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.

Eg. Social Networks
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 $O(V)$.
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:    $Level[v] \leftarrow \infty$, $Count[v] \leftarrow 0$
3: for $v \in V$ do
4:    if $Level[v] = \infty$ then
5:        $Q[0] \leftarrow v$, $Q_{\text{start}} \leftarrow 0$, $Q_{\text{end}} \leftarrow 1$
6:        $Level[v] \leftarrow 0$, $C_{id}[v] \leftarrow v$
7:    while $Q_{\text{start}} \neq Q_{\text{end}}$ do
8:        $Q_{\text{stop}} \leftarrow Q_{\text{end}}$
9:        for $i \leftarrow Q_{\text{start}}$ to $Q_{\text{stop}}$ in parallel do
10:           for each neighbor $d$ of $Q[i]$ do
11:              if $Level[d] = \infty$ then
12:                  $Q[Q_{\text{end}}] \leftarrow d$
13:                  $Q_{\text{end}} \leftarrow Q_{\text{end}} + 1$
14:                  $Level[d] \leftarrow Level[Q[i]] + 1$
15:                  $C_{id}[d] \leftarrow C_{id}[Q[i]]$
16:          if $Count[d] < \text{thresh}_{PN}$ then
17:              if $Level[Q[i]] < Level[d]$ then
18:                  $PN_{d}[Count[d]] \leftarrow Q[i]$
19:                  $Count[d] \leftarrow Count[d] + 1$
20:              else if $Level[Q[i]] = Level[d]$ then
21:                  $PN_{d}[Count[d]] \leftarrow -Q[i]$
22:                  $Count[d] \leftarrow Count[d] + 1$
23:        $Q_{\text{start}} \leftarrow Q_{\text{stop}}$
24:    $Size[v] \leftarrow Q_{\text{end}}$
To insert (s, d), the edge is either either 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: \( G(V, E), \tilde{E}_t, C_{id}, \text{Size}, \text{Level}, \text{PN}, \text{Count} \)
Output: \( C_{id}, \text{Size}, \text{Level}, \text{PN}, \text{Count} \)

1: for all \((s, d) \in \tilde{E}_t\) in parallel do
2: \( E \leftarrow E \cup \langle s, d \rangle \)
3: \( C_{id}[s] = C_{id}[d] \) then
4: \( \text{if Level}[s] > 0 \) then
5: \( \text{if Level}[d] < 0 \) then
6: // d is not “safe”
7: if Level[s] > Level[d] then
8: \( \text{if Count}[d] < \text{thresh}_{PN} \) then
9: \( \text{PN}_d[\text{Count}[d]] \leftarrow s \)
10: \( \text{Count}[d] \leftarrow \text{Count}[d] + 1 \)
11: else
12: \( \text{PN}_d[0] \leftarrow s \)
13: \( \text{Level}[d] \leftarrow -\text{Level}[d] \)
14: else if \( \text{Count}[d] < \text{thresh}_{PN} \) then
15: \( \text{if Level}[s] < \text{Level}[d] \) then
16: \( \text{PN}_d[\text{Count}[d]] \leftarrow s \)
17: \( \text{Count}[d] \leftarrow \text{Count}[d] + 1 \)
18: else if Level[s] = Level[d] then
19: \( \text{PN}_d[\text{Count}[d]] \leftarrow s \)
20: \( \text{Count}[d] \leftarrow \text{Count}[d] + 1 \)
21: else if Level[s] < Level[d] then
22: for \( i \leftarrow 0 \) to \( \text{thresh}_{PN} \) do
23: \( \text{PN}_d[i] \leftarrow s \)
24: Break for-loop
25: end for
26: end if
27: end if
28: end if
29: end for
30: \( s \neq C_{id}[d] \) then
31: \( \text{if Size}[s] = 1 \) then
32: \( \text{Size}[s] \leftarrow 0 \)
33: \( \text{Size}[d] \leftarrow \text{Size}[d] + 1 \)
34: \( C_{id}[s] \leftarrow C_{id}[d], \text{PN}_s[0] \leftarrow d \)
35: \( \text{Level}[s] \leftarrow \text{abs}(\text{Level}[d]) + 1, \text{Count}[s] \leftarrow 1 \)
36: else
37: connectComponent(Input, s, d)
38: end if
39: end for
**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:** \(G(V, E), \tilde{E}_R, C_{id}, \text{Size, Level, PN, Count} \)

**Output:** \(C_{id}, \text{Size, Level, PN, Count} \)

1. for all \((s, d)\) in parallel do
2. \(E \leftarrow E \setminus (s, d)\)
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. \(PN_d[p] \leftarrow PN_d[\text{Count}[d]]\)
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\([d]\)
12. for all \((s, d)\) in parallel do
13. for all \(p \in PN_d\) do
14. if \(p \geq 0\) or Level\([\text{abs}(p)] > 0\) then
15. \(E_R \leftarrow E_R \setminus (s, d)\)
16. \(PREV \leftarrow C_{id}\)
17. for all \((s, d)\) in \(E_R\) do
18. unsafe \(\leftarrow (C_{id}[s] = C_{id}[d] = PREV)\)
19. for all \(p \in PN_d\) do
20. if \(p \geq 0\) or Level\([\text{abs}(p)] > 0\) then
21. unsafe \(\leftarrow\) false
22. if unsafe then
23. if \(\{(u, v) \in G(E, V) : u = s\} = \emptyset\) then
24. Level\([s]\) \(\leftarrow\) 0, \(C_{id}[s] \leftarrow s\)
25. \(\text{Size}[s] \leftarrow 1, \text{Count}[s] \leftarrow 0\)
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), Eq, Cid, Size, Level, PN, Count, s, d
Output: Cid, Size, Level, PN, Count
1: Q[0] ← d, Qstart ← 0, Qend ← 1
2: SLQ ← ∅, SLQ_start ← 0, SLQ_end ← 0
3: Level[d] ← 0, Cid[d] ← d
4: disconnected ← true
5: while Qstart ≠ Qend do
6: Q Stop ← Q_end
7: for i ← Qstart to Q_stop in parallel do
8: u ← Q[i]
9: for each neighbor v of u do
10: if Cid[v] = Cid[u] then
11: if Level[v] ≤ abs(Level[d]) then
12: Cid[v] ← Cid[d]
13: disconnected ← false
14: SLQ[SLQ_end] ← v
15: SLQ_end ← SLQ_end + 1
16: else
17: Cid[v] ← Cid[d]
18: Count[v] ← 0
19: Level[v] ← Level[u] + 1
20: Q[Q_end] ← v
21: Q_end ← Q_end + 1
22: if Count[v] < threshPN then
23: if Level[v] < Level[u] then
24: PNv[Count[v]] ← u
25: Count[v] ← Count[v] + 1
26: else if Level[v] = Level[u] then
27: PNv[Count[v]] ← −u
28: Count[v] ← Count[v] + 1
29: Q Start ← Q_stop
30: else if disconnected then
31: Size[d] ← Q_end
32: for i ← SLQ_start to SLQ_end in parallel do
33: Cid[i] ← Cid[s]
34: while SLQ_start ≠ SLQ_end do
35: SLQ_stop ← SLQ_end
36: SLQ_start ← SLQ_end
37: for i ← SLQ_start to SLQ_stop in parallel do
38: u ← SLQ[i]
39: for each neighbor v of u do
40: if Cid[v] = Cid[u] then
41: Cid[v] ← Cid[u]
42: Count[v] ← 0
43: Level[v] ← Level[u] + 1
44: SLQ[SLQ_end] ← v
45: SLQ_end ← SLQ_end + 1
46: if Count[v] < threshPN then
47: if Level[u] < Level[v] then
48: PNv[Count[v]] ← u
49: Count[v] ← Count[v] + 1
50: else if Level[v] = Level[u] then
51: PNv[Count[v]] ← −u
52: Count[v] ← Count[v] + 1
53: Q_start ← 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 $threshPN$. 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