Paper Introduction

Of Hammers and Nails: An Empirical Comparison of Three Paradigms for Processing Large Graphs

発表者: Kazuhiro Inaba

この論文について

- WSDM ’12
  - ACM Conference on Web Search & Data Mining

Of Hammers and Nails: An Empirical Comparison of Three Paradigms for Processing Large Graphs

Marc Najork
Microsoft Research
Mountain View, CA, USA
najork@microsoft.com

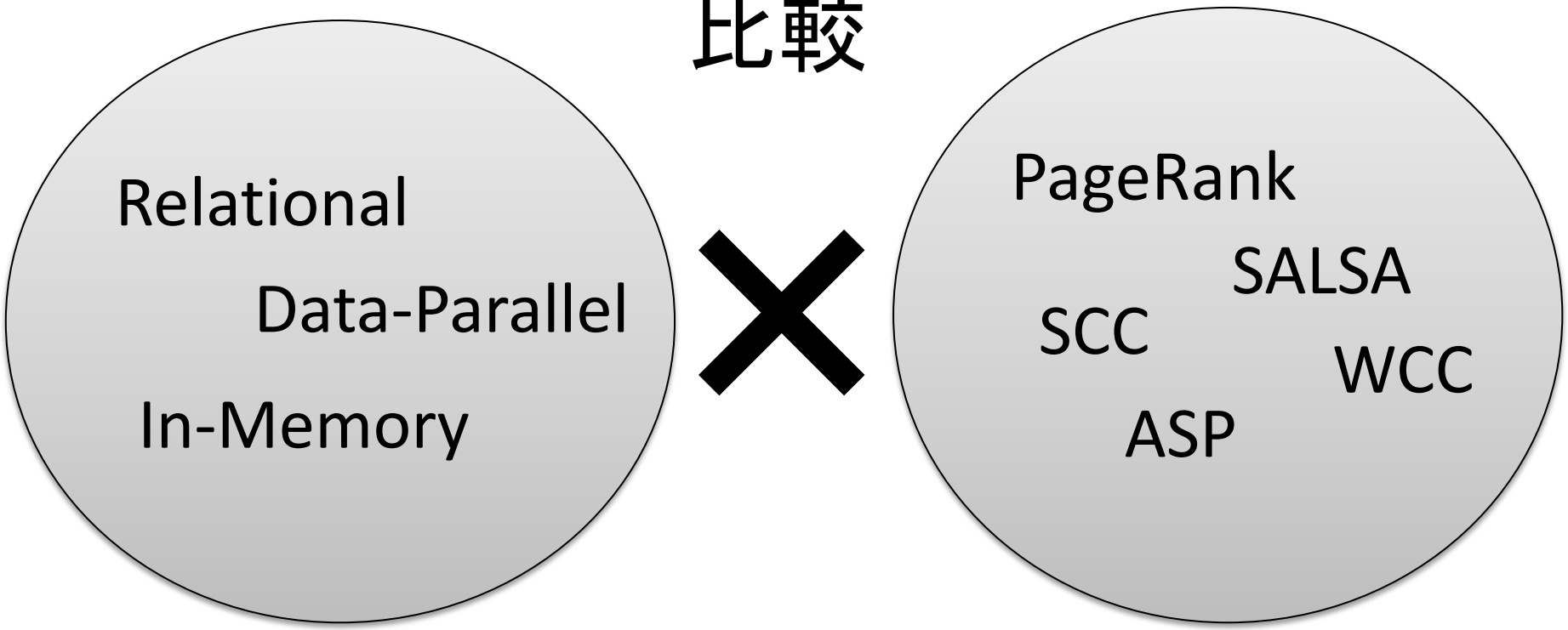
Dennis Fetterly
Microsoft Research
Mountain View, CA, USA
fetterly@microsoft.com

Alan Halverson
Microsoft Corporation
Madison, WI, USA
alanhal@microsoft.com

Krishnaram Kenthapadi
Microsoft Research
Mountain View, CA, USA
krisken@microsoft.com

Sreenivas Gollapudi
Microsoft Research
Mountain View, CA, USA
sreenig@microsoft.com

巨大Webグラフを扱う計算パラダイムの比較



3つの計算パラダイム　を、　5つのアルゴリズムで比較

データセット

- ClueWeb 09 Dataset
  - http://lemurproject.org/clueweb09/
  - Category A:
    - 4.77G nodes, 7.94G edges
    - 71GB uncompressed, 29GB compressed
  - Category B:
    - 0.428G nodes, 0.454G edges

Algorithm 1: PageRank

1: **for** each $u \in V$ **do**
2: $s[u] := 1/n,\ s'[u] := 0$
3: **for** $k = 1$ to z **do**
4: **for** each $u \in V$ **do**
5: $links[u] := \{v | (u, v) \in E\}$
6: **for** each $v \in links[u]$ **do**
7: $s'[v] := s'[v] + \frac{s[u]}{|links[u]|}$
8: **for** each $u \in V$ **do**
9: $s[u] := d/n + (1-d)s'[u]$
10: $s'[u] := 0$

全ノードのスコアを
1/|V| に初期化

z 回繰り返し

スコアを in-edge の
スコアの和に更新

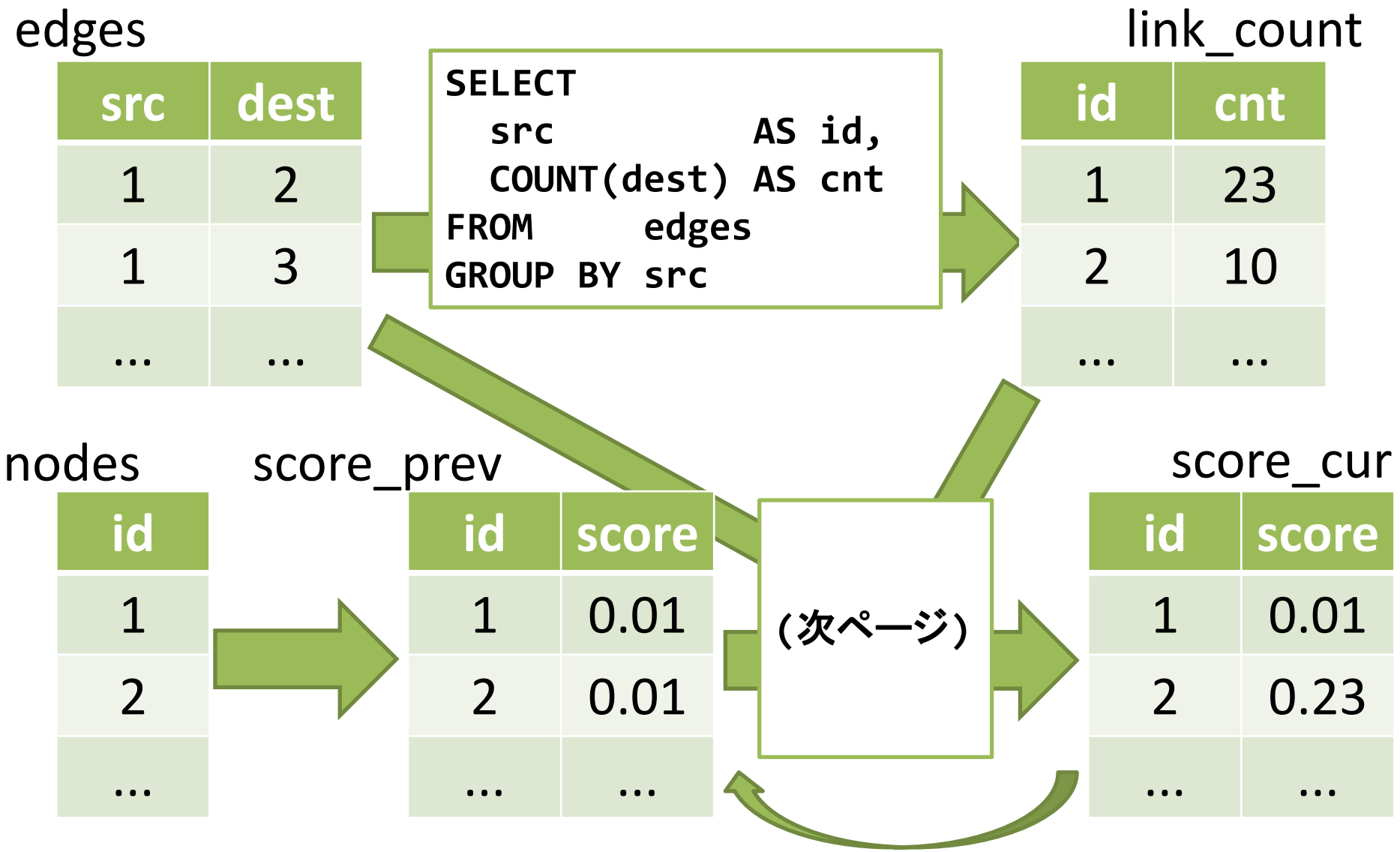
確率 d で
ランダムジャンプ

Paradigm 1: Relational
(実験対象: SQL Server 2008 Parallel DataWarehouse)

- Relation (= Table) (= Finite set of Tuples) に対する集合演算で計算処理を表現する
  - 長い研究・実用の歴史 [Codd 1970]
  - 関係代数などに基づいた最適化・並列化
  - データ表現の柔軟性が低い (なんでもRelationにしないといけない)

src	dest	weight
1	2	1.5
1	3	12.3
2	3	0

PageRank × SQLServer



```
While @ITER < 100
BEGIN
      CREATE TABLE score_cur WITH (DISTRIBUTION=HASH(id)) AS
      SELECT s.curid AS id, SUM(s.newscore) AS score FROM
      (SELECT * FROM (
      SELECT e.dest AS curid,
            SUM((1.0-@d)*(sp.score/lc.cnt)) AS newscore
      FROM score_prev sp, edges e, link_counts lc
      WHERE sp.id = lc.id AND sp.id = e.src
      GROUP BY e.dest
      WITH (DISTRIBUTED_AGG)) A
      UNION ALL
      SELECT sp.id AS curid, (@d / @CNODES) AS newscore
      FROM score_prev sp
      WHERE sp.id <> @CNODES) s GROUP BY s.curid;

      INSERT INTO score_cur
      SELECT CAST(@CNODES AS bigint) AS id,
      SUM(CASE WHEN lc.cnt IS NULL
          THEN sp.score ELSE 0 END) * (1.0 - @d)
      FROM score_prev sp
           LEFT OUTER JOIN link_counts lc on sp.id = lc.id;

      DROP TABLE score_prev;
      RENAME OBJECT score_cur TO score_prev;
      SET @ITER += 1;
END
```

PageRank × SQLServer

e.src	e.dest
1	2
1	3
2	3

lc.id	lc.cnt
1	23
2	10
…	…

sp.id	sp.scr
1	0.01
2	0.01
…	…

```
SELECT
   ...
FROM e, lc, sp
```

e.src	e.dest	lc.id	lc.cnt	sp.id	sp.scr
1	2	1	23	1	0.01
1	3	1	23	1	0.01
2	3	2	10	2	0.01

```
WHERE
   sp.id=lc.id &
   sp.id=e.src
```

e.src	e.dest	lc.id	lc.cnt	sp.id	sp.scr
(1)	2	(1)	(23)	(1)	(0.01)
(1, 2)	3	(1,2)	(23,10)	(1,2)	(0.01,0.01)

```
GROUP BY
   e.dest
```

curid	newscr
2	…
3	…

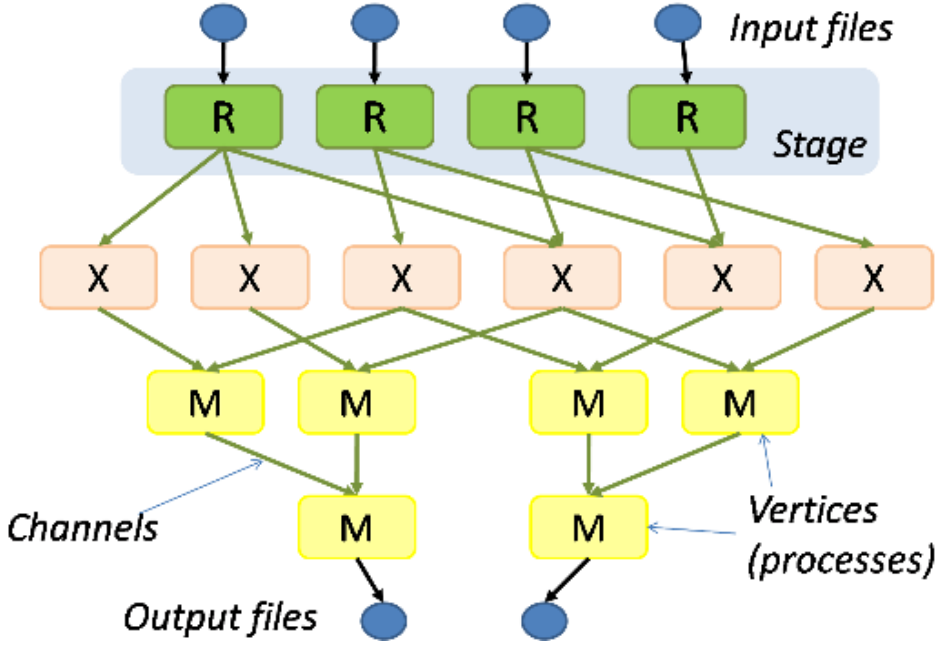
```
SELECT
 e.dest AS curid
 SUM((1-d)/(sp.scr/lc.cnt)) AS newscr
```

Paradigm 2: Data-Parallel
(実験対象 : DryadLINQ)

「データの列に一斉に同じ処理をする」
「その結果を特定のキーで集計し別の列を作る」
の多段重ねで計算を記述

- Dryad : Microsoft の Data-Parallel インフラ
- LINQ : MS の言語拡張インフラ

- Relational Model より扱えるデータやプログラムは柔軟
- 計算の一斉同時適用意外のことは苦手



PageRank × DryadLINQ

pages
1 => {2, 3}
2 => {3, 4}
3 => {}
…

```
from p in pages
join s in scores on p.Key equals s.Key
from v in p.Value
select new LD(v, s.Value / p.Value.Length)
```

scores1
2 => 0.005
3 => 0.005
3 => 0.005
4 => 0.005

scores
2 => …
3 => …
4 => …
…

scores
1 => 0.01
2 => 0.01
3 => 0.01
…

```
from s in scores1
group s.Value by s.Key into g
select new LD(g.Key, d / n + (1.0 - d) * g.Sum())
```

Paradigm 3: In-Memory Store
(SHS : Scalable Hyperlink Store)

- この論文の第一著者の研究 (HT'09)
- Webのリンク構造を分散・圧縮して保持するデータストア
  - ただリンク構造を記憶・取り出しできるだけなので、それ自体に並列計算機構はない
    （以下の実験でも1マシンで直列実行）
  - 計算部分は普通の小規模プログラムと変わりないので記述は楽

PageRank × SHS

```
for (int k = 0; k < 100; k++) {
  scores.SetAll(x => d / n);
  var ib = new InBuf(Name(k));
  foreach (long u in shs.Uids) {
    uidBatch.Add(u);
    if (uidBatch.Full || shs.IsLastUid(u)) {
      var linkBatch = shs.BatchedGetLinks(true, uidBatch);
      var linkMap = new Shs.UidMap(linkBatch);
      var scoreArr = scores.GetMany(linkMap);
      foreach (var links in linkBatch) {
        double f = (1.0 - d) * ib.ReadDouble() / links.Length;
        foreach (long link in links) {
          scoreArr[linkMap[link]] += f;
        }
      }
      scores.SetMany(linkMap, scoreArr);
      uidBatch.Reset();
    }
  }
```

for each u in V ...

データストアに
都合のいい単位で
隣接辺集合を
まとめて取得

s'[v] += (1-d)*s[u] / |links|

PageRank: 速度 (単位:秒)

DataSize : Machine	SQL	Dryad	SHS
4G : 16台	156,982	68,791	836,445
0.4G : 16台	8,970	4,513	90,942
0.4G : 1台	122,305	83,472	63,711

スケールしている

スケールしている

Algorithm 2: SALSA

"Stochastic Approach to Link-Sensitivity Analysis"

– 条件Rにマッチするwebページ群での"authority度"

1: $V_R := \bigcup_{r \in R} \{r\} \cup C_a(\{u|(u,r) \in E\}) \cup C_b(\{v|(r,v) \in E\})$
2: $E_R := \{(u,v) \in E | u \in V_R \wedge v \in V_R\}$
3: $V_R^A := \{u \in V_R | in(u) > 0\}$
4: **for** each $u \in V_R$ **do**
5: $s[u] := \frac{1}{|V_R^A|}$ if $u \in V_R^A$; 0 otherwise
6: **repeat**
7: **for** each $u \in V_R^A$ **do**
8: $s'[u] := \sum_{(v,u) \in E_R} \sum_{(u,w) \in E_R} \frac{s[w]}{out(v) \cdot in(w)}$
9: **for** each $u \in V_R^A$ **do**
10: $s[u] := s'[u]$
11: **until** s converges

R とその近傍

in-Edge があるノードが Authority 候補

収束まで繰り返し

2部グラフ上を
ランダムウォーク
in/out双方で正規化

Implementation

- SQL
  - PageRank の場合と似たような実装
- DryadLINQ
  - Stream join “s.Key equals p.Key” の組み合わせで V_R, E_R を求める
  - ローカルノードでメインの計算
- SHS
  - 擬似コードの通りに V_R, E_R を求める
  - ローカルノードでメインの計算

SALSA: 速度 (単位:秒)

DataSize : Machine	SQL	Dryad	SHS
4G : 16台	2,199	2,221	124
0.4G : 16台	2,034	439	163
0.4G : 1台	5,873	4,843	37

速い

Algorithm 3: 強連結成分分解

Algorithm 3 SCC(G)

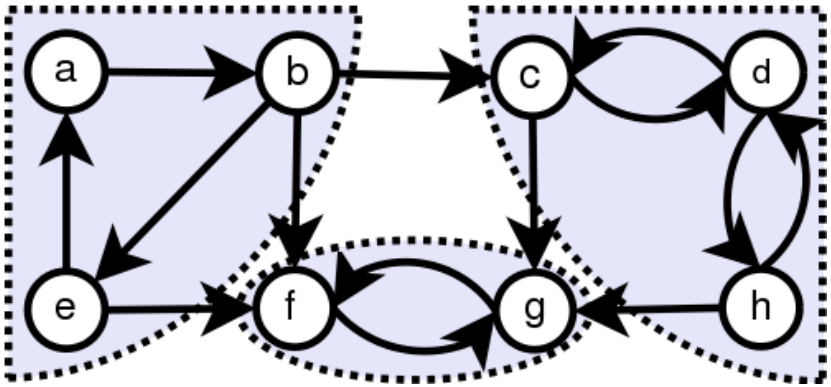
Input: A directed graph $G = (V, E)$.
Output: A list of SCCs where each SCC is represented as a set of nodes belonging to the SCC.

```
1: for each u ∈ V do
2:     fwdVisited[u] := false, fwdParent[u] := Nil
3: time := 0
4: for each u ∈ V do
5:     if fwdVisited[u] = false then
6:         DFS-VISIT(u, E, fwdVisited, fwdParent, f)
7: E^T := {(v,u)|(u,v) ∈ E}
8: for each u ∈ V do
9:     bwdVisited[u] := false, bwdParent[u] := Nil
10: for each u ∈ V in decreasing order of f[u] do
11:     if bwdVisited[u] = false then
12:         DFS-VISIT(u, E^T, bwdVisited, bwdParent, b)
13: Output the vertices of each tree in the backward depth-first forest as a
    separate strongly connected component.
```

Algorithm 3 DFS-VISIT($u, E, visited, parent, f$)

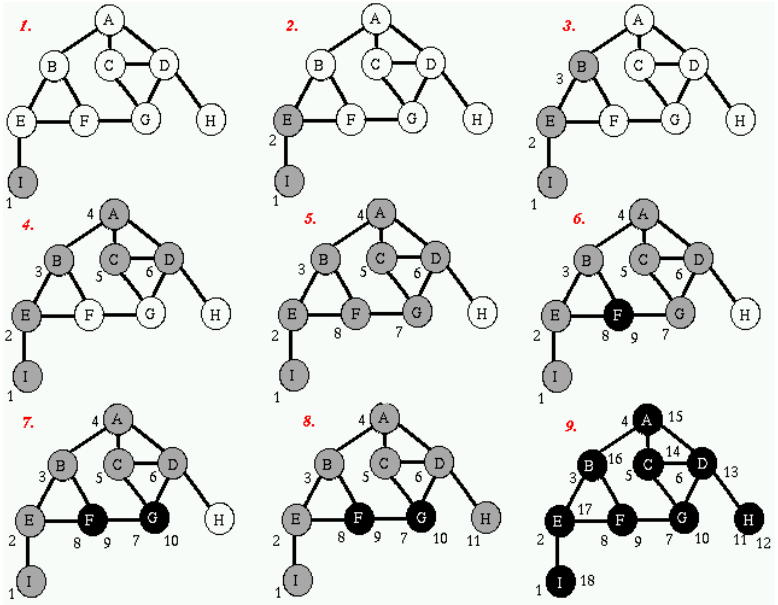
```
time := time + 1
links[u] := {v|(u,v) ∈ E}
for each v ∈ links[u] do
    if visited[v] = false then
        parent[v] := u
        DFS-VISIT(v, E, visited, parent, f)
visited[u] := true, f[u] := time
```

ノードを、相互にreachableなノード集合に分解



Algorithm 3: 強連結成分分解

深さ優先探索



を2回実行

→ これは本質的に sequential なので遅すぎる

workaround: 次数1のノードは先に除く

SCC: 速度 (単位:秒)

DataSize : Machine	SQL	Dryad	SHS	SHS
4G : 16台	7,306	6,294	15,903	-
0.4G : 16台	475	446	1,073	214,858
0.4G : 1台	1,147	3,243	816	94,836

Prune degree-1 node
+ local naive computation

Naive

Algorithm 4: 連結成分(無向)

```
 1: for each u ∈ V do
 2:     rep[u] := u
 3: repeat
 4:     progress := false
 5:     for each u ∈ V do
 6:         links[u] := {v|(u, v) ∈ E}
 7:         minid := rep[u]
 8:         for each v ∈ links[u] do
 9:             if rep[v] ≠ minid then
10:                 minid := min(minid, rep[v])
11:                 progress := true
12:         rep[u] := minid
13:         for each v ∈ links[u] do
14:             rep[v] := minid
15: until ¬progress
```

各ノードを
単一のグループに分類
（代表元＝自分）

収束まで繰り返し

各ノードについて

隣接ノードの代表元の
最小のIDで

自分(の近傍)の
代表元を置き換え

なぜもっと速いアルゴリズムを使わないのか？

- Disjoint-Set Forest (Galler & Fischer 64)
  - a.k.a. Union-Find
  - 全ノードで代表元を覚えるのではなく、代表元に近づく“親”を覚えて木構造を作る。木の高さがlog N になるように工夫する

→ これも sequential な計算方法なので、SQLやData-Parallel では書きにくい

WCC: 速度 (単位:秒)

$O(mn/p) > O(m\ \alpha(n))$

DataSize : Machine	SQL	Dryad	SHS
4G : 16台	214,479	160,168	26,219
0.4G : 16台	4,207	3,844	1,976
0.4G : 1台	63,972	74,359	1,801

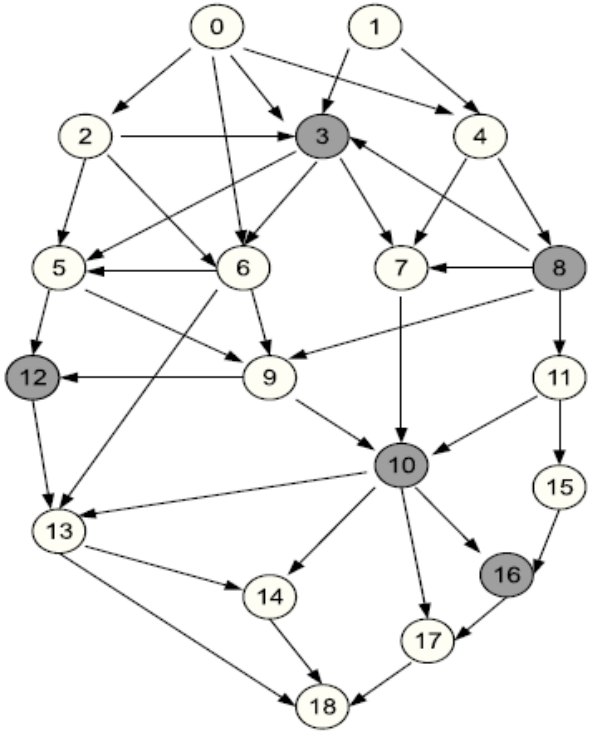
Naive

Disjoint-set DataStructure

Algorithm 5: 近似最短距離クエリ
(Sketch-based: 第1,第5著者ら ’10)

1: $C := \{u \in V | degree(u) \geq 2\}$
2: $I := \{0, 1, \ldots, \lfloor log|C| \rfloor\}$
3: **for** each $i \in I$ **do**
4: $S_i :=$ Random sample of 2^i elements of C
5: **for** each $u \in V$ and each $i \in I$ **do**
6: $seed[u][i] := u$
7: $dist[u][i] := 0$ if $u \in S_i$; ∞ otherwise
8: **repeat**
9: $progress := false$
10: **for** each $(u, v) \in E$ and each $i \in I$ **do**
11: **if** $dist[u][i] < dist[v][i]$ **then**
12: $seed[v][i] := seed[u][i]$
13: $dist[v][i] := dist[u][i] + 1$
14: $progress := true$
15: **until** $\neg progress$

S_i = 2^i 個の “seed”



各ノードについて、
最近seedとその距離を計算
(Bellman-Ford)

ASP: 速度 (単位:秒)

さほど
速くない

DataSize : Machine	SQL	Dryad	SHS
4G : 16台	671,142	749,016	2,381,278
0.4G : 16台	30,379	17,089	246,944
0.4G : 1台	138,911	175,839	77,214

index作成の計算時間

まとめ & 感想

- まとめ
  - PageRank => inherently parallel. good.
  - SALSA => 局所的な解析は並列化の恩恵があまりない
  - SCC, ASP vs WCC => sequential vs arbitrary-order access
- 感想
  - 当然の結果が確かめられていた
  - もっと複雑ネットワークの性質に特化した計算モデル？