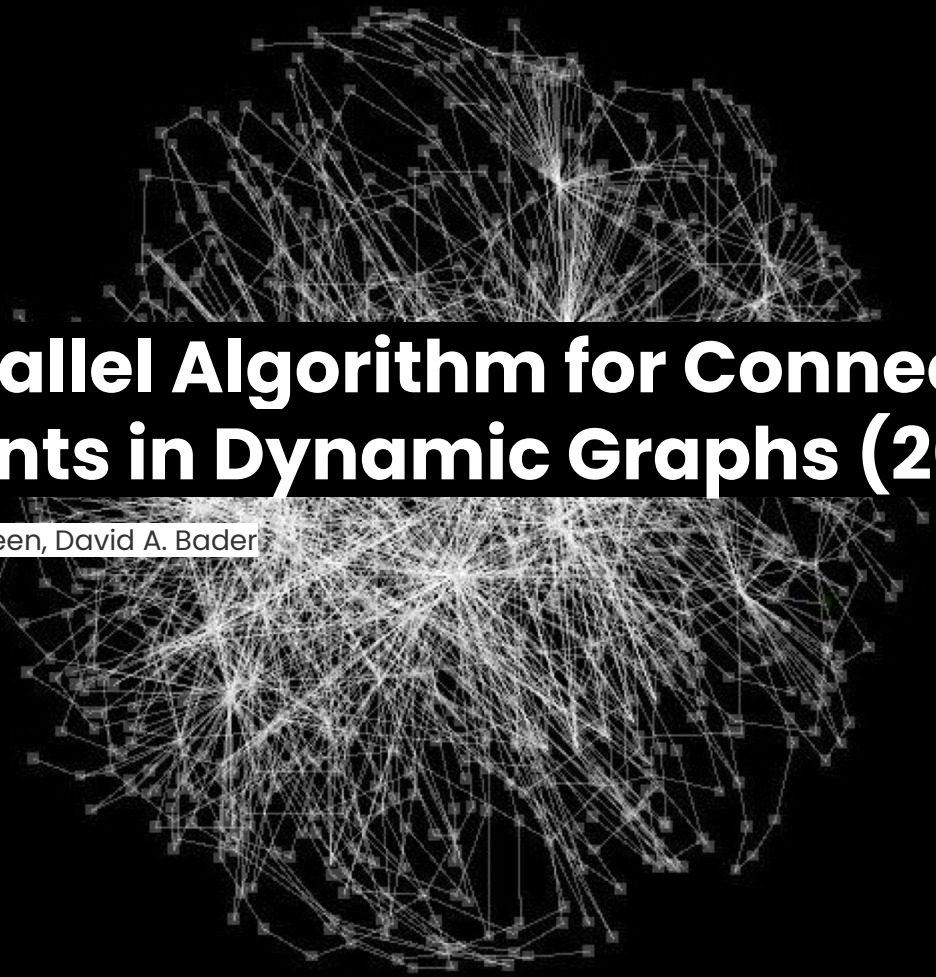


A New Parallel Algorithm for Connected Components in Dynamic Graphs (2013)

By Robert McColl, Oded Green, David A. Bader

Presentation by Qing Feng

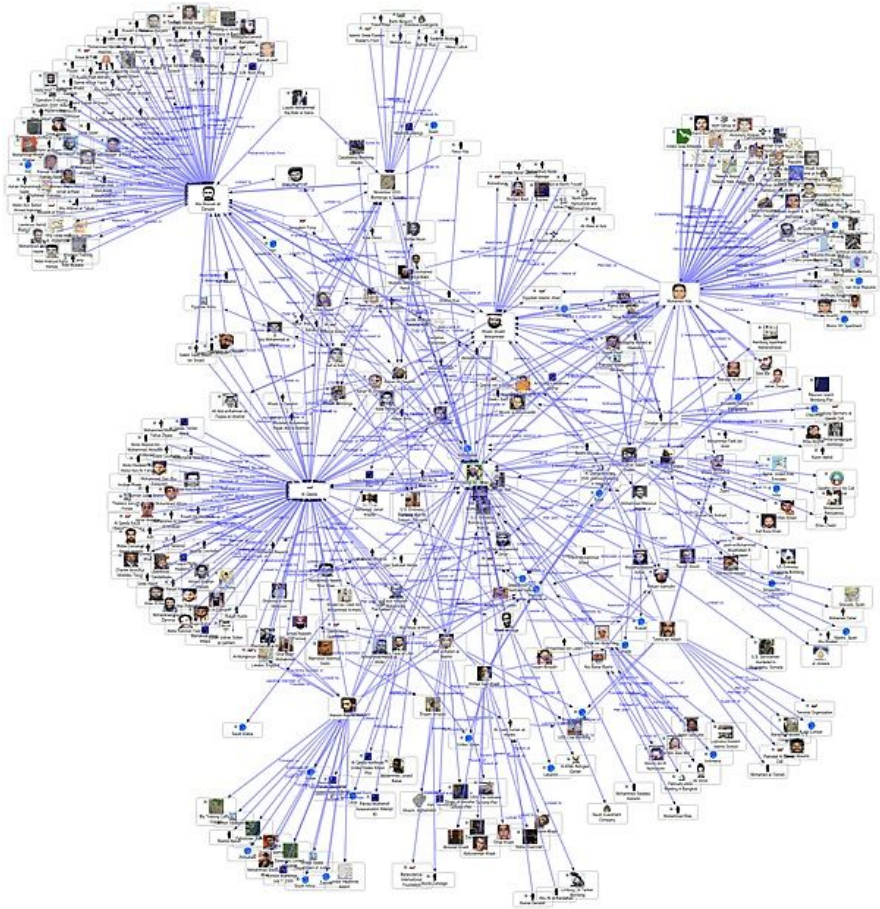
Mar 31 2022

Dynamic Graph Algorithm

A dynamic graph can be viewed as **a discrete sequence of static graphs**.

Relationships represented in the graph are changing quickly, making computation on static snapshots expensive. Hence a demand for:

Algorithms for dynamic graphs in which edges can be inserted or deleted.



facebook

Eg. Social Networks

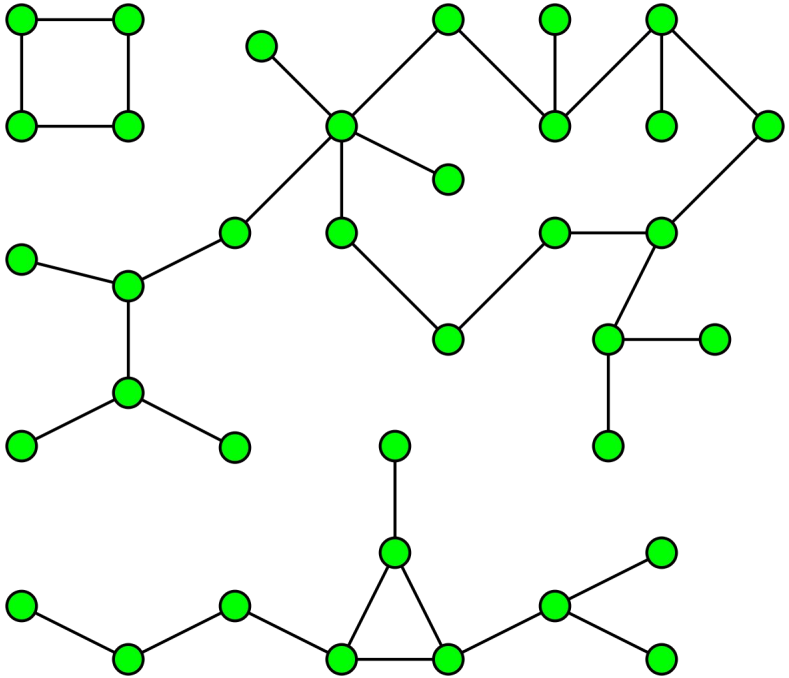
Friendship ...

Message ...

Connected Components

Given an **undirected graph** $G = (V, E)$:
A connected component (CC) $C \subseteq V$ ensures that for each $s, t \in C$ there is a path between s and t.

Each CC can be detected by an DFS or BFS.
Conducting a **full-DFS/BFS** on each static snapshot takes $O(V+E)$ time and space.



Connected Components

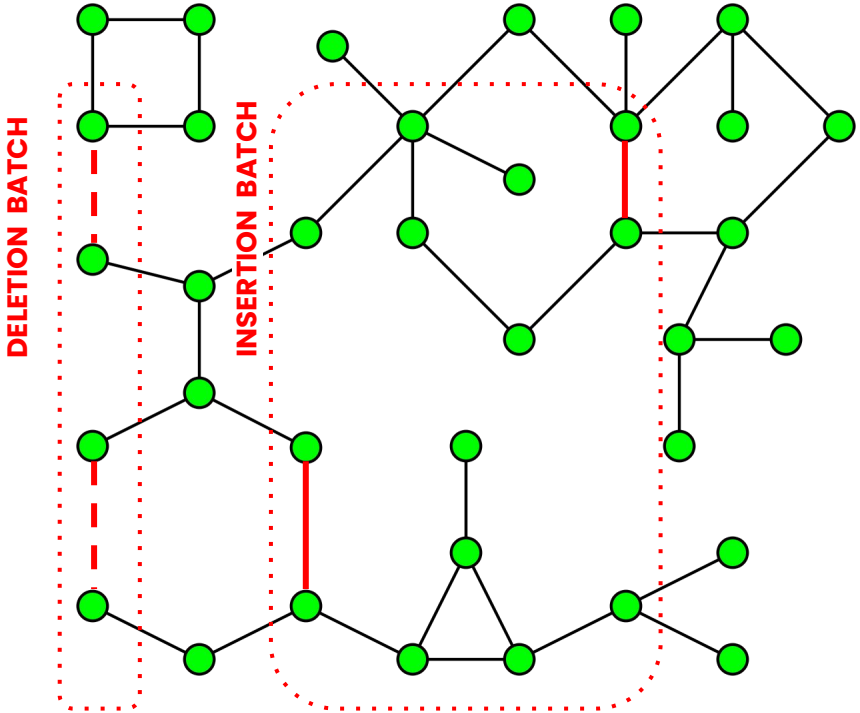
Considering connected components of dynamic graphs, **edge insertions** may join two different components, and **edge deletions** may split one component into two.

Intuitively:

Given the graph G and the components labels C, **determining if an insertion has joined two components can be done in O(1) time.**

However, **determining if an deletion splits a component will potentially lead to a SPSP** taking worst case O(V+E) time.

Can **aggregate updates into a batch** over time or until a number are collected which can then be applied in parallel, to reduce redundant computation.



CC Algorithm for Dynamic Graphs

McColl et al present a parallel algorithm for CC in dynamic graphs with below attributes:

- **Correctness:** Results are correct and consistent at fixed points in time for the graph meanwhile
- **Parallel:** minimize synchronization and communication
- **Time efficient:** At least asymptotically equivalent with better performance in practice
- **Space efficient:** preferably O(V) extra storage as graph itself covers O(V+E)

Data Structure

The main approach is to maintain a **"parent-neighbor" subgraph structure** based on BFS trees.

For each vertex, **parents** are its adjacent vertices that are in the level above and **neighbors** are ones that are in the same level. Use an array to store parents and neighbors with an **constant upper bound** to keep extra space O(V).

Table I

THE DATA STRUCTURES MAINTAINED WHILE TRACKING DYNAMIC CONNECTED COMPONENTS.

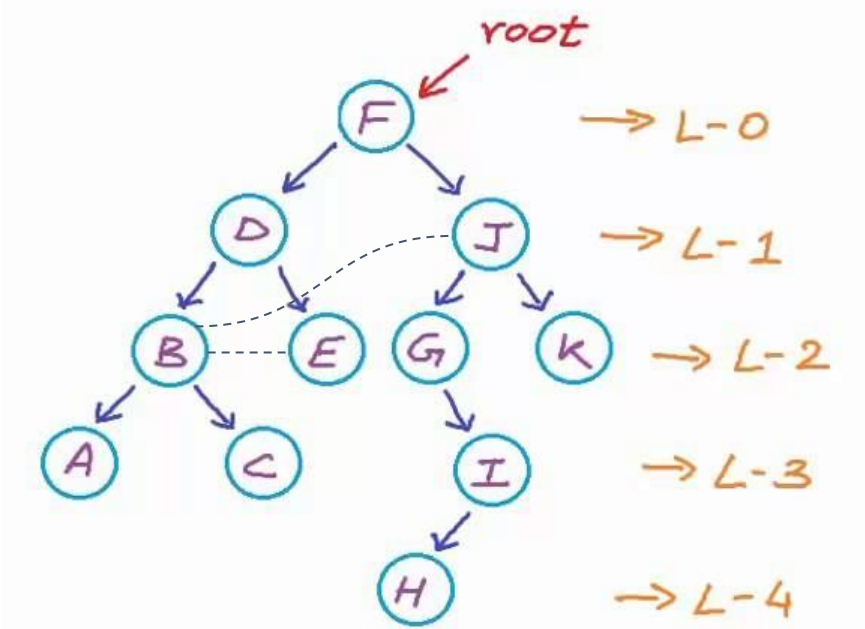
Name	Description	Type	Size (Elements)
C	Component labels	array	$O(V)$
$Size$	Component sizes	array	$O(V)$
$Level$	Approximate distance from the root	array	$O(V)$
PN	Parents and neighbors of each vertex	array of arrays	$O(V \cdot thresh_{PN}) = O(V)$
$Count$	Counts of parents and neighbors	array	$O(V)$
$thresh_{PN}$	Maximum count of parents and neighbors for a given vertex	value	$O(1)$
$\tilde{E}_I$	Batch of edges to be inserted into graph	array	$O(batch\ size)$
$\tilde{E}_R$	Batch of edges to be deleted from graph	array	$O(batch\ size)$

Main Idea

The high level idea is that upon deleting an edge, using the data structure introduced above, it is desirable to be able to determine "safe" deletion in O(1) time correctly, and **minimize the examination of "unsafe" deletion which is actually safe**.

For example, vertex B has 2 parents D and J, and neighbor E. Deletion of (B, D) or (B, J) or (B, E) is safe as there are alternative paths to root.

Can use parents and neighbors' level value to determine deletion safety. **Negative level value** indicates vertices that may have lost all their parents but still have neighbors so other vertices cannot rely on it for path to root.



Build PN subgraph

Use a **parallel BFS** to extract the BFS trees. For each vertex, store a list of parents and neighbors.

Distinguish parents and neighbors with marks. Positive label for parents while negative label for neighbors.

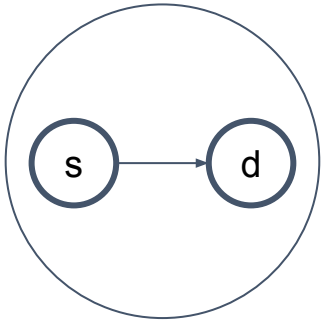
The list may be filled as there is a constant upper bound.

Algorithm 1 A parallel breadth-first traversal that extracts the parent-neighbor subgraph.

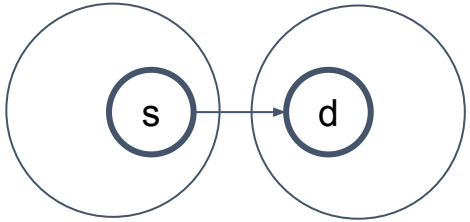
Input: $G(V, E)$
Output: C_{id}, $Size$, $Level$, PN, $Count$

1: **for** $v \in V$ **do**
2: $\quad Level[v] \leftarrow \infty$, $Count[v] \leftarrow 0$
3: **for** $v \in V$ **do**
4: $\quad$ **if** $Level[v] = \infty$ **then**
5: $\quad\quad Q[0] \leftarrow v$, $Q_{start} \leftarrow 0$, $Q_{end} \leftarrow 1$
6: $\quad\quad Level[v] \leftarrow 0$, $C_{id}[v] \leftarrow v$
7: $\quad\quad$ **while** $Q_{start} \neq Q_{end}$ **do**
8: $\quad\quad\quad Q_{stop} \leftarrow Q_{end}$
9: $\quad\quad\quad$ **for** $i \leftarrow Q_{start}$ **to** Q_{stop} **in parallel do**
10: $\quad\quad\quad\quad$ **for each neighbor** d **of** $Q[i]$ **do**
11: $\quad\quad\quad\quad\quad$ **if** $Level[d] = \infty$ **then**
12: $\quad\quad\quad\quad\quad\quad Q[Q_{end}] \leftarrow d$
13: $\quad\quad\quad\quad\quad\quad Q_{end} \leftarrow Q_{end} + 1$
14: $\quad\quad\quad\quad\quad\quad Level[d] \leftarrow Level[Q[i]] + 1$
15: $\quad\quad\quad\quad\quad\quad C_{id}[d] \leftarrow C_{id}[Q[i]]$
16: $\quad\quad\quad\quad\quad$ **if** $Count[d] < thresh_{PN}$ **then**
17: $\quad\quad\quad\quad\quad\quad$ **if** $Level[Q[i]] < Level[d]$ **then**
18: $\quad\quad\quad\quad\quad\quad\quad PN_d[Count[d]] \leftarrow Q[i]$
19: $\quad\quad\quad\quad\quad\quad\quad Count[d] \leftarrow Count[d] + 1$
20: $\quad\quad\quad\quad\quad\quad$ **else if** $Level[Q[i]] = Level[d]$ **then**
21: $\quad\quad\quad\quad\quad\quad\quad PN_d[Count[d]] \leftarrow -Q[i]$
22: $\quad\quad\quad\quad\quad\quad\quad Count[d] \leftarrow Count[d] + 1$
23: $\quad\quad\quad Q_{start} \leftarrow Q_{stop}$
24: $\quad\quad Size[v] \leftarrow Q_{end}$

Edge Insertion



1) Intra-connecting



2)Inter-connecting

To insert (s, d), the edge is either with in a CC (**intra-connecting**) or between CCs (**inter-connecting**).

The first case can be handled in parallel. If d's PN is not full, try to add s as a parent or neighbor. If d's PN is filled and s is a parent, replace a neighbor with s.

The latter is handled **serially** that a parallel BFS starts at the joining vertex to relabel the smaller CC and add vertices to the larger CC's BFS tree.

Algorithm 2 The algorithm for updating the parent-neighbor subgraph for inserted edges.

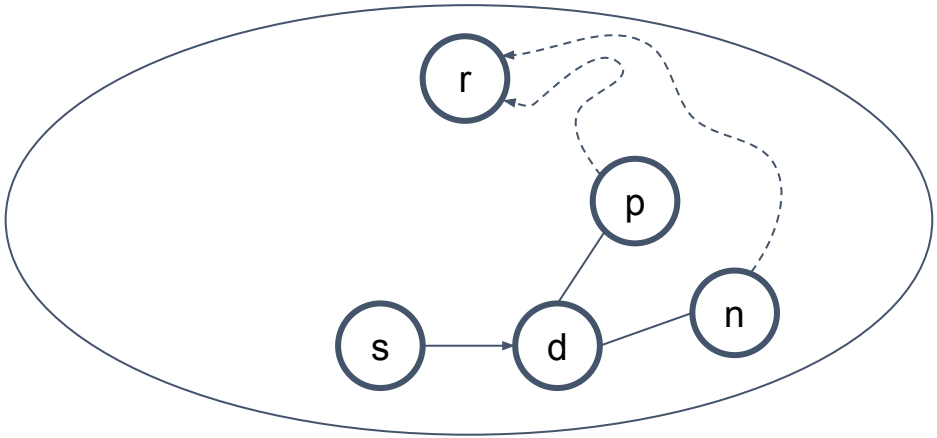
Input: $G(V, E)$, $\tilde{E}_I$, C_{id}, $Size$, $Level$, PN, $Count$
Output: C_{id}, $Size$, $Level$, PN, $Count$

1: **for all** $\langle s, d\rangle \in \tilde{E}_I$ **in parallel do** $E \leftarrow E \cup \langle s, d\rangle$
2: **insert**$(E, \langle s, d\rangle)$
3: **if** $C_{id}[s] = C_{id}[d]$ **then**
4: **if** $Level[s] > 0$ **then**
5: **if** $Level[d] < 0$ **then**
6: **// d is not "safe"**
7: **if** $Level[s] < -Level[d]$ **then**
8: **if** $Count[d] < thresh_{PN}$ **then**
9: $PN_d[Count[d]] \leftarrow s$
10: $Count[d] \leftarrow Count[d] + 1$
11: **else**
12: $PN_d[0] \leftarrow s$
13: $Level[d] \leftarrow -Level[d]$
14: **else**
15: **if** $Count[d] < thresh_{PN}$ **then**
16: **if** $Level[s] < Level[d]$ **then**
17: $PN_d[Count[d]] \leftarrow s$
18: $Count[d] \leftarrow Count[d] + 1$
19: **else if** $Level[s] = Level[d]$ **then**
20: $PN_d[Count[d]] \leftarrow -s$
21: $Count[d] \leftarrow Count[d] + 1$
22: **else if** $Level[s] < Level[d]$ **then**
23: **for** $i \leftarrow 0$ **to** $thresh_{PN}$ **do**
24: **if** $PN_d[i] < 0$ **then**
25: $PNV_d[i] \leftarrow s$,
26: **Break for-loop**
27: $\tilde{E}_I \leftarrow \tilde{E}_I \backslash \langle s, d\rangle$
28: **for all** $\langle s, d\rangle \in \tilde{E}_I$ **do**
29: **if** $C_{id}[s] \neq C_{id}[d]$ **then**
30: **if** $Size[s] = 1$ **then**
31: $Size[s] \leftarrow 0$
32: $Size[d] \leftarrow Size[d] + 1$
33: $C_{id}[s] \leftarrow C_{id}[d], PN_s[0] \leftarrow d$
34: $Level[s] \leftarrow$ **abs**$(Level[d]) + 1, Count[s] \leftarrow 1$
35: **else**
36: **connectComponent**$($**Input**$, s, d)$

Edge Deletion

Deleting (s, d) has two safe cases:
1) If d has at least one remaining parent with **non-negative level**, a path to the root must exist.
2) If d has no parent, **negate its level value**. Then its **neighbors** are checked for non-negative level values to be safe.

Edges are processed **from two ends respectively** as undirected.



Algorithm 3 The algorithm for updating the parent-neighbor subgraph for deleted edges.

Input: $G(V, E)$, $\tilde{E}_R$, C_{id}, $Size$, $Level$, PN, $Count$
Output: C_{id}, $Size$, $Level$, PN, $Count$

1: **for all** $\langle s, d\rangle \in \tilde{E}_R$ **in parallel do**
2: $E \leftarrow E \backslash \langle s, d\rangle$
3: $hasParents \leftarrow$ false
4: **for** $p \leftarrow 0$ **to** $Count[d]$ **do**
5: **if** $PN_d[p] = s$ **or** $PN_d[p] = -s$ **then**
6: $Count[d] \leftarrow Count[d] - 1$
7: $PN_d[p] \leftarrow PN_d[Count[d]]$
8: **if** $PN_d[p] > 0$ **then**
9: $hasParents \leftarrow$ true
10: **if (not** $hasParents$**) and** $Level[d] > 0$ **then**
11: $Level[d] \leftarrow -Level[d]$

UPDATE

12: **for all** $\langle s, d\rangle \in \tilde{E}_R$ **in parallel do**
13: **for all** $p \in PN_d$ **do**
14: **if** $p \geq 0$ **or** $Level$[**abs**(p)] > 0 **then**
15: $\tilde{E}_R \leftarrow \tilde{E}_R \backslash \langle s, d\rangle$
16: $PREV \leftarrow C_{id}$
17: **for all** $\langle s, d\rangle \in \tilde{E}_R$ **do**
18: $unsafe \leftarrow (C_{id}[s] = C_{id}[d] = PREV_s)$
19: **for all** $p \in PN_d$ **do**
20: **if** $p \geq 0$ **or** $Level$[**abs**(p)] > 0 **then**
21: $unsafe \leftarrow$ false

VERIFICATION

22: **if** $unsafe$ **then**
23: **if** $\{\langle u, v\rangle \in G(E, V) : u = s\} = \emptyset$ **then**
24: $Level[s] \leftarrow 0, C_{id}[s] \leftarrow s$
25: $Size[s] \leftarrow 1, Count[s] \leftarrow 0$
26: **else**
27: **Algorithm 4**
28: **repairComponent**(**Input**, s, d)

Repairing upon Unsafe Deletion

Unsafe deletion will demand **a partial BFS** to correctly examine the CCs, starting from d back to the root searching vertices in the equal or lower level. If no path found, split the CC by adding new the BFS tree. Otherwise, the first BFS ends and a new BFS traces back to relabel and rebuild part of the CC.

Algorithm 4 The algorithm for repairing the parent-neighbor subgraph when an unsafe deletion is reported.

Input: $G(V, E)$, E_R, C_{id}, $Size$, $Level$, PN, $Count$, s, d
Output: C_{id}, $Size$, $Level$, PN, $Count$

$Q[0] \leftarrow d$, $Q_{start} \leftarrow 0$, $Q_{end} \leftarrow 1$
$SLQ \leftarrow \emptyset$, $SLQ_{start} \leftarrow 0$, $SLQ_{end} \leftarrow 0$
$Level[d] \leftarrow 0$, $C_{id}[d] \leftarrow d$
$disconnected \leftarrow$ true
while $Q_{start} \neq Q_{end}$ **do**
  $Q_{stop} \leftarrow Q_{end}$
  for $i \leftarrow Q_{start}$ **to** Q_{stop} **in parallel do**
    $u \leftarrow Q[i]$
    for each neighbor v **of** u **do**
      if $C_{id}[v] = C_{id}[s]$ **then**
        if $Level[v] \leq$ **abs**$(Level[d])$ **then**
          $C_{id}[v] \leftarrow C_{id}[d]$
          $disconnected \leftarrow$ false
          $SLQ[SLQ_{end}] \leftarrow v$
          $SLQ_{end} \leftarrow SLQ_{end} + 1$
        else
          $C_{id}[v] \leftarrow C_{id}[d]$
          $Count[v] \leftarrow 0$
          $Level[v] \leftarrow Level[u] + 1$
          $Q[Q_{end}] \leftarrow v$
          $Q_{end} \leftarrow Q_{end} + 1$
      if $Count[v] < thresh_{PN}$ **then**
        if $Level[u] < Level[v]$ **then**
          $PN_v[Count[v]] \leftarrow u$
          $Count[v] \leftarrow Count[v] + 1$
        else if $Level[v] = Level[v]$ **then**
          $PN_v[Count[v]] \leftarrow -u$
          $Count[v] \leftarrow Count[v] + 1$
  $Q_{start} \leftarrow Q_{stop}$
if $disconnected$ **then**
  $Size[d] \leftarrow Q_{end}$
else
  for $i \leftarrow SLQ_{start}$ **to** SLQ_{end} **in parallel do**
    $C_{id}[i] \leftarrow C_{id}[s]$
  while $SLQ_{start} \neq SLQ_{end}$ **do**
    $SLQ_{stop} \leftarrow SLQ_{end}$
    for $i \leftarrow SLQ_{start}$ **to** SLQ_{stop} **in parallel do**
      $u \leftarrow SLQ[i]$
      for each neighbor v **of** u **do**
        if $C_{id}[v] = C_{id}[d]$ **then**
          $C_{id}[v] \leftarrow C_{id}[u]$
          $Count[v] \leftarrow 0$
          $Level[v] \leftarrow Level[u] + 1$
          $SLQ[SLQ_{end}] \leftarrow v$
          $SLQ_{end} \leftarrow SLQ_{end} + 1$
        if $Count[v] < thresh_{PN}$ **then**
          if $Level[u] < Level[v]$ **then**
            $PN_v[Count[v]] \leftarrow u$
            $Count[v] \leftarrow Count[v] + 1$
          else if $Level[v] = Level[v]$ **then**
            $PN_v[Count[v]] \leftarrow -u$
            $Count[v] \leftarrow Count[v] + 1$
    $Q_{start} \leftarrow Q_{stop}$

Evaluation: Tuning threshPN

Higher threshPN causes more updates, but also fewer unsafe deletes.

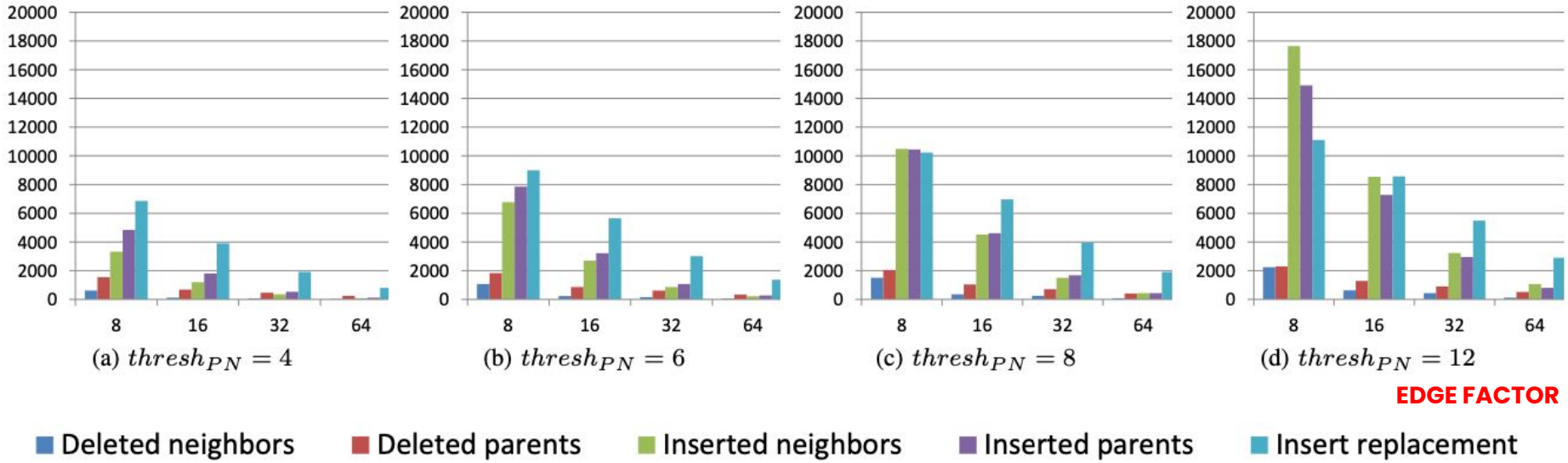


Figure 1. Average number of inserts and deletes in PN array for batches of $100K$ updates for RMAT-22 graphs. The subfigures are for different values of $thresh_{PN}$. Note that the ordinate is dependent on the specific bar chart. The charts for RMAT-21 graphs had very similar structure and have been removed for the sake of brevity.

Evaluation: Tuning threshPN

Higher threshPN causes more updates, but also much fewer unsafe deletes.

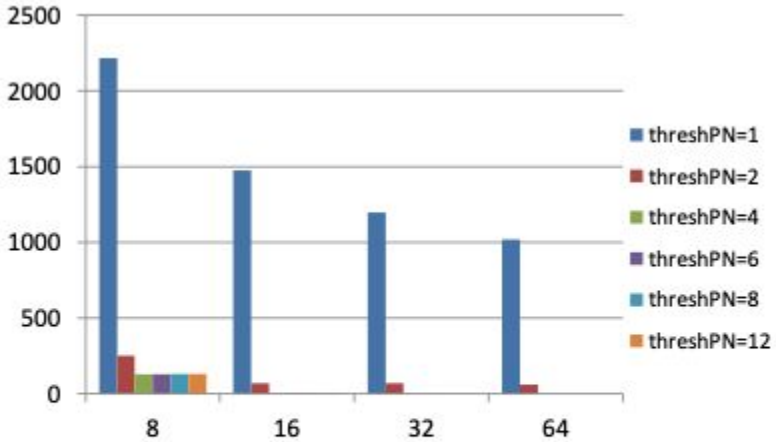


Figure 2. Average number of unsafe deletes in PN data structure for batches of $100K$ updates as a function of the average degree (x-axis) and $thresh_{PN}$ (bars).

Evaluation: Performance

Compared with recalculating the CCs using the parallel static Shiloach-Vishkin implementation:

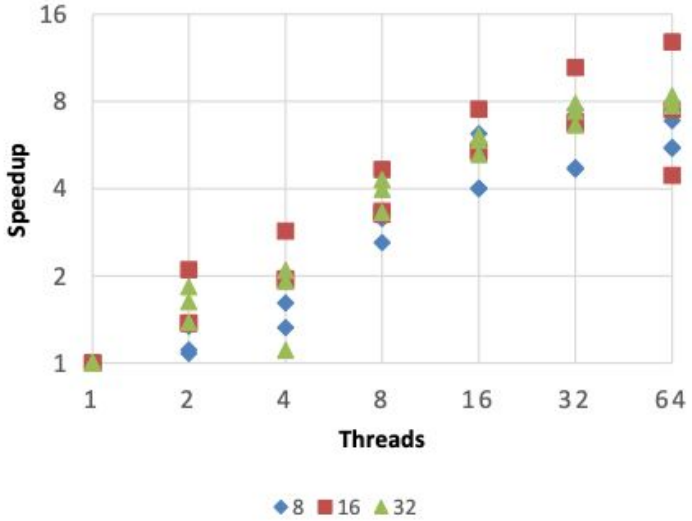


Figure 4. Speed up of the new algorithm over performing parallel static recomputation after each batch on three different RMAT-22 graphs with each average degree as a function of the number of threads.

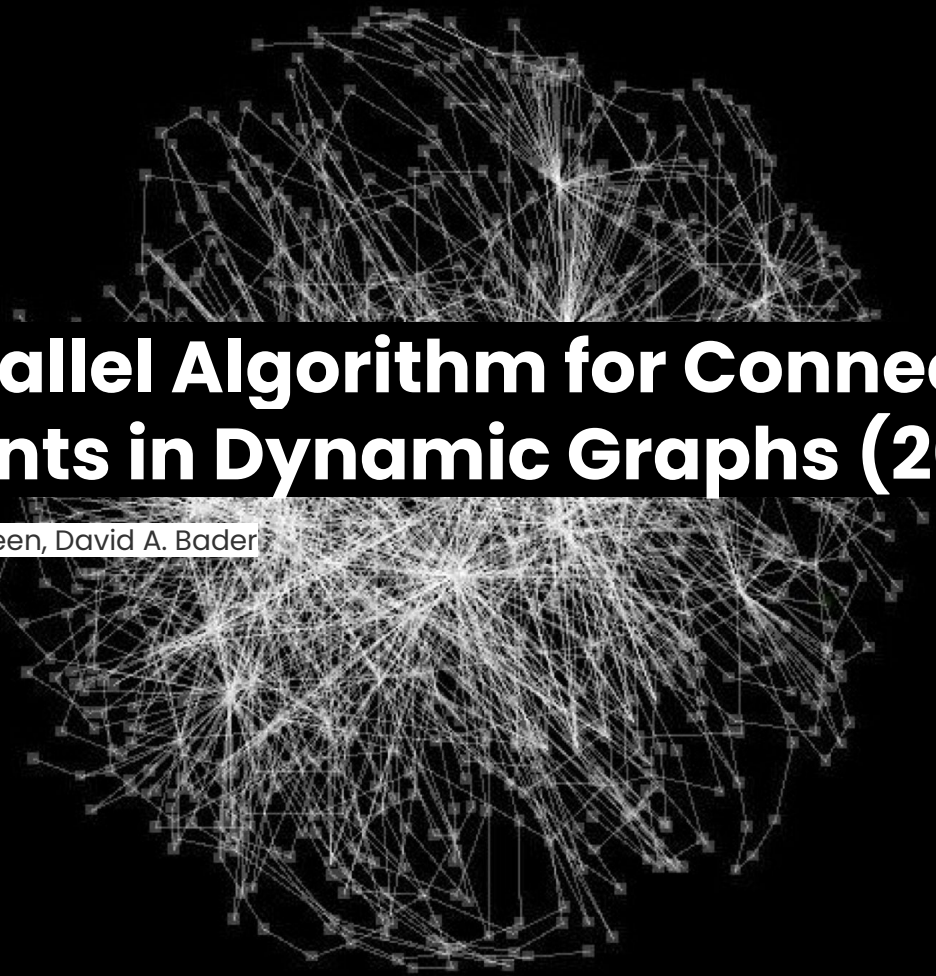
Strength and Weakness

Pros

- Can process larger graph size as O(V) extra space needed
- Works well in practice as real-world graphs typically have low diameter

Cons

- ThreshPN seems a trade-off costing time for space which may not be desired
- Tuning threshPN makes the algorithm “graph-aware”
- New parallel BFS can be integrated to further lift performance



A New Parallel Algorithm for Connected Components in Dynamic Graphs (2013)

By Robert McColl, Oded Green, David A. Bader

Presentation by Qing Feng

Mar 31 2022