フカシギおねえさん問題の高速計算アルゴリズム

岩下 洋哲

JST ERATO 湊離散構造処理系プロジェクト

2013/7/26

Joint work with

中澤 吉男	アマチュアプログラマー
川原 純	奈良先端科学技術大学院大学情報科学研究科
宇野 毅明	国立情報学研究所情報学プリンシプル研究系
湊 真一	北海道大学大学院情報科学研究科

発表の流れ

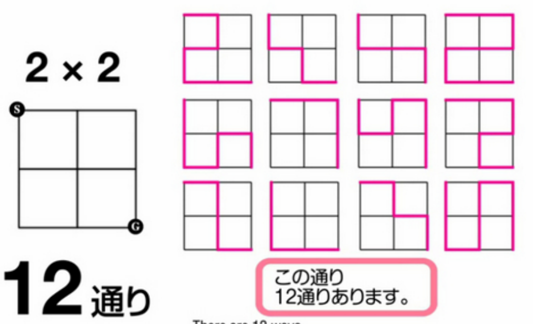
1. はじめに
2. 数え方の基本
3. 状態の圧縮表現
4. 高速化と省メモリ化の手法
5. 実験結果
6. おわりに

発表の流れ

1. はじめに
   - 格子グラフ上の道順を数え上げる問題
   - 数え上げの記録
2. 数え方の基本
3. 状態の圧縮表現
4. 高速化と省メモリ化の手法
5. 実験結果
6. おわりに

問題: スタートからゴールまでの道順は何通り？

同じところを２度通らないが、遠回りしても良い



数え上げの記録

	記録	計算時間	計算機
おねえさん*	9 × 9	6 年	スーパーコンピュータ
オネエサン*	10 × 10	25 万年	スーパーコンピュータ

*フィクションです。

数え上げの記録

	記録	計算時間	計算機（スレッド数）
おねえさん*	9 × 9	6 年	スーパーコンピュータ
オネエサン*	10 × 10	25 万年	スーパーコンピュータ
Bousquet-Mélou (2005)	19 × 19	3 日	1GHz Alpha (8)
Iwashita (Sep 2012)	21 × 21	3 日	2.67GHz Xeon (1)
Spaans (Feb 2013)	24 × 24	数週間	??? (30)
Iwashita (Apr 2013)	25 × 25	1 週間	2.67GHz Xeon (30)

*フィクションです。

計算結果

Sequence A007764 of the On-Line Encyclopedia of Integer Sequences

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	3137475105013710272042053813738221451310331219369872365306135199134643337938938579396557699224602131646386 8
22	7559702866673453396615191233152226193531037320724094811673914104795179257927436312349870388833176349872711714044397 92
23	5543542935523747700991431848906143793069037997096433133255695864648400840733488554456638692402087571124206008540851348293394572 0
24	1237171223120706475833874486267357083237304198901294353967872708048495169551593048564139455079215303719185802821251228092660030458138679109 4
25	840297485788113347100708374543680912729605429377538354982474262393702849789821525692917857708397096012162560250602731654971840210649404997837560424740 8

$$23 \times 23: \quad 5.544 \times 10^{127}$$

$\downarrow$ 2231 億倍

$$24 \times 24: \quad 1.237 \times 10^{139}$$

$\downarrow$ 6792 億倍

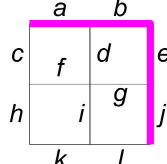
$$25 \times 25: \quad 8.403 \times 10^{150}$$

発表の流れ

1. はじめに
2. 数え方の基本
   - 探索空間
   - 状態の圧縮表現
   - 等価な状態の併合
   - 数え上げのアルゴリズム
3. 状態の圧縮表現
4. 高速化と省メモリ化の手法
5. 実験結果
6. おわりに

探索空間

- 道順 $\Longleftrightarrow$ 線分の選び方



$$a \to b \to e \to j \quad \Longleftrightarrow \quad \{a, b, e, j\}$$

$$c \to f \to d \to b \to e \to j \quad \Longleftrightarrow \quad \{b, c, d, e, f, j\}$$

※ 選ぶ順番は無関係 $\cdots$ 探索空間は 2^{12}

- 左上から順に、横線の選択で場合分け

1

横線 1 を
含めない

横線 1 を
含める

2

2

横線 2 を
含めない

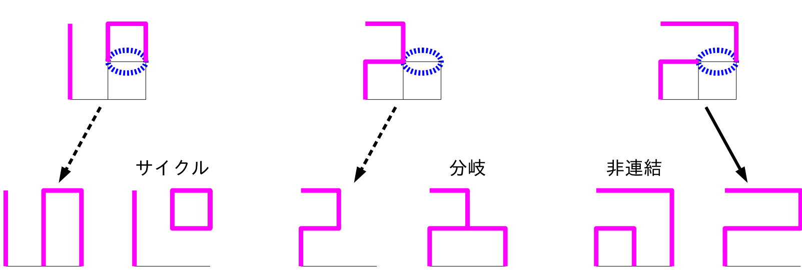
横線 2 を
含める

横線 2 を
含めない

横線 2 を
含める

※ 縦線は自動的に決まる · · · 探索空間は 2^6

- パス完成の見込みが無い状態は除外



2 × 2の探索結果

状態の圧縮表現

状態の併合

) ○ ○ 1

) ○ ○ 1　○) ○ 1

) ○ ○ 1　) ( ) 1　○) ○ 1　○ ○) 1

) ○ ○ 1　○) ○ 2　) ( ) 1　○ ●) 1　) ● ○ 1　○ ○) 1　( )) 1

) ○ ○ 3　) ( ) 2　○) ○ 3　○ ○) 4

○) ○ 6　○ ●) 2　○ ○) 4

○ ○) 12

アルゴリズムの概要

1: *count* の全要素を 0 で初期化;
2: $count[\overbrace{\boxed{)}\boxed{\circ}\boxed{\circ}\cdots\boxed{\circ}}^{n+1}] \leftarrow 1$;
3: **for** $i = 1$ **to** $n+1$ **do**
4: **for** $j = 1$ **to** n **do**
5: *tmp* の全要素を 0 で初期化;
6: **for all** 状態 s **do**
7: $s_0 \leftarrow j$ 番目の横線を含めないときの s からの遷移先;
8: $s_1 \leftarrow j$ 番目の横線を含めるときの s からの遷移先;
9: $tmp[s_0] \leftarrow tmp[s_0] + count[s]$;
10: $tmp[s_1] \leftarrow tmp[s_1] + count[s]$;
11: **end for**
12: $count \leftarrow tmp$;
13: **end for**
14: **end for**
15: **return** $count[\overbrace{\boxed{\circ}\boxed{\circ}\cdots\boxed{\circ}\boxed{)}}^{n+1}]$;

発表の流れ

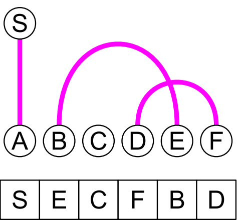
1. はじめに
2. 数え方の基本
3. 状態の圧縮表現
   - フロンティア状態
   - 全てのフロンティア状態の集合
4. 高速化と省メモリ化の手法
5. 実験結果
6. おわりに

フロンティア状態

注目する頂点集合におけるパス断片の接続関係

汎用的なパス数え上げ手法（Simpath）では

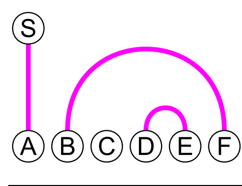
- パスの端点なら、もう一方の端点
- どこにも接続していなければ、それ自身
- パスの通過点なら、0



平面グラフでは入れ子構造が保証されるため
さらに圧縮が可能

- 左側のパス端点なら、(
- 右側のパス端点なら、)
- どこにも接続していなければ、○
- パスの通過点なら、●

※ 始点 S は最も左にあるとみなす



フロンティア状態の更新

禁止
（枝分かれ発生）

※ ● は新しい横線の右端に現れ、次の遷移で消える

状態遷移表

遷移前	横線を含めないとき	横線を含めるとき
・・・○○・・・	・・・○○・・・	・・・()・・・
・・・○)・・・	・・・○)・・・	・・・)●・・・
・・・○(・・・	・・・○(・・・	・・・(●・・・
・・・)○・・・	・・・)○・・・	・・・○)・・・
・・・(・・・))・・・	・・・(・・・))・・・	・・・)・・・○●・・・
・・・)(・・・	・・・)(・・・	・・・○●・・・
・・・(○・・・	・・・(○・・・	・・・○(・・・
・・・((・・・)・・・	・・・((・・・)・・・	・・・○●・・・(・・・
・・・()・・・	・・・()・・・	禁止（サイクル発生）
・・・●○・・・	・・・○○・・・	禁止（枝分かれ発生）
・・・●)・・・	・・・○)・・・	禁止（枝分かれ発生）
・・・●(・・・	・・・○(・・・	禁止（枝分かれ発生）

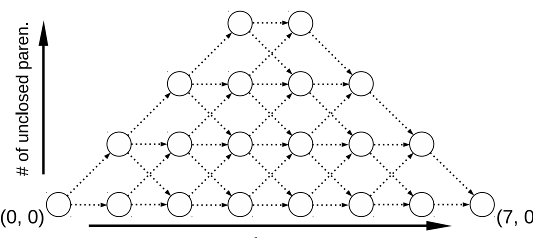
入れ子構造と Motzkin 数

Motzkin 数

$$M_0 = M_1 = 1$$

$$M_n = \frac{3(n-1)M_{n-2} + (2n+1)M_{n-1}}{n+2}$$

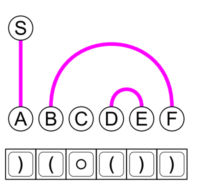
M_n は、3 種類の動き $(1,0), (1,1), (1,-1)$ によって座標 $(0,0)$ から $(n,0)$ に至る、負の座標を通らないパスの数に等しい

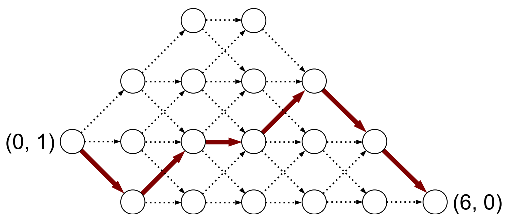


$M_7 = 127$

フロンティア状態の数と Motzkin 数

- [○] $= (1, 0)$,　[( ] $= (1, 1)$,　[ )] $= (1, -1)$
- 始点と接続している [ )] が一つ余るので $(0, 1)$ からスタート





[●] を含まない長さ L の文字列で構成されたフロンティア状態は

$$(M_{L+1} - M_L) \text{ 通り}$$

$n \times n$ 問題におけるフロンティア状態は

⊡ を含まない長さ $n+1$ の文字列: $(M_{n+2} - M_{n+1})$ 通り

⊡ を 1 つ含む長さ $n+1$ の文字列
$\leftrightarrow$ ⊡ を含まない長さ n の文字列: $(M_{n+1} - M_n)$ 通り

合計 $(M_{n+2} - M_n)$ 通り

フロンティア状態の集合 S が明らかになったことにより

最小完全ハッシュ関数 $\varphi : S \to \{1, 2, \ldots, |S|\}$ を定義できる

$$\Downarrow$$

ハッシュ表 $\{s \mapsto k \mid s \in S,\ k \text{ は整数}\}$ の代わりに配列を使い

$count[\varphi(s)]$ のようにアクセス可能

計算量のオーダーは変わらないが、現実的には大きな

高速化、省メモリ化

発表の流れ

1 はじめに

2 数え方の基本

3 状態の圧縮表現

4 高速化と省メモリ化の手法
- In-place 更新
- 高速な最小完全ハッシュ関数
- 高速な状態列挙
- 共有メモリ並列処理
- 中国の剰余定理

5 実験結果

アルゴリズムの概要（再掲）

1: *count* の全要素を 0 で初期化;
2: $count[\overbrace{\boxed{)}\boxed{\circ}\boxed{\circ}\cdots\boxed{\circ}}^{n+1}] \leftarrow 1$;
3: **for** $i = 1$ **to** $n + 1$ **do**
4: **for** $j = 1$ **to** n **do**
5: *tmp* の全要素を 0 で初期化;
6: **for all** 状態 s **do**
7: $s_0 \leftarrow j$ 番目の横線を含めないときの s からの遷移先;
8: $s_1 \leftarrow j$ 番目の横線を含めるときの s からの遷移先;
9: $tmp[s_0] \leftarrow tmp[s_0] + count[s]$;
10: $tmp[s_1] \leftarrow tmp[s_1] + count[s]$;
11: **end for**
12: $count \leftarrow tmp$;
13: **end for**
14: **end for**
15: **return** $count[\overbrace{\boxed{\circ}\boxed{\circ}\cdots\boxed{\circ}\boxed{)}}^{n+1}]$;

count 配列をその場で書き換える（In-place 更新）

( ) ←1 ○ ○ ←2 ● ○ = ○ ● ←3))
←3) (
←3 ((

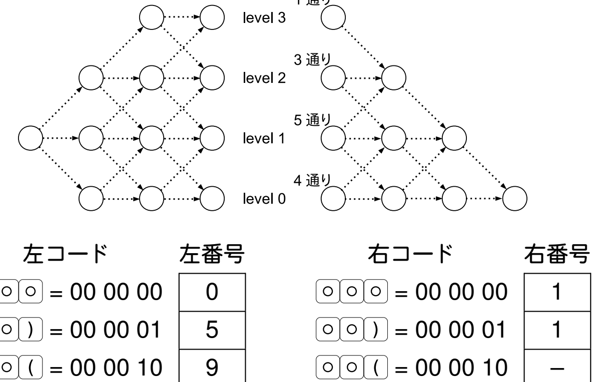
●) =) ● ↔1 ○) ←2) ○

● (= (● ↔1 ○ (←2 (○

依存関係と更新順序

tmp 配列を使わない ⇒ メモリ半減

ハッシュ関数: 2分割コードからの変換表で実現

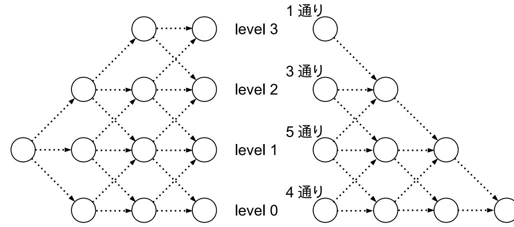


左コード		左番号
○○○	= 00 00 00	0
○○)	= 00 00 01	5
○○(	= 00 00 10	9
	00 00 11	–
○)○	= 00 01 00	12
		⋮

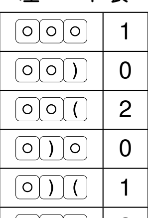
右コード		右番号
○○○	= 00 00 00	1
○○)	= 00 00 01	1
○○(	= 00 00 10	–
	00 00 11	–
○)○	= 00 01 00	2
		⋮

ハッシュ値 = 左番号 + 右番号

状態列挙の高速化: 階層化された状態コード表を参照

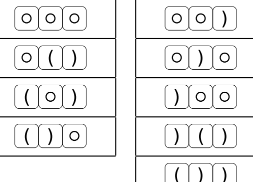


左コード表



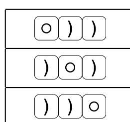
状態コード	値
○○○	1
○○)	0
○○(	2
○)○	0
○)(	1
○(○	2

⋮



右コード表 0
○○○
○()
(○)
()○

右コード表 1
○○)
○)○
)○○
)()
())



右コード表 2
○))
)○)
))○

右コード表 3
)))

並列処理: データ更新の局所性を活用

フロンティア状態の変化パターン

1. 横線位置の 2 文字だけ変化する場合
2. 横線位置の 2 文字に加え、他の 1 箇所で (↔) となる場合

遷移前	横線を含めないとき	横線を含めるとき
··· ○ ○ ···	··· ○ ○ ···	··· ( ) ···
··· ○) ···	··· ○) ···	···) ● ···
··· ○ (···	··· ○ (···	··· (● ···
···) ○ ···	···) ○ ···	··· ○) ···
··· (···)) ···	··· (···)) ···	···) ··· ○ ● ···
···) (···	···) (···	··· ○ ● ···
··· (○ ···	··· (○ ···	··· ○ (···
··· ((···) ···	··· ((···) ···	··· ○ ● ··· (···
··· ( ) ···	··· ( ) ···	禁止（サイクル発生）
··· ● ○ ···	··· ○ ○ ···	禁止（枝分かれ発生）
··· ●) ···	··· ○) ···	禁止（枝分かれ発生）
··· ● (···	··· ○ (···	禁止（枝分かれ発生）

並列処理: データ更新の局所性を活用

フロンティア状態の変化パターン

1. 横線位置の 2 文字だけ変化する場合
2. 横線位置の 2 文字に加え、他の 1 箇所で (↔) となる場合

- 横線位置以外の m 箇所（$1 \leq m \leq n+1-2$）の文字に注目
- m ビットの 2 進数を ID とする 2^m 個の状態グループを作成
- 0 は ○、1 は) or (に対応

（$m=2$ の例）

00	=	···	○	···	○	···
01	=	···	○	···	) or (	···
10	=	···	) or (	···	○	···
11	=	···	) or (	···	) or (	···

- 状態グループを越える依存関係は無い ⇒ 並列計算

- メモリ使用のほとんどは（多倍長）整数の巨大な配列
- 25 × 25 問題の解を表現するには 502 ビットの整数が必要 ··· そこで、

中国の剰余定理　（中国人の剰余定理／孫子の定理）

与えられた整数 $m_1, m_2, \ldots, m_k$ がどの二つも互いに素ならば、任意の整数 $a_1, a_2, \ldots, a_k$ に対し

$$\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \\ &\vdots \\ x &\equiv a_k \pmod{m_k} \end{aligned}$$

を満たす整数 x が $m_1 m_2 \cdots m_k$ を法として一意的に存在する。

64 ビット整数のモジュロ演算を使えば 1 回のメモリ使用量が約 1/8 に

発表の流れ

1. はじめに
2. 数え方の基本
3. 状態の圧縮表現
4. 高速化と省メモリ化の手法
5. **実験結果**
6. おわりに

実験

- 4 つのプログラム

  Original 通常のハッシュ表を使用（任意のグラフに適用可能）
  Grid-BS 最小完全ハッシュ法と In-place 更新を使用
  Grid-FS Grid-BS にハッシュ関数と状態列挙の高速化手法を追加
  Grid-FP Grid-FS を共有メモリ並列処理を追加

- 計算機
  - CPU: Intel Xeon E7-8837 / 2.67GHz / 32 cores
  - Memory: 1.5TB
- 並列処理には 12 スレッドを使用

CPU 時間

1000000
100000
10000
1000
100
10
1

Original
Grid-BS
Grid-FS
Grid-FP

11 12 13 14 15 16 17 18 19 20 21 22 23

最小完全ハッシュで法 1/3 → ハッシュ関数などの高速化で 1/3
→ 12 スレッド並列処理で 1/10

メモリ使用量

1000000
100000
10000
1000
100
10
1

11 12 13 14 15 16 17 18 19 20 21 22 23

Original
Grid-BS
Grid-FS
Grid-FP

最小完全ハッシュ法と In-place 更新で 1/5 以下
（中国の剰余定理の併用でさらに削減）

発表の流れ

1 はじめに
2 数え方の基本
3 状態の圧縮表現
4 高速化と省メモリ化の手法
5 実験結果
6 おわりに

格子グラフであることを利用した高速なパス数え上げアルゴリズム

- 途中状態の圧縮表現を定義
  - 全ての状態と状態遷移を明確化
- 様々な高速化／省メモリ化手法を開発
  - In-place 更新アルゴリズム
  - テーブル参照による高速なハッシュ関数／状態列挙
  - 共有メモリ並列処理

25 × 25 の実行結果

- 条件
  - Intel Xeon E7-8837 / 2.67GHz / 32 cores / 1.5TB memory
  - 30 スレッド並列実行、64 ビット計算（中国の剰余定理）
- 結果
  - 計算時間　18 時間 × 9 回 = 1 週間
  - メモリー使用量　480GB

26 × 26 は 3 週間、1400GB で計算可能