A New Algorithmic Model for Graph Analysis of Streaming Data

Chunxing Yin
Georgia Institute of Technology
cyin9@gatech.edu

Jason Riedy
Georgia Institute of Technology
jason.riedy@cc.gatech.edu

David A. Bader
Georgia Institute of Technology
bader@cc.gatech.edu

Abstract

The constant and massive influx of new data into analysis systems needs to be addressed without assuming we can pause the onslaught. Here we consider one aspect: non-stop graph analysis of streaming data. We formalize a new and practical algorithm model that includes both single-run analysis as well as efficiently updating analysis results only around changed data. In our model, a massive graph undergoes changes from an input stream of edge insertions and removals. These changes occur concurrently with analysis. Algorithms do not pause or stop the input stream. Assuming basic data access safety, we consider an algorithm *valid* for our model if the output is correct for a graph consisting of the initial graph and some implicit subset of concurrent changes.

Our technical contributions include 1. the first formal model for graph analysis with concurrent changes, 2. properties of the model including how our model is the strongest possible without point-in-time graph views, 3. demonstrations of our model on connected components and PageRank, and 4. an extension to updating results incrementally.

CCS Concepts • **Theory of computation → Dynamic graph algorithms**; • **Information systems →** *Network data models*; *Data streams*;

ACM Reference Format:
Chunxing Yin, Jason Riedy, and David A. Bader. 2018. A New Algorithmic Model for Graph Analysis of Streaming Data. In *Proceedings of 14th International Workshop on Mining and Learning with Graphs (MLG'18).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.nnnn/nnnnnnn.nnnnnnn

1 Introduction

Applications in fields like computer network security and social media analyze an ever-changing environment. The data is rich in relationships and lends itself to graph analysis. Network security applications analyze nearly *one million events per second*[21] to shut down threats immediately. Social networks use the relationships in over 140 thousand "tweets" per second[20] to over 510 thousand comments per second[23] to find the best advertisements for one's current needs or desires.

There are many computational models and software frameworks that address *updating* graph analysis results given a starting result and point-in-time or snapshot views of the changing graph, see Section 3 and our STINGER[8] framework. To our knowledge none of these address computing that initial, starting analysis result without stopping the world to provide an initial snapshot view. On graphs with billions of vertices and tens to hundreds of billions of edges, providing a snapshot view for the initialization imposes a large performance cost on the entire analysis system, affecting not only the initializing query but also all concurrently running analyses. For example, STINGER can ingest ten million graph updates per second[8]. The updating kernels peak at hundreds of thousands[7] to around a million[13] updates per second. *Initializing* those kernels requires time proportional to the graph size[14] and not the update size. These limit the peak performance of the system as a whole.

We address the initialization problem by presenting the *first formal model* for graph analysis on streaming data in which algorithms *run concurrently* with graph updates (Section 4). Our model also applies to updating analysis results (Section 8). Analysis updates occur concurrently with changes, permitting many more simultaneous analysis clients on a single massive graph. This is an extreme model for extreme rates that assumes only memory consistency. We are interested in possibilities without fine-grained locking or versioning to enable far more simultaneous applications referencing the same massive graph store.

Not all algorithms are appropriate for our execution model (Section 2), but some core algorithms like breadth-first search work *if* we consider the algorithms as traversing a graph that consists of the starting graph plus some implicit subset of the concurrent graph changes. We consider this result *valid* for our model (not incorrect!) as defined formally in Section 4.

This paper's contributions are as follows:

- We provide a formal algorithmic model for applying graph analysis algorithms to graphs being updated concurrently from a live stream (Section 4). Algorithms considered *valid* may not require large-scale copying of the graph nor pausing the data stream.
- We prove multiple properties of our model. Invalid algorithms can be demonstrated with a single change (Corollary 5.5), and algorithms that produce subgraphs of their inputs (*e.g.* tree construction) cannot be proven

MLG'18, 20 August, 2018, London, United Kingdom
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.nnnn/nnnnnnn.nnnnnnn

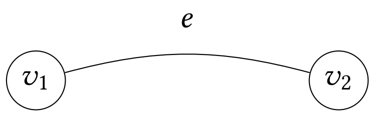


Figure 1. Invalid instance for degree counting algorithm

to output a result corresponding to any snapshot view and thus our model is the strongest possible without snapshot views (Theorem 5.7).

- We demonstrate our model on connected components (Section 6) and PageRank (Section 7).
- We then apply our model to *updating* results without full recomputation (Section 8).

This paper elides proofs for space.

Note that any algorithm can be made valid by copying the entire graph and running on the copy. That is not feasible for massive graphs but could be an option for small graphs or subgraphs. If the rate of change is sufficiently slow, then various techniques could provide a snapshot view without copying[2, 6, 16] while introducing overhead acceptable for relatively slow rates of change. Our model is not bound to any software framework and applies not only to STINGER but also distributed systems like DegAwareRHH[11] and Tegra[12].

2 Illustrating Validity with Degrees

A simple example illustrates the model we formalize in Section 4. Given an undirected graph $G = (V, E)$ without self-loops, consider computing the degree $\deg(v)$ of every vertex $v \in V$. Algorithm 1 shows a simple algorithm that iterates over every vertex v in parallel and stores the number of edges found to be incident to v. We intentionally gloss over details of the data structure and algorithm. In STINGER, which represents an undirected graph by two directed edges, Algorithm 1 would loop over vertices, walk adjacent edges, increment a local count, then store that count to $\deg(v)$.

Algorithm 1 Invalid degree counting algorithm

```
for all $v \in V$ do in parallel
    $\deg(v) \leftarrow$ number of edges incident to $v$;
end parallel for
```

Algorithm 1 is correct applied to a static graph. If the graph changes during Algorithm 1's execution, however, the computed degrees may not correspond to *any* undirected graph. Figure 1 provides a simple example. Graph G contains two vertices and one edge. The algorithm could process v_1 and assign $\deg(v_1) \leftarrow 1$. A concurrent change could remove the edge before processing v_2, in which case $\deg(v_2) \leftarrow 0$.

There is no two-vertex undirected graph with those vertex degrees. So this result cannot correspond to the initial graph plus some implicit subset of concurrent changes, and we consider this algorithm *invalid* for our model.

Algorithm 2 Valid degree counting algorithm

```
$\deg(v) \leftarrow 0 \; \forall v \in V$;
for all $(u, v) \in E$ do in parallel
    atomically: $\deg(u) \leftarrow \deg(u) + 1$;
    atomically: $\deg(v) \leftarrow \deg(v) + 1$;
end parallel for
```

A different method that loops over edges, Algorithm 2, is *valid* for our model. All initial edges are counted unless removed before the algorithm traverses that edge. In that case, the subset of concurrent changes includes such removals and does not include any that remove previously counted edges. Inserted edges are counted and in the subset unless the insertion occurs after the loop would traverse it. So the algorithm's result includes a subset of concurrent changes and is *valid* for our model. Systems that store an undirected graph using pairs of edges apply a tie-breaking rule like executing the loop's body only when $u < v$ for the same result.

3 Related Work

There are many different models for dynamic graphs for different applications. To our knowledge all rely on an initial starting point that requires static computation over a potentially massive graph. And only the evolving graph model considers updates with concurrent changes. We compare related existing models to our new model.

Dynamic graph model Classic dynamic graph algorithms[5] efficienctly update analysis results rather than recomputing. These algorithms copy the graph into data structures specialized and useful only for each query. In contrast, our model assumes that maintaining specialized data structures is infeasible for massive graphs.

Data stream model Here algorithms compute results by treating a graph as a stream of edges[9, 15]. These algorithms make a constant or logarithmic number of passes over the edge stream and are restricted to using limited memory. Many streaming algorithms compute approximate results. These algorithms apply in situations like streaming a graph from external storage or sensors. In contrast, our model assumes there is a massive, changing graph that can be accessed arbitrarily.

Evolving graph model Algorithms in this model probe a changing graph to update approximated metrics[1]. The changes occur in discrete time steps but not during computation. Their probing method could permit concurrent changes, but their analysis would need extended to execution spanning multiple time steps. Our model considers exact results on a "nearby" graph in terms of incorporated concurrent changes.

Streamed graphs The algorithm in [3] maintains a sparsified graph from a stream of edge insertions and deletions to provide an estimate of triangle counts. The algorithm contains two phases, one that constructs the sampled, sparsified graph and another separate phase that counts triangles. The sparsified graph is not altered in the second phase. This algorithm could be applied to our original problem by maintaining snapshots of the sparsified graph, and that is a different approach than our model's.

Current software frameworks, including STINGER [8], DegAwareRHH [11], Tegra [12], and Floe [22] provide point-in-time snapshot views to implement algorithms in these models. Their focus is on rapidly updating existing, initial results. Our new model adds computing those initial results at the cost of a different mental model. Other frameworks like PHISH[17] implement a version of the data stream model but permit repeatedly streaming edges through distributed computing units for complex queries like subgraph isomorphism. PHISH and similar frameworks do not focus on dynamically changing graphs but rather on limiting computational resources.

4 Validity Model

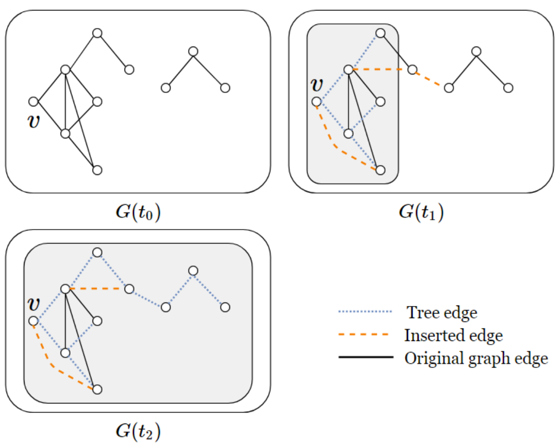
In this model, we consider a weighted and directed graph $G = (V, E)$ on vertex set V and edge set E. We convert an undirected graph to a directed one by replacing each edge with two directed edges in opposite directions. In the following discussion we denote by $n = |V|$, the number of vertices and $m = |E|$, the number of edges. Let $C = \{c_1, c_2, \dots\}$ be the (finitely many) kinds of changes to graph G defined on the system, such as edge insertion, edge deletion, edge weight changes. We will mainly focus on edge insertions and deletions in the following discussion. We consider new vertices only once a connecting edge is inserted.

Definition 4.1. An **input stream** S is a sequence of tuples $C_i = (c_i, e_i, t_i)$, where $c_i \in C$ is type of the change, e_i is the edge to be modified, and t_i is the arrival time of the change. Note that $t_i \leq t_{i+1}$ for all i.

Definition 4.2. A **sub-stream** of an input stream S is a subsequence of it, $S' = ((c_1^*, e_1^*, t_1^*), (c_2^*, e_2^*, t_2^*), \dots)$ where $(c_i^*, e_i^*, t_i^*) \in S$ and $t_i^* \leq t_{i+1}^*$.

In our *validity model*, the input stream keeps making changes to the graph while the algorithm runs. However, the algorithm is not notified of a change and it has no access to the current input stream. A change that happens at time t will instantly modify the corresponding edge (or vertex) and thus if the algorithm traverses this edge (or vertex) at time t, only the modified status will be seen. In the following discussion, "input stream" refers to the part of stream that arrives during the algorithm's execution.

Figure 2. Breadth first search illustration



We are interested in extreme cases where the graph is too large for the platform to maintain snapshots and the rate of change is too high to accept the overhead of localized revision control[2, 6, 16]. This could apply to a Raspberry Pi used as inexpensive network instrumentation during peak times as well as a large-scale system maintaining a massive data set. Because we are not providing point-in-time snapshot views, the output of the algorithm may not match either the initial state or the final state of the graph. We show later that the result need not match any intermediate time point either (Theorem 5.7).

Throughout the remainder of the paper, let G be a graph and $S = (C_1, C_2, \dots, C_k)$ be a finite input stream. We use $G \cup S$ to denote the graph obtained by making all changes defined in S to G in the order of $C_1, C_2, \dots, C_k$. We regard the output obtained from running the algorithm on G with input stream S as $OUT(G, S)$. Moreover, we denote the output produced by running the same algorithm on static graph $G \cup S$ as $OUT(G \cup S)$. We use $G(t)$ to denote the graph at time t (abstractly, not necessarily as traversed by the algorithm).

We provide a simple illustration using degree counting in Section 2. Here we consider a breadth first search (BFS) in more detail. The algorithm begins with vertex v as its *frontier* and repeatedly constructs new frontiers of neighboring and unexplored vertices. Additionally, the algorithm saves a discovered parent to produce a breadth first search tree. The algorithm proceeds unaware that the graph may change underneath. Low-level data access correctness must be provided by the underlying framework.

In Figure 2 the algorithm begins at time t_0 on $G(t_0)$ with the left most vertex v. Let the gray area represent the set of vertices that are discovered at time $t_i > t_0$. Suppose all three insertions arrive at time t_1. The back-edge from the next frontier leads to an already visited vertex and will be ignored. Similarly the edge entirely within the explored region will be ignored. Both of these changes would alter the BFS tree, but our model permits not including some concurrent changes. The inserted edge leading from the frontier to unexplored vertices will be included, and that edge is one element in the implicit set of concurrent changes.

We can see the output of this algorithm is not a BFS tree for the graph $G(t_0)$ nor for $G(t')$ at any single time point $t' \geq t_1$. But the result does include one of the concurrent changes. In our model, the basic BFS algorithm is *valid* because it produces a result corresponding to the input graph and a subset of concurrent changes. In Section 8 valid algorithms are extended to catch up to the last change during execution, producing updating algorithms that maintain valid results after every set of changes.

Now we formalize our new model of *validity* on concurrently changing data.

Definition 4.3. Let $\mathcal{A}$ be an algorithm running on a graph with input stream S. Suppose that the execution time window is $[t, t']$ with output $OUT(G_0, S)$, where G_0 is the initial state of the graph at time t. We say $\mathcal{A}$ **is valid for stream** S if there exists a sub-stream of S, say $S' = ((c_1^*, e_1^*, t_1^*), ..., (c_k^*, e_k^*, t_k^*))$, where $t \leq t_1^* \leq t_k^* \leq t'$, that makes $OUT(G_0, S)$ a solution for $\mathcal{A}$ on graph $G_0 \cup S'$.

Although we define S' to be an arbitrary sub-stream, in Section 5 we will show that S' is indeed a subset of the changes that are seen by the algorithm. To make the definitions rigorous, we keep S' to be an arbitrary subset of S in this section.

Definition 4.4. An algorithm is **valid** if for any input stream S, the algorithm is valid for S.

Note that the sub-stream S' can be empty. To facilitate our analysis, we introduce the following definitions.

Definition 4.5. A change (c, e, t) is **visible** if the algorithm traverses e at time $t^* \geq t$ and there are no other changes to e between time $(t, t^*]$. In this case, we say that the algorithm **traverses the change** C at time t^*. We use S_v to denote the sub-stream containing all visible changes in S.

The algorithm must traverse an edge in order to check its status, such as existence and weight. In order to describe the algorithms behavior when it encounters deleted edges, we assume that an edge deletion does not erase the data but only marks the edge as deleted. This is a convenience for analysis but not necessary for implementation. Moreover, if the input stream modifies an edge multiple times, the algorithm only notices the last change before the traversal according to Definition 4.5.

5 Validity Model Properties

We first show that we can reorder the input stream while keeping the streaming output invariant. We then prove that validity defined in the previous section is the strongest form of correctness under the assumption that the algorithm is not provided with the equivalent of a snapshot view.

Lemma 5.1. *Given graph G and input stream S, suppose change $C = (c, e, t)$ is visible and that C is traversed for the first time at $t^* \geq t$. Let $C' = (c, e, t')$ for any $t \leq t' \leq t^*$. If we replace C with C' in S, then (i) the sub-stream of visible changes S_v does not change except for this substitution, and (ii) the streaming output $OUT(G, S)$ remains the same.*

Corollary 5.2. *Let G be a graph, and $S_v = (C_1, C_2, \ldots, C_k)$ be the sub-stream of all visible changes in S, where $C_i = (c_i, e_i, t_i)$ for all $1 \leq i \leq k$. Suppose the algorithm traverses each C_i for the first time at t_i^*. We define another stream $S^* = S$ except for replacing each $C_i \in S_v$ with $C_i^* = (c_i, e_i, t_i^*)$. Then $\{C_i^*\}$ is the visible sub-stream of S^* and $OUT(G, S) = OUT(G, S^*)$.*

Note that in S^* and S_v^*, all the visible changes are ordered by non-decreasing t_i^*.

Corollary 5.3. *Let S_v^* be defined as in Corollary 5.2. We define a new input stream to be $S' = (C_1^*, C_2^*, \ldots, C_i^*)$, which contains the first i changes in S_v^*. Then every change in S' will be visible for any $1 \leq i \leq |S_v^*|$.*

Theorem 5.4. *Let S_v be the sub-stream of all visible changes in an input stream S, then an algorithm is valid if and only if for any input stream S, the algorithm is valid for S_v.*

Corollary 5.5. *If an algorithm is invalid, then there exists a single change input stream S that makes the algorithm invalid for S.*

Corollary 5.6. *Let S and S' be two different input streams. For a deterministic valid algorithm $\mathcal{A}$, if $S_v = S_v'$, then $OUT(G, S) = OUT(G, S')$.*

We notice that in most cases, the information of G required for verifying an algorithm's validity for an input stream usually is as much as that for recomputing an output statically on G. By Theorem 5.4, to verify if an algorithm is valid for S, we only need to check if $OUT(G, S)$ equals $OUT(G \cup S_v)$. Since we have no assumptions about the graph G, in most cases, it is impossible to determine the correctness of the output by checking a partial graph described by the input stream. Moreover, if $S_v = \emptyset$, it requires to verify if the output is correct for G without a snapshot of G.

Next we show that validity is the strongest form of correctness in this model if the algorithm satisfies the following properties:

1. Given a graph G and an input stream S, $OUT(G, S)$ is a subgraph of $G \cup \{\text{inserted edges}\}$. Moreover, there should exist an output that contains at least two edges
2. The algorithm is not able to traverse all edges or vertices belonging to $OUT(G, S)$ at the same moment.

Theorem 5.7. *For any valid algorithm satisfying the two conditions mentioned above, there exist a graph G and a finite input stream S such that $OUT(G, S) \neq OUT(G_i)$ for any $0 \leq i \leq k$, where $G_0 = G$ and $G_i = G_{i-1} \cup C_i$.*

Making a copy of the graph G satisfies the assumed properties for graphs with more than two edges. A copy with concurrent changes cannot be proven identical to any snapshot view, so no algorithm can create a snapshot view without being provided such a view as input. This is why we claim our model is the strongest possible without snapshot views.

6 Case Study: Connected Components

Given an undirected graph $G = (V, E)$, a connected component is a maximal subgraph of G such that there is a path between any two vertices in that subgraph. In this section, we adapt the parallel Parent-Neighbor subgraph algorithm [13] to our model and show the result is valid.

A *parent neighbor subgraph PN* is a subgraph of G sampled during breadth first search (BFS) such that each vertex tracks a constant number of its parents and neighbors. The *parents* of v are adjacencies of v in the previous frontier; the *neighbors* of v are adjacencies of v in the same frontier. To distinguish parents from neighbors, all parents are presented as a positive label while neighbors are stored in the form of $-v$.

We define the parameters used in the algorithm as follows:

- $C(v)$: component label of vertex v
- $Level(v)$: distance from v to the component root
- $thresh_{PN}$: maximum number of allowed parents and neighbors for each vertex
- $PN(v)$: parents and neighbors of each vertex
- $Label(v)$: $Label(v) \geq 0$ represents v has a path to the root through its parent; otherwise $Label(v) < 0$, all v's paths to root rely on v's neighbors

Algorithm 3 Extract a parent-neighbor subgraph

```
1:  for $v \in V$ do $Level(v) \leftarrow \infty$
2:  for $v \in V$ do
3:      if $Level(v) = \infty$ then
4:          add $v$ to $Frontier$
5:          $Level(v) \leftarrow 0, C(v) \leftarrow v$
6:          while $Frontier \neq \emptyset$ do
7:              for all $u \in Frontier$ do
8:                  remove $u$ from $Frontier$
9:                  for all $d \in N(u)$ do
10:                     if $Level(d) = \infty$ then
11:                         add $d$ to $Frontier$
12:                         $Level(d) = Level(u) + 1$
13:                         $C(d) = C(u)$
14:                     if $|PN(d)| < thresh_{PN}$ then
15:                         if $Level(u) = Level(d) - 1$ then
16:                             Add $u$ to $PN(d)$
17:                         else if $Level(u) = Level(d)$ then
18:                             Add $-u$ to $PN(d)$
```

We claim that $Level(v)$ is a valid labeling of the graph, which implies that $\cup_{v \in V} PN(v)$ is a valid parent-neighbor subgraph. To show its correctness, we will prove a stronger statement, that the edges (u, d) used in line 9-11 form a valid BFS tree T.

Theorem 6.1. *Let T be the BFS tree defined by Algorithm 3 with input stream S, then there exists a sub-stream of S that makes the output BFS tree T valid.*

Corollary 6.2. *Let T be the BFS tree defined by Algorithm 3 with input stream S, a sub-stream that make T valid is $\{C = (c_i, e_i, t_i) \in S | c_i = insert \ \& \ e_i \in T \text{ or } c_i = delete \ \& \ e_i \notin T\}$*

7 Case Study: PageRank

A directed graph with vertex set V can be represented by a sparse, unsymmetric matrix A with $a_{ij} = 1$ where there is an edge $i \longrightarrow j$. An undirected graph can be represented by a symmetric matrix similarly. Here we ignore self edges and let the diagonal of A be zero. Define D to be the diagonal matrix of out-degrees, or diag $D = A\mathbf{1}$ where $\mathbf{1}$ is a $|V|$-long vector with unit entries. If a vertex i is the source of no edges, let $d_{ii} = 1$ so that $1/d_{ii} = 1$. The definitions generalize to graphs with arbitrary non-negative weights.

While the original PageRank defines an eigenvector problem, a little algebraic manipulation as in [4, 10] finds an equivalent linear system

$$(I - \alpha A^T D^{-1})x = (1 - \alpha)v, \qquad (1)$$

where α is the "teleportation" constant in $(0, 1)$, v is a personalization vector, and x is the PageRank vector. For this discussion, v is a vector with entries $1/|V|$ denoting a uniformly random start, and $\|v\|_1 = 1$, although personalized PageRank applications use non-uniform vectors v. Solving this system with inexact arithmetic produces an approximate solution x. The backward error, or the distance to the nearest system solved exactly, is measured by the residual $r = (1 - \alpha)v - (I - \alpha A^T D^{-1})x$ up to a normwise scaling factor. Because PageRank defines a probability distribution, the normwise relative backward error is scaled by a constant independent of the graph structure.

Solving Equation (1) by Jacobi iteration, a simple splitting method, iterates the following computation:

$$x^{(k+1)} = \alpha A^T D^{-1} x^{(k)} + (1 - \alpha)v.$$

The graph operation, applying $A^T D^{-1}$, is valid in our non-stop model for implementations that examine each edge at most once *and* track the number of outward edges traversed per vertex. Such implementations will gather the row of A before computing its contribution to the output. The software frameworks referenced treat the graph as a sparse matrix and can satisfy the additional restriction easily. Insertions and removals during any single iteration may be missed, but the algorithm always applies the graph plus all changes before the iteration and some subset of changes concurrent during each iteration. Typical convergence criteria compare the iterates $x^{(k+1)}$ and $x^{(k)}$. Conveniently, this translates to backward error via the residual,

$$r^{(k)} = (1-\alpha)v - (I - \alpha A^T D^{-1})x^{(k)} = x^{(k+1)} - x^{(k)}.$$

So a single pass over the graph produces both the current residual and next iterate. If the residual $r^{(k)}$ is smaller than some tolerance, then the iterate $x^{(k)}$ solves a system acceptably close to the graph plus the subset of changes encountered during the iteration that produced $x^{(k+1)}$. Hence the method is valid in our model for the solution $x^{(k)}$, although common practice delivers $x^{(k+1)}$. The backward error is small no matter the path taken to $x^{(k)}$. Only the last iteration matters.

As an example of an invalid approach, consider using a stored out-degree or one computed separately from the edge walk. The resulting operator will no longer preserve the one-norm and will lead to a solution that is not a probability distribution. That cannot be a PageRank vector of any graph.

Being a valid implementation does not ensure that solving the system by Jacobi iteration or other iterative method actually converges. In the extreme, the graph could be entirely rewritten during each iteration, and the iteration never converges to a solution with small residual. Future work will characterize what rates of change lead to convergence.

8 Updating Algorithms

On a quiescent graph with no concurrent changes, an initial valid result could be updated to match the "true" result given the changes that had occured during execution. If the updating algorithm is valid even when the graph is changing concurrently, then the result will be valid. Recording changes during each execution and repeatedly catching up provides a valid result tracking the "true" result by fixing previously missed changes. Ideally such updating algorithms compute incrementally to avoid full recomputation and should be much faster than the initial computation. Such algorithms exist and are available in frameworks like STINGER[7, 13, 18, 19]. Here we formalize the updating method and present valid algorithms for updating PageRank and connected components (in Section 9).

An algorithm $\mathcal{A}$ runs consecutively at times $0 < t_1 < t_2 < t_3 < \cdots$ on the graph. During the i^{th} execution, $\mathcal{A}$ is given with Out_{i-1}, the output (or configuration) from the previous execution, and the set of changes Δ_{i-1}, the set of all changes that happen during the time $[t_{i-1}, t_i]$ spent computing Out_{i-1}. The initial configuration is computed by a valid algorithm (assume starting at time 0), and Δ_0 is the set of changes happening from time 0 to t_1.

Definition 8.1. An algorithm $\mathcal{A}$ is a **valid updating algorithm** if for all $i = 1, 2, 3, \ldots$, Out_i is a correct output for graph $G(t_i) \cup \delta_i$ for some $\delta_i \subseteq \Delta_i$.

If there is no concurrent change ($\Delta_i = \emptyset$), a valid updating algorithm will produce a snapshot result $OUT(G(t_i))$, which is also a snapshot result for all $t_i \le t \le t_{i+1}$.

The theoretical results in Section 5 also hold for updating algorithms. We note that all valid algorithms are valid updating algorithm by doing recomputations every time. However we will focus on algorithms that only perform local updates or converge faster than the initial computation. We apply the most important results in Section 5 here. All the proofs are similar to the original ones.

Corollary 8.2. *Let S_i be the sub-stream of all visible changes in Δ_i for i–$1, 2, \cdots$. An algorithm is a valid updating algorithm if and only if for all possible Δ_i's, the algorithm is valid for S_i.*

Corollary 8.3. *If an updating algorithm is invalid, there exists a single change input stream Δ_i for some i that makes the algorithm invalid for Δ_i.*

Corollary 8.4. *For any algorithm satisfying the condition*

1. *Given a graph G and an input stream Δ, $OUT(G, \Delta)$ is a subgraph of $G \cup \{inserted\ edges\}$.*
2. *The algorithm is not able to traverse all edges or vertices belonging to $OUT(G, \Delta)$ at the same moment.*

there exist a graph G and a finite input stream Δ_i for all i such that $OUT(G_i, \Delta_i) \ne OUT(G(t))$ for any $t_i < t < t_{i+1}$.

In Theorem 5.7, we showed the algorithm is not able to produce any snapshot result for any $t_i \le t < t_{i+1}$. In the updating model, the changes up to time t_i is known, and therefore the algorithm might be able to update the result only for Δ_{i-1} and obtain a snapshot for graph $G(t_i)$.

For a simple example updating algorithm, consider the PageRank algorithm using Jacobi iteration in Section 7. If the change Δ_{i-1} is relatively small in matrix norm, restarting the iteration from the previously computed solution x_{i-1} should converge quickly. As in Section 7, the iteration is a valid algorithm in our model, hence restarting is a valid *updating* algorithm. Algorithms that do not traverse the entire graph exist[19], but computing the convergence criteria in a valid manner is challenging and will be addressed in future work.

9 Updating Connected Components

In this section we will discuss the algorithms to maintain a valid parent-neighbor subgraph of an undirected graph G. The parent-neighbor subgraph serves as a sparse sample of the graph that maintains the connected components and contains a valid breadth-first search forest.

We simply summarize the algorithm in this section. Our algorithm is a variation of the initial algorithm from [13] as presented in Section 6. In the worst case, the algorithm may essentially recompute a parent-neighbor subgraph, but that should be rare[13]. The algorithm process insertions and deletions contained in Δ_i separately.

Case 1. $e = (s, d)$ is an insertion. There are two types of insertions:

1. If s and d are in a same component, without loss of generality, assume $Level(s) \le Level(d)$. If d has less than $thresh_{PN}$ parents and neighbors, add s to $PN(d)$ and if $Label(d) < 0$, $Label(d) \longleftarrow -Label(d)$.
2. If s and d are in different components, a parallel BFS on the stored parent-neighbor subgraph starts at the vertex for the smaller of two components. The BFS relabels the smaller component.

Note that the algorithm process all type 1 insertions first, and then type 2. Neither case accesses the graph G. Indeed, updating connected components considering only insertions need never access the graph. All complications come from deletions that may split components.

Case 2. $e = (s, d)$ is a deletion, assume $Level(s) \le Level(d)$. The following process will be repeated from s's perspective as well.

1. Remove s from $PN(d)$ if s is in the list. If d has parents in PN, there is nothing else to do.
2. If d does not have any other parents, let $Level(d) \longleftarrow -Level(d)$ to indicate that d is dependent on its neighbors to other vertices. If d has neighbors with positive levels, then d still have a path to the root through its neighbors. Hence this deletion is safe.
3. If d no longer has a path to its root in PN, start a BFS on induced subgraph $G[\{v : v, d \text{ in a same component}\}]$ to search for a path to the root (in fact any positive level vertex in the component). If such path is not found, take d as the new root and relabel the vertices reached during the BFS.

The second kind of deletion is the only time the updating algorithm accesses G.

We notice that the levels of some vertices becomes outdated after updates, however we claim that the levels do not impact the correctness of the algorithm.

For all insertions inside a component, adding s to $PN(d)$ or adding d to $PN(s)$ does not modify the component. For deletions, the algorithm must maintain a path from affected vertices to the parent-neighbor root. This will not be true if some vertex v is dependent on a positive labeled vertex u where u actually has no path to the root. Assume u, v are the closest pair to the root. We first observe that $|Level(u)| \ne |Level(v)|$, which means u and v are not neighbors. To see this, $Level(u) > 0$ implies that u has a path to the root through some parent p. In this case v should also have a path to root through u and v. Therefore v must rely on u as a parent. Since all vertices p with $Level(p) > Level(u)$ are correct by assumption, that implies u is dependent on some "parent" p' that $Level(p') < Level(u)$. Since we fix the levels of all vertices, such insertions can never happen.

When joining two components or splitting a component, the algorithm recomputes the levels of vertices and the partial PN graph. During the joining process, the levels of the smaller component become consistent with the levels of the larger one. For splitting, the two components become independent and so in both situations vertices that have their decedents as parents cannot appear.

To show the correctness of the algorithm, we generalize PN graph to any subgraph $H \subseteq G$ that contains a spanning forest of G. The following lemmas will show that after updates, there will always be a spanning forest inside the subgraph H, and so the connected components in H and G are identical. In the following discussion, the $Level$ of each vertex can be different from those in Algorithm 3, which approximates the BFS level. However, the $Label$ of each vertex is defined in the same way as in Algorithm 3 according to H and $Level$, that

- If $Label(v) \ge 0$, then v has a path to the root through its parent p ($Level(p) = Level(v) - 1$)
- If $Label(v) < 0$, all v's paths to root rely on v's neighbors u ($Level(u) = Level(v)$)

Lemma 9.1. *At the i^{th} execution, let F be a spanning forest of graph $G_\delta = G(t_{i-1}) \cup \delta$ for some subset of changes $\delta \subseteq \Delta_{i-1}$, and let H be a subgraph of $G(t_{i-1}) \cup \delta$ containing F. We use Δ_I to denote the set of insertions specified in Δ_{i-1}. Assume H_I is the graph obtained by updating Δ_I to H by the algorithm above, then H_I contains a spanning forest of graph $G_\delta \cup \Delta_I$.*

Corollary 9.2. *The connectivity of H_I is the same as the connectivity of $G_\delta \cup \Delta_I$.*

Lemma 9.3. *At the i^{th} execution, let F be a spanning forest of graph $G_\delta = G(t_{i-1}) \cup \delta$ for some subset of changes $\delta \subseteq \Delta_{i-1}$, and let H be a subgraph of $G(t_{i-1}) \cup \delta$ containing F. We use Δ_D to denote the set of deletions specified in Δ_{i-1}. Assume H_D is the graph obtained by updating Δ_D to H by the algorithm above, then H_D contains a spanning forest of graph $G_\delta \cup \Delta_D \cup \delta_i$ for some $\delta_i \subseteq \Delta_i$.*

Corollary 9.4. *Let H_I and other notations be from Lemma 9.1, and let $G_I = G_\delta \cup \Delta_I$. We use Δ_D to denote the set of deletions specified in Δ_{i-1}. Assume H_D is the graph obtained by updating Δ_D to H_I by the algorithm above, then H_D contains a spanning forest of graph $G_I \cup \Delta_D \cup \delta_i = G(t_i) \cup \delta_i$ for some $\delta_i \subseteq \Delta_i$.The connectivity of H_D is the same as the connectivity of $G(t_i) \cup \delta_i$ for some $\delta_i \subseteq \Delta_i$.*

Theorem 9.5. *Let H be the subgraph traversed by BFS during updating deletions. Let $\delta_i \subseteq \Delta_i$ be the set of changes that have both endpoints in H. The algorithm above outputs the connected components for graph $G_i \cup \delta_i$.*

10 Conclusion

Our new *validity model* addresses analysis of graphs undergoing non-stop change. Graphs that stretch the capability of an analysis platform cannot provide point-in-time snapshot views of the entire graph. Analysis results may not include all concurrent changes, but we consider an algorithm *valid* if it always corresponds to the initial graph and some implicit subset of concurrent changes. Updating algorithms that could "catch up" to time T without concurrent changes are considered valid if their results correspond to their input graph at time T and some subset of concurrent changes. These updating algorithms continue producing valid results, enhancing scalability both in size of the graph and number of concurrent analyses.

Additional valid algorithms include direction-optimized breadth-first search (BFS), single-source shortest paths (SSSP), label propagation algorithms, careful triangle counting, and extracting subgraphs.

There are many directions for continuing this work. One immediate concern is quantifying any potential benefits. Meaningful performance comparisons between snapshot views and concurrent analysis require careful construction. Comparisons between the analysis results at any given point in time also are difficult to construct for discrete labelings like connected components. Algorithms like triangle counting appear to require copying *just enough* of the graph for validity. Others like computing betweenness centrality require multiple simultaneous views of the same graph and may require copying *all* of the graph to be valid. The required space trade-offs could provide insights into concurrent data analysis in general. Initial work towards problems with matroid structure suggests these problems may be amenable to concurrent analysis, and this would generalize the model beyond graph analysis.

Acknowledgments

This work partially was supported by NSF Grant ACI-1339745 (XScala) and DARPA award HR0011-13-2-0001. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors. Many referees have offered invaluable suggestions.

References

[1] Aris Anagnostopoulos, Ravi Kumar, Mohammad Mahdian, Eli Upfal, and Fabio Vandin. 2012. Algorithms on evolving graphs. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ACM, 149–160. https://doi.org/10.1145/2090236.2090249

[2] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *Comput. Surveys* 13, 2 (June 1981), 185–221. https://doi.org/10.1145/356842.356846

[3] Laurent Bulteau, Vincent Froese, Konstantin Kutzkov, and Rasmus Pagh. 2016. Triangle Counting in Dynamic Graph Streams. *Algorithmica* 76, 1 (2016), 259–278. https://doi.org/10.1007/s00453-015-0036-4

[4] Gianna M. Del Corso, Antonio Gullí, and Francesco Romani. 2005. Fast PageRank Computation via a Sparse Linear System. *Internet Math.* 2, 3 (2005), 251–273. http://projecteuclid.org/euclid.im/1150474883

[5] Camil Demetrescu, David Eppstein, Zvi Galil, and Giuseppe F. Italiano. 2010. Algorithms and Theory of Computation Handbook. Chapman & Hall/CRC, Chapter Dynamic Graph Algorithms, 9–9.

[6] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. 2012. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems* 23, 2 (Feb. 2012), 375–382. https://doi.org/10.1109/TPDS.2011.159

[7] David Ediger, Karl Jiang, E. Jason Riedy, and David A. Bader. 2010. Massive Streaming Data Analytics: A Case Study with Clustering Coefficients. In *4th Workshop on Multithreaded Architectures and Applications (MTAAP)*. Atlanta, GA. https://doi.org/10.1109/IPDPSW.2010.5470687

[8] David Ediger, Robert McColl, Jason Riedy, and David A. Bader. 2012. STINGER: High Performance Data Structure for Streaming Graphs. In *The IEEE High Performance Extreme Computing Conference (HPEC)*. Waltham, MA. https://doi.org/10.1109/HPEC.2012.6408680 Best paper award.

[9] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. 2005. On graph problems in a semi-streaming model. *Theoretical Computer Science* 348, 2 (2005), 207–216.

[10] David Gleich, Leonid Zhukov, and Pavel Berkhin. 2004. *Fast parallel PageRank: A linear system approach.* Technical Report YRL-2004-038. Yahoo! Research. 22 pages. http://research.yahoo.com/publication/YRL-2004-038.pdf

[11] K. Iwabuchi, S. Sallinen, R. Pearce, B. V. Essen, M. Gokhale, and S. Matsuoka. 2016. Towards a Distributed Large-Scale Dynamic Graph Data Store. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 892–901. https://doi.org/10.1109/IPDPSW.2016.189

[12] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. 2016. Time-evolving graph processing at scale. *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems* (2016). https://doi.org/10.1145/2960414.2960419

[13] R. McColl, O. Green, and D. A. Bader. 2013. A new parallel algorithm for connected components in dynamic graphs. In *20th Annual International Conference on High Performance Computing*. 246–255. https://doi.org/10.1109/HiPC.2013.6799108

[14] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A. Bader. 2014. A Performance Evaluation of Open Source Graph Databases. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications (PPAA '14)*. ACM, New York, NY, USA, 11–18. https://doi.org/10.1145/2567634.2567638

[15] Andrew McGregor. 2014. Graph Stream Algorithms: A Survey. *SIGMOD Rec.* 43, 1 (May 2014), 9–20. https://doi.org/10.1145/2627692.2627694

[16] Paul E. McKenney and Jack Slingwine. 1998. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *10th IASTED International Conference on Parallel and Distributed Computing and Systems*. IEEE Computer Society.

[17] Steven J. Plimpton and Tim Shead. 2014. Streaming data analytics via message passing with application to graph algorithms. *J. Parallel and Distrib. Comput.* 74, 8 (Aug. 2014), 2687–2698. https://doi.org/10.1016/j.jpdc.2014.04.001

[18] E. Jason Riedy and David A. Bader. 2013. Multithreaded Community Monitoring for Massive Streaming Graph Data. In *7th Workshop on Multithreaded Architectures and Applications (MTAAP)*. Boston, MA. https://doi.org/10.1109/IPDPSW.2013.229

[19] Jason Riedy. 2016. Updating PageRank for Streaming Graphs. In *Graph Algorithms Building Blocks (GABB 2016)*. Chicago, IL.

[20] Twitter. 2013. (Aug. 2013). https://blog.twitter.com/2013/new-tweets-per-second-record-and-how, retrieved in Aug 2017.

[21] Matthias Vallentin, Vern Paxson, and Robin Sommer. 2016. VAST: A Unified Platform for Interactive Network Forensics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 345–362.

[22] Charith Wickramaarachchi, Alok Kumbhare, Marc Frincu, Charalampos Chelmis, and Viktor K. Prasanna. 2015. Real-Time Analytics for Fast Evolving Social Graphs. *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (May 2015). https://doi.org/10.1109/ccgrid.2015.162

[23] Zephoria. 2017. (Aug. 2017). https://zephoria.com/top-15-valuable-facebook-statistics/, retrieved on 23 Aug 2017.