_{(i,j)} = \{v_{(i,1)}, \ldots, v_{(i,j)}, v_{(i-1,j+1)}, \cdots v_{(i-1,n)}\}$ を $v_{(i,j)}$ の処理ステップにおけるフロンティアと呼ぶ。フロンティアは常に各列から 1 つずつの頂点を含む（$|V_{(i,j)}| = n$）。

Simpath アルゴリズムにおける mate 配列に代わるものを、ここではフロンティア状態と呼ぶ。mate 配列では相手側の頂点がどれであるかを記録することでフロンティア上の端点対の関係を保持していたが、対象がグリッドグラフであることを利用すれば、より圧縮された形でフロンティア状態を表現することができる。平面グラフではパスの交差が起こらないためフロンティア上に現れる端点対が必ず入れ子構造になる。したがって相手側の頂点を明示的に記録する必要はなく、左右どちらの端点であるかだけを記録しておけば良い。そこで、$v_{(i,j)}$ の処理ステップにおけるフロンティア状態 $s = (s_0, \ldots, s_n)$ を次のように定義する。

$$s_k = \begin{cases} \boxed{(} & \text{if } k \text{ 列目の頂点は左側のパス端点} \\ \boxed{)} & \text{if } k \text{ 列目の頂点は右側のパス端点} \\ \boxed{\bullet} & \text{if } k = j \text{ かつ } k \text{ 列目の頂点はパスの通過点} \\ \boxed{\circ} & \text{if それ以外} \end{cases}$$

Vol.2013-AL-143 No.8
2013/3/1



(a) (b)

図 3 6×6 グリッドグラフのパス列挙における状態遷移の例

遷移前	採用しないとき	採用するとき
$\cdots\boxed{\circ}\boxed{\circ}\cdots$	$\cdots\boxed{\circ}\boxed{\circ}\cdots$	$\cdots\boxed{(}\boxed{)}\cdots$
$\cdots\boxed{\circ}\boxed{)}\cdots$	$\cdots\boxed{\circ}\boxed{)}\cdots$	$\cdots\boxed{)}\boxed{\bullet}\cdots$
$\cdots\boxed{\circ}\boxed{(}\cdots$	$\cdots\boxed{\circ}\boxed{(}\cdots$	$\cdots\boxed{(}\boxed{\bullet}\cdots$
$\cdots\boxed{)}\boxed{\circ}\cdots$	$\cdots\boxed{)}\boxed{\circ}\cdots$	$\cdots\boxed{\circ}\boxed{)}\cdots$
$\cdots\boxed{(}\cdots\boxed{)}\boxed{)}\cdots$	$\cdots\boxed{(}\cdots\boxed{)}\boxed{)}\cdots$	$\cdots\boxed{)}\cdots\boxed{\circ}\boxed{\bullet}\cdots$
$\cdots\boxed{)}\boxed{(}\cdots$	$\cdots\boxed{)}\boxed{(}\cdots$	$\cdots\boxed{\circ}\boxed{\bullet}\cdots$
$\cdots\boxed{(}\boxed{\circ}\cdots$	$\cdots\boxed{(}\boxed{\circ}\cdots$	$\cdots\boxed{\circ}\boxed{)}\cdots$
$\cdots\boxed{(}\boxed{(}\cdots\boxed{)}\cdots$	$\cdots\boxed{(}\boxed{(}\cdots\boxed{)}\cdots$	$\cdots\boxed{\circ}\boxed{\bullet}\cdots\boxed{(}\cdots$
$\cdots\boxed{(}\boxed{)}\cdots$	$\cdots\boxed{(}\boxed{)}\cdots$	サイクルのため禁止
$\cdots\boxed{\bullet}\boxed{\circ}\cdots$	$\cdots\boxed{\circ}\boxed{\circ}\cdots$	休止状態のため禁止
$\cdots\boxed{\bullet}\boxed{)}\cdots$	$\cdots\boxed{\circ}\boxed{)}\cdots$	休止状態のため禁止
$\cdots\boxed{\bullet}\boxed{(}\cdots$	$\cdots\boxed{\circ}\boxed{(}\cdots$	休止状態のため禁止

図 4 横線の選択による状態遷移

ここではより小さい列番号を持つ頂点を左側とし、フロンティアから抜けた始点 $v_{(0,0)}$ はどの頂点よりも左側にあると見なす。$\boxed{\bullet}$ は s_j にしか現れない特殊な値であり、次の横線 $\{v_{(i,j)}, v_{(i,j+1)}\}$ を採用してはならないという点で $s_j = \boxed{\circ}$ の場合とは異なる状態であることを示している。6×6 グリッドグラフにおけるフロンティア状態の例を図 3 に示す。図には、$v_{(3,3)}$ の処理ステップにおいて縦線 $\{v_{(2,3)}, v_{(3,3)}\}$ と横線 $\{v_{(3,2)}, v_{(3,3)}\}$ に関する選択処理を行う前後のフロンティア状態が示されている。

一般に、縦線 $\{v_{(i-1,j)}, v_{(i,j)}\}$ に関しては常に 1 通りの選択しかない。途中で分岐したり分断されたりしないためには、$v_{(i-1,j)}$ が端点のときは $\{v_{(i-1,j)}, v_{(i,j)}\}$ を採用することができず（図 3a）、$v_{(i-1,j)}$ が端点でないときは必ず $\{v_{(i-1,j)}, v_{(i,j)}\}$ を採用しなければならない（図 3b）。一方、横線 $\{v_{(i,j-1)}, v_{(i,j)}\}$ を採用するかどうかによってフロンティア状態の分岐が起こる。横線を採用しない場合にはフロンティア状態は変化せず、採用した場合は変化する。ただし例外として $s_{j-1} = \boxed{\bullet}$ のときは横線を採用することができず、その場合には採用しなくても $s_{j-1} = \boxed{\circ}$ に変化する。$\boxed{\bullet}$ を 1 つ含むフロンティア状態を休止状態と呼び、それ以外の $\boxed{\bullet}$ を含まないフロンティア状態を主状態と呼ぶことにする。横線の選択による状態遷移の全ケースを図 4 にまとめた。

フロンティア状態として有効な値列には括弧の対応関係があることから、その数は Motzkin 数 [12] に深く関連していることがわかる。平面上の座標 (x_1, y_1) からの 3 種類の動き $(1,1)$, $(1,0)$, $(1,-1)$ によって (x_2, y_2) に至るパスのうち y 座標が負の点を通らないものの集合を $\mathscr{M}_{(x_1,y_1)\to(x_2,y_2)}$ と書くことにすると、Motzkin 数は $M_n = \left|\mathscr{M}_{(0,0)\to(n,0)}\right|$ で与えられる。また、M_n は次の漸化式でも求めることもできる。

$$\begin{cases} M_0 = M_1 = 1 \\ M_n = \dfrac{3(n-1)M_{n-2} + (2n+1)M_{n-1}}{n+2} \end{cases} \tag{1}$$

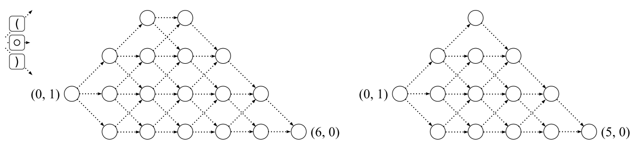
一方、$\boxed{(}$, $\boxed{\circ}$, $\boxed{)}$ をそれぞれ $(1,1)$, $(1,0)$, $(1,-1)$ に対応させると、$n \times n$ グリッドグラフにおける主状態の集合は $\mathscr{M}_{(0,1)\to(n,0)}$ に対応する（図 5a）。したがって主状態の数は次の式で与えられる。

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

同じステップにおける休止状態はどれも同じ位置に $\boxed{\bullet}$ を含んでいるため、それらを削除した長さ $n-1$ の値列として取り扱っても情報は失われない。したがってその集合は $\mathscr{M}_{(0,1)\to(n-1,0)}$ に対応し（図 5b）、休止状態の数は N_{n-1} である。以上より、$n \times n$ グリッドグラフのパス列挙における各ステップでのフロンティア状態数の最大値は

$$N_n + N_{n-1} = M_{n+1} - M_{n-1} \tag{3}$$

となることがわかる。



(a) 主状態の集合 $\mathscr{M}_{(0,1)\to(6,0)}$ (b) 休止状態の集合 $\mathscr{M}_{(0,1)\to(5,0)}$

図 5 6×6 グリッドグラフのパス列挙におけるフロンティア状態

3.2 数え上げのアルゴリズム

パス数を計算するアルゴリズムの概要を Algorithm 1 に示す。S はフロンティア状態の集合、$count[s]$ はフロンティア状態 $s \in S$ に到達する場合の数を記録した変数である。ステップ 7 では図 4 の状態遷移表に基づいて $count$ を更新する。状態 s を遷移先に持つ状態の集合を $prev(s)$ とするとき、s に対する新しい値は

$$count'[s] = \sum_{r \in prev(s)} count[r]$$

で求められる。ただし、横線 $\{v_{(i,n)}, v_{(i,n+1)}\}$ は存在しないので $j = n$ のときは横線を選択するケースを除外して計算する。

Algorithm 1 パス数え上げの概要

for all $s \in S$ **do**
  $count[s] \leftarrow 0$;
end for
$count[\overbrace{\boxed{)}\boxed{\circ}\boxed{\circ}\cdots\boxed{\circ}}^{n}] \leftarrow 1$;
for $i = 1$ **to** n **do**
  for $j = 1$ **to** n **do**
    横線 $\{v_{(i,j)}, v_{(i,j+1)}\}$ の選択にともなう状態遷移のルールに従って *count* を更新;
  end for
end for
return $count[\overbrace{\boxed{\circ}\boxed{\circ}\cdots\boxed{\circ}\boxed{)}}^{n}]$;

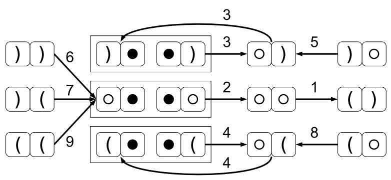


図 6　状態遷移の関係と数値の更新順序

4. 高速化と省メモリ化の手法

4.1 メモリ使用量の最小化

有効なフロンティア状態を列挙することができたので、それらに通し番号のインデックスを与えることができる。我々は、フロンティア状態を主状態と休止状態に分け、それぞれに $1,\ldots,N_n$ および $1,\ldots,N_{n-1}$ のインデックスを与えることにした。メモリ消費量かかわる作業領域は次の 2 つである。

- 主状態に到達した場合の数を保持する多倍長整数の配列 $A[1\ldots N_n]$
- 休止状態に到達した場合の数を保持する多倍長整数の配列 $B[1\ldots N_{n-1}]$

例えば 24×24 グリッドグラフの計算には、固定長 56 バイトの整数を使用する。したがって A と B に必要なメモリ量の合計は $(N_{24}+N_{23})\times 56$ bytes $=$ 413 Gigabytes となる。

状態遷移表に基づく数値更新のとき、実際に前節の $count'$ に相当するような配列全体の一時コピーを作ったとすると、メモリ使用量は一気に倍増することになってしまう。我々は更新順序の工夫でこれを回避する。

状態遷移の関係を図 6 に示す。休止状態では、例えば遷移元の状態 $\boxed{\bullet}\boxed{)}$ と遷移先の $\boxed{)}\boxed{\bullet}$ は同じインデックスを持つ（$\boxed{\bullet}$ を除去したパターンでインデックス計算される）ため、領域を共有している。基本的な処理は、遷移元の状態が保持する数値を遷移先の状態が保持する数値に加算していくことである。遷移枝に付けられたラベルはその処理順序を示している。先ずはそのステップで読み出す必要のない $\boxed{(}\boxed{)}$ が更新され、次に読み出しの終わった $\boxed{\circ}\boxed{\circ}$ が更新される。$\boxed{)}\boxed{\bullet}$, $\boxed{\bullet}\boxed{)}$ と $\boxed{\circ}\boxed{)}$ は情報を交換し合う関係なので、わずかな一時領域を使用するだけで同時に更新することができる。

4.2 フロンティア状態からインデックスへの高速変換

フロンティア状態からそのインデックスへの変換処理は頻繁に呼び出されるため、その高速化が全体の性能に大きく影響を与えることになる。

フロンティア状態を構成するコードは $\boxed{\circ}$, $\boxed{)}$, $\boxed{(}$ の 3 種類であるため、我々はそれぞれ 2 ビットを使用して 00, 01, 10 とコード化する。そうすると 32×32 グリッドグラフまでは 64 ビット CPU の 1 ワード内に保持することができ、効率が良い。その 1 ワードのコードには使用されない値が散在するため、それを効率的にインデックス値に変換する手段が必要である。そこで、フロンティア状態を構成するビット列を左側 $n/2$ 列に対応する n ビットほどのコードと残りの右側に対応する n ビットほどのコードに分割し、それぞれのコード値を直接使って参照する 2 つのテーブルを用意する。テーブルの大きさはどちらも 2^n 程度であり、これはフロンティア状態の総数（$O(3^n)$）に比べて十分に小さい。それぞれのテーブルには左側／右側コードに対応する上位／下位インデックス値を記録しておき、2 つの値の和を計算するだけで最終的なインデックス値が得られる。

この方法がうまく働くのは、フロンティア状態が Motzkin パス（のようなもの）を構成しているためである。有効なフロンティア状態を構成するコードであれば左側コードの右端と右側コードの左端は必ず同じ y 座標で出会うため、そこでテーブルを 2 つに分割しても副作用が発生しない。

5. 実験結果

本手法によるパス数え上げプログラムを C++で実装し、性能を測定した。実験に使った計算機は 2.67GHz Intel Xeon E7-8837 CPU で、1TB の主記憶を搭載している。OS は 64-bit SUSE Linux Enterprise Server 11 である。

表 1 にメモリ使用量を、表 2 に CPU 時間を示す。任意のグラフを対象とした実装 [11] に対して 5 倍のメモリ使用効率向上と 10 倍の高速化を確認することができた。本手法 1 は 4.2 節の技法を使わない場合、本手法 2 は使った場合の結果である。本手法 2 の方が本手法 1 に対してわずかに多くのメモリを消費するが、実行速度は 3 倍に向上している。

本手法によって、$(n+1)\times(n+1)$ グリッドグラフのパス数え上げにおける $n = 22, 23$ に対する解を初めて求めることができた。表 2 にその計算結果を掲載しておく。

表 3　$(n+1)\times(n+1)$ グリッドグラフの対角頂点を結ぶパスの数

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
22	755970286667345339661519123315222619353103732072409481167391410479517925792743631234987038883317634987271171404439792
23	55435429355237477009914318489061437930690379970964331332556958646484008407334885544566386924020875711242060085408513482933945720

表 1　メモリ使用量（MB）

n	従来法 [11]	本手法 1	本手法 2
10	3	1	1
11	7	2	3
12	18	6	6
13	44	15	15
14	145	40	41
15	432	110	111
16	1290	304	306
17	3676	847	849
18	9993	2367	2376
19	34298	6641	6663
20	95329	18688	18729
21	297260	52723	52791
22	>512000	149108	149254
23	>512000	422634	422861

表 2　CPU 時間（秒）

n	従来法 [11]	本手法 1	本手法 2
10	0.2	0.2	0.0
11	0.7	0.5	0.1
12	2.4	1.6	0.3
13	8.6	5.1	1.0
14	38.0	16.7	4.7
15	140.1	53.7	14.0
16	508.1	172.5	45.8
17	1763.7	554.1	146.4
18	6003.0	1755.0	459.3
19	17961.9	5687.2	1759.6
20	61570.0	18121.4	4616.1
21	208001.7	56263.6	17917.2
22	N/A	178439.6	53671.1
23	N/A	554159.8	170475.7

6. さらなる改善

6.1 点対称性の利用

問題の点対称性を利用すると、グリッドグラフを全体の半分の行まで処理したところで途中結果を 180 度回転したものとのマッチングを取ることで最終結果を求めることができる。図 7 の例では、フロンティア状態 $A=\boxed{(}\boxed{(}\boxed{)}\boxed{)}\boxed{\circ}\boxed{)}$ に対して 180 度回転後にマッチするフロンティア状態には $B=\boxed{(}\boxed{\circ}\boxed{)}\boxed{)}\boxed{(}\boxed{)}$ と $C=\boxed{(}\boxed{\circ}\boxed{(}\boxed{)}\boxed{)}\boxed{)}$ があることを示している。フロンティア状態 s に到達する場合の数が $count[s]$ に記録されているとき、上半分のフロンティア状態が A になるようなパスの総数は $count[A]\times(count[B]+count[C])$ である。

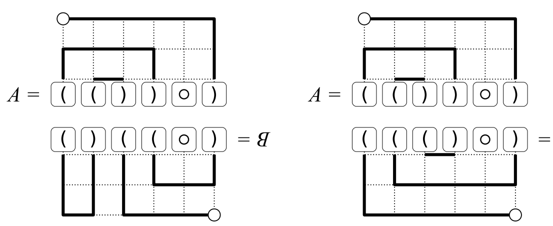


図 7　マッチングの例

2 つのフロンティア状態のマッチングにおいては、必ず、線のあるところは線のあるところと、線のないところは線のないところと接続する。線のあるところは $\boxed{(}$ と $\boxed{)}$ の 2 種類のコードがあり、どの組み合わせでも連結する可能性があるが、線のないところは $\boxed{\circ}$ しかなく、その場所には無

条件に $\boxed{\circ}$ を割り当てればよい。そこで、$\boxed{\circ}$ を除いて $\boxed{(}$ と $\boxed{)}$ のみで表される縮退状態を考え、まずは縮退後の状態を使って、同じ長さの縮退状態の間でのマッチングを列挙する。次に、それぞれの列挙結果に対してそれをフロンティア状態のマッチングに復元するような $\boxed{\circ}$ の挿入パターンをさらに列挙する。後段の列挙は高速に実行できるので、全体として高速な処理を達成できる。

count に記録される数値の大きさは処理済みのグリッド行数に対して指数的に増大し、その桁数はほぼ一定の速度で増えていく。マッチング処理を行うことで処理するグリッド行数は半分で済み、*count* に記録する数値の桁数も半分で済むため、必要なメモリ量も約半分になる。

計算時間については、問題が大きくなるほど前半の数え上げよりも後半のマッチングにかかる処理時間の比率が大きい傾向が見られる。予備実験でのこの手法による速度改善率は、$(n+1)\times(n+1)$ グリッドグラフにおいて $n=14$ のとき 3.2 倍、$n=21$ では 1.7 倍であった。

6.2 並列化

この問題は、[6] の実装のようにモジュロ演算と Chinese remainder theorem を利用すれば、合計の CPU 時間を使用する代わりに、より小さいメモリで、かつ分散並列環境でも計算できることが知られている。また、今回は考慮しなかったが、密な並列計算による高速化にも可能性があるだろう。

7. おわりに

正方形を縦横それぞれ n 分割してできる $(n+1)\times(n+1)$ グリッドグラフにおいて対角の 2 頂点を結ぶパスの数を求める「おねえさんの問題」において、$n=23$ までの解を求めることに初めて成功した。従来の方法では途中状態と数値の組を保存する巨大なハッシュテーブルが必要があったが、対象をグリッドグラフに特化した本手法では出現する途中状態を正確に数え上げ、それに対する最小完全ハッシュ関数を構築することに成功した。これにより、メモリ使用量を従来の 5 分の 1、処理速度を従来の 10 倍に向上させることができた。

参考文献

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

[2] Madras, N. and Slade, G.: *The Self-Avoiding Walk*, Birkhäuser (1993).

[3] Weisstein, E. W.: Self-Avoiding Walk, MathWorld—A Wolfram Web Resource, `http://mathworld.wolfram.com/Self-AvoidingWalk.html`.

[4] 日本科学未来館：『フカシギの数え方』おねえさんといっしょ！みんなで数えてみよう！, `http://www.youtube.com/watch?v=Q4gTV4r0zRs` (2012).

[5] of Integer Sequences, T. O.-L. E.: A007764 Number of nonintersecting (or self-avoiding) rook paths joining opposite corners of an $n\times n$ grid, `http://oeis.org/A007764`.

[6] Bousquet-Mélou, M., Guttmann, A. J. and Jensen, I.: Self-avoiding Walks Crossing a Square, *Journal of Physics A: Mathematical and General*, Vol. 38, pp. 9159–9181 (2005).

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

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

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

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

[11] H.Iwashita, Kawahara, J. and Minato, S.: ZDD-Based Computation of the Number of Paths in a Graph, Technical Report TCS-TR-A-12-60, Division of Computer Science, Graduate School of Information Science and Technology, Hokkaido University (2012).

[12] Donaghey, R. and Shapiro, L. W.: Motzkin Numbers, *Journal of Combinatorial Theory, Seires A*, Vol. 23, No. 3, pp. 291–301 (1977).