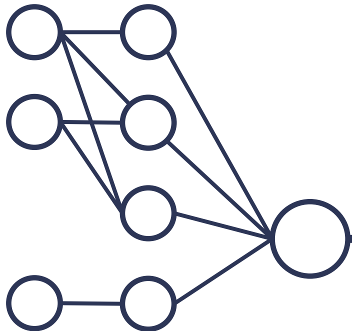
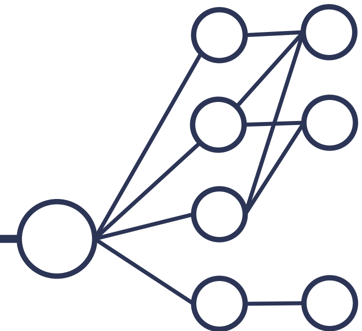
ICDE 2014 勉強会

ℝ6 Graphs I: Fundamental Algorithms

R6-2: Efficient Top-k Closeness Centrality Search
R6-3: Contract & Expand: I/O Efficient SCCs computing



担当：駒水（筑波大）



R6-2: Efficient Top-k Closeness Centrality Search

Paul W. Olse Jr., Alan G. Labouseur, Jeong-Hyon Hwang (Univ. of Albany)

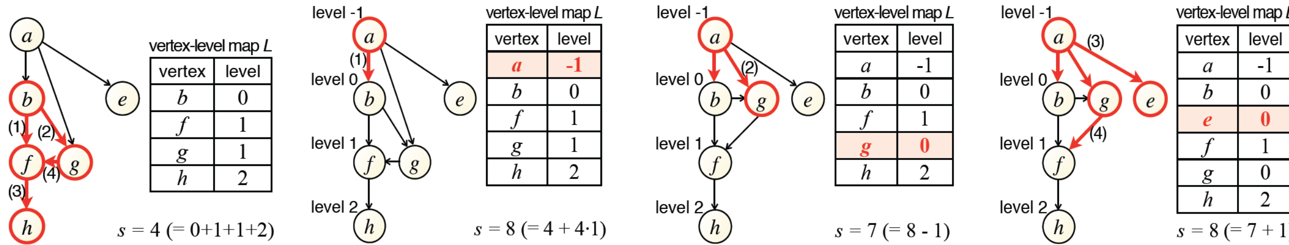
- Closeness Centrality: 他のノードからの距離に基づく中心性
  - 10-year best paper の計算に使われる
- 空間・時間の両面において効率的に中心性を計算したい
  - 愚直的解法：距離行列を構築（空間: $\Omega(|V|^2)$）
  - 現実的解法：単一始点最短路計算（Dijkstra + Fibonacci heap）を利用
    - 空間：$O(|V|+|E|)$，時間：$O(|V||E| + |V|^2\log|V|)$
  - 既存アルゴリズム[11,12,13]は無向・重みなしグラフが対象，正確でない
- 基本アイデア：ノード間で途中計算結果の共有化と枝刈り
  - どのノードの途中結果を利用するかが計算時間に影響
    - 計算時間の推定＋スケジューリング
  - Top-*k* に入り得ないノードのスキップ
    - ノードの中心性の上限を計算

R6-2: Efficient Top-k Closeness Centrality Search

Paul W. Olse Jr., Alan G. Labouseur, Jeong-Hyon Hwang (Univ. of Albany)

※図は論文から引用

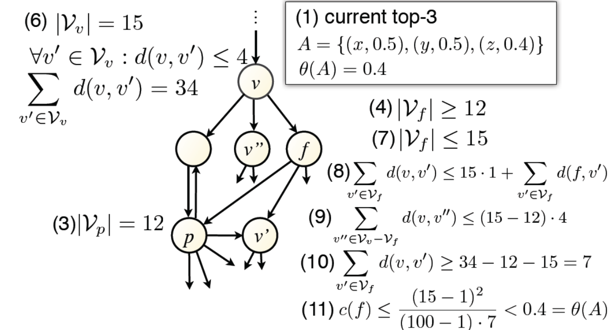
- 途中計算結果の共有化
  - 中心性計算に必要：到達可能なノード数 (|L|)と最短距離の総和 (s)



中心性の計算

$$\frac{(|\mathcal{V}_a|-1)^2}{(|V|-1)\sum_{v'\in\mathcal{V}_a} d(a,v')} = \frac{(|L|-1)^2}{(|V|-1)s} = \frac{(6-1)^2}{(6-1)8} = \frac{5}{8}$$

- 枝刈り
  - 中心性の上限：到達可能なノード数の上限 + 最短距離の総和の下限
  - ノード (f) の親ノード (v) の値を利用



- スケジューリング
  - エッジの重みを推定処理時間としたグラフの最小全域木を利用
  - 処理時間の推定：木の探索（$O(|E_v| + |V_v| \log |V_v|) \rightarrow O(|V_v| \log |V_v|)$）

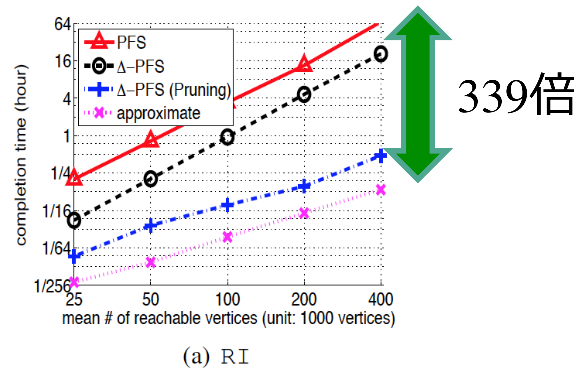
R6-2: Efficient Top-k Closeness Centrality Search

Paul W. Olse Jr., Alan G. Labouseur, Jeong-Hyon Hwang (Univ. of Albany)

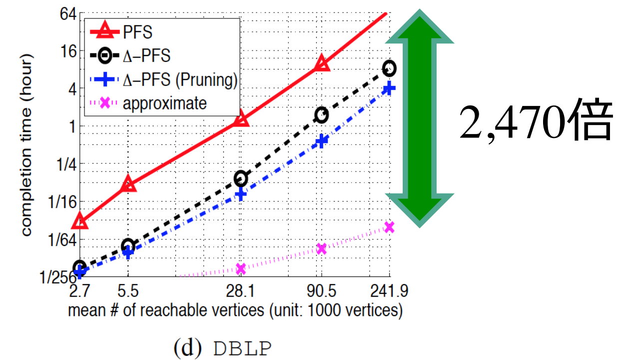
※図は論文から引用

- 評価実験
  - 比較対象：PFS (Dijkstra + Fibonacci heap), Δ-PFS, Δ-PFS (Pruning), approximation
  - 評価項目：速度

速度

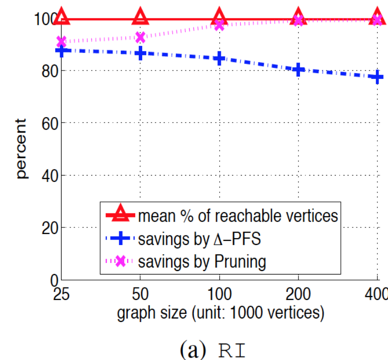


(a) RI

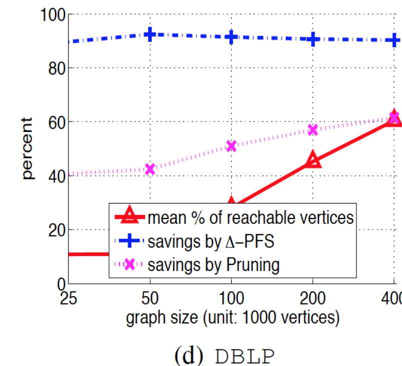


(d) DBLP

到達可能な平均ノード数の割合



(a) RI



(d) DBLP

R6-3: Contract & Expand: I/O Efficient SCCs computing

Zhiwei Zhang[1], Lu Qin[2], Jeffrey Xu Yu[1] ([1]CUHK, [2]Univ. of Technology, Sydney)

※図は論文から引用

- グラフの強連結成分 (SCC) を見つける問題
  - 応用例：トポロジカルソート，到達可能判定，パターンマッチ
- ノード全体がメモリに乗らない D:
  - Facebook: 1.11b users, Webgraph (UK2007): 105m pages
- 従来のアルゴリズム
  - in-memory: Kosaraju-Sharir algo. [3]➔グラフサイズに対して線形オーダ
  - semi-external : 先行研究 [26, SIGMOD'13] ➔ **"メモリにのる"** 全域木
- アプローチ: I/O効率なアルゴリズム
  - メモリに乗るようになるまでグラフ収縮 (contraction)
  - ディスクへはシーケンシャル
  - 2 フェイズ（右図）
    - contraction フェイズ
    - expansion フェイズ

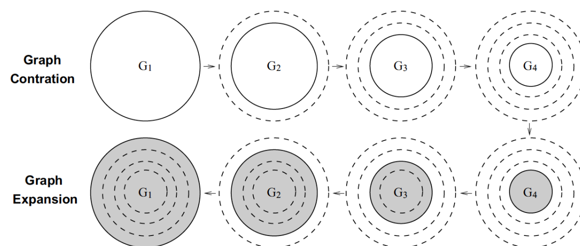


Fig. 2. A Solution Overview

Algorithm 2 Ext-SCC(G)

Input: a directed graph $G(V, E)$.
Output: all the SCCs in G.

1: $G_1(V_1, E_1) \leftarrow G(V, E)$; $i \leftarrow 1$;
2: **while** V_i can not fit in memory **do**
3: contract $G_i(V_i, E_i)$ to $G_{i+1}(V_{i+1}, E_{i+1})$ with $V_{i+1} \subset V_i$;
4: $i \leftarrow i + 1$;
5: compute all SCCs in $G_i(V_i, E_i)$ using Semi-SCC;
6: **while** $i > 1$ **do**
7: $i \leftarrow i - 1$;
8: compute SCCs for all nodes in $V_i - V_{i+1}$;
9: combine SCCs in $V_i - V_{i+1}$ with SCCs in V_{i+1};
10: output all SCCs in V_1;

R6-3: Contract & Expand: I/O Efficient SCCs computing

Zhiwei Zhang[1], Lu Qin[2], Jeffrey Xu Yu[1] ([1]CUHK, [2]Univ. of Technology, Sydney)

- グラフ収縮 (contraction)：徐々にノードを削減
  - SCC を正しく見つけられるようにノードを削減
  - 各ステップ i での削減条件
    - Contractible：最低でも1ノード削減（$|V_{i-1}| - |V_i| > 0$）
    - SCC-preservable：同じ成分に属するノードペアを同じ成分内に保持（$SCC(u, G_{i-1}) = SCC(v, G_{i-1}) \Leftrightarrow SCC(u, G_i) = SCC(v, G_i)$）
    - Recoverable：G_{i-1} から削減したノード v の隣接ノード（$nbr(v, G_{i-1})$）は必ず削減後のグラフ G_i に含まれる

これらを満たす削減方法を提案
➔ 詳しくは論文で

- グラフ拡張 (expansion)
  - 削除の逆順（i をデクリメント）でノードを追加
  - G_i に対して SCC_i を計算 ➔ SCC_i を利用して SCC_{i-1}を計算
    - $v \in V_{i-1} - V_i$ が SCC に入るかの判定には $nbr(v, G_{i-1})$ を見る
- I/O コストの最小化
  - 計算に不要なノード・エッジの削除

R6-3: Contract & Expand: I/O Efficient SCCs computing

Zhiwei Zhang[1], Lu Qin[2], Jeffrey Xu Yu[1] ([1]CUHK, [2]Univ. of Technology, Sydney)

※図は論文から引用

- 実験
  - データ
    - Webグラフ UK2007
    - SCC のサイズをコントロールした人工データ
  - 評価方法
    - 実行時間
    - I/O 数

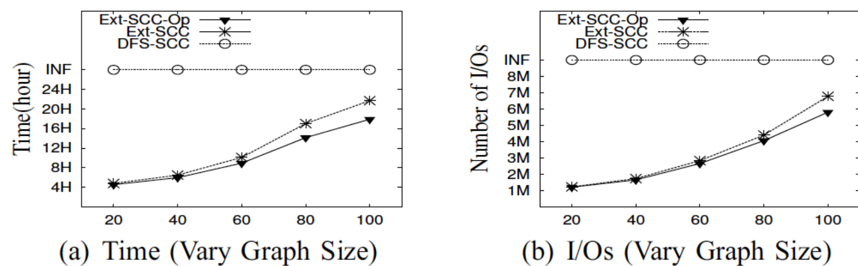


Fig. 6. WEBSPAM-UK2007: Varying Graph Size (Percent)

データサイズを変化

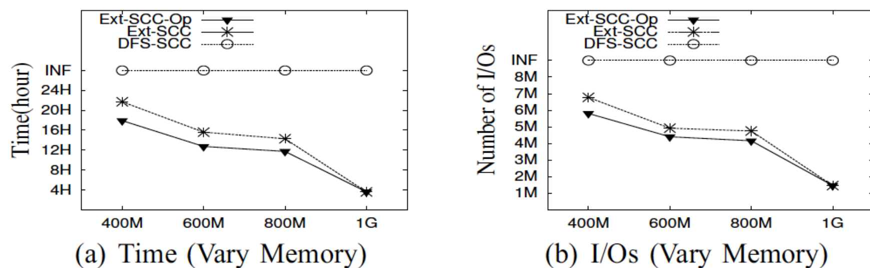


Fig. 7. WEBSPAM-UK2007: Varying Memory Size

メモリサイズを変化

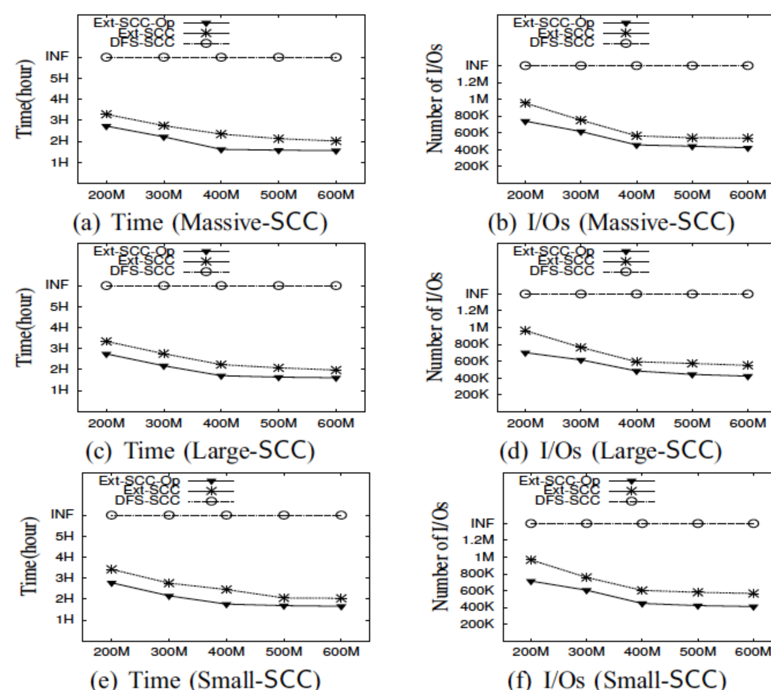


Fig. 8. Synthetic Data: Vary Memory Size

異なる SCC のサイズに対して