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

By: Robert McColl, Oded Green, David A. Bader

Presented by Ian Gatlin

What is a Dynamic Graph Algorithm?

Dynamic Graph - Ordered series of discrete static graphs. There are a lot of applications where updates to a graph are happening quickly. For example, you could imagine on Facebook there are thousands of friendship updates every minute.

Dynamic Graph Algorithm - Theoretically is able to find insights from graphs such as spanning trees, shortest paths, and connected components by using updates to the graph rather recomputing with traditional static graph algorithms.

What Does This Paper Address?

- Their proposed data structure allows us to keep track of connected components with a dynamic graph algorithm
- Runs up to 128x faster than static algorithms
- Achieves 14x parallel speedup
- Takes O(V) space

Why is this important?

- Connected components is a well studied graph problem with applications in social media networks, 3-d image processing, and recommendation systems

Two Cases

Adding an edge -> just check the component membership of each of the edge's two vertices.

- If vertices are in the same component then we do nothing, if they are different, combine them into the same component

Deleting an edge -> more complicated, naively takes O(V+E) to run a BFS or DFS on the edges in that component to determine connectivity and potentially relabel new connected components. This is too slow for dynamic graph usage.

Main Idea

Can we find a way that determines if an edge deletion doesn’t break up a connected component (is “safe”) with 100% true positive rate and in constant time?

False negatives (marked unsafe when safe) will be handled with our naive solution. We need to minimize these.

If we can find a way to store in our data structure alternative paths between vertices in a component in linear space (O(V)), we can determine if a path is safe by looking at those alternative paths while also being space and time efficient.

Parents-Neighbors Sub-graph Data Structure

Constant

Table I
THE DATA STRUCTURES MAINTAINED WHILE TRACKING DYNAMIC CONNECTED COMPONENTS.

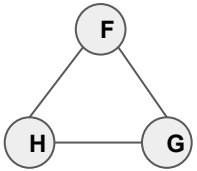
Name	Description	Type	Size (Elements)
C	Component labels	array	$O(V)$
$Size$	Component sizes	array	$O(V)$
$Level$	Approximate distance from the root	array	$O(V)$
PN	Parents and neighbors of each vertex	array of arrays	$O(V \cdot thresh_{PN}) = O(V)$
$Count$	Counts of parents and neighbors	array	$O(V)$
$thresh_{PN}$	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	$O(batch\ size)$

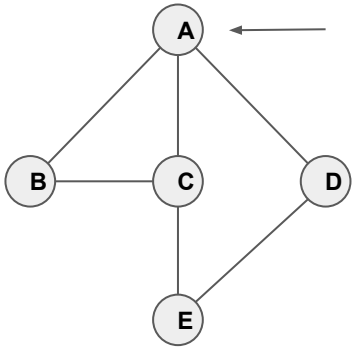
- This turns our undirected graph into a directed graph representation, therefore, we need to apply transformations for prospective edge {s, d} as both <s, d> and <d, s>

Overall: O(V)

Undiscovered Potential to change Finished

Initialization





Start parallel
Full-BFS

∀ vertices @ start

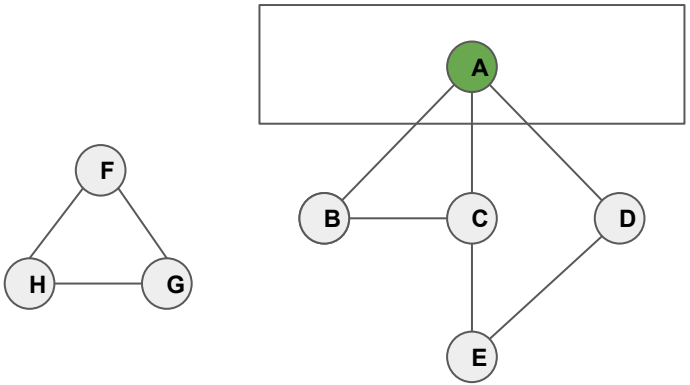
C (Label)	-
Size	-
Level	∞
Count	0
PN	[ ]

Thresh_pn => 2

Q_index	0	1	2	3	4	5	6	7
Label								
Queue								

Undiscovered Potential to change Finished

Initialization



A

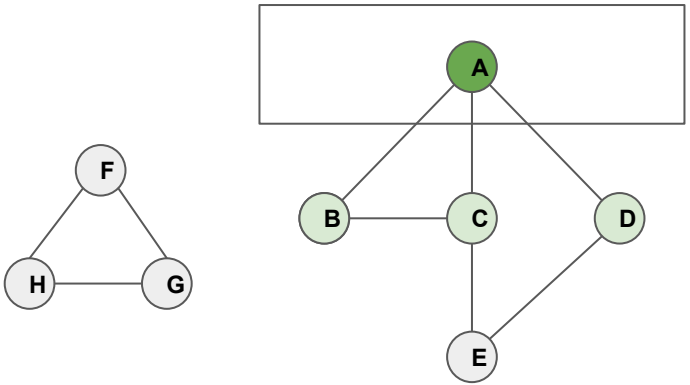
C (Label)	A
Size	-
Level	0
Count	0
PN	[ ]

Thresh_pn => 2

Q_index	0	1	2	3	4	5	6	7
Label	start	stop, end						
Queue	A							

Undiscovered Potential to change Finished

Initialization



A

C (Label)	A
Size	-
Level	0
Count	0
PN	[ ]

B, C, D

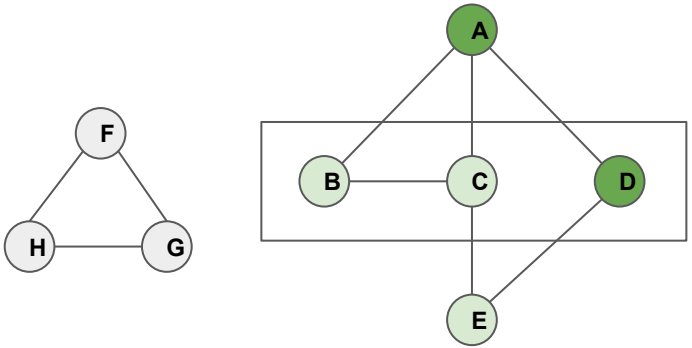
C (Label)	A
Size	-
Level	1
Count	1
PN	[A]

Thresh_pn => 2

Q_index	0	1	2	3	4	5	6	7
Label	start	stop			end			
Queue	A	B	C	D				

Undiscovered Potential to change Finished

Initialization



B

C (Label)	A
Size	-
Level	1
Count	2
PN	[A, -C]

C

C (Label)	A
Size	-
Level	1
Count	2
PN	[A, -B]

E

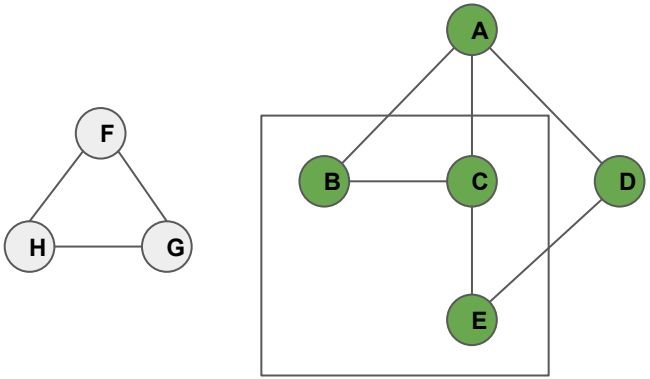
C (Label)	A
Size	-
Level	2
Count	2
PN	[C, D]

Thresh_pn => 2

Q_index	0	1	2	3	4	5	6	7
Label		start			stop	end		
Queue	A	B	C	D	E			

Undiscovered Potential to change Finished

Initialization



B

C (Label)	A
Size	-
Level	1
Count	2
PN	[A, -C]

C

C (Label)	A
Size	-
Level	1
Count	2
PN	[A, -B]

E

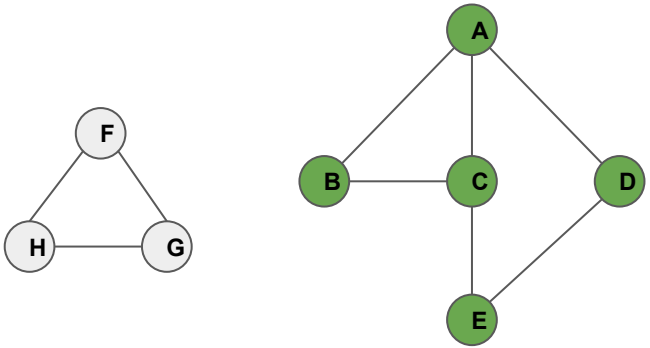
C (Label)	A
Size	-
Level	2
Count	2
PN	[C, D]

Thresh_pn => 2

Q_index	0	1	2	3	4	5	6	7
Label					start	stop, end		
Queue	A	B	C	D	E			

Undiscovered Potential to change Finished

Initialization



A

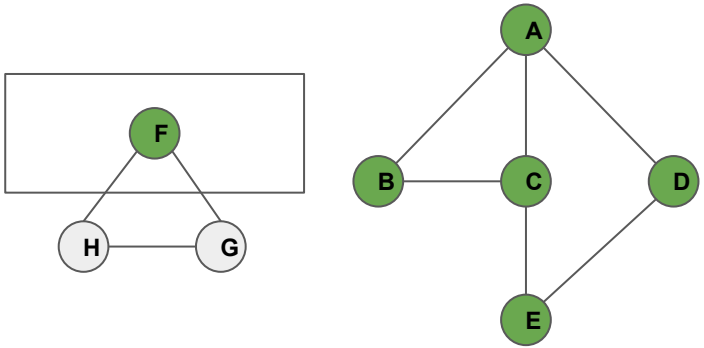
C (Label)	A
Size	5
Level	0
Count	0
PN	[ ]

Thresh_pn => 2

Q_index	0	1	2	3	4	5	6	7
Label						start, stop, end		
Queue	A	B	C	D	E			

Undiscovered Potential to change Finished

Initialization



F

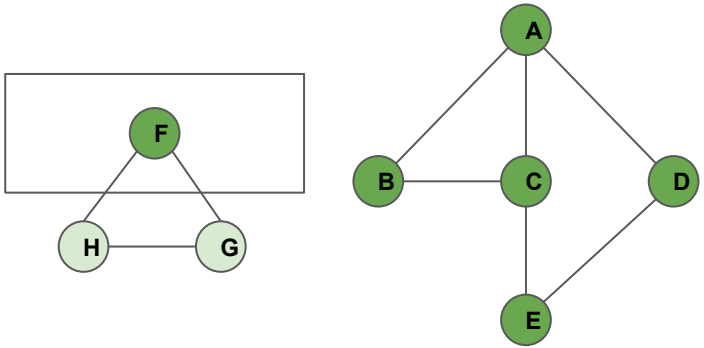
C (Label)	F
Size	-
Level	0
Count	0
PN	[ ]

Thresh_pn => 2

Q_index	0	1	2	3	4	5	6	7
Label	start	stop, end						
Queue	F							

Undiscovered Potential to change Finished

Initialization



F

C (Label)	F
Size	-
Level	0
Count	0
PN	[ ]

H, G

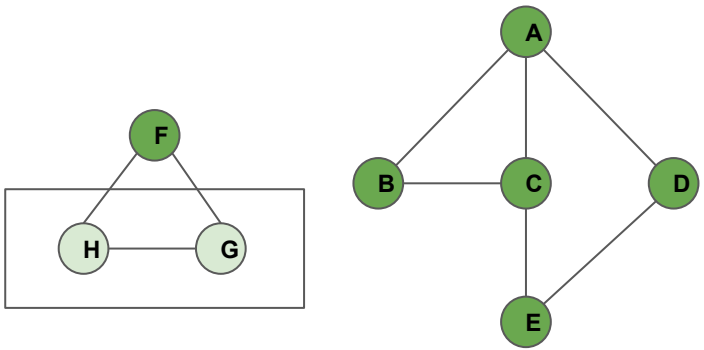
C (Label)	F
Size	-
Level	1
Count	1
PN	[F]

Thresh_pn => 2

Q_index	0	1	2	3	4	5	6	7
Label	start	stop		end				
Queue	F	H	G					

Undiscovered Potential to change Finished

Initialization



H

C (Label)	F
Size	-
Level	1
Count	2
PN	[F, -G]

G

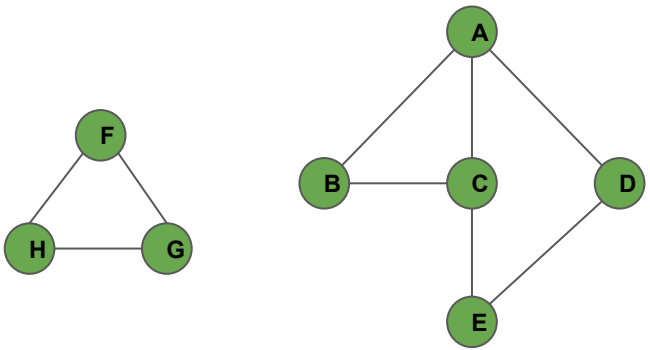
C (Label)	F
Size	-
Level	1
Count	2
PN	[F, -H]

Thresh_pn => 2

Q_index	0	1	2	3	4	5	6	7
Label		start		stop, end				
Queue	F	H	G					

Undiscovered Potential to change Finished

Initialization



F

C (Label)	F
Size	3
Level	0
Count	0
PN	[ ]

Thresh_pn => 2

Q_index	0	1	2	3	4	5	6	7
Label				start, stop, end				
Queue	F	H	G					

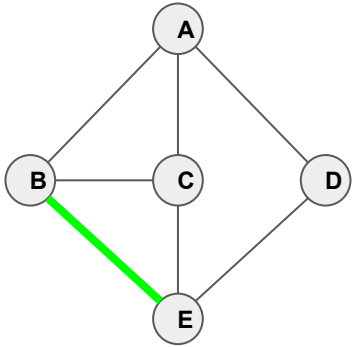
Additional Notes about Initialization Algorithm

- If length of array PN_d (count) == thresh_pn, then the potential parent or neighbor is simply not added to the array
- Because of the BFS traversal order, parents will always be put into array before neighbors. Parents are more important than neighbors
- Each frontier is searched in parallel

Adding Edges

- Done as a batch
- Edges within a component are processed in parallel, edges across components are processed sequentially after intra-component edges are finished.

Adding “Safe” Edges Within Component



E

C (Label)	A
Size	-
Level	2
Count	2
PN	[C, D]

→

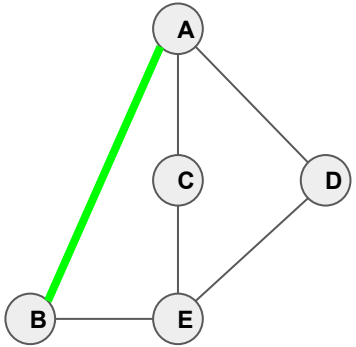
E

C (Label)	A
Size	-
Level	2
Count	3
PN	[C, D, B]

Edge <B, E>

- Try to create parent relationship in PN_E no matter what
  - (replace a neighbor relationship)
- If there is space in PN_E, add a neighbor relationship

Adding “Unsafe” Edges Within Component



B

C (Label)	A
Size	-
Level	-2
Count	1
PN	[-E]

→

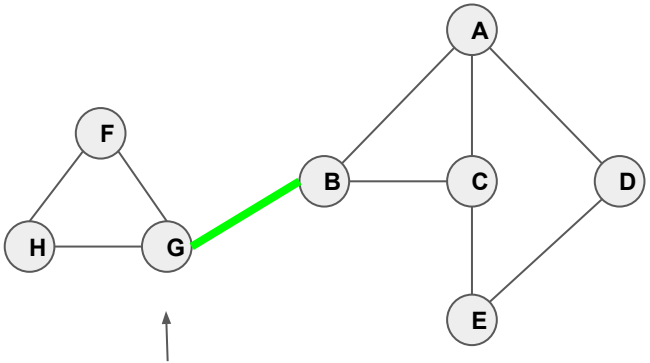
B

C (Label)	A
Size	-
Level	2
Count	2
PN	[-E, A]

Edge <A, B>

- If parent relationship possible, then add and set level of destination to positive level of neighbors
- Level becomes an approximation

Adding Edges Across Components



Start parallel BFS into its previous component's members

G

C (Label)	A
Size	-
Level	2
Count	1
PN	[B]

F/H

C (Label)	A
Size	-
Level	3
Count	2
PN	[G, -H/-F]

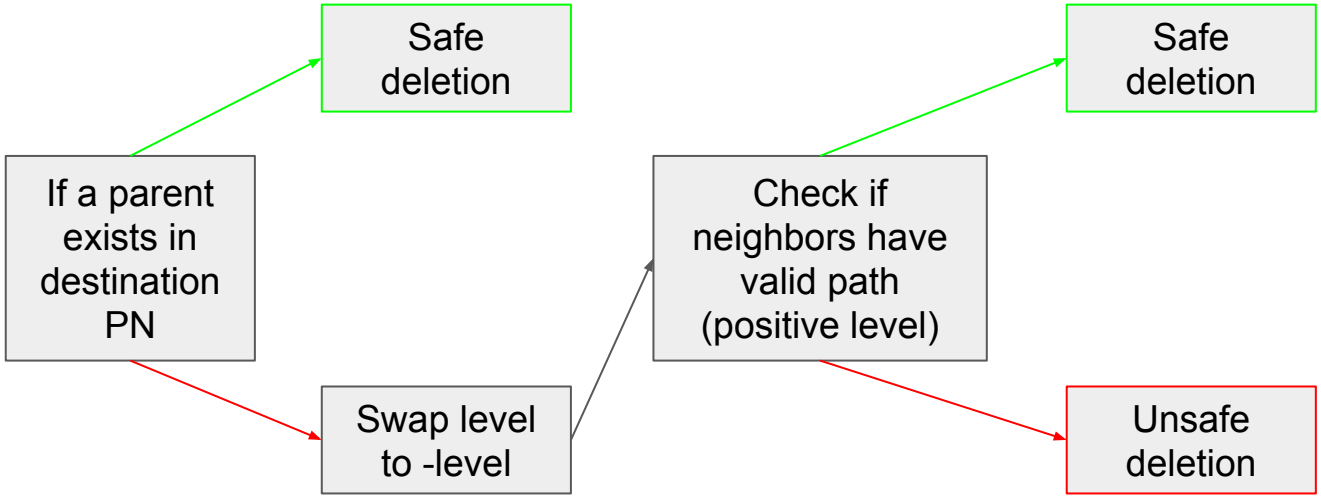
Edge <B, G>

- Rerun's parallel BFS from initialization on smaller component
- If component size = 1, then manually update state of that vertex

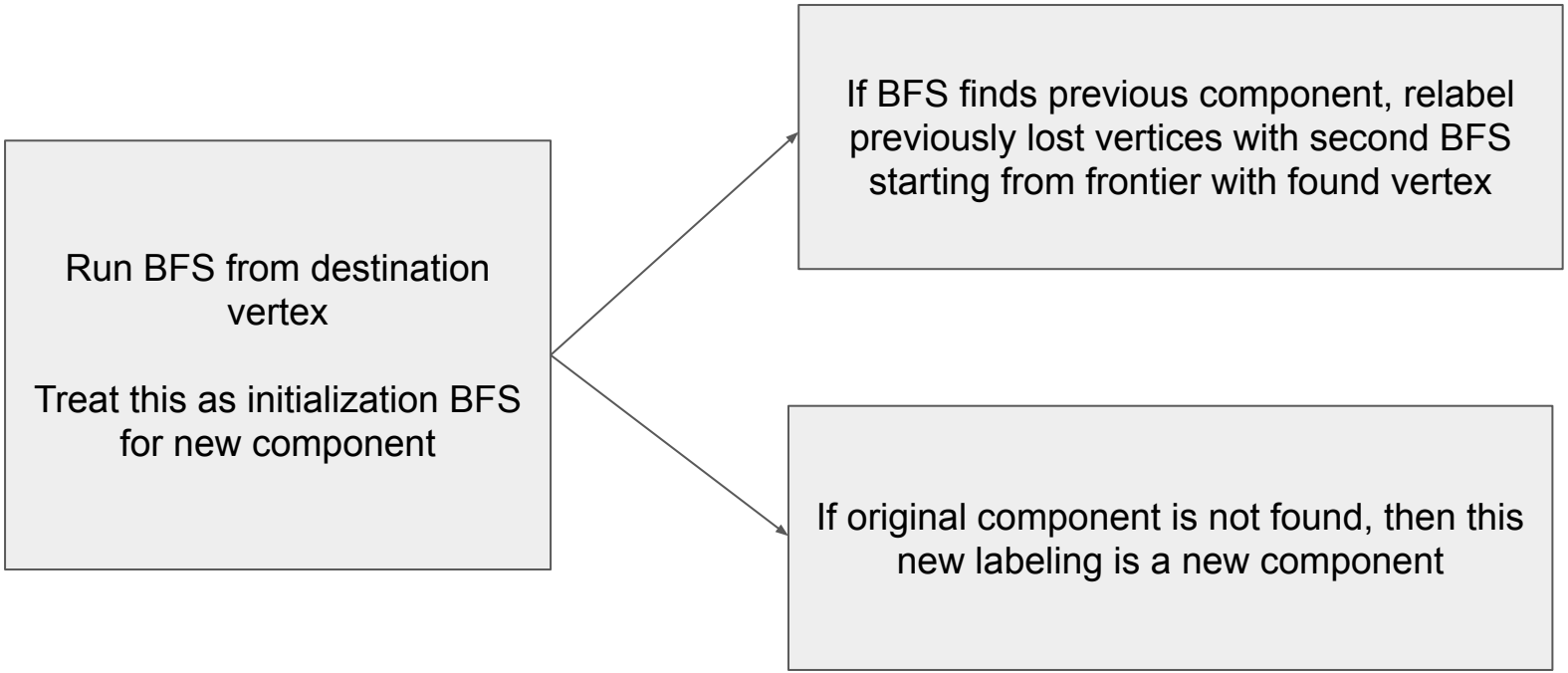
Deleting Edges

- Also done as a batch
- Delete all batched edges and update PN and Level in parallel, and then check “safety” of points by looking at each PN array of vertices that had deletions
- Safe deletions can be processed with no extra restructuring of data

Detecting “Unsafe” Edge Deletion



To Resolve an Unsafe Deletion



Experimental Methodology

- Used synthetic graphs with:
  - Skewed degree distribution
  - Power law
  - Few large components
  - Many small components
- Graphs used had scale S and edge factor E
  - |V| = 2^S
  - |E| = E * 2^S.
  - E corresponds to the average degree.
- 10 batches with 100K updates are used

Table II
GRAPH SIZES USED IN OUR EXPERIMENTS FOR TESTING THE ALGORITHM. MULTIPLE GRAPHS OF EACH SIZE WERE USED.

	Totals edge per average degree			
Vertices	8	16	32	64
2M	16M	32M	64M	128M
4M	32M	64M	128M	256M
16M	128M	256M	512M	—

Failed Experiments

- Finding two-hop connecting paths with adjacency list intersection
  - 750 unsafe deletes = full static recompute
- Maintaining spanning trees for each component
  - Lots of recomputation if tree edge was deleted. 90% of deletions were safe
- Maintaining two spanning trees in each component
  - 99.7% of deletions were safe, and computationally challenging
- Path between vertices with BFS
  - Because of power law, frontier get very large quickly

Quantitative Results

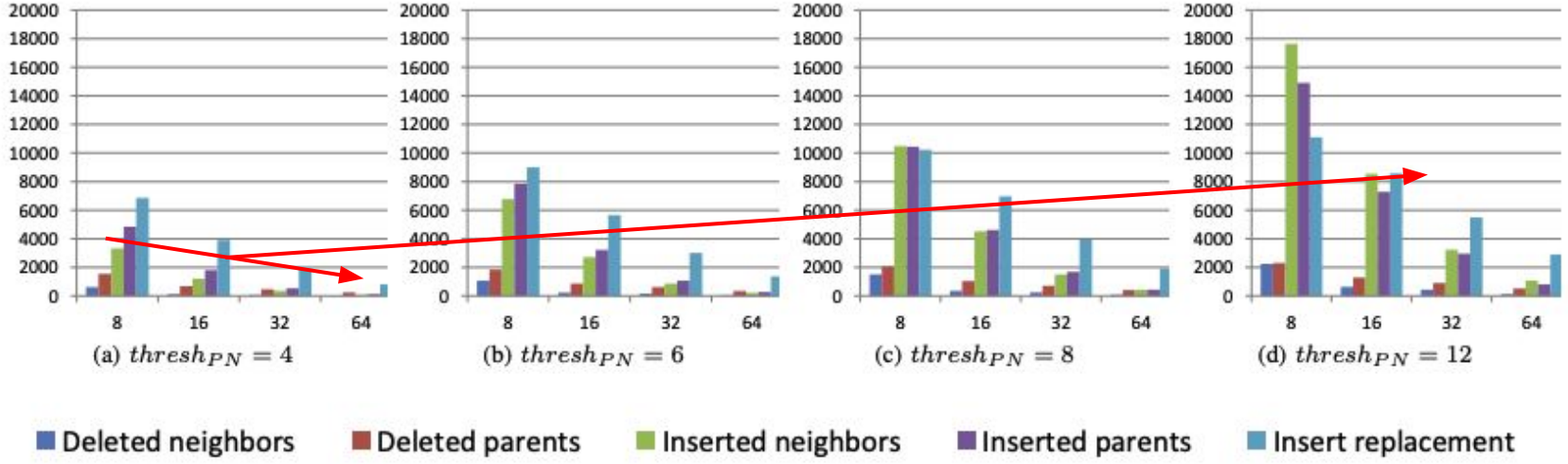


Figure 1. Average number of inserts and deletes in PN array for batches of $100K$ updates for RMAT-22 graphs. The subfigures are for different values of $thresh_{PN}$. Note that the ordinate is dependent on the specific bar chart. The charts for RMAT-21 graphs had very similar structure and have been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

Quantitative Results

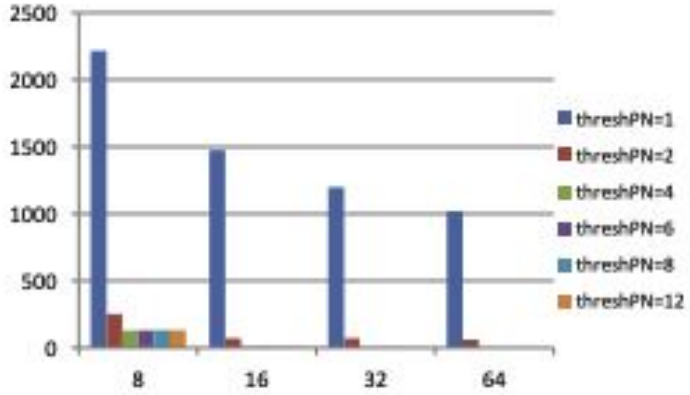


Figure 2. Average number of unsafe deletes in PN data structure for batches of $100K$ updates as a function of the average degree (x-axis) and $thresh_{PN}$ (bars).

Performance Results

Colors just represent 3 different graphs of same properties tested on

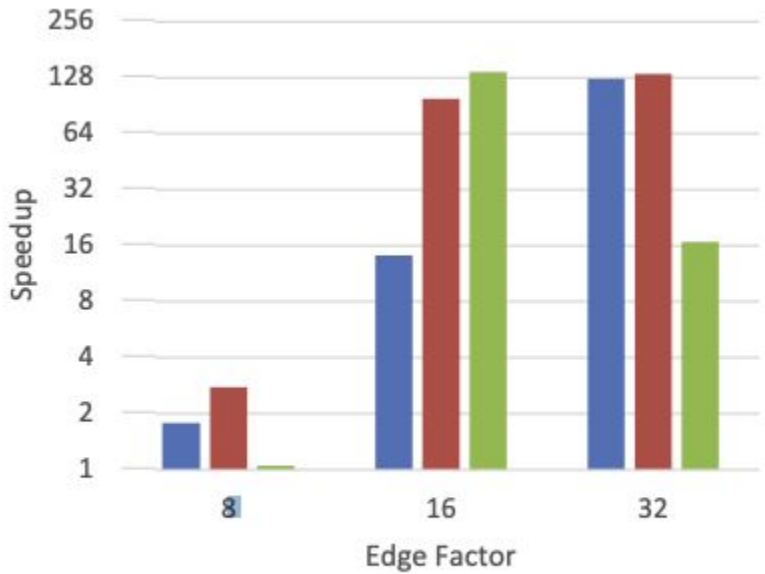


Figure 6. Speed up over performing static recomputation after each batch on scale 24 graphs for three graphs at each edge factor using 64 threads.

Pros

- Algorithm is simple and concise, easily implementable
- Thresh_pn value for achieving constant size is effective
- Tested on graphs with properties on which they would be useful

Cons

- Figures and explanation could have been more clear
- Would be nice to compare unsafe deletion rate to failed experiments
- Need testing on larger graphs (2^24 ~ 16 million)

Thank you! Questions

Appendix

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: $Level[d] \leftarrow -Level[d]$

Appendix

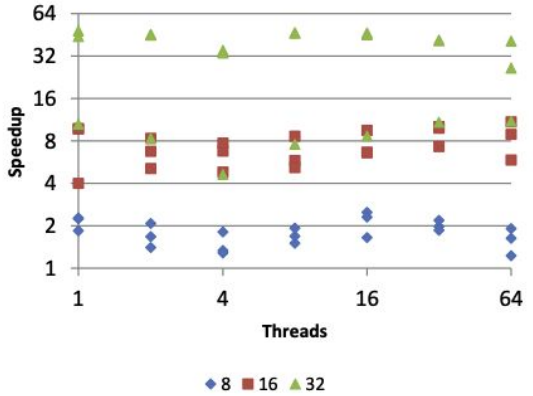


Figure 3. Strong scaling results on RMAT-22 graphs with different average degree as a function of the number of threads. Results include three graphs at each average degree.

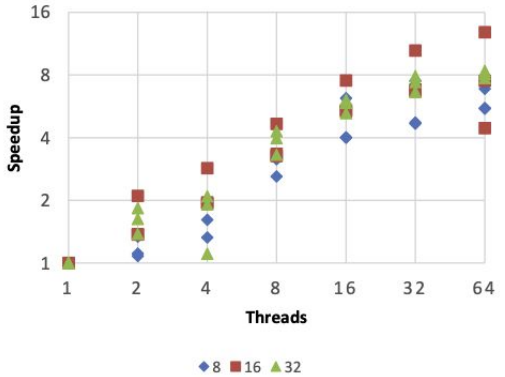


Figure 4. Speed up of the new algorithm over performing parallel static recomputation after each batch on three different RMAT-22 graphs with each average degree as a function of the number of threads.