グリッドグラフを対象としたパス数え上げ手法の最適化

岩下 洋哲 [13]　宇野 毅明 [2]　川原 純 [13]　湊 真一 [31]

[1]JST ERATO 湊離散構造処理系プロジェクト

[2] 国立情報学研究所情報学プリンシプル研究系

[3] 北海道大学大学院情報科学研究科

2012/10/15

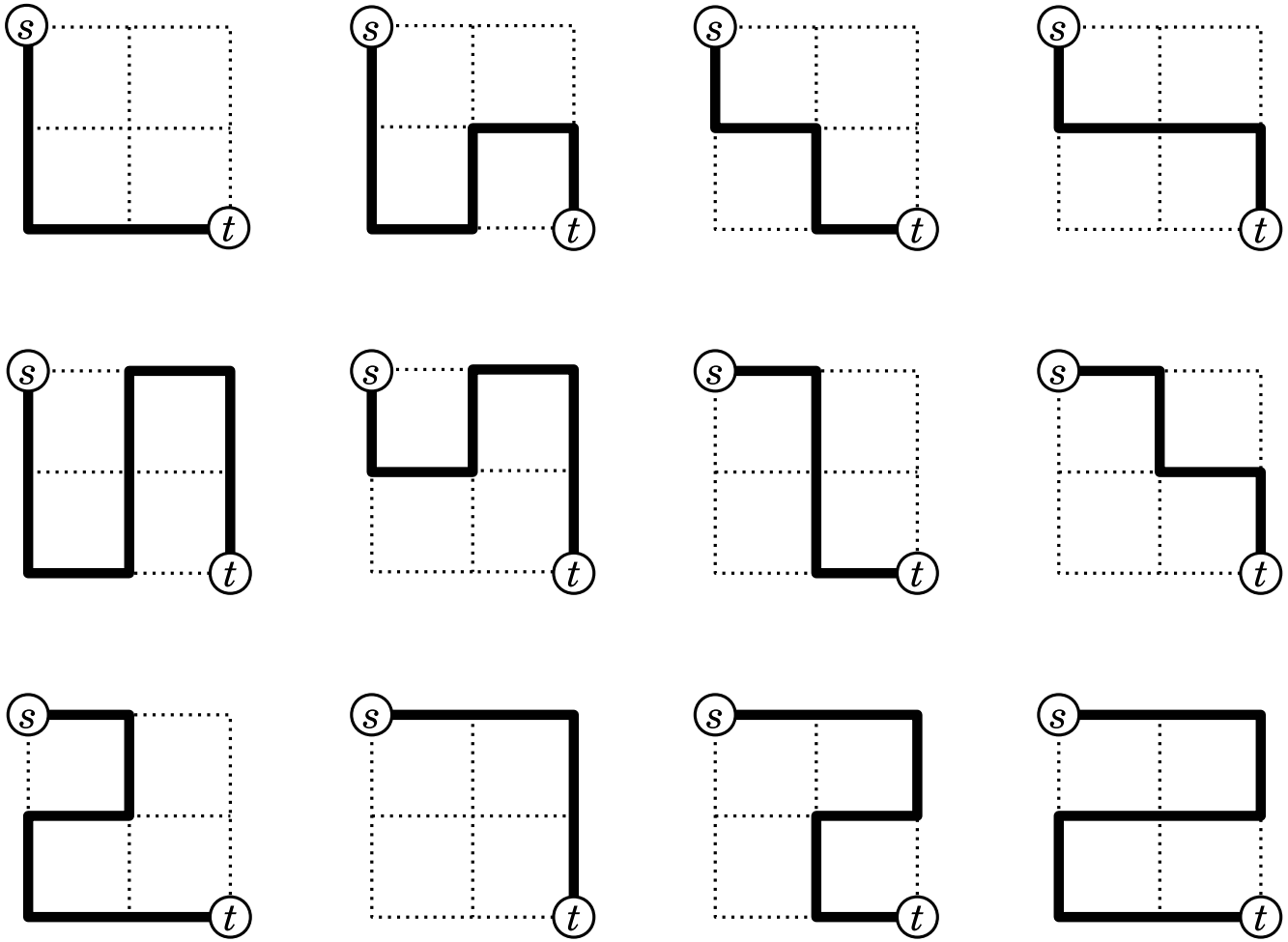
発表の流れ

1. はじめに
   - グリッドグラフのパスを数え上げる問題
   - フロンティア法によるパスの数え上げ
2. グリッドグラフであることを利用したパス数え上げ
   - 状態数 $O(3^n)$
   - 完全ハッシュ関数
   - アルゴリズム
3. 実験結果
   - 任意のグラフを対象とした実装との比較

発表の流れ

1. はじめに
   - グリッドグラフのパスを数え上げる問題
   - フロンティア法によるパスの数え上げ
2. グリッドグラフであることを利用したパス数え上げ
   - 状態数 $O(3^n)$
   - 完全ハッシュ関数
   - アルゴリズム
3. 実験結果
   - 任意のグラフを対象とした実装との比較

スタートからゴールまでの通り方（同じところを２度通らない）

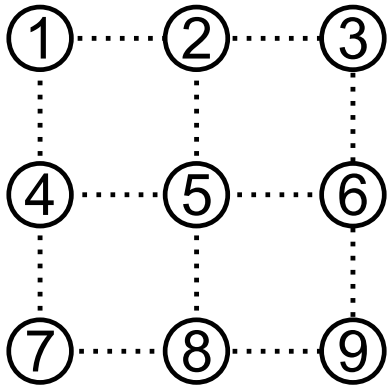


２×２のとき: 12 通り

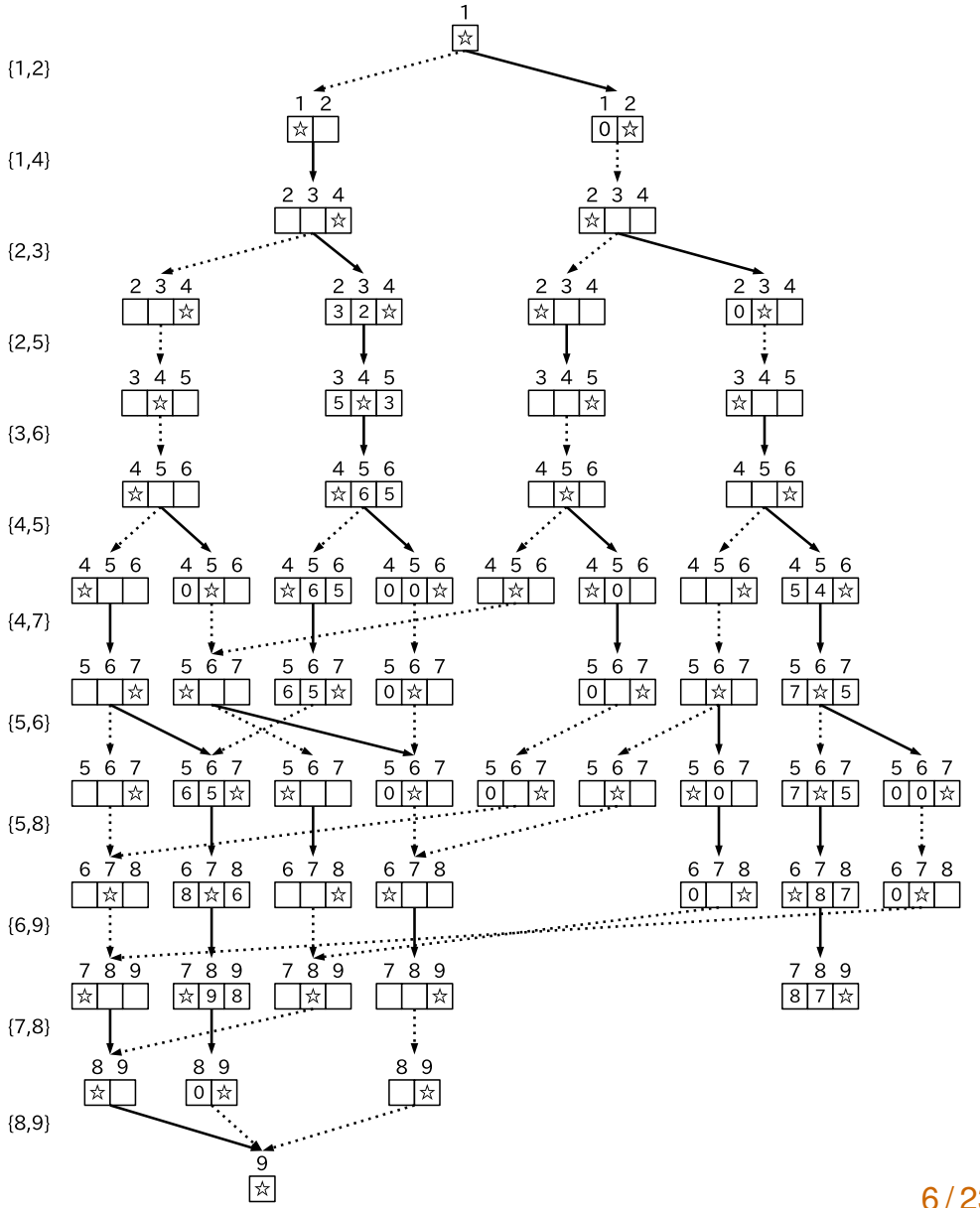
数え上げの記録

	記録	対象	年
Bousquet-Mélou et al.	19×19	グリッドグラフ	2005
Iwashita/Kawahara/Minato	21×21	任意のグラフ	2012
本日の発表	22×22	グリッドグラフ	2012
おねえさん	9×9	任意のグラフ	2018
オネエサンロボット	10×10	任意のグラフ	252018

フロンティア法によるパスの数え上げ



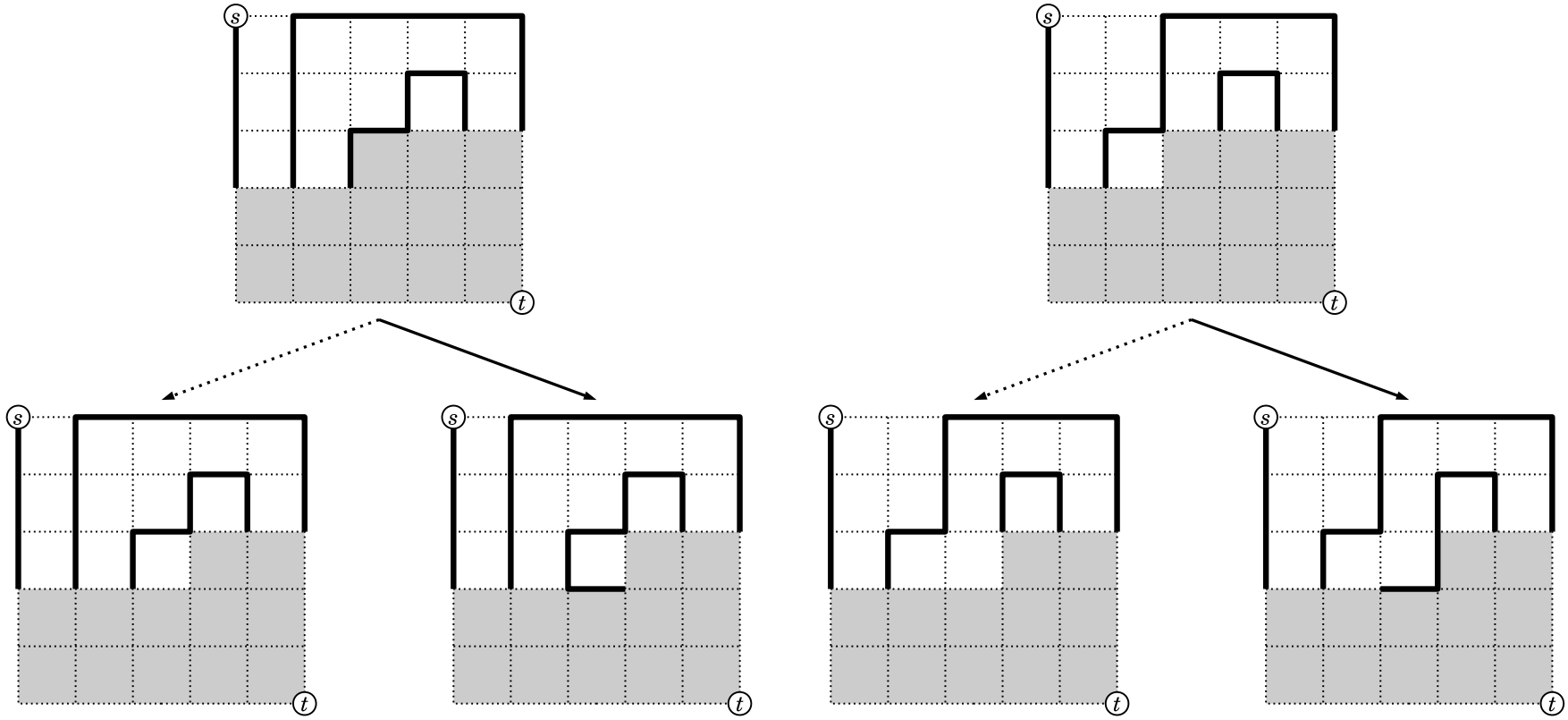
- 辺を 使う／使わない の２択
- 状態 $\mapsto$ 場合の数（bignum）をハッシュ表に記録
- 不要になったエントリを削除しながら幅優先探索



発表の流れ

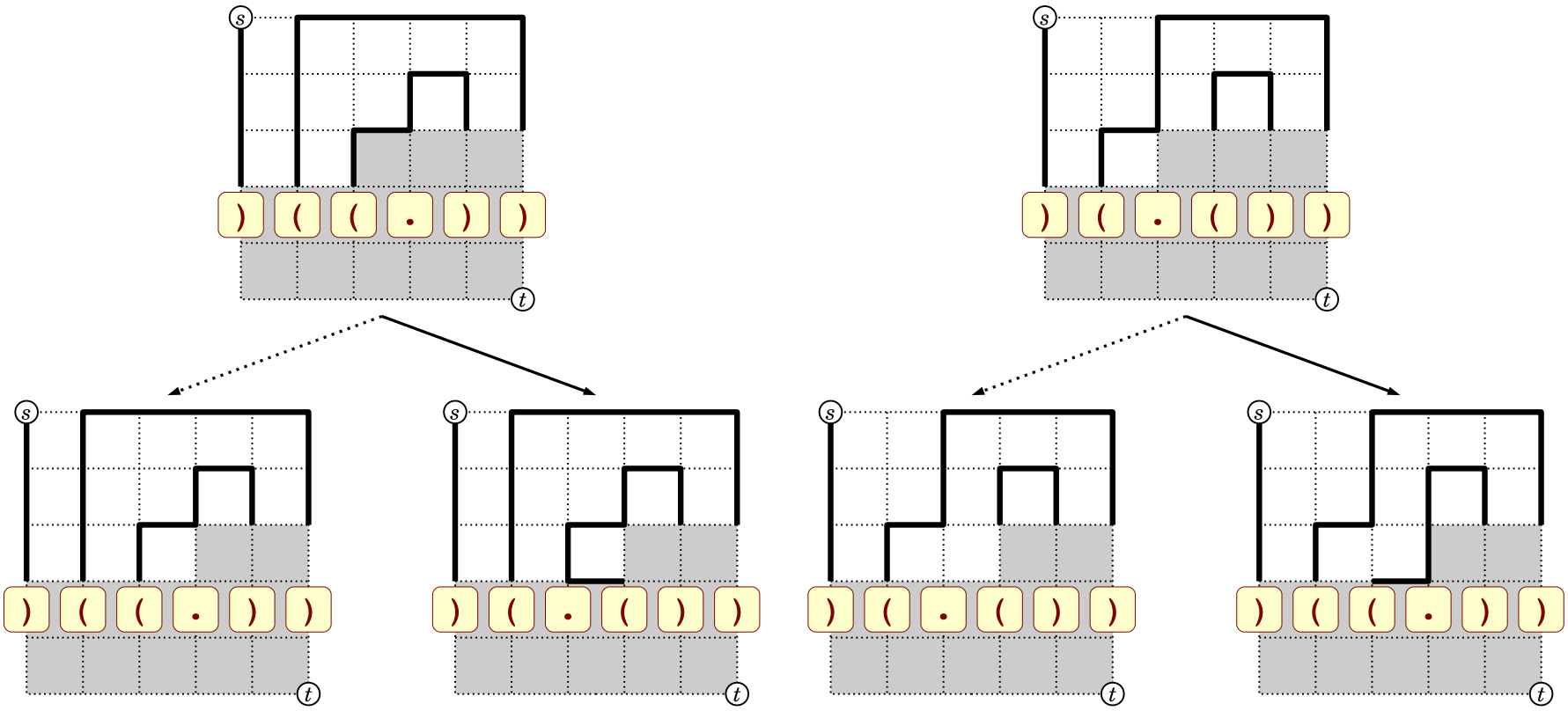
1. はじめに
   - グリッドグラフのパスを数え上げる問題
   - フロンティア法によるパスの数え上げ
2. グリッドグラフであることを利用したパス数え上げ
   - 状態数 $O(3^n)$
   - 完全ハッシュ関数
   - アルゴリズム
3. 実験結果
   - 任意のグラフを対象とした実装との比較

フロンティア法によるグリッドグラフのパス数え上げ



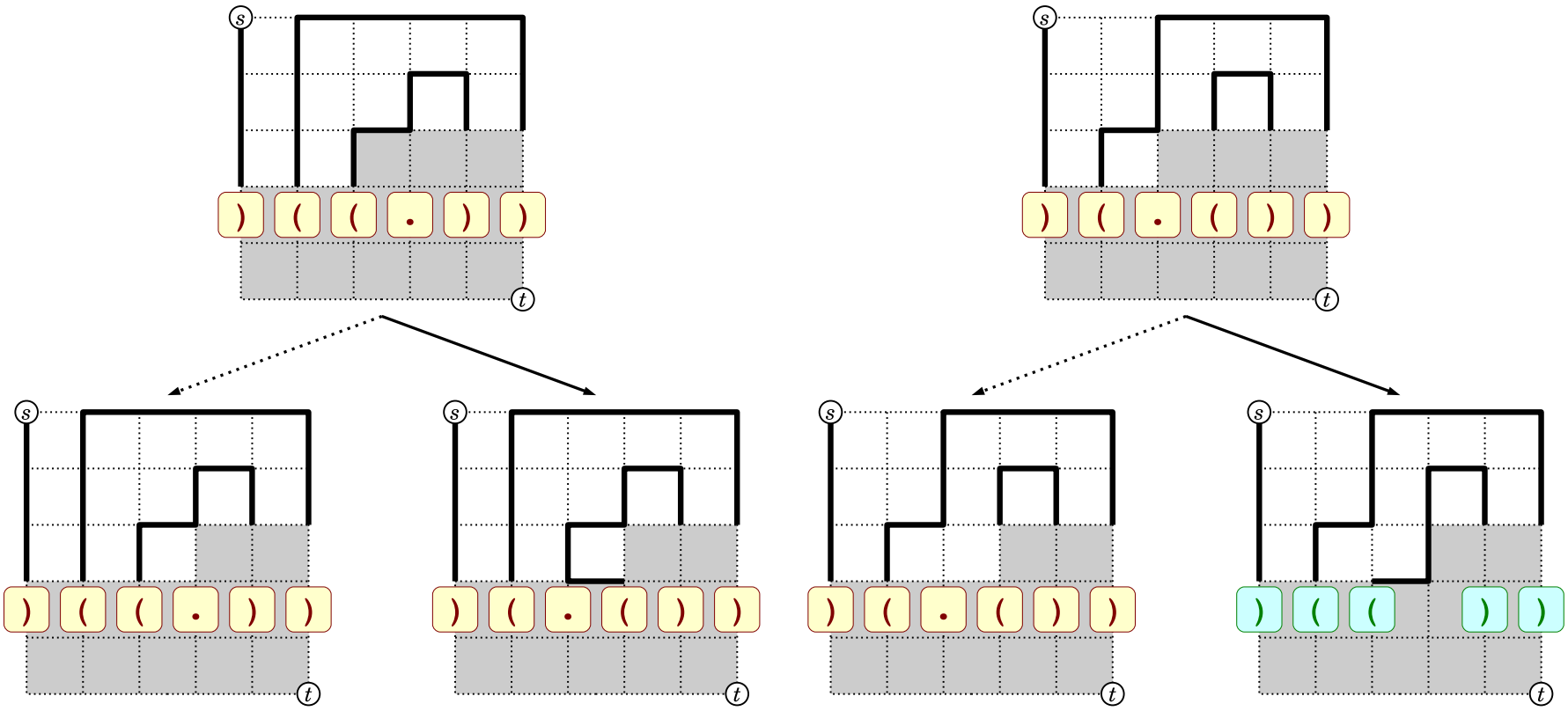
縦辺は「端点なら伸ばす」で一意に決まる。
横辺は 使う／使わない の 2 択。

状態数 $O(3^n)$



各ステップの状態は、$\left\{ \underset{\text{端点でない}}{.},\ \underset{\text{左の端点}}{(},\ \underset{\text{右の端点}}{)} \right\}$ による長さ $n+1$ の文字列で表現できる。

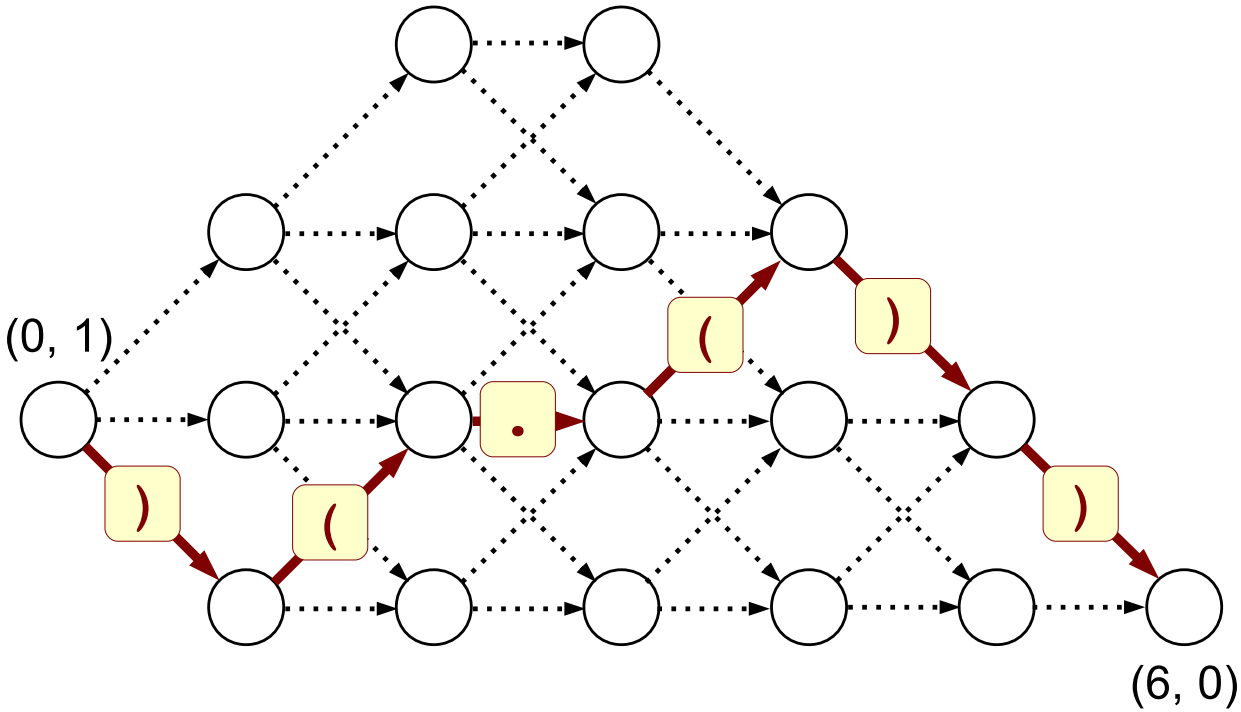
状態数 $O(3^n)$



ただし、新しい縦辺と横辺を共に使うケース（右端）は
直後の横辺を使えない 一回休み状態 として別に扱う。
→ 長さ n の文字列で表現

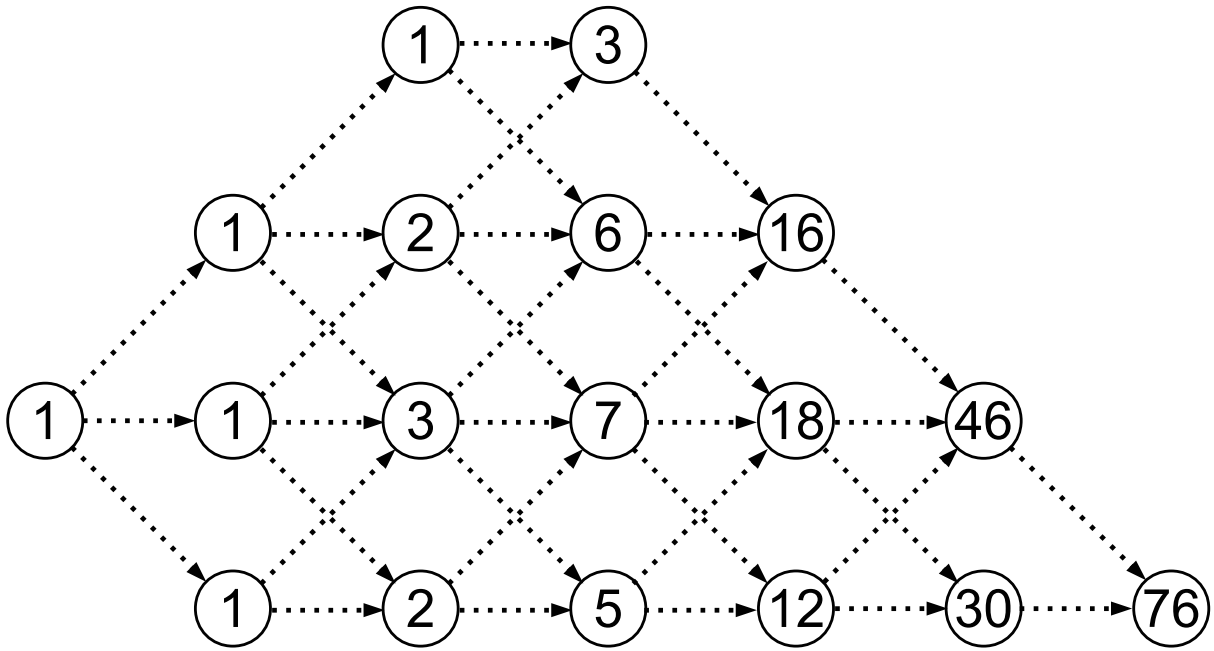
Motzkin パス*みたいな状態だけが出現

グリッドグラフの辺は交差しないので、( ) は必ず入れ子構造になる。
ただし、始点と接続している) が一つ余るので (0, 1) からスタート。



*http://en.wikipedia.org/wiki/Motzkin_number

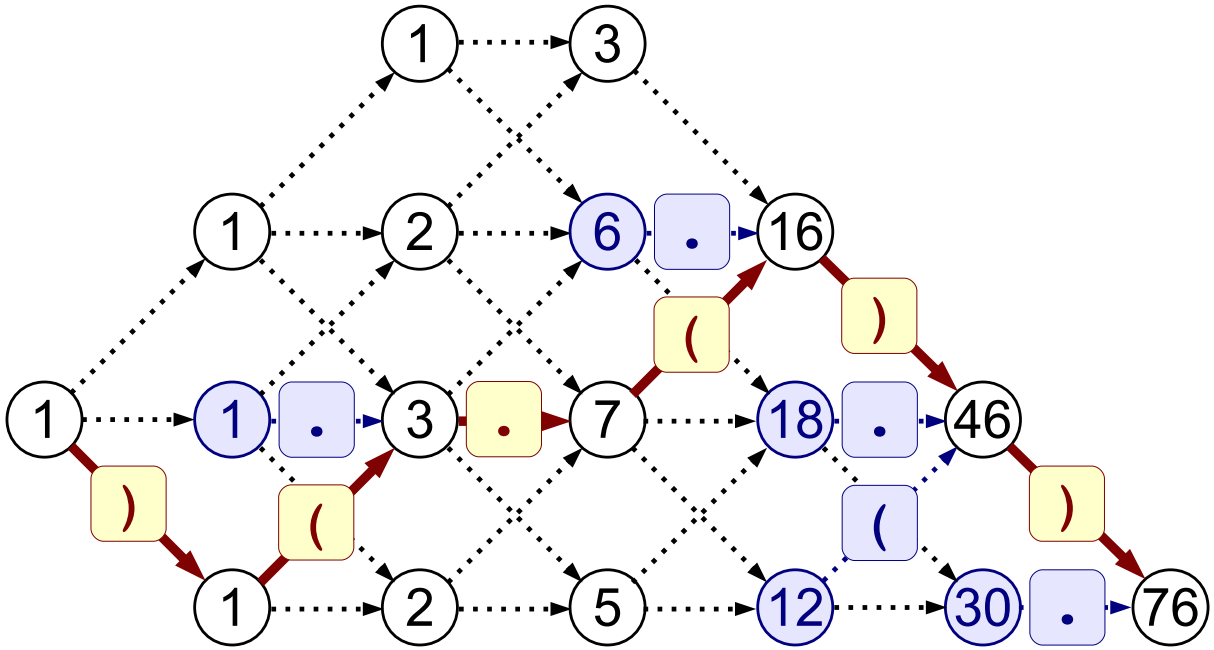
可能な状態を数え上げると



$$N_6 = 76$$

状態のインデックスを計算できる

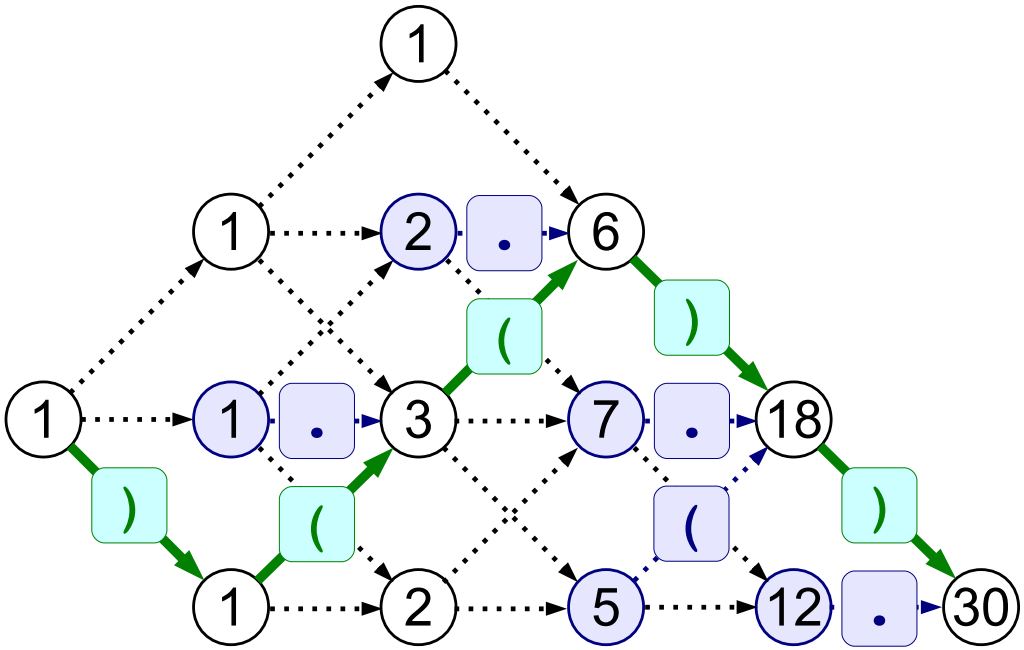
. < (<) を枝ラベルの順序とし、
より小さいラベルを持つ枝から合流するパスの数を加えていく。



$$index_6(\,)\,(\,.\,(\,)\,) = 0 + 1 + 0 + 6 + (18 + 12) + 30 = 67$$

反対に、$state_6(67)$ =) (. ( )) も計算可能。

「一回休み状態」も同様に



$$N_5 = 30$$

$$\mathrm{index}_5(\,)\,(\,(\,)\,) \;=\; 0 + 1 + 2 + (7 + 5) + 12 = 27$$

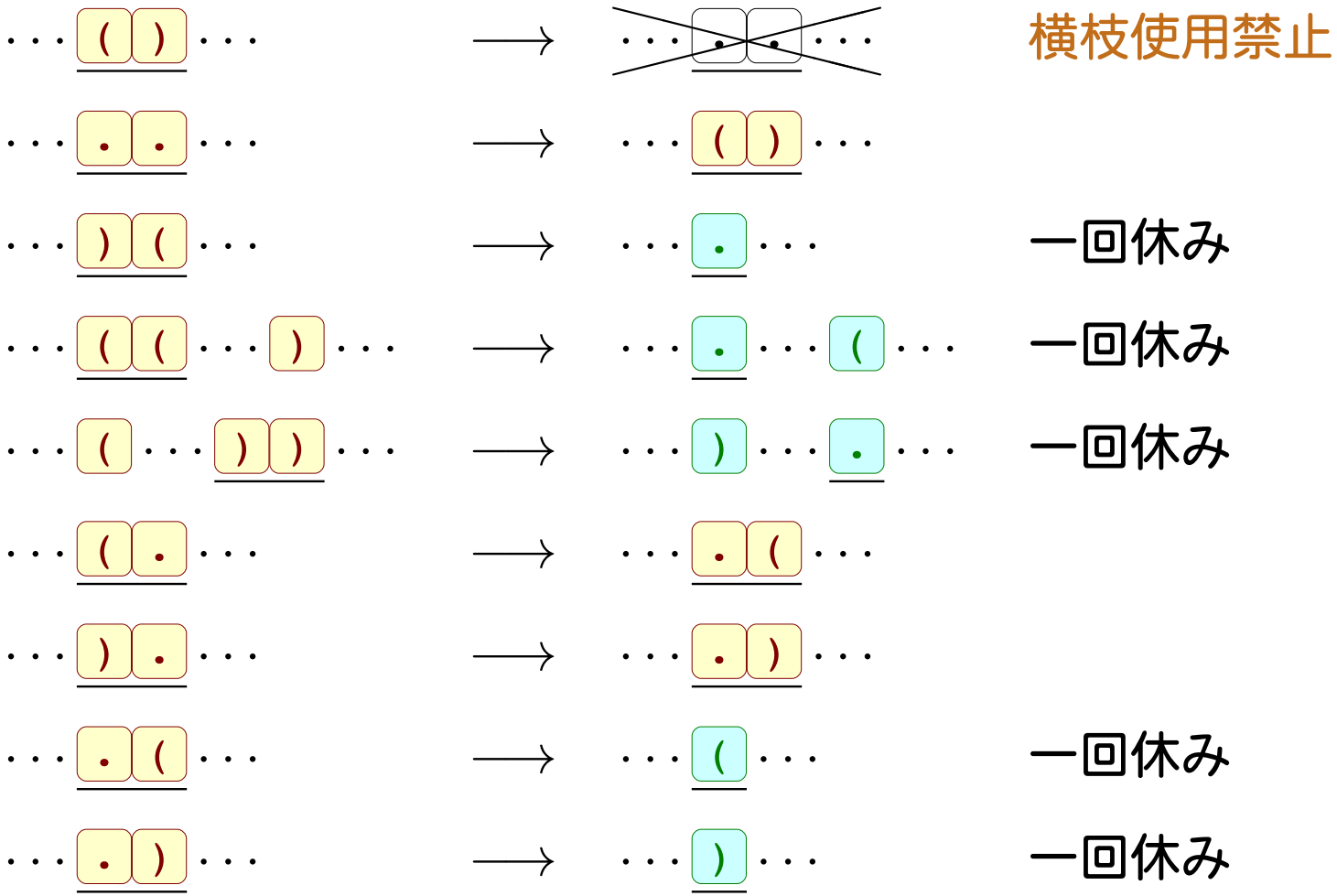
データ構造

各状態に到達する場合の数を保持した bignum の配列

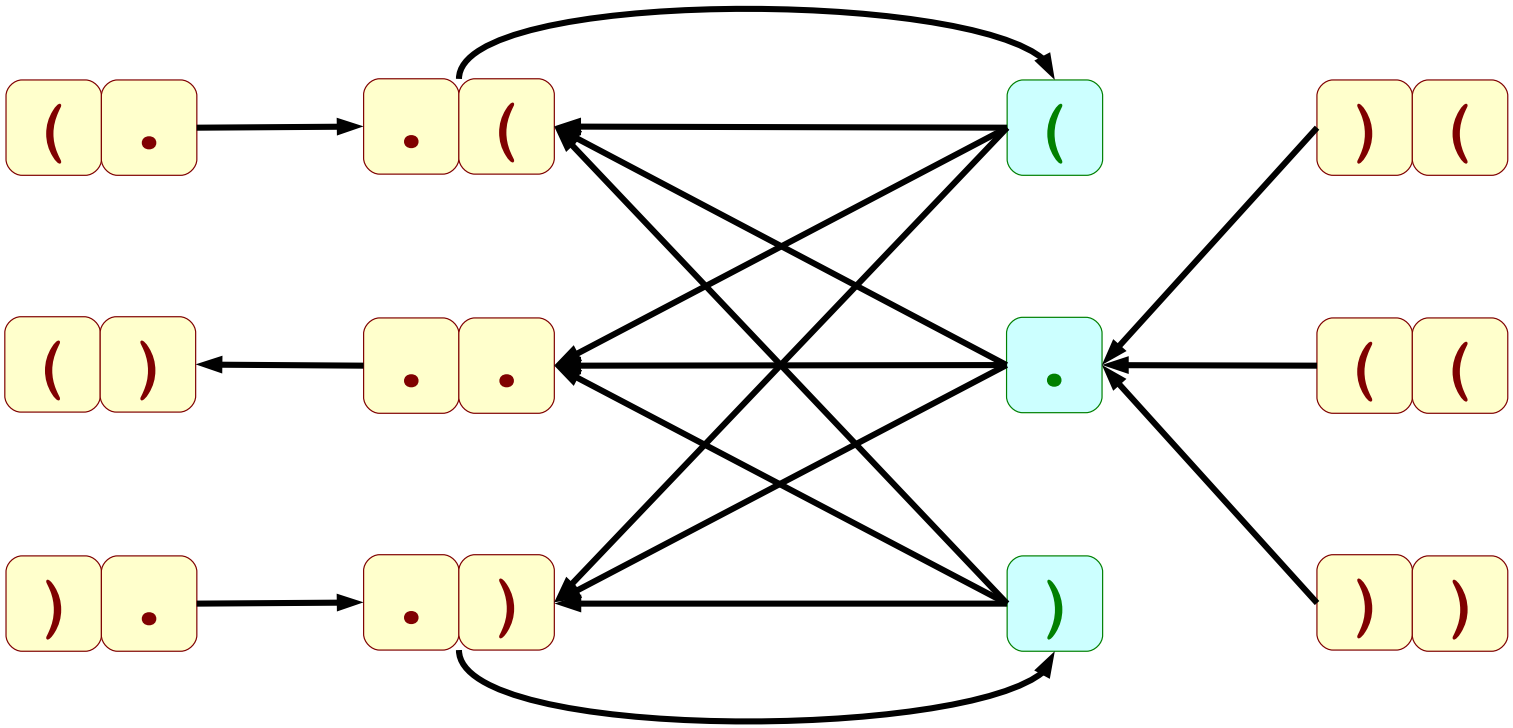
のみ

いわゆる 完全ハッシュ法。

横枝による状態変化



1ステップの状態遷移



アルゴリズムの概要

1: $A[0], \ldots, A[N_{n+1} - 1]$ および $B[0], \ldots, B[N_n - 1]$ を 0 で初期化;
2: $A[index(\boxed{)}\boxed{.}\boxed{.}\cdots\boxed{.})] \leftarrow 1$;
3: **for** $row = 1$ **to** $n + 1$ **do**
4: **for** $col = 1$ **to** n **do**
5: 状態遷移のルールに従って A と B を更新;
6: **end for**
7: B の各要素を対応する A の要素に足し込み、B を初期化;
8: **end for**
9: **return** $A[index(\boxed{.}\boxed{.}\cdots\boxed{.}\boxed{)})]$;

発表の流れ

1. はじめに
   - グリッドグラフのパスを数え上げる問題
   - フロンティア法によるパスの数え上げ
2. グリッドグラフであることを利用したパス数え上げ
   - 状態数 $O(3^n)$
   - 完全ハッシュ関数
   - アルゴリズム
3. 実験結果
   - 任意のグラフを対象とした実装との比較

実験結果

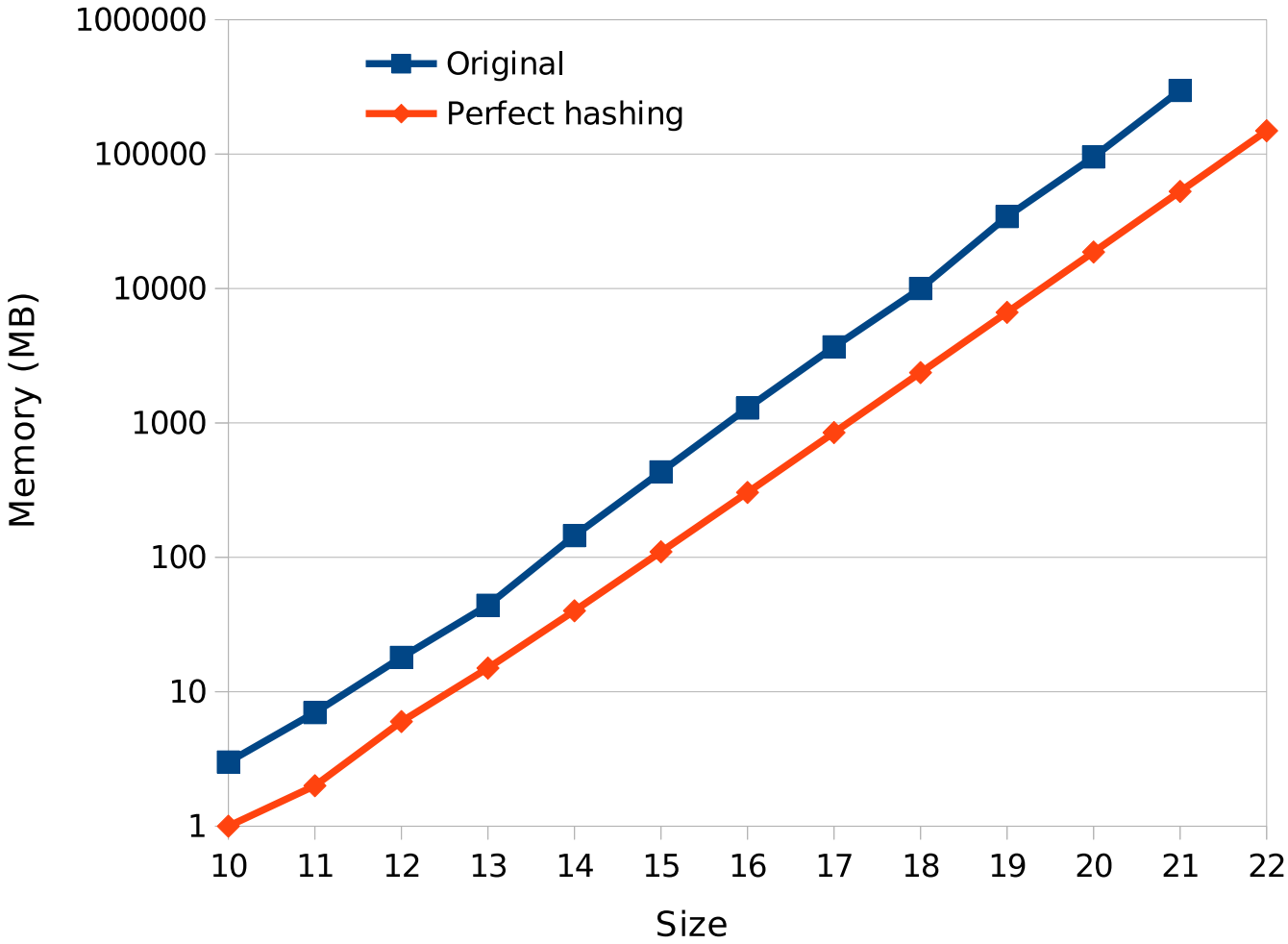
メモリ使用量を 1/5 以下、実行時間を 1/3 以下に削減。

Size	Original		Perfect hashing	
	Mem (MB)	Time (sec)	Mem (MB)	Time (sec)
18×18	9993	6003	2367	1755
19×19	34298	17962	6641	5687
20×20	95329	61570	18688	18121
21×21	297260	208002	52723	56264
22×22			149108	178440
23×23			422640	実行中

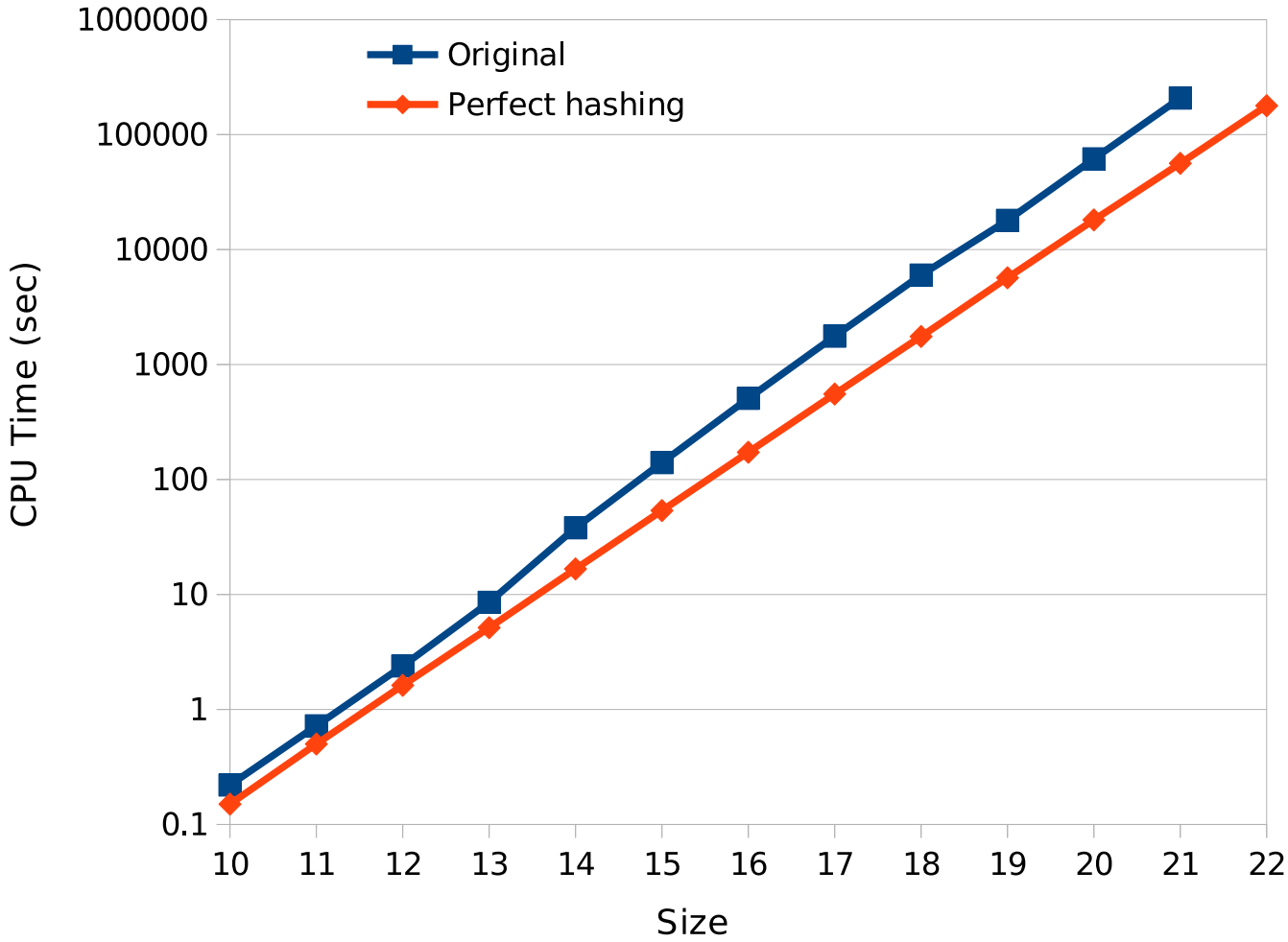
22×22 のメモリ使用量

$$
\begin{aligned}
&\simeq (N_{23} + N_{22}) \times \text{sizeof(bignum)} \\
&= (2,062,967,382 + 728,997,192) \times 56 \text{ bytes} \\
&= 149\text{G bytes}
\end{aligned}
$$

メモリ使用量


Original
Perfect hashing
Memory (MB)
1000000
100000
10000
1000
100
10
1
10
11
12
13
14
15
16
17
18
19
20
21
22
Size

CPU 時間


1000000
100000
10000
1000
100
10
1
0.1
CPU Time (sec)
Original
Perfect hashing
10
11
12
13
14
15
16
17
18
19
20
21
22
Size

パス数え上げの実装を最小完全ハッシュ関数で効率化

- 任意のグラフを対象とした実装に対して
  - 5 倍のメモリ使用効率改善。
  - 3 倍の計算速度改善。
- 23×23 まで計算できそう。

もっと ···

- bignum による計算を int による複数回の計算に代えると、メモリ使用量の削減は可能。（中国の剰余定理）
- 並列化のアイディアを募集中。