Incremental Algorithm for Updating Betweenness Centrality in Dynamically Growing Networks

Miray Kas, Matthew Wachs, Kathleen M. Carley, L. Richard Carley
Carnegie Mellon University
Pittsburgh, PA, USA
{miraykas@gmail.com, misc@mwachs.com, Kathleen.carley@cs.cmu.edu, carley@ece.cmu.edu}

Abstract— **The increasing availability of dynamically growing digital data that can be used for extracting social networks has led to an upsurge of interest in the analysis of dynamic social networks. One key aspect of social network analysis is to understand the central nodes in a network. However, dynamic calculation of centrality values for rapidly growing networks might be unfeasibly expensive, especially if it involves recalculation from scratch for each time period. This paper proposes an *incremental algorithm* that effectively updates betweenness centralities of nodes in dynamic social networks while avoiding re-computations by exploiting information from earlier computations. Our performance results suggest that our incremental betweenness algorithm can achieve substantial performance speedup, on the order of thousands of times, over the state of the art, including the best-performing non-incremental betweenness algorithm and a recently proposed betweenness update algorithm.**

Keywords — Betweenness Centrality, Incremental Algorithms, Dynamic Networks, All-Pairs Shortest Paths.

I. INTRODUCTION

For decades, social network analysis has been an important tool for solving a number of problems such as revealing patterns of information dissemination, assessing the impact of business decisions in organizational structures, and identifying influential actors in social networks.

There are a large number of algorithms in network science that seek to identify the most prominent nodes and relevant characteristics of nodes. These can be roughly classed into the following categories based on the underlying key calculation: counting local edges, calculating shortest paths, counting two mode connections, correlation, etc. Of these, the shortest path and correlation measures are the most costly to calculate, and that cost increases dramatically when there is temporal variation in what nodes and edges are present. To address this problem we examine betweenness, which is the canonical and most widely used shortest path based metric, while recognizing that the basic steps here can be generalized to the other shortest path based metrics. We propose an incremental algorithm, which reduces the calculation costs for shortest path based metrics, and breaks up the shortest path calculation into steps so that the entire calculation can then be done in a distributed architecture.

The betweenness centrality of a node x is defined as the fraction of the shortest paths that pass through x across all pairs of nodes in a network. Traditional techniques used for computing betweenness centrality involve solving the well-studied all-pairs shortest paths problem. The all-pairs shortest paths problem has complexity on the order of $O(nm + n^2 \log n)$ when it is computed by invoking a single-source shortest path computation using each social actor (node) as a source, where m denotes the number of social ties (edges), and n denotes the number of social actors (nodes). When an all-pairs shortest path algorithm such as the Floyd-Warshall algorithm [1] is used the complexity increases to $O(n^3)$.

The initial design point for all of these centrality metrics, including betweenness, was static snapshots of small networks (e.g. 20-30 nodes) [2] and the limiting algorithmic complexities and computation times of centrality measures were not a significant problem for such small, static networks. However, restricting the representation of social networks to static snapshots results in substantial information loss, especially when the dynamism of social relationships is of interest in the research.

Increasingly, network data is available through sensors and on-line resources from networks where the participants may be changing and/or the level of participation is changing. Examples include SMS networks, Twitter networks, and inter-organizational alliance networks.

Dynamic network analysis (DNA) is useful for analyzing social networks that evolve over time and serves as a response to the concerns about the limitations of analyses performed on static snapshots. However, keeping the costly-to-compute centrality metrics up to date in dynamic networks that rapidly change/grow becomes computationally very expensive. This is because many important centrality metrics such as betweenness require re-solving the all-pairs shortest path problem with every update/change made to the network. Expensive computation times inhibit many social network researchers from analyzing over-time variations of centrality values on time-variant networks.

To facilitate solutions to costly problems on continually changing networks, *incremental algorithms* have been commonly used. An incremental algorithm is an algorithm that updates the solution to a problem after an incremental change is made on its input [3]. Incremental algorithms arrive at solutions for computationally complex problems in an efficient manner without recomputing everything from scratch by preserving significant information from prior computations. This paper extends existing incremental algorithms for solving the all pairs shortest path problem [4] to the case of incrementally computing betweenness centrality of nodes for each time snapshot of a dynamically updated, growing network. This requires significant additional computation and memory over the incremental all pairs

shortest path algorithms. We present results that indicate how the proposed incremental algorithm for betweenness centrality will scale with realistic social network data sets.

We broadly classify network updates that evolve a network into two categories: (*i*) growing network updates, and (*ii*) shrinking network updates. The first group of updates, the growing network updates, includes (*i*) inserting a new node, (*ii*) inserting a new edge, and (*iii*) decreasing the cost of an existing edge. We call them 'growing network updates' because they are usually observed due to new actors/agents joining the network or more/new interactions.

Growing network updates can be handled by a single incremental algorithm. Insertion of a new node with no edges (i.e. an isolated node) has no effect on the shortest paths in the network; therefore, no further action would be required to complete the update. Insertion of a new node with one or more edges is equivalent to inserting one or more edges to or from the new node. Therefore, an algorithm designed to handle inserting new edges into a network can also handle inserting new nodes to the network. Inserting a new edge can be represented as a special case of the network update that decreases the cost of an edge. Because, inserting a new edge corresponds to decreasing the cost of an edge from infinity to a real, positive value in the adjacency matrix. Hence, as mentioned, a single incremental algorithm will be sufficient to cover all three sub-types of growing network updates.

Many real-life dynamic networks that can be obtained online or by other digital means evolve only by growing network updates, and do not exhibit shrinking behavior. Thus, even with only support for growing network updates, a significant number of real life networks can be studied. For instance, consider a network of co-authorship. As researchers continue to publish, new nodes and edges are added to the network where nodes represent the authors and the edges represent coauthorship, and once the paper is published, the edge is expected to remain permanently. Hence, handling growing network updates is important and we primarily focus on this class of updates.

II. Computation of Betweenness Centrality

A. Notation

A directed network G consists of a set of nodes $V(G)$ and edges $E(G)$ where n is the number of nodes, and m is the number of edges in the network. $x \rightarrow y \in E(G)$ represents an edge directed from node x to node y, where $x \in V(G)$ is a predecessor of y, and $y \in V(G)$ is a successor of x. $Pred(x)$ is used to denote all predecessors of x in the network. $P_x(y)$ denotes the set of predecessors of node y on the shortest paths from node x. $\tilde{G}$ is the transpose (reverse) of network G where all edges in network G are reversed in direction. The set of edges, nodes, and edge costs are also defined for network $\tilde{G}$.

In weighted networks, each edge e in the network has a traversal cost of $C(e)$ where $C(x \rightarrow y) > 0$ for $x \rightarrow y \in E(G)$. The length of a path *Path* is the sum of the costs of the edges on *Path*. The distance from node x to y is the length of the minimum-length path from x to y that is also called the shortest path. $D(x, y)$ denotes the shortest distance while $\sigma(x, y)$ denotes the number of shortest paths from node x to y. The vector B holds the betweenness centrality value of each node. Finally, $SP(x, y, z)$ is true if the edge $x \rightarrow y \in E(G)$ is on a shortest path from x to z, satisfying the two conditions: (*i*) there is a path from x to z (i.e. the distance from x to z is $D(x, z) \neq \infty$) and (*ii*) $D(x, z) = C(x, y) + D(y, z)$. SP is false otherwise [4].

The algorithms presented in this paper are designed to handle weighted, directional, dynamic networks with positive edge weights/costs. Undirected networks can be represented as directed networks where the edge $\{x - y\}$ is represented using two directed edges $\{x \rightarrow y\}$ and $\{y \rightarrow x\}$. Binary networks can also be represented as weighted networks where existing edges' weights/costs are always equal to 1.

B. Overview of Betweenness Centrality

Betweenness centrality of a node i is defined as the fraction of shortest paths that pass through node i across all pairs of nodes. Let $\sigma_{(j,k)}$ be the number of shortest paths from j to k and $\sigma_{(j,k)}(i)$ be the number of shortest paths from j to k that contain node i.

$$B(i) = \sum_{\substack{j \in V(G) \\ k \in V(G)}} \frac{\sigma_{(j,k)}(i)}{\sigma_{(j,k)}} \; where \; i \neq k, i \neq j, \; j \neq k$$

C. State of the Art

Herein we focus on betweenness centrality, as it is one of the most commonly used metrics in the field of social network analysis. The original argument for an algorithm for calculating betweenness was introduced by Freeman [2]. Currently, the majority of the implementations for betweenness centrality use Brandes' algorithm or a variant of it [5] which yields $O(nm + n^2 \log n)$ performance for a weighted network where n is the number of nodes in the network and m is the number of edges. The Brandes' algorithm exploits the sparsity of real life networks to avoid some of the superfluous work done in $\Theta(n^3)$ algorithms, by following an idea similar to that of Dijkstra's algorithm.

D. Algorithm Variants for Computing Betweenness

Many researchers have provided algorithms for variants of betweenness centrality. One set of variants of betweenness centrality focus on incorporating over-time information into the definition of betweenness for dynamically changing graphs (e.g. [6] [7] [8] [9]). In contrast, we do not change or extend the definition of betweenness; we rather focus on faster computation of the original betweenness metric in dynamically growing networks. Another recent study focusing on speeding up the exact computation of betweenness centrality is [10]. The authors use two different heuristics: structural equivalence and partitioning the network into smaller components. Although [10] focuses on speeding up betweenness computation, it targets static networks; and does not maintain betweenness centrality dynamically. Another approach that is closely related to ours is QuBE, which focuses on updating betweenness centralities without computing all-pairs shortest paths in the network [11]. We provide comparisons with the QuBE algorithm later in the results section.

E. Work on Dynamic Shortest Path Computations

We also draw on earlier research on dynamic shortest path computations. Computation of betweenness centrality is

tightly coupled with solving the all-pairs shortest paths problem. In the literature, there are many different techniques proposed for solving the all-pairs shortest paths problem dynamically [4] [12] [13]. However, some of these techniques come with a number of restrictions. For instance, [12] solves the all-pairs shortest paths problem in networks that have positive integer edge costs that are less than a certain number, b, which is inapplicable for networks whose edges are positive real valued numbers. The Demetrescu and Italiano algorithm [13] depends on the notions of locally shortest paths and locally historical paths. The main idea is to maintain dynamically the set of locally historical paths, which is a path that has been identified as a shortest path at some point and has not been modified since then. In this study, we use the dynamic all-pairs shortest path algorithm proposed by Ramalingam and Reps in [4] as our building block to maintain all-pairs shortest paths dynamically.

There are many reasons why we use Ramalingam and Reps algorithm as our basic building block. First, Ramalingam and Reps algorithm is the most commonly used dynamic all-pairs shortest paths algorithm in the literature. Second, it has good performance compared to other dynamic all-pair shortest path algorithms available in the literature considering the experiments presented in [14]. Ramalingam and Reps algorithm performs quite well on sparse, real-life networks and the compute times of Ramalingam and Reps algorithm and Demetrescu and Italiano's algorithm are very close. In experiments done with real life networks (presented in [14]), Ramalingam and Reps have the lowest or one of the lowest compute times among all dynamic all-pairs shortest paths algorithms. Third, Ramalingam and Reps algorithm is shown to scale better as the number of nodes increases because Demetrescu & Italiano's algorithm maintains more global structures and requires more memory while Ramalingam and Reps algorithm requires less space and exhibits better locality in its memory access pattern. Since supporting an increasing number of nodes is important for dynamically growing social networks, we decided to use Ramalingam and Reps algorithm as a building block in our algorithm.

III. INCREMENTAL BETWEENNESS ALGORITHM

A. Background on Incremental Algorithm Design

An incremental algorithm updates the solution to a problem after an incremental change is made on its input [3]. In the application of an incremental algorithm, the initial run is conducted by an algorithm that performs the desired computation from scratch and the incremental algorithm is used in the subsequent runs (*i*) using information from earlier computations and (*ii*) to reflect the update on the network while avoiding re-computations as much as possible.

The computation of betweenness centrality depends on the number of shortest paths in a network and the intermediate nodes on these paths. A network update such as an edge insertion or edge cost decrease might result in creation of new shortest paths in the network. However, a considerable portion of the older paths might remain intact, especially in the unaffected parts of the network. Therefore, accurate maintenance of the number of shortest paths and the predecessors on the shortest paths will suffice for accurately updating betweenness values in the case of dynamic network updates. This is the key observation we make in the design of our incremental betweenness centrality algorithm.

Our incremental betweenness centrality algorithm extends the dynamic, all-pairs shortest path algorithm proposed by Ramalingam and Reps in [4]. The original Ramalingam and Reps algorithm [4] is a dynamic all-pairs shortest-path algorithm which maintains *only* the shortest distances when a network is updated. While the Ramalingam and Reps algorithm can be used as a basic building block for the proposed incremental betweenness centrality algorithm, *several major extensions* are required for detecting the newly formed shortest paths of equal length, maintaining the number of shortest paths, and maintaining the predecessors on these shortest paths. Due to space constraints, we describe only the algorithms we have designed for our incremental betweenness computation. The pseudocodes are provided in the Appendix.

B. Procedures for Incremental Betweenness Centrality

1) InsertEdge Procedure

When there is a network update (e.g. edge insertion or edge cost decrease), the entry point of execution is the INSERTEDGE procedure. In the first phase of the INSERTEDGE procedure, INSERTUPDATE is invoked twice (Lines 3–4 of Algorithm-1) to determine the sets of affected sinks and affected sources, passing once the source, once the destination of the inserted edge as a parameter to it. INSERTUPDATE is then invoked for each affected sink and source node (Lines 5–8) to modify the information required for accurate maintenance of betweenness values. The modified information includes the shortest distances (D), the number of distinct shortest paths (σ), and the predecessors on these shortest paths. After all the shortest distances, predecessors, and the shortest path counts are updated accurately for the affected nodes, betweenness values of the intermediate nodes that lie on the paths between affected source and affected sink nodes are adjusted (Line 9). The first two lines prepare auxiliary data that are only used during the current update, and are not maintained across different updates.

2) InsertUpdate Procedure

The INSERTUPDATE procedure examines the impact of the updated edge $\{src \rightarrow dest\}$ on the network, for each affected sink or affected source node z. The update process continues until there are no edges that were on the shortest paths that would propagate the update further. The INSERTUPDATE procedure consists of three phases:

- If a strictly shorter path is found, the shortest path distance is updated. The predecessors and the number of shortest paths are cleared. Betweenness values for the intermediate nodes on the cleared paths are also reduced opportunistically. (Lines 7–14 of Algorithm-2).
- If the shortest paths have changed in any way (number or length), the predecessors and the number of shortest paths are adjusted accordingly (Lines 15–25 of Algorithm-2).
- Propagation of the update across the network is continued if appropriate (Lines 26–29 of Algorithm-2).

Consider the case where the edge $\{x \rightarrow y\}$ is updated. In the first phase of the algorithm, assume that there is now a path from node x to z passing through the edge $\{x \rightarrow y\}$, which is strictly shorter than the previously known shortest path(s) from x to z (Lines 7–14 of Algorithm-2).

In this first phase of the algorithm (Lines 7–14 of Algorithm-2), since a strictly shorter path from x to z is found, the previously known shortest paths are no longer the shortest. Hence, the number of known shortest paths and the predecessors should be cleared (Line 11 of Algorithm-2). Before we clear the number of shortest paths ($\sigma(x, z)$) and the update distance from x to z ($D(x, z)$) to be equal to the new distance (*alt*), we temporarily record their values in $\sigma_{old}(x, z)$ and $D_{old}(x, z)$ and reduce the betweenness values of the old predecessors because these intermediates do not have any contribution from the (x, z) pair anymore. Attempt to retrieve σ_{old} and D_{old} values returns the temporarily stored values if they exist and returns current σ and D values otherwise.

At the beginning of the first phase, we check if it is the first time a strictly shorter path is found from x to z by checking if σ_{old} contains any information on the pair (x, z) (Line 8 of Algorithm-2). We check σ_{old} because a change in betweenness values is required if the number of shortest paths changes even if the shortest distance does not necessarily change. For every updated pair, we record the original number of shortest paths from x to z known before the update to ensure accurate reduction of betweenness values for the nodes that were on the paths that are not shortest paths anymore.

Since the original Ramalingam and Reps algorithm is concerned with only updating the D values, in their algorithm, the AffectedVertices set only covers the nodes with lower-cost paths to/from z that pass through the modified edge $\{src \rightarrow dest\}$. However, for computing betweenness centrality, we need to maintain the number of shortest paths (σ) and the predecessors (P) accurately as well. Hence, we need to consider the alternative shortest paths of equal length and expand the AffectedVertices to include nodes that have new alternative shortest path(s) to/from node z passing through the originally modified edge $\{src \rightarrow dest\}$.

The second phase of the algorithm (Lines 15–25 of Algorithm-2) checks if the shortest distance from x to z is now equal to the cost of the alternative shortest path passing through the edge $\{x \rightarrow y\}$ (Line 15 of Algorithm-2). If they are equal, then we need to update the number of shortest paths from x to z and the predecessors on these shortest paths.

The entry condition of the second phase (Line 15 of Algorithm-2) is not a condition that is tied to the if block between Lines 7–14. Once the condition in Line 7 ($alt < D(x, z)$) is satisfied (i.e. a strictly shorter path is found), the value of $D(x, z)$ is updated in Line 9 to be the newly found alternative distance *alt*. Therefore, the condition in Line 15 of Algorithm-2 ($alt = D(x, z)$) is satisfied for all cases that originally satisfied the initial check of $alt < D(x, z)$ in Line 7. However, the condition $alt = D(x, z)$ covers additional cases where there are newly formed alternative shortest paths of equal length. Such cases would not satisfy the condition on Line 7 which checks for strictly shorter paths, but would still satisfy the $alt == D(x, z)$ condition in Line 15. This new part of the algorithm, not handled by Ramalingam and Reps, is required for accurate maintenance of betweenness. Finally, in Line 25 of Algorithm-2, we mark node x as affected whose predecessors should be further checked to understand if the update has a wider impact on the network.

When updating the number of shortest paths from x to z, we increase the number of shortest paths only by the number of shortest paths that are newly formed due to the change made in the network. To obtain the number of newly formed shortest paths, the number of shortest paths from x to z that use the modified or inserted edge $\{src \rightarrow dest\}$ should be counted. The number of newly discovered paths is calculated as $\sigma(x, src) * 1 * \sigma(dest, z)$, and then added to the $\sigma(x, z)$ to calculate the total number of shortest paths from x to z. From *src* to *dest*, there may be other shortest paths that might already be counted in. Hence, to avoid double counting, we only consider the modified edge, which is represented with the '1' in the above given formulation (Line 21 of Algorithm-2).

This second phase of the algorithm also updates the predecessors on the shortest paths from x to z due to formation of new shortest paths that pass through the edge $\{x \rightarrow y\}$. Hence, the new shortest paths can be represented in the following form: $x \rightarrow y \rightarrow v_i \ldots\ldots v_n \rightarrow z$. In this case, x becomes a predecessor of y, and the predecessors on the shortest path(s) from y to z become predecessors on the shortest path(s) from x to z. Predecessors denote the nodes that are the last stop(s) before the final destination node, which is z in this case. The predecessors are updated in Lines 22–23 of Algorithm-2.

The final phase of the INSERTUPDATE procedure (Lines 26–29 of Algorithm-2) is for pruning the parts of the network that are not affected by the changes in the shortest paths. For each of the edges to/from the affected node x, it is checked to see if they are on the inspected shortest paths. If SP returns true, and if the other end of the edge (node u) is not in the list of already processed nodes, the edge $u \rightarrow x$ is inserted in the set of edges that would need inspection for subsequent processing. This part of the algorithm is responsible for propagating updates further if required. The ripples of updates expand outwards as much as required starting at the modified or inserted edge in the center. In this case, the edge $u \rightarrow x$ would carry the network update to the next ripple level.

3) ReduceBetweenness Procedure

This procedure opportunistically reduces the betweenness values of intermediates on the old set of shortest paths from x to z that are no longer shortest paths. To be able to construct the shortest paths, we only store predecessors; not the whole path. The shortest paths from node x to z are constructed on demand by following the predecessors. However, since these paths are constructed on demand, there might be subpaths that might have already been updated before the network update propagation reaches the shortest paths/distance from x to z. In such cases, there will be some intermediate nodes that are already cleared and not reachable anymore. The nodes that are already deleted from the shortest paths are stored in *trackLost*.

In Lines 1–20 of Algorithm-3, first, we reduce the betweenness of each node v that is found to be an intermediate from x to z and remove the contribution of the node pair (x, z) from the betweenness of node v. Then, we process currently unreachable intermediate nodes that originally belonged to the shortest paths from x to z and reduce their betweenness values as required (Lines 21–29 of Algorithm-3).

4) IncreaseBetweenness Procedure

By the time INCREASEBETWEENNESS is invoked in the INSERTEDGE procedure (Line 9 of Algorithm-1), all the shortest paths, the number of distinct shortest paths, and the predecessors affected by the network update have accurately been adjusted. Since we have also reduced the betweenness

values of the intermediate nodes on the invalidated shortest path by invoking REDUCEBETWEENNESS, the only remaining action is updating the betweenness values for the new set of intermediate nodes on the paths from the affected source nodes to the affected sink nodes. For each node pair (x, z) that is recorded in the σ_{old} set, we increase the betweenness value of each intermediate n on the shortest paths from x to z by $(\sigma(x, n) * \sigma(n, z) / \sigma(x, z))$. With this step, incremental update of betweenness centralities is complete.

5) Discussion on Algorithmic Complexity

Next, we discuss the time complexities of the proposed algorithms. Earlier, it has been shown that an incremental algorithm can perform asymptotically no better than its static counterpart for some dynamic problems [15] because in the worst case an incremental algorithm needs to solve the entire problem set. In our case, the proposed algorithms' complexities are not any lower than that of Brandes algorithm. We express the complexity analysis for incremental algorithms by incorporating the complexity of changes for expressing the time complexity of the incremental function.

The INCREASEBETWEENNESS procedure runs a for loop for σ_{old} many iterations and inside the outer for loop, there is one for loop, and one while loop. These two loops should be considered in combination because the intermediate nodes on the shortest paths from *src* to *dest* are handled by one or the other and the distinction is irrelevant. The complexity of the bodies of these loops are O(1), and they are executed once for each intermediate node. So, the overall complexity of the procedure is $O(|\sigma_{old}| I)$ where I represents the total number of intermediates processed for all node pairs listed in σ_{old}. In the REDUCEBETWEENNESS procedure, the run time is dominated by the if block at the end (Lines 25 – 29 of Algorithm-3). This block performs a search over the map of all known intermediate nodes on the shortest paths from a to z and uses two intermediates at a time to form the key to the map. Hence, its complexity is $O(I_{a,z}^2)$ where $I_{a,z}$ represents the number of intermediates on the shortest paths from a to z.

The overall complexity of the INSERTUPDATE procedure is dominated by the complexity of the priority queue Workset. Workset is used to track all the affected nodes as the propagation of the update progresses. INSERTUPDATE essentially performs a traversal in the neighborhood of every AffectedSink and AffectedSource. The work performed inside the while loop is $O(||\text{Affected}|| \log ||\text{Affected}||) + I^2)$ where ||Affected|| is used to denote the sum of the number of the edges and the nodes in the subgraph formed by AffectedSource and AffectedSink nodes' neighborhoods. Finally, the INSERTEDGE procedure invokes the INSERTUPDATE procedure for each AffectedSink and AffectedSource node once, followed by an invocation of the INCREASEBETWEENNESS procedure, yielding $O((|\text{AffectedSink}|+|\text{AffectedSource}|)||\text{Affected}||\log ||\text{Affected}||) + I^2 + |\sigma_{old}| I)$ time complexity overall.

The proposed algorithms depend on the dynamic all-pairs shortest path algorithms proposed in [4] to incorporate the computation of betweenness centrality. Incremental algorithms usually provide faster solutions at the cost of more memory usage. The incremental betweenness algorithm also takes quadratic space, using memory on the order of $O(n^2 + m)$. Accurate maintenance of betweenness centralities depends on the accurate maintenance of shortest distances, whose correctness was proved in [4]. The reader is referred to [4] for more details on the proof of correctness.

IV. IMPLEMENTATION, DATASETS, AND RESULTS

A. Implementation Environment

We implement our algorithms in an open source, dynamic Java graph library [16]. Our performance results are collected on a machine with a 3.20Ghz CPU and 256 GB of RAM.

B. Synthetic Networks

For synthetic networks, we use preferential attachment networks (PF) [17], Erdos-Renyi (ER) networks [18], and small-world (SW) networks [19]. We vary the number of nodes from 1000 to 5000 with a step size of 2000, and fix the average degree to 6. For small world networks, the rewiring probability is 0.5. We generate these synthetic networks with all but 100 edges that are selected randomly. We insert the last 100 edges incrementally and get the average update performance in terms of execution time over the repeated invocations of Brandes' algorithm, which is the best performing algorithm used in standard implementations.

TABLE 1 - PERFORMANCE IMPROVEMENTS OBTAINED ON DIFFERENT NETWORKS WITH DIFFERENT TOPOLOGIES/SIZES.

#(Nodes)	PF	ER	SW
1000	1178.66 x	7.99 x	17.48 x
3000	971.40 x	18.98 x	18.53 x
5000	3760.48 x	31.19 x	22.54 x

TABLE 2 - PERCENTAGE OF AFFECTED NODES (AFFECTEDSINKS + AFFECTEDSOURCES)

#(Nodes)	PF	ER	SW
1000	3.54%	79.16%	32.52%
3000	1.98%	85.7%	33.35%
5000	1.16%	87.36%	31.86%

Table 1 lists average speedup obtained per new edge insertion while Table 2 shows the percentage of nodes that are affected in terms of the sizes of AffectedSinks and AffectedSources. These results indicate that the incremental betweenness algorithm performs best with the preferential attachment networks. Comparing the network statistics and the speedup obtained on different networks (Table 3), the speedup obtained using the incremental betweenness update algorithm increases with the increased network size. It is also observed that other parameters such as network diameter, characteristic path length, and min/max betweenness values are inversely related with the performance obtained. The values in Table 1 and the speedup column in Table 4 describe the speedup obtained over Brandes' algorithm averaged across 100 updates on the network. For instance, for a single update on 1000-node Erdos-Renyi network, the incremental betweenness algorithm is 7.99 times faster on average than invoking Brandes algorithm for the same update, resulting in a 799x faster cumulative execution time for a sequence of 100 updates.

For instance, in preferential attachment networks, the average path length and the diameter are lower than they are in other topologies. However, the average betweenness value and the network size exhibit the strongest correlation with the

obtained speedup. The performance benefits of the incremental betweenness algorithm increase with the increasing network size. When the average betweenness values are considered, the difference across different topologies is very large. This is because in preferential attachment networks, there are fewer nodes that are on the shortest paths of many other nodes than in other network topologies. Hence, when there is a network update, there are fewer nodes whose betweenness values should be adjusted. Another factor is the average and maximum of the shortest path lengths (i.e. average path length and diameter). When the average distances are low, fewer nodes lie on the shortest paths which also results in tracking of fewer predecessors when there is need for reconstructing the shortest paths.

TABLE 3 - NETWORK STATISTICS.

Topology	Size	Max Btw	Avg. Btw	Std. Dev. Btw	Diameter	Avg Path Length	Clust Coef
PF	1000	1953.97	94.37	177.47	10	3.45	0.014
PF	3000	5183.26	197.59	434.828	14	4.126	0.007
PF	5000	12987.22	292.48	749.003	16	4.442	0.005
ER	1000	25429.36	4777.28	4249.81	15	6.305	0.003
ER	3000	76713.80	18136.7	10087.4	14	7.086	0.001
ER	5000	108061.5	32073.4	16062.9	14	7.492	0.001
SW	1000	12401.46	2685.67	2255.69	33	7.612	0.044
SW	3000	82585.03	11296.2	10401.4	55	10.33	0.039
SW	5000	147015.4	21003.3	20449.2	71	11.93	0.039

TABLE 4 - PERFORMANCE BENEFITS AND NETWORK STATISTICS OBTAINED ON SMALL WORLD NETWORKS (1000 NODES, AVERAGE DEGREE = 6).

p	Speed up	Affect %	Max Btw	Avg. Btw	Std. Dev. Btw	Diameter	Avg Path Len	Clust Coef
0.2	1.36	47.78	34020	4305	3104	35	9.71	0.154
0.4	9.97	36.47	15036	3183	2395	30	8.02	0.071
0.6	18.3	28.00	14763	2268	2463	31	7.79	0.024
0.8	67.3	13.06	6779	833	1162	22	6.44	0.005
1.0	72.2	2.23	1026	100	144	12	3.86	0.003

In addition, small world networks have different topological characteristics and performance values depending on the rewiring probability, *p*, chosen. We perform a sweep of *p* values covering the range of 0.2 - 1.0 with a step size of 0.2 on 1000-node networks, with an average degree of 6. As shown in Table 4, with the increasing rewiring probability, clustering coefficient, diameter, and the characteristic path length reduce. This reflects as a reduction in the average of unscaled betweenness values along with an increase in the speedup obtained using the incremental betweenness algorithm similar to the results presented in Table 1 and Table 3. In addition, the speedup obtained over repeated invocations of Brandes' algorithm increases with the reducing percentage of affected nodes, in line with the results presented earlier.

C. Real Life Networks

Next, we evaluate the performance of our algorithm using a number of real life networks that are of different magnitudes and that grow incrementally over time. The networks used in our evaluations are prepared as weighted networks where the cost of an edge is inversely proportional to the strength of relationship. We consolidate multiple updates for the same pair of nodes in a single edge. For instance, if an interaction between two nodes x and y has been recorded twice up to a certain point, then the edge $x \rightarrow y$ has the cost of 1/2. When a third update is recorded between x and y, then the cost of the edge $x \rightarrow y$ is updated to be 1/3. We first describe the datasets we have used, and then compare the performance of our incremental betweenness update algorithm against the best-performing non-incremental betweenness algorithm (Brandes' algorithm [5]). We use four different real life networks: SocioPatterns (communication between conference attendees) [20], Facebook-like (online-forum communication between students) [21], HEP Co-Authorship Network (coauthorship relations between High-Energy Physics researchers) [22], and P2P Communication Network (P2P file sharing) [23].

TABLE 5- PERFORMANCE OF INCREMENTAL BETWEENNESS ALGORITHM ON REAL LIFE NETWORKS.

Network	D?	#(N)	#(E)	Avg Speedup	Affect%
SocioPatterns	U	113	4392	9.58 x	38.26%
FB-like	D	1896	20289	18.48 x	27.67%
HEP Coauthor	U	7507	19398	357.96 x	42.08%
P2P Comm.	D	6843	7572	36732 x	0.02%

TABLE 6- NETWORK STATISTICS COLLECTED ON REAL LIFE NETWORKS.

Network	Max Btw	Avg. Btw	Std. Dev. Btw	Diameter	Avg. Path Len.	Clus Coef
SocioPatterns	423.477	36.752	51.139	3	1.65	0.53
FB-like	146171.2	2848.62	9753.8	8	3.19	0.08
HEP Coauthor	820318.2	13553.29	38024	15	5.74	0.46
P2P Comm	1515.99	0.3298	18.870	3	1.24	0

For evaluating the performance of our incremental betweenness update algorithm, we first compute the betweenness centrality values for each network modeling all but 100 interactions. Then, we incrementally update the network and record the average speedup obtained over Brandes' algorithm. Table 5 presents the performance improvements obtained along with basic information on the networks, while Table 6 lists additional information about other topological properties of the networks.

The results presented in Table 5 and Table 6 suggest that the incremental betweenness update algorithm can obtain substantial performance benefits, but these benefits vary with the network topology. The avg. speedup column in Table 5 describes the speedup obtained over Brandes' algorithm averaged across 100 updates on the network. For instance, for a single update the incremental betweenness algorithm is 9.58 times faster on average than invoking Brandes algorithm for the same update; resulting in 958x faster cumulative execution time for a sequence of 100 updates.

The performance benefits improve with the increasing network size and decreasing characteristic path length, diameter, and average betweenness as shown in Table 5 and Table 6. For instance, on the HEP co-authorship network, there are several close-knit groups and it is a relatively more connected network than the P2P communication network, where only a few users act as servers for the other users providing them with files to download. Hence, in the P2P communication network, very few nodes can lie on the shortest paths between other nodes. Consequently, when a network update occurs, few shortest paths tend to be changed, and thus few betweenness values are affected, resulting in a dramatic average speedup per each update (36732x) over Brandes' algorithm. The rightmost column of Table 5 shows the percentage across the entire set of nodes that were affected. In undirected (bidirectional) networks, the percentage of affected nodes tends to be higher as each inserted edge causes the network update to propagate in multiple directions.

D. Comparison with QuBE Algorithm [11]

The idea of the QuBE algorithm depends on estimating the nodes whose betweenness values might change due to an update in a network while avoiding computation of all-pairs shortest paths. In contrast, our algorithm depends on dynamic maintenance of all-pairs shortest paths and the related auxiliary data. The QuBE algorithm covers edge insertions/deletions, leaving out node insertions for growing networks and edge cost modifications for weighted network types. In contrast, our algorithm supports node/edge insertion and edge cost modifications for the weighted networks.

Providing support for weighted networks makes the algorithm more complex. For example, assume that there is a path from x to y. Then, with a network update an edge from node x to y is inserted into the network. In binary networks, it is obvious that no path between x and y can be smaller than a direct edge between x and y, and several changes on the shortest paths can be maintained by considering the number of hops. However, in weighted networks, when an edge from x to y is inserted, it is still necessary to check the paths of equivalent length before ruling out all previously known shortest paths between x and y.

TABLE 7- PERFORMANCE COMPARISON OF QUBE AND OUR PROPOSED ALGORITHM.

Network	Type	#(Node)	#(Edge)	QuBE	Incremental Betweenness
Eva [24]	Ownership	4457	4562	2418.17	25425.87
CAGrQc [25]	Collaboration	4158	13422	2.06	67.86

We compare our algorithm against the QuBE algorithm using the datasets the authors used in their paper [11]. We select two of their datasets: the dataset on which QuBE performs the best (Eva), and the dataset on which QuBE performs the lowest (CAGrQc). Table 7 reports the average performance results for 100 random updates on the networks. For purposes of fair comparison, the updates included shrinking network updates as well, which were handled by an incremental shrinking network update algorithm we have under development and excluded due to space reasons. Both QuBE and our algorithm are compared against the Brandes' algorithm as baseline. Our algorithm performs 10-30 times better than the QuBE algorithm while providing substantial improvements over Brandes' algorithm. Additional analyses of speedup and memory consumption are presented in [26].

V. CONCLUSION

This paper proposes an incremental betweenness algorithm that performs dynamic maintenance of betweenness values in the cases of a new edge/node insertion and/or edge cost decrease. The goal is to avoid re-computations involved in the analysis of dynamic social networks and reflect changes triggered by a network update as efficiently as possible. The approach in this paper has already been extended to other types of centrality measures and to networks that grow and shrink over time [27]. While the underlying behavior of incremental all-pairs shortest path computation has been studied, the memory and computation required to extend the shortest path algorithm to a particular centrality metric can result in significantly different scaling of computation time and memory requirements with network size and type. Our performance results indicate substantial performance improvements over the state of the art including non-incremental and dynamic update algorithms on realistic social network data.

VI. ACKNOWLEDGEMENTS

This work is supported in part by the Defense Threat Reduction Agency (HDTRA11010102), and by the center for Computational Analysis of Social and Organizational Systems (CASOS). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied by the DTRA or the U.S. government.

APPENDIX

Algorithm-1: INSERTEDGE (*src*, *dest*, *cost*)

1. $\sigma_{old} \leftarrow [\ ]$; $D_{old} \leftarrow [\ ]$; *trackLost* $\leftarrow [\ ]$; *PairsDone* = [ ]
2. $C(src, dest) \leftarrow cost$
3. *Sinks* $\leftarrow$ INSERTUPDATE (*dest*, *src*, *src, PairsDone*)
4. *Sources* $\leftarrow$ INSERTUPDATE (*src*, *dest*, *dest, PairsDone*)
5. for $s \in$ *Sinks*
6. INSERTUPDATE (*src*, *dest*, *s, PairsDone*)
7. for $s \in$ *Sources*
8. INSERTUPDATE (*dest*, *src*, *s, PairsDone*)
9. INCREASEBETWEENNESS( )

Algorithm-2: INSERTUPDATE (*src*, *dest*, *z, PairsDone*)

1. Workset $\leftarrow \{src \rightarrow dest\}$
2. VisitedVertices $\leftarrow \{src\}$
3. AffectedVertices $\leftarrow \emptyset$
4. while Workset $\neq \emptyset$
5. $\{x \rightarrow y\} \leftarrow$ pop (Workset)
6. $alt \leftarrow C(x, y) + D(y, z)$
7. if $alt < D(x, z)$
8. if $\langle x, z\rangle \notin \sigma_{old}$
9. $D_{old}(x, z) \leftarrow D(x, z)$; $\sigma_{old}(x, z) \leftarrow \sigma(x, z)$;
10. REDUCEBETWEENNESS (x, z);
11. $\sigma(x, z) \leftarrow 0$; Clear $P_x(z)$;
12. if $[u, z] \in$ *PairsDone*
13. Remove $[x, z]$ from *PairsDone*
14. $D(x, z) \leftarrow alt$
15. if $alt == D(x, z)$ and $D(x, z) \neq \infty$
16. if $[x, z] \notin$ *PairsDone*
17. if $\langle x, z\rangle \notin \sigma_{old}$
18. REDUCEBETWEENNESS (x, z);
19. if $\sigma(x, z) \neq 0$
20. $\sigma_{old}(x, z) \leftarrow \sigma(x, z)$
21. $\sigma(x, z) \leftarrow \sigma(x, z) + (\sigma(x, src) * 1 * \sigma(dest, z))$
22. Append x to $P_x(y)$ and $P_y(z)$ to $P_x(z)$
23. Insert $[x, z]$ into *PairsDone*
24. Insert x into AffectedVertices
25. for $u \in$ Pred (x) sorted w.r.t. edge costs in asc. order
26. if SP $(u, x, src) = 1$ && $u \notin$ VisitedVertices
27. push $\{u \rightarrow x\}$ into Workset
28. Insert u into VisitedVertices
29. return AffectedVertices

Algorithm-3: REDUCEBETWEENNESS (a, z)

1. if $\sigma_{old}(a, z) = 0$
2. return;
3. *Known* ← ∅; *Stack* ← ∅
4. for $n \in P_a(z)$
5. if $D(a, z) \neq D_{old}(a, n) + D_{old}(n, z)$
6. continue;
7. else if $a \neq n$ & $n \neq z$
8. $B(n) = B(n) - (\sigma_{old}(a, n) * \sigma_{old}(n, z) / \sigma_{old}(a, z))$
9. Add $<a, z, n>$ to *trackLost*
10. Add n to *Stack* and *Known*
11. while *Stack* ≠ ∅
12. p ← pop (*Stack*)
13. Add p to *Known*
14. for $n \in P_a(p)$
15. if $D(a, z) \neq D_{old}(a, n) + D_{old}(n, z)$
16. continue;
17. else if $a \neq n$ & $n \neq z$ & $n \notin$ *Known*
18. $B(n) = B(n) - (\sigma_{old}(a, n) * \sigma_{old}(n, z) / \sigma_{old}(a, z))$
19. Add $<a, z, n>$ to *trackLost*
20. Add n to *Stack* and *Known*
21. *AlreadyDone* ← (*Known* ∪ a)
22. if $D(v, z) = D_{old}(a, v) + D_{old}(v, z)$ where $v_1, v_2 \in$ *Known* and $<v_1, v_2, v> \in$ *trackLost*
23. if $v \notin$ *AlreadyDone*
24. $B(v) = B(v) - (\sigma_{old}(a, v) * \sigma_{old}(v, z) / \sigma_{old}(a, z))$
25. Add v to *AlreadyDone*
26. Add $<a, z, v>$ to *trackLost*

Algorithm-4: INCREASEBETWEENNESS ()

1. for $(src, dest) \in \sigma_{old}$
2. *Known* ← ∅; *Stack* ← ∅
3. for $n \in P_{src}(dest)$
4. Add n to *Stack* and *Known*
5. if $src \neq n$ & $n \neq dest$
6. $B(n) \leftarrow B(n) + (\sigma(src, n) * \sigma(n, dest) / \sigma(src, dest))$
7. while *Stack* ≠ ∅
8. n ← pop (*Stack*)
9. Add n to *Known*
10. for $p \in P_{src}(n)$
11. if $p \neq src$ & $p \neq dest$ & $p \notin$ *Known*
12. Add p to *Stack* and *Known*
13. $B(p) = B(p) + (\sigma(src, p) * \sigma(p, dest) / \sigma(src, dest))$

REFERENCES

[1] R.W. Floyd, "Algorithm 97: Shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, June 1962.

[2] L. C. Freeman, "A Set of Measures of Centrality based on Betweenness," *Sociometry*, vol. 40, no. 1, pp. 35-41, 1977.

[3] A. M. Berman, "Lower and upper bounds for incremental algorithms," Computer Science, The State University of New Jersey at Rutgers, New Brunswick, NJ, PhD Dissertation 1992.

[4] G. Ramalingam and T. Reps, "On the Computational Complexity of Incremental Algorithms," CS, Univ. of Wisconsin at Madison, Tech. Report 1991.

[5] U. Brandes, "A Faster Algorithm for Betweenness Centrality," *Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163--177, 2001.

[6] H. Kim and R. Anderson, "Temporal node centrality in complex networks," *PHYSICAL REVIEW E*, vol. 85, no. 026107, pp. 1-8, 2012.

[7] K. Lerman, R. Ghosh, and J. H. Kang, "Centrality Metric for Dynamic Networks," in *8th Workshop on Mining and Learning with Graphs (MLG)*, 2010, pp. 70-77.

[8] J. Tang, M. Musolesi, C. Mascolo, V. Latora, and V. Nicosia, "Analysing information flows and key mediators through temporal centrality metrics," in *3rd Workshop on Social Network Systems (SNS)*, 2010.

[9] H. Habiba, C. Tantipathananandh, and T. Berger-Wolf, "Betweenness centrality measure in dynamic networks," Department of Computer Science, University of Illinois at Chicago, Chicago, 2007.

[10] R. Puzis, P. Zilberman, Y. Elovici, S. Dolev, and U. Brandes, "Heuristics for Speeding up Betweenness Centrality Computation," in *Social Computing and on Privacy, Security, Risk and Trust*, 2012, pp. 302-311.

[11] M. J. Lee, J. Lee, J. Y. Park, R.H. Choi, and C. W. Chung, "QUBE: a Quick algorithm for Updating BEtweenness Centrality," in *WWW*, 2012, pp. 351--360.

[12] V. King, "Fully Dynamic Algorithms for Maintaining All-Pairs Shortest Paths and Transitive Closure in Digraphs," in *40th Annual Symposium on Foundations of Computer Science*, 1999, pp. 81--89.

[13] C. Demetrescu and G. F. Italiano, "A New Approach to Dynamic All Pairs Shortest Paths," *Journal of the ACM (JACM)*, vol. 51, no. 6, pp. 968--992, November 2004.

[14] C. Demetrescu and G. F. Italiano, "Experimental Analysis of Dynamic All Pairs Shortest Path Algorithms," *ACM Transactions on Algorithms (TALG)*, vol. 2, no. 4, pp. 578 - 601, 2006.

[15] S. Even and H. Gazit, "Updating distances in dynamic graphs," *Methods of Operations Research*, vol. 49, pp. 371--387, 1985.

[16] GraphStream Team. (2010) GraphStream. [Online]. http://graphstream-project.org/

[17] A.L. Barabasi and R. Albert, "Emergence of Scaling in Random Networks," *Science*, vol. 286, no. 5439, pp. 509-512, 1999.

[18] A. Renyi and P. Erdos, "On Random Graphs," *Publicationes Mathematicae*, vol. 6, 1959.

[19] D. Watts and S. Strogatz, "Collective Dynamics of 'Small-World' Networks," *Nature*, vol. 393, 1998.

[20] L. et al. Isella, "What's in a crowd? Analysis of face-to-face behavioral networks," *Journal of Theoretical Biology*, vol. 271, no. 1, pp. 166–180, 2011.

[21] T. Opsahl and P. Panzarasa, "Clustering in weighted networks," *Social Networks* , vol. 31, no. 2, pp. 155-163, 2009.

[22] M. Kas, K. M. Carley, and L. R. Carley, "Trends in science networks: understanding structures and statistics of scientific networks," *Social Network Analysis and Mining (SNAM)*, vol. 2, no. 2, pp. 169-187, 2012.

[23] F. et al. Gringoli, "GT: picking up the truth from the ground for Internet traffic," *Computer Communication Review*, vol. 39, no. 5, pp. 13--18, 2009.

[24] K. Norlen, G. Lucas, M. Gebbie, and J. Chuang, "EVA: Extraction, visualization and analysis of the telecommunications and media ownership network," in *International Telecommunications Society 14th Biennial Conference*, 2002.

[25] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph evolution: Densification and shrinking diameters.," *ACM Trans. KDD*, vol. 1, no. 2, pp. 1-41, 2007.

[26] M. Kas, "Incremental Centrality Algorithms for Dynamic Network Analysis," ECE, Carnegie Mellon University, Pittsburgh, PA, Ph.D. Dissertation 2013.

[27] M. Kas, K. M. Carley, and L.R. Carley, "Incremental Closeness Centrality for Dynamically Changing Social Networks," in *Workshop on the Semantic and Dynamic Analysis of Information Networks (ASONAM).*, 2013.