# Single-Source Shortest Path Tree for Big Dynamic Graphs

Sara Riazi
*University of Oregon*
*Eugene, OR 97403, USA*
*Email: riazi@uoregon.edu*

Sriram Srinivasan
*University of Nebraska*
*Omaha, NE 68106, USA*
*Email: sriramsrinivas@unomaha.edu*

Sajal K. Das
*Missouri University of Science and Technology*
*Rolla, MO 65409, USA*
*Email: sdas@mst.edu*

Sanjukta Bhowmick
*University of North Texas*
*Denton, TX 76203, USA*
*Email: sanjukta.bhowmick@unt.edu*

Boyana Norris
*University of Oregon*
*Eugene, OR 97403, USA*
*Email: norris@cs.uoregon.edu*

***Abstract*—Computing single-source shortest paths (SSSP) is one of the fundamental problems in graph theory. There are many applications of SSSP including finding routes in GPS systems and finding high centrality vertices for effective vaccination. In this paper, we focus on calculating SSSP on big dynamic graphs, which change with time. We propose a novel distributed computing approach, SSSPIncJoint, to update SSSP on big dynamic graphs using GraphX. Our approach considerably speeds up the recomputation of the SSSP tree by reducing the number of map-reduce operations required for implementing SSSP in the gather-apply-scatter programming model used by GraphX.**

***Keywords*-Single-Source Shortest Path (SSSP), Map-Reduce, Apache Spark, Big Dynamic Graphs**

## I. Introduction

Discovering the single-source shortest path (SSSP) tree is a classical graph theory problem with many real-world applications such as finding routes in maps and social network analysis. However, many graphs evolve over time, which necessitates the recomputation of the SSSP tree. For very large dynamic graphs, this recomputation requires significant resources and is time consuming, thus motivating the development of new algorithms that can quickly recover the updated SSSP tree without recomputing it from scratch.

This problem is exacerbated when graphs are so large that they do not fit in the memory of a single machine. In these cases, the graphs are analyzed using scalable parallel approaches that can use distributed memory and out-of-core processing. An example of such distributed memory software is GraphX [7], which has been developed on top of Apache Spark. GraphX enjoys the fault-tolerance and distributed computing provided by the data-parallel environment of Apache Spark. Apache Spark supports map-reduce (MR) operations over immutable distributed data structures called resilient distributed dataset (RDD) [19], which requires the algorithms such as SSSP to be defined as a set of (expensive) MR operations over RDD representation of graphs. However, GraphX does not come with built-in support for dynamic graphs; instead, we can apply batch updates by mapping the current snapshot of a graph to the next snapshot.

We can re-execute the SSSP algorithm on the new snapshot to obtain the new SSSP tree. However, reusing the computation from the previous snapshot may save significant execution time, especially for very large graphs. Reducing the execution time is even more critical when expensive computing services are being used for parallel processing.

In this paper, we explore an algorithmic approach toward reusing the computation from previous snapshot in order to compute the SSSP tree for the current snapshot. Specifically, we introduce SSSPIncJoint, a new parallel incremental SSSP algorithm, which recovers the SSSP tree over a series of graph snapshots that represent a dynamic graph. Our **key contribution** is this new algorithm, which reduces high-overhead data-parallel operations by tracking the changes among snapshots that affect the SSSP tree.

We experimentally show that SSSPIncJoint is more

efficient (up to 2.2x speedup) than recomputing the SSSP for every snapshot of large dynamic graphs.

## II. Static SSSP on Spark

To enable computations on large graphs that do not fit in a single machine's memory, GraphX provides a vertex-centric gather-apply-scatter (GAS) distributed-memory parallel programming model (first introduced by Pregel [12]). In a GAS model, an algorithm is developed from a vertex point of view, and in general includes three different steps: (i) gathering messages from its neighboring vertices, (ii) updating its state, and (iii) generating messages for its neighbors. GraphX iteratively executes these steps, and each iteration of these steps is called a *superstep*. GraphX stores a graph as two RDDs, one for edges and another for vertices. It also provides *triplets* view as a joint representation of an edge attribute and the attributes on its incident vertices. As it provided by the name view, the triplets are dynamically constructed by shipping vertex attributes to the computation nodes where the corresponding edge partitions are located. This makes MR operations on triplets more expensive than MR operations on edge or vertex RDDs.

Each superstep of a GAS model can viewed as a set of MR operations over the triplet, edge and vertex RDDs. To gather the messages for each vertex, each triplet is mapped to messages using a *sendMessage* function that has access to edge and vertex attributes of its source and destination vertices, and then a *reduceMessage* function combines the messages to generate an RDD containing pairs of vertex ID and message data. To apply the messages, a new vertex RDD for the vertices that received any message is constructed by joining the existing vertex RDD and the new message RDD, and then the old vertex attributes and the message data are mapped to new vertex attributes using a *vertexProgram* function. Finally the graph's vertex RDD is updated by joining the new vertex RDD with the existing one to make sure that the vertex partitioning remains the same, otherwise, constructing the triplet view becomes very expensive for the next round.

## III. Dynamic SSSP on Spark

Dynamic graphs can be viewed as a series of graph snapshots that evolve over time, where each snapshot is constructed by applying an update batch to its predecessor snapshot. In our setting, we assume that the update batches are queued until the computation on the current snapshot is completed. GraphX does not have built-in support for dynamic graphs since it depends on immutable RDDs for graph representation.[1]

An updated graph can be constructed by mapping the old edge RDD to the new one to reflect the new changes (edge insertion and deletion) and constructing a new graph using the new edge RDDs. A simple approach for computing SSSP over dynamic graphs is to re-run SSSP for each snapshot separately. However, the main goal is to expedite the repetitious computation on dynamic graphs by reusing the state of vertices in the current snapshot as much as possible, so we have to transfer the old vertex attributes to the new graph using the *join* operations over RDDs.

Reusing computation for GAS is introduced by GraphInc [4], which memoizes received messages and vertex states from all supersteps. In each superstep, a vertex participates in GAS if its current state is different from the memoized state for the same superstep on the previous snapshot. A vertex runs the vertexProgram using the received messages and also using the memoized messages from its neighbors that have participated in the same superstep of the previous snapshot, but not in the current snapshot. Therefore, GraphInc runs SSSP for the new snapshot for the same number of supersteps, but with fewer messages in each superstep as shown in Figure1.c.

A naive implementation of GraphInc on top of GraphX suffers from **two problems**: **first**, it does not reduce the number of supersteps, which are executed using expensive join operations over large RDDs, and **second**, in an MR framework such as GraphX, we have to store the memoized information as the vertex attributes and frequently ship them across different computation nodes (workers in Spark), which makes memoization impractical for large networks, especially for the social networks with power-law degree distributions. In order to scale to large social networks, the size of vertex attributes must not depend on the degree of vertices, which motivates using fixed-size attributes such as tuples. An example of variable-size attributes

[1] IndexedRDD [1] was introduced to expedite modifying a graph, however, it is not officially supported by GraphX due to fault-tolerance issues. Therefore, we focused on constructing dynamic graphs merely using the functionality provided by GraphX.

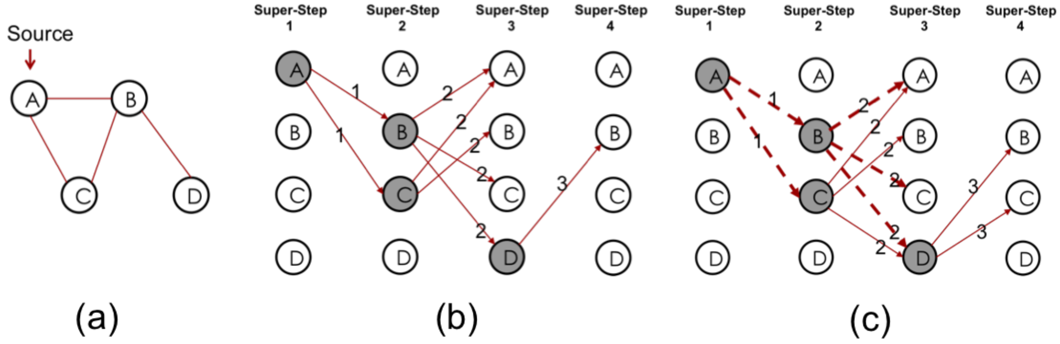


Figure 1. a) The original graph, weights not shown for readability. b) The SSSPBase algorithm based on GAS model. The gray nodes indicate the vertices that participate in a superstep. The red arrows is the messages labeled with shortest paths. c) The GraphInc execution after adding an edge between vertices C and D. The dotted edges shows the memoized messages that have been saved.

would be if each vertex keeps the distance to source of all its neighbors. We only store the distance to source, the parent of each vertex, and an extra flag for capturing the affected vertices due to a batch update.

In contrast to GraphInc, our memoized state does not provide enough information to recompute the state of vertices, thus requiring message propagation to take place. However, we can limit the number of required messages by considering the details of the SSSP algorithm.

An update batch includes a set of edge insertions and deletions.[2] Inserting or deleting edges directly affects the target vertices of the edges (immediate affected) or indirectly affects the descendants of the target vertices (causal affected). We call a vertex *insert-affected* or *delete-affected* if it is affected (immediate or causal) by edge insertion or deletion, respectively.

If the update batch only includes edge insertions, the states of affected vertices (both immediate or causal) converge to the correct states if we continue running the SSSP algorithm. This happens because edge insertion can only shorten the distance of a vertex to the source, and a vertex generates a message for its neighbor only if it can reduce the distance-to-source (DTS) of the target vertex, otherwise the vertex does not participate in the superstep. Therefore, the neighbors of insert-affected vertices will participate in the message passing in order to adjust the state of the insert-affected vertices. This update propagates to adjust the DTS of all insert-affected vertices.

[2]For simplicity, we only discuss edge insertion and deletion, but the same reasoning applies for weight decrease and increase.

The situation for edge deletion is more complicated because deleting an edge may increase the DTS of affected vertices, and in turn, the neighbors of delete-affected vertices may not participate in message passing because the DTS of delete-affected vertices is at least as large as their DTS before the edge deletion happening.

If an update batch contains any edge deletion, the SSSP algorithm may not correct the delete-affected vertices, so we have to mark or invalidate them to make sure that we can correct their states using the SSSP algorithm. This marking phase (invalidation) starts with the immediate delete-affected vertices and propagates to their descendants in the SSSP tree using the GAS model. Therefore, in each superstep of the invalidation phase, each marked vertex generates messages for its children in the SSSP tree. If a vertex receives a message, it changes its status to marked. After convergence, all the delete-affected vertices are marked.

By setting the state of the marked vertices to $\infty$, we can make sure the SSSP does converge to the correct values. Therefore, after the invalidation phase, we can rely on the SSSP algorithm as a correction phase to adjust the state of all affected vertices (insert-affected and delete-affected).

These two steps (invalidation and correction) comprise our vanilla SSSPInc Algorithm 1, which exactly computes the SSSP tree for dynamic graphs. However, the invalidation phase is also expensive since it requires join operations over large RDDs to propagate the marks to all delete-affected vertices, and experimentally we observe that the invalidation phase may take as long as

the correction phase (that considers all delete-affected and insert-affected vertices). Therefore, we introduce two variations of the basic SSSPInc: SSSPIncApprox and SSSPIncJoint in order to reduce the required time for recomputing SSSP for large dynamic graphs.

**Algorithm 1** High-level SSSPInc

```
Run SSSP on the primary graph.
for each update batch do
    Invalidate all delete-affected vertices.
    Apply the update batch.
    Adjust the state of vertices.
```

**Algorithm 2** High-level SSSPIncApprox

```
Run SSSP on the primary graph.
for each update batch do
    Invalidate the immediate delete-affected vertices.
    Apply the update batch.
    Adjust the state of vertices.
```

## IV. SSSPIncJoint

The invalidation propagation phase of SSSPInc is extremely expensive because it requires multiple MR and join operations over very large RDDs in order to pass few messages (with respect to the size of the graph). The number of required supersteps for the invalidation phase depends on the position of immediate delete-affected vertices in the SSSP tree. Therefore, SSSPInc may have even more supersteps than running the SSSP algorithm from scratch on the current snapshot.

To expedite the message propagation for the invalidation phase, one can prune the graph to only the SSSP tree, which significantly reduces the size of the edge RDD. But we should also note that pruning requires an MR operation over triplets. The overhead cost of pruning may be amortized over message propagation steps, but in our setting we didn't find it useful.

An alternative approach is to ignore the incorrect state of causal delete-affected vertices. In that case, the DTS of the causal delete-affected vertices is only an approximation of the true value. We call this approach SSSPIncApprox, and is described in Algorithm 2.

To achieve the same efficiency as SSSPIncApprox, but with more accurate DTS values, we try to run the invalidation and correction phase jointly. Although the joint execution may result in inexact values, we can guarantee that if it converges, it would be to the exact values. We revisit the convergence assumption after describing the algorithm.

To jointly execute the invalidation and correction, we must make sure that the correction does not truncate the invalidation phase. We first mark all the immediate delete-affected vertices. In each superstep, a marked vertex sends marking messages to its children in the current SSSP tree. To make sure that a delete-affected vertex at least remains marked for one superstep, the neighbors of a marked vertex do not send any DTS value for the marked vertex. Therefore, if a vertex is marked it can propagate the mark to its children in one superstep. After propagating the mark, the vertex clears itself and sets its DTS value to $\infty$, then removes its parent in the tree. This happens in the vertexProgram. Therefore, in the next supersteps, its neighbors start sending their DTS to the already cleared vertex. To avoid loops, a vertex never sends DTS to its current parent in the SSSP tree, however, longer cycles are still possible but less likely. Algorithm 3 shows the GAS model for SSSPIncJoint.

**Proposition:** SSSPIncJoint converges to the exact single-source shortest path value or never converges.

Proof. Suppose that the edge $e_{uv}$ is removed and also suppose that there exists an edge $e_{yv}$ such that $y$ belongs to the subtree rooted at $v$. Based on these assumptions, there exists a cycle including $v$ and $y$. Let $z$ be any vertex in this cycle, including $u$ and $v$, with an edge $e_{xz}$ such that $x$ belongs to the subtree rooted at the source of the original SSSP tree. Note that if the latter condition is not met, the graph is not strongly connected after removing edge $e_{uv}$. In SSSPIncJoint, $v$ is marked, and $y$ sends $y.distance + e_{yv}.weight$ to $v$ and becomes the parent of $v$. However, $y$ is also a descendant of $v$, so it will receive the mark token and a new DTS value from its parent based on the DTS of vertex $v$, and since node $y$ is the parent of $v$, it passes the mark token to $v$ and the new DTS value. This cycle monotonically increases the DTS values of vertices in the cycle. Therefore, eventually $x.distance + e_{xz}.weight < z.distance$, so $z$ changes its parent to $x$ and breaks the cycle. And after another round of message passing in the cycle, all DTS values become exact. If there is no such vertex $x$

(i.e. the graph is not strongly connected after removing the edges), then DTS values of vertices in the cycle increase infinitely, and the algorithm never converges. □

**Algorithm 3** SSSPIncJoint

```
//s: Source vertex for the SSSP algorithm
//e_uv : edge from u to v.
//msg: (source, distance, mark)
//vertex attributes: (isMarked, distance, parent)
//u → v : msg means u generates msg for v
procedure SENDMESSASGE(e_uv)
    if u.isMarked then
        if v.isMarked then
            No message
        else if v.parent = u then //v is a child of u
            u → v: (u, ∞, true)
        else
            No message
    else if v.isMarked then
        if u.parent ≠ v then //v is not the parent of u
            u → v: (u, e_uv.weight + u.distance, false)
        else
            No message
    else if e_uv.weight + u.distance < v.distance then
        u → v: (u, e_uv.weight + u.distance, false)
    else
        No message
procedure MERGEMESSAGES(a, b)
    mark ← a.mark or b.mark
    if a.distance < b.distance then
        (a.source, a.distance, mark)
    else
        (b.source, b.distance, mark)
procedure VERTEXPROGRAM(u, msg)
    if u = s then
        (false, 0.0, s)
    else
        if msg.mark then
            (true, ∞, ∞)
        else if u.distance > msg.distance then
            (false, msg.distance, msg.source)
        else
            (false, u.distance, u.source)
```

## V. Experiments

We evaluate the performance of SSSPInc, SSSPIncApprox, and SSSPIncJoint on three very large real-world social network graphs: Friendster, Twitter-MPI, and Twitter[3]. We also run our experiments on a very large syntactic random graph generated by R-MAT: with parameters: a=0.55, b=0.15, c=0.15, d=0.15. Table I shows the characteristics of these datasets.

We assume that a primary graph and an update batch in the form of edge events (insert or delete) are given as input. To construct a primary graph and update batch from a static graph, we randomly select an $\alpha$ fraction of edges of the static graph without replacement. $\beta$ percent of events are edge deletion, and the rest are edge insertion. A primary graph is formed by removing the edges corresponding to insertion events from the static graph. The number of edge insertions and deletions for each update batch is shown in Table II.

Inserting an edge may introduce a new vertex if the source or destination vertices are not in the graph. Therefore, we remove standalone vertices appearing as a result of edge removal from the static graph; they will be added to the graph as new vertices when we add the edges back.

Table I
VERTICES AND EDGES OF THE REAL-WORLD AND SYNTHETIC GRAPHS IN OUR TEST SUITE.

| Name | Num. of Vertices | Num. of Edges | Type |
|---|---|---|---|
| RMAT | 339,201,984 | 4,252,445,904 | Directed |
| Friendster | 68,349,466 | 2,586,147,869 | Directed |
| Twitter-MPI | 52,579,682 | 1,963,263,821 | Directed |
| Twitter | 41,637,597 | 1,453,833,084 | Directed |

The baseline is to re-run the SSSP algorithm for each snapshot without considering the dynamic nature of the graph. We call this method SSSPBase.

We use the vertex with the highest degree as the source for the SSSP algorithm. All algorithms are implemented using the GraphX library of Apache Spark v. 2.3. For GraphX, we use ten Spark workers on a cluster with ten dual Intel Xeon E5-2690v4 processors. Each worker has access to 20 cores (for a total of 200 cores) and 120GB of memory (total 1.2TB memory).

We do not use GraphInc in our comparison because by using the suggested memoization, we have to store all messages in attributes of vertices. This would dramatically increase the size of vertex attributes, making shipping the vertices to the computation nodes very

[3]These graphs are the three largest graphs available on the Konnect graph repository: http://konect.uni-koblenz.de/networks/

Table II
THE CHARACTERISTICS OF UPDATE BATCHES FOR DIFFERENT GRAPHS.

| | $\alpha = 0.1\%, \beta = 1\%$ | | $\alpha = 0.1\%, \beta = 10\%$ | | $\alpha = 1\%, \beta = 1\%$ | |
|---|---|---|---|---|---|---|
| | Insert | Delete | Insert | Delete | Insert | Delete |
| RMAT | 4,250,418 | 42,647 | 3,867,390 | 429,833 | 42,521,392 | 429,474 |
| Friendster | 2,559,344 | 25,949 | 2,327,102 | 258,373 | 25,592,403 | 258,992 |
| Twitter-MPI | 1,941,750 | 19,856 | 1,764,937 | 196,681 | 19,444,189 | 196,481 |
| Twitter | 1,453,304 | 14,796 | 1,321,018 | 146,888 | 14,532,098 | 147,270 |

costly. Moreover, the number of messages depends on the degree of vertices, thus for social network graphs with power-law degree distributions, some of vertices have to store prohibitively large number of messages.

### A. *Results and Discussion*

We report the execution time of SSSPBase, SSSPInc, SSSPIncApprox, and SSSPIncJoint for our three different update batches in Figure 2. The execution time depends on the number of supersteps, as well as the size of the graph (number of edges and vertices), which determines execution time of each superstep. Figure 3 shows the number of supersteps of different algorithms for batch $\alpha = 0.1\%, \beta = 1\%$. Comparing to the same execution time for the same batch in Figure 2, we conclude that the ranking of algorithms with respect to the number supersteps is often the same as their ranking with respect to the execution time. The differences are explainable by the execution time of each superstep, which also depends on the number of active vertices participating in the message passing.

In general, SSSPInc is often slower than SSSPBase, and the difference is significant when we increase the number of edge deletions as in the batch $\alpha = 0.1\%, \beta = 10\%$. This happens because the invalidation phase is expensive since it needs to run several supersteps. We also show the number of supersteps required for the invalidation and correction phases, as well as the execution time for each phase in Figure 4. The reported numbers are for batch $\alpha = 0.1\% \beta = 1\%$. The execution time of invalidation phase is considerable comparing to the execution time of the correction phase.

SSSPIncApprox, which only has one step of invalidation (for immediate delete-affected vertices), is always better than SSSPInc by saving multiple supersteps of invalidation phase. SSPIncApprox is also always better than SSSPBase. The one-step invalidation of SSSPIncApprox has not been included in the number of supersteps required for SSSPIncApprox. We notice, from the execution of SSSPInc, that the number of invalidated vertices is negligible compared to the number of the vertices in the graph (less than $0.001\%$ of vertices), which indicates that the accuracy of shortest-path distance values found by SSSPIncApprox is above 99.9% comparing to the exact SSSP.

SSSPIncJoint is always better or equivalent to SSSPIncApprox and is always better than SSSPBase and SSSPInc. SSSPIncJoint and SSSPIncApprox often share the same number of supersteps, which suggests that SSSPIncJoint successfully combines the correction and invalidation phases.

As we mentioned earlier, SSSPIncJoint may not converge if deleting the edges partitions the graph into disconnected components, but SSSPIncJoint in all of the experiments converges and finds the exact DTS for all the vertices comparing to our SSSPBase.

Finally, to see how balanced the workload distribution over the workers is, we show the processing time of each worker for batch $\alpha = 0.1\%, \beta = 1\%$ applied to the Friendster graph in Figure 5. We find that the workload is evenly distributed among the workers.

## VI. RELATED WORK

GraphTau [11] proposes a paradigm of pause-shift-resume, in which whenever a new batch of updates is ready, GraphTau pauses the current computation and updates the underlying graph and resume the computation with the previous state of the vertices. GraphTau cannot guarantee the correctness of the computation.

Chronos [8] and ImmortalGraph [13] optimize GAS operations across different snapshots. They suppose accessing to all snapshots in advance and batch the operations for each vertex/edge over different snapshots and run batches in parallel using a locality-aware batch scheduling. In the incremental setting, when given a set

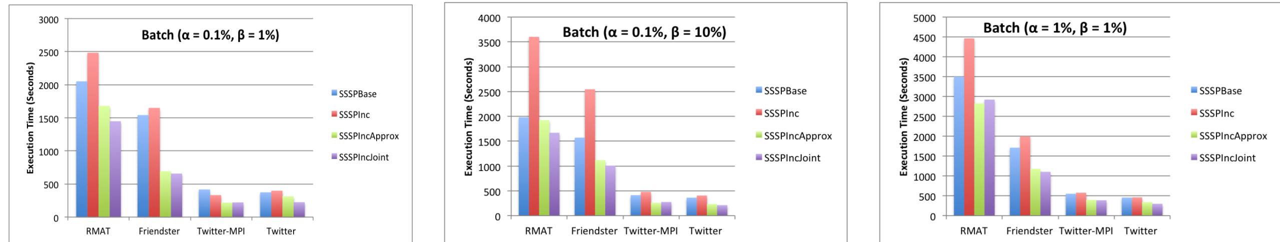


Figure 2. The execution time (in seconds) of SSSPBase, SSSPInc, SSSPIncApprox, and SSSPIncJoint for different update batches.

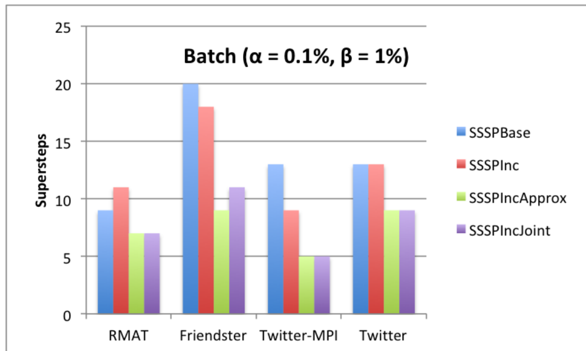


Figure 3. Total number of GAS supersteps for running each algorithm.

of graph snapshots, it processes the first snapshot and batches the other snapshot reusing the computation of the first snapshot.

Similarly, Tegra [10] operates over all snapshots, however, Tegra does not batch all snapshots together but runs every GAS round over all snapshots before continuing with the next round, so save the redundant computation.

BLADYG [2] is a block-centric framework, which partition the graph into blocks and assigns each block to a worker. When a new edge comes, it updates the corresponding block and the corresponding worker may communicate to other workers to propagate the update.

In [5] they use GraphInc which uses memoization and is not scalable for large networks which we use for our experiments. Among other related papers [18] and [6] do not report any experimental scalability results and their code base is not available for comparison.

There are a few implementations of sequential SSSP on dynamic networks such as Ramalingam et al. [15], Narvez et al. [14]. Bauer et al. [3] proposed SSSP algorithm for dynamic networks using batch updates. Vora et al. [17] have proposed an approach that uses approximation while calculating SSSP on streaming graphs. Srinivasan et al. [16] recently proposed an approach for finding SSSP on dynamic networks, however it is based on shared-memory parallelism. Ingole et al. [9] have proposed a GPU implementation of SSSP on dynamic networks.

## VII. Conclusion

We introduce an algorithmic approach to compute the SSSP tree for dynamic graphs on GraphX. Our approach, SSSPIncJoint [4], jointly finds the vertices with incorrect state and corrects their states. SSSPIncJoint is computationally more efficient than computing the SSSP from scratch and also more efficient than two-phase approaches that complete finding the vertices with incorrect states before start correcting their values.

## Acknowledgment

Sanjukta Bhowmick and Sriram Srinivasan are supported by the NSF CCF Award #1533881 and #1725566. Boyana Norris and Sara Riazi are supported by the NSF CCF Award #1725585. Sajal Das is supported by the NSF CCF Awards #1533918 and #1725755.

## References

[1] IndexedRDD for Apache Spark. https://github.com/amplab/spark-indexedrdd.

[2] Sabeur Aridhi, Alberto Montresor, and Yannis Velegrakis. BLADYG: A graph processing framework for large dynamic graphs. *Big Data Research*, 9:9–17, September 2017. arXiv: 1701.00546.

[3] Reinhard Bauer and Dorothea Wagner. Batch dynamic single-source shortest-path algorithms: An experimental study. In Jan Vahrenhold, editor, *Experimental Algorithms*, pages 51–62, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[4] Source code and instructions to reproduce our scalability results are available on https://github.com/DynamicSSSP/SSSPIncJoint-

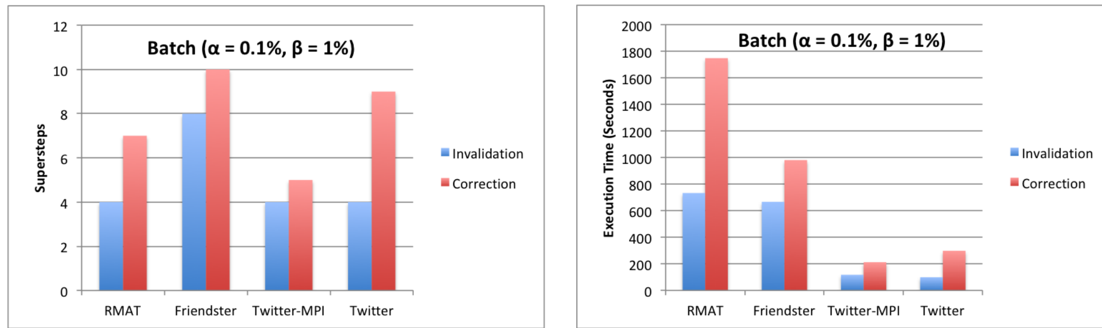


Figure 4. Total number GAS supersteps (left) and execution time (right) for invalidation and correction phase in SSSPInc.

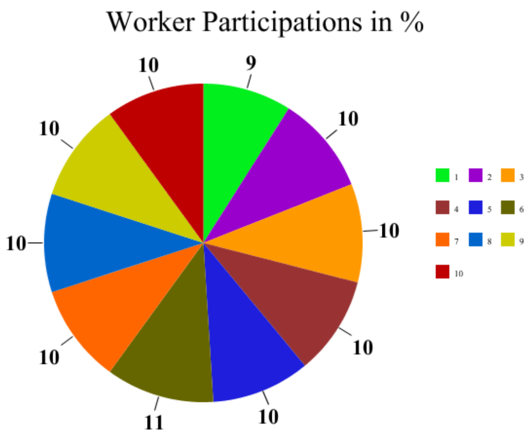


Figure 5. Apache Spark workers participation in SSSPInc for update batch $\alpha = 0.1\%, \beta = 0.1\%$ for Friendster graph.


[4] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. Facilitating real-time graph mining. In *CloudDb*, 2012.

[5] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. Facilitating real-time graph mining. pages 1–8. ACM Press, 2012.

[6] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, Jiaxin Jiang, Zeyu Zheng, Bohan Zhang, Yang Cao, and Chao Tian. Parallelizing Sequential Graph Computations. pages 495–510. ACM Press, 2017.

[7] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI 14, pages 599–613, Broomfield, CO, USA, 2014.

[8] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, page 1. ACM, 2014.

[9] A. Ingole and R. Nasre. Dynamic shortest paths using javascript on gpus. 2015.

[10] Anand Padmanabha Iyer. Time-evolving graph processing on commodity clusters, 2017.

[11] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-evolving graph processing at scale. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems - GRADES '16*, pages 1–6, Redwood Shores, California, 2016. ACM Press.

[12] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[13] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. Immortalgraph: A system for storage and analysis of temporal graphs. *ACM Transactions on Storage (TOS)*, 11(3):14, 2015.

[14] P. Narvaez, Kai-Yeung Siu, and Hong-Yi Tzeng. New dynamic algorithms for shortest path tree computation. *IEEE/ACM Transactions on Networking*, 8(6):734–746, December 2000.

[15] Ganesan Ramalingam and Thomas Reps. On the computational complexity of dynamic graph prob-

lems. *Theoretical Computer Science*, 158(1-2):233–277, 1996.

[16] Sriram Srinivasan, Sara Riazi, Boyana Norris, Sajal Das, and Sanjukta Bhowmick. A shared-memory algorithm for updating single-source shortest paths in large weighted dynamic networks. In *Proceedings of the 25th IEEE International Conference on. High Performance Computing, Data, and Analytics (HIPC)*, November 2018.

[17] Keval Vora, Rajiv Gupta, and Guoqing Xu. Kick-Starter: Fast and accurate computations on streaming graphs via trimmed approximations. pages 237–251. ACM Press, 2017.

[18] Charith Wickramaarachchi, Charalampos Chelmis, and Viktor K. Prasanna. Empowering fast incremental computation over large scale dynamic graphs. pages 1166–1171. IEEE, May 2015.

[19] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012.