Incremental Closeness Centrality for Dynamically Changing Social Networks

Miray Kas, Kathleen M. Carley, L. Richard Carley
Carnegie Mellon University
Pittsburgh, PA, USA
{mkas@ece.cmu.edu, kathleen.carley@cs.cmu.edu, carley@ece.cmu.edu}

Abstract— **Automation of data collection using online resources has led to significant changes in traditional practices of social network analysis. Social network analysis has been an active research field for many decades; however, most of the early work employed very small datasets. In this paper, a number of issues with traditional practices of social network analysis in the context of dynamic, large-scale social networks are pointed out. Given the continuously evolving nature of modern online social networking, we postulate that social network analysis solutions based on *incremental algorithms* will become more important to address high computation times for large, streaming, over-time datasets. Incremental algorithms can benefit from early pruning by updating the affected parts only when an incremental update is made in the network. This paper provides an example of this case by demonstrating the design of an incremental closeness centrality algorithm that supports efficient computation of all-pairs of shortest paths and closeness centrality in dynamic social networks that are continuously updated by addition, removal, and modification of nodes and edges. Our results obtained on various synthetic and real-life datasets provide significant speedups over the most commonly used method of computing closeness centrality, suggesting that *incremental algorithm design* is a fruitful research area for social network analysts.**

Keywords—Closeness Centrality; Incremental Algorithms; Dynamic All-Pair Shortest Path; Dynamic Networks.

I. INTRODUCTION

Today, with more people actively using the Internet in their daily lives, social network data that can be collected online is undergoing dramatic growth. This trend has caused the analysis of online social networks to emerge as an important tool for many social and business opportunities including creating and following trends, discovering new markets, running election and advertisement campaigns, and influencing national politics.

On the research side, due to this dramatic growth in available online social network data, along with the possibility of automated data collection, there is increasing interest in *dynamic network analysis* which focuses on developing custom-designed methods for very large, dynamic (over-time), multi-mode (where nodes can be in different categories), multiplex (where there are multiple types of links) networks. However, there still exist major challenges to be overcome in this transition from traditional social network analysis on small static networks to social network analysis on large-scale dynamic networks.

In today's online world, these links are constructed using information provided by email exchanges, file or photo sharing, or other features like 'friend lists', 'membership', 'tag', 'like', 'share this', 'follow', 'send a message', etc. Such social graphs are constantly expanding and growing; presenting major challenges in identification of the most central actors in a given social network as it evolves over time. A majority of early research in the field of social network analysis targeted eliciting the most central/prominent actors in small, static networks that model small social groups.

To date, hundreds of social centrality metrics have been designed and discussed in the literature. However, a significant number of publications analyzing social networks consider only a handful of metrics: degree centrality, eigenvector centrality [1], closeness centrality [2], and betweenness centrality [3]. Out of these four metrics, the latter two are shortest-path based metrics. The shortest-path based metrics consider the shortest communication paths in a given network topology and focus on the position of a node with respect to the shortest paths in the network.

Computations of the shortest path based centrality metrics usually require solving the costly all-pairs shortest path problem. Given that the most commonly used centrality metrics are designed for static networks, attempting to compute traditional centrality metrics on dynamic social networks boils down to fixing a dynamic network momentarily, performing computations on it, and then performing similar computations from scratch on an updated version of the network.

The goal of this paper is to draw the attention of the social network analysis community to the use of incremental computation of shortest paths in dynamic network analysis and to discuss how incremental algorithm design techniques would be beneficial to improve the traditional techniques used in social network analysis. Incremental algorithms are algorithms that are custom-designed for dynamically changing networks and respond to the over-time changes in the analyzed network by performing early pruning and propagating the updates only to the affected parts of the network. In general, incremental algorithms are saving redundant computations at the expense of the need to store information about prior computations. Another contribution of the algorithms designed in this paper is the fast computation of closeness centrality for large-scale static networks as we will show later in the results section. That is, applying the incremental algorithm at every step in building up a network can be less computationally expensive than running the traditional algorithm a single time on the final network.

This paper discusses the general class of centrality metrics based on the shortest paths across all possible pairs of nodes. As a case study, we present the design of an incremental closeness centrality algorithm that handles various types of network updates including addition, removal, and modification of nodes and edges. Closeness centrality was selected as the focus for this paper for two reasons. First, closeness centrality is one of the most commonly used metrics in social network analysis. Second, the definition of closeness centrality depends entirely on the shortest path information across all pairs of nodes. The information on the shortest distance between pairs of nodes is inherently required by all shortest-path based metrics. This means that the incremental methods discussed in this paper are generalizable to other metrics with shortest-path computation as their core computational limitation. Most other shortest path based centrality metrics require information such as the number of the shortest paths between nodes, the predecessors and/or successors on these shortest paths. Therefore, this paper will focus on closeness centrality as the example incremental metric.

II. Background

A. Definition & Computation of Closeness

The closeness centrality of node x, $C_c(x)$, is defined as the inverse of the sum of the distances from x and all other nodes in the network: $C_c(x) = \frac{1}{\sum_{x \neq y} D(x,y)}$ where $D(x,y)$ denotes the shortest distance from node x to node y. Closeness centrality is traditionally best computed by running a single-source shortest path algorithm using each node as the source node once. At each iteration, the distances found are summed up to obtain the total distance from the given source node, and this distance is inverted to obtain the closeness value of the source.

In unweighted (binarized) networks, a breadth-first search algorithm may be used to discover the shortest paths from a source nodes, which is bounded by $O(n+m)$ time complexity per source node, resulting in $O(nm)$ complexity in total. In weighted networks, Dijkstra's algorithm [4] has $O((n + m)\log n)$ complexity, where n denotes the number of nodes, m denotes the number of edges in the network. This complexity is achieved when a binary min-heap is used in the implementation of the priority queue. A faster run-time of $O(m + n\log n)$ can be achieved by implementing the priority queue using a Fibonacci heap [5]. When Dijsktra's algorithm is invoked using every node in the network as the source node to compute all-pairs shortest paths, the overall complexity is $O(mn + n^2\log n)$. Computation of closeness centrality can be performed by running an all-pair shortest paths algorithm (e.g. Floyd-Warshall [6]), which results in $O(n^3)$ time complexity.

The algorithmic complexities of Dijkstra and Floyd-Warshall are sufficiently high that they are very difficult to invoke at every time step in dynamically changing, large-scale networks. Hence, this paper proposes the use of incremental algorithms that avoid the cost of recomputing all of the shortest paths from scratch every time period.

B.Example Use Cases for Closeness

Closeness centrality is a commonly used social centrality metric and it can have different uses in different contexts. Closeness centrality of a social actor describes actor's efficiency for information propagation across the entire network. In other words, social actors with high closeness centrality values are considered to be efficient at making contact with others in the network. High closeness centrality is also regarded as representing high potential for independent communication.

In the context of technological networks, such as wireless networks, closeness centrality identifies nodes that have rapid access to information (e.g. nodes that are close to many other nodes on average). Since closeness centrality is inversely proportional to the sum of the distances to all other nodes, it also provides an estimate of how long it will take information to spread from a node to all others. Hence, it can also be used as a performance measure in technological networks [7].

As another example, in [8], the authors discuss the use of closeness centrality for policy-making networks (e.g. drug policy making). In the context of policy-making networks, the actors that have information that is crucial to all other actors in the network should have high closeness centrality if the network is to function effectively.

As we mentioned earlier, closeness centrality is one of the most commonly used metrics in social network analysis. Hence, it should be understood that there are several other papers that employ closeness centrality in the research literature on social network analysis and that the preceding discussion simply provides a few examples of the applications of closeness centrality in social network analysis. There exist other studies that discuss the extensions of closeness centrality metrics for dynamic, complex networks [9]. There has also been research on new methods to select top-*k* nodes in terms of closeness in large-scale networks [10] and on algorithms for approximation of closeness [11].

C. Dynamic All-Pairs Shortest Path Computation

To date, several algorithms have been proposed for solving the all-pairs shortest paths problem dynamically [12], [13], [14]. However, some of these solutions come with a number of restrictions and only work for certain conditions. For instance, [12] requires all edge costs to be integers below a certain threshold to be able to solve the all-pairs shortest paths problem dynamically.

In this study, we use the dynamic all-pairs shortest path algorithm proposed in [13] as our starting point. In general, Ramalingam and Reps algorithms define a full framework that works with all non-negative edge weights and its procedural structure enables distinguishing between the methods required for each network update type easily (e.g. inserting edges/nodes versus deleting edges/nodes). Second, Ramalingam and Reps algorithms are one of the most commonly used dynamic computation of all-pairs shortest paths algorithms. Third, Ramalingam and Reps algorithms have been shown to perform quite well on sparse, real-life networks/graphs.

In terms of computational time and memory requirement, [14] and [13] usually achieve similar performance. In [14], it has been discussed that the underlying computational platform is an important factor in deciding which algorithm performs better. The authors of [14] state that Ramalingam and Reps' algorithm is likely to become faster as the number of nodes increases because it requires less space compared to Demetrescu & Italiano's algorithm and exhibits better locality in the memory access pattern. Real life network experiments presented in [14] indicate that Ramalingam and Reps have the lowest or one of the lowest execution times among all dynamic all-pairs shortest path algorithms compared in that paper. Hence, we have decided to use Ramalingam and Reps algorithm as a building block in the closeness centrality algorithms proposed in this paper.

III. Incremental Closeness Algorithm

In order to handle the special needs of very dynamic large-scale social networks, this paper presents an *incremental algorithm* design approach. An incremental algorithm is different from its static counterpart that performs all computations from scratch. The application of an incremental algorithm is as follows. At one point, an initial run is performed by an algorithm that performs the desired computation from scratch (e.g. computation of closeness values in a given network). The incremental algorithm is then used in subsequent runs to handle various network updates such as edge cost modifications (in the case of weighted networks), node/edge insertions, and node/edge deletions. The incremental algorithm uses information from earlier computations such that the changes in the network are reflected on the closeness values as well. The benefit of an incremental algorithm is that, by being able to build on prior computations, it is able to perform early-pruning and update *only* the affected parts of the network while avoiding recomputation to a significant extent.

After briefly reviewing the notation we use in this paper, the pseudocodes are provided for the proposed incremental closeness algorithms in Section 3.B and Section 3.C.

In Section 3.B, an incremental algorithm for the following network update types is presented: (*i*) inserting a new node, (*ii*) inserting a new edge, and (*iii*) decreasing the cost of an existing edge. The same algorithm can handle these three update types because inserting a new node can be handled by invoking insert a new edge for every edge that comes along with the new node and inserting a new edge corresponds to decreasing the cost of an edge from infinity to a real value. We call these update types *growing network updates* as social networks usually grow by new people joining a community, formation of new relationship, and relationships becoming closer which reflects as a decrease in the cost of communication.

Section 3.C provides the pseudocodes for the incremental algorithms for handling the remaining network update types: (*iv*) deleting an existing node, (*v*) deleting an existing edge, and (*vi*) increasing the cost of an existing edge. We call this group of updates *shrinking network updates* because social networks shrink by people departing from a community, ending relationships or by relationships that become more distant. The algorithms in Section 3.B and Section 3.C focus on unit updates, handling each edge modification (e.g. addition/deletion/cost modification) one at a time. Similar to growing network updates, deletion of a node with several edges reduces to several invocations of the algorithms provided in Section 3.C to handle the removal of each edge emanating from and/or entering into the processed node.

In addition, 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 be represented as weighted networks where existing edges' weights/costs are always equal to 1. Therefore, directed, weighted networks provide the most generalized coverage of different network types. Thus, the algorithms in Section 3.B and Section 3.C consider weighted, directed networks.

A. Notation

This section briefly describes the notation we use in this paper. A directed network G is composed of a set of nodes $V(G)$ and edges $E(G)$. $\{x \rightarrow y\} \in E(G)$ represents an edge directed from node x to node y. $\tilde{G}$ is the transpose (reverse) of network G where all edges in network G are reversed in direction. Similar to network G, the set of edges, nodes, and edge costs are defined for network $\tilde{G}$ as well. Pred(x) is used to denote the predecessor neighbors of node x that have direct edges going into node x. Similarly, Succ(x) is used to denote the successor neighbors of node x that have direct edges emanating from node x going into the successor nodes.

$D(x, y)$ is used to hold the pairwise shortest path distances from node x to y while $W(x, y)$ is used to denote the cost of the edge from x to y. In other words, when the network is weighted, $D(x, y)$ denotes the sum of the edge costs over all of the edges along the shortest path(s) from x to y. And, when the network is binary, $D(x, y)$ gives the shortest path distance from x to y in terms of number of edges along the shortest path. The algorithms discussed in this paper are applicable to networks with non-negative edge costs. The closeness values of nodes with a vector of length $|V(G)|$ are represented by C_C. 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, false otherwise [13].

B. Incremental Closeness Algorithm: Handling Growing Network Updates

To compute the closeness values incrementally for streaming, dynamically changing social networks, the incremental all-pairs shortest-paths algorithm proposed by Ramalingam and Reps [13] is extended such that closeness values are incrementally updated in line with the changing shortest path distances in the network.

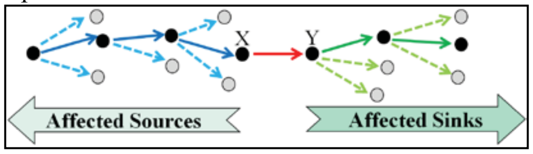

Figure 1 - An abstract picture describing how affected sink and source nodes are selected and how early pruning is done.

Before moving on to low level details of how the shortest path distances and closeness values are updated, a high level description of early pruning of shortest paths is provided in Figure 1. When the edge {X → Y} is inserted to a network G depicted in Figure 1, the maintenance of the shortest paths starts with the inserted edge. The edges that are on the shortest paths are tracked and further processed to ensure propagation of the shortest path updates as far as needed. In Figure 1, the set of black nodes gives us the affected sink and source nodes. The edges drawn with solid lines are the edges on the shortest paths, and they are considered for subsequent processing later in the execution. The edges that are drawn with dashed lines and the gray nodes that are accessible from those edges that are not on the shortest paths. Such nodes are not affected by the incoming network update; hence pruned early from recomputation.

When a growing network update is observed, the incremental computation of closeness centrality is handled by two sub-algorithms: *InsertEdgeGrowing* and

INSERTUPDATEGROWING. The main entry point for execution is *INSERTEDGEGROWING*. The *INSERTEDGEGROWING* (Algorithm-1) invokes *INSERTUPDATEGROWING* (Algorithm-2) several times to ensure identification of all affected source and sink nodes and to maintain closeness centrality values and the shortest distances to/from those nodes accurately. *INSERTEDGEGROWING* first invokes *INSERTUPDATEGROWING* to find the set of AffectedSink and AffectedSource nodes, passing once the source, once the destination of the inserted edge as a parameter to it (Lines 2–3 of Algorithm-1). Then, the *INSERTUPDATEGROWING* is invoked for each AffectedSink and AffectedSource (Lines 4–7 of Algorithm-1) to update the information required for accurate maintenance of closeness values in line with the newly-discovered, shortest paths (e.g. *D* for the shortest distance between each node pair).

```
Algorithm-1: INSERTEDGEGROWING (G, src, dest, c)

1. $W(src, dest) \leftarrow c$;  $\widetilde{W}(dest, src) \leftarrow c$
2. AffectedSinks    ← INSERTUPDATEGROWING ($\tilde{G}$, dest, src, src)
3. AffectedSources ← INSERTUPDATEGROWING (G, src, dest, dest)
4. for s ∈ AffectedSinks
5.     INSERTUPDATEGROWING (G, src, dest, s)
6. for s ∈ AffectedSources
7.     INSERTUPDATEGROWING ($\tilde{G}$, dest, src, s)
```

In Algorithm-2, the list Workset holds the set of edges that should be processed to detect formation of new shortest paths or existing paths becoming shorter. Since closeness centrality is computed as the inverse of sum of the distances from a node to all other nodes in a network, the *only* information it needs is the shortest distances between all pairs of nodes (represented as *D*). It is not necessary to know the number of shortest paths, the predecessors on these shortest paths, etc.

Assume that the shortest path distances and closeness centralities are already computed for a given network. In the case of a network update, we only need to update the closeness of a node *x* if the shortest distance from node *x* to any other node in the network changes. We check for the changes in the shortest distances using the condition given in Line-6 of Algorithm-2. If this condition holds, it means that there is now a shorter path from node *x* to *z* which passes through the edge $\{x \rightarrow y\}$, and the distance from *x* to *z* (i.e. $D(x, z)$) should be updated accordingly (Line-14).

Before the previous value of $D(x, z)$ is overridden with the new value, Lines 8-13 of Algorithm-2 handle the accurate maintenance of closeness centrality. To update closeness value of node *x* accurately, it is first necessary to check if node *z* was previously reachable from node *x* (e.g. $D(x, z) \neq \infty$). If it was reachable before, then it means the distance from *x* to *z* had a contribution to the closeness value of node *x*. In this case, it is necessary to first subtract the previously known shortest distance from node *x* to *z* (e.g. $D(x, z)$), and then add the new shortest distance (e.g. $W(x, y) + D(y, z)$) to the sum of distances to all nodes from node *x*. Otherwise, nothing is subtracted, only the new shortest distance is added to the sum of distances from node *x*. The closeness of node *x* is obtained by inverting the total distance (Algorithm - 2, Line 13).

The final part of the *INSERTUPDATEGROWING* algorithm (Algorithm-2, Lines 15-17) performs checks for subsequent processing. In this part of the algorithm, the portions of the network that are not affected by the changes in the shortest paths are pruned. 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 for subsequent processing.

```
Algorithm-2: INSERTUPDATEGROWING (G, src, dest, z)
1.  Workset ← {src → dest};
2.  VisitedVertices ← {src};
3.  AffectedVertices ← ∅
4.  while Workset ≠ ∅
5.      {x → y} ← pop (Workset)
6.      if $W(x, y) + D(y, z) < D(x, z)$
7.          Add x to AffectedVertices
8.          TotDist(x) = $\frac{1}{C_c(x)}$
9.          if $D(x, z) \neq \infty$
10.             TotDist(x) = TotDist(x) - $D(x, z) + W(x, y) + D(y, z)$
11.         else
12.             TotDist(x) = TotDist(x) + $W(x, y) + D(y, z)$
13.         $C_c(x) = \frac{1}{\mathrm{TotDist}(x)}$
14.         $D(x, z) \leftarrow W(x, y) + D(y, z)$
15.         for u ∈ Pred(x)
16.             if SP (u, x, src) == 1 && u ∉ VisitedVertices
17.                 push {u → x} into Workset
18.                 Insert u into VisitedVertices
19. return AffectedVertices
```

C. Incremental Closeness Algorithm: Handling Shrinking Network Updates

In this section, the part of the incremental closeness algorithm that handles shrinking network updates (e.g. deletion of a node/edge or edge cost increase) is presented. The shrinking network updates are handled by two sub-algorithms: *DELETEEDGESHRINKING* (Algorithm-3) and *DELETEUPDATESHRINKING* (Algorithm-4).

DELETEEDGESHRINKING (Algorithm-3) follows a very similar logic to that of *INSERTEDGEGROWING* (Algorithm-1). After updating the adjacency matrix for the modified/deleted edge, *DELETEUPDATESHRINKING* (Algorithm-4) is invoked several times: first, to identify affected sink and source nodes and then, to process each sink and source nodes separately.

```
Algorithm-3: DELETEEDGESHRINKING (G, src, dest, c)
1. $W(src, dest) \leftarrow c$;   $\widetilde{W}(dest, src) \leftarrow c$
2. AffectedSinks ←  DELETEUPDATESHRINKING($\tilde{G}$, dest, src, src)
3. AffectedSources←DELETEUPDATESHRINKING(G, src, dest, dest)
4. for s ∈ AffectedSinks
5.     DELETEUPDATESHRINKING (G, src, dest, s)
6. for s ∈ AffectedSources
7.     DELETEUPDATESHRINKING ($\tilde{G}$, dest, src, s)
```

The *DELETEUPDATESHRINKING* algorithm (Algorithm-4) has two distinct phases. The first phase of Algorithm-4 is between Lines 1-14 while the second phase is between Lines 15-38. The first phase of the algorithm identifies the set of affected vertices. In this case, affected vertices are those nodes whose shortest distances to node *z* (the third parameter of the algorithm) have increased. The shortest path distance from node *x* to *z* may only increase if the network update is made on an edge which used to lie on the shortest paths between those two nodes and all the available shortest paths pass through the modified/deleted edge (i.e. when there is no alternative shortest paths that would still be shorter). The check for this condition is in Lines 12-13 of Algorithm-4.

The second phase of Algorithm-4 determines the new shortest path distance to node *z* for all nodes in the set of affected vertices identified in the first phase of the algorithm. The maintenance of new distances to node *z* is handled by min-key priority queue where the priority of a node corresponds to its distance to node *z*. Closeness values are also updated in the second phase of Algorithm-4. To be more precise, closeness values are updated whenever a change on a *D* value is observed but right before this is change is recorded, overriding the prior knowledge on previous *D* value (Lines 18-24 and Lines 28-34). The idea behind closeness updates is similar to the idea described in Section 3.B. However, in this case additional checks are performed to avoid updating closeness values with a distance that was just set to infinity.

Algorithm-4: *DELETEUPDATESHRINKING* (*G, src, dest, z*)

```
1.  AffectedVertices ← ∅
2.  at_least_one_exists = false;
3.  for each $x$ c Succ($src$)
4.      if (SP($src$, $x$, $z$) == true)
5.          at_least_one_exists = true;
6.          break;
7.  if (at_least_one_exists == false)
8.      Workset ← {$src$};
9.      while Workset ≠ ∅
10.         $u$ ← pop (Workset)
11.         Add $u$ to AffectedVertices
12.         for each $x \in$ Pred($u$) such that SP($x$, $u$, $z$) == true
13.             if (all $y \in$ Succ($x$) s.t. SP($x$, $y$, $z$) == true and y ∈ AffectedVertices)
14.                 push $x$ into Workset
15.     PriorityQueue← ∅
16.     for ($a \in$ AffectedVertices)
17.         $min\_dist$ = min ({$W(a, b) + D(b, z)$ | {$a \rightarrow b$} ∈ $E(N)$ & $b \notin$ AffectedVertices}, {∞})
18.         TotDist($a$) = $\frac{1}{C_c(a)}$
19.         if $D(a, z) \neq \infty$
20.             TotDist($a$) = TotDist($a$) - $D(a, z)$
21.         $D(a, z) \leftarrow min\_dist$
22.         if $D(a, z) \neq \infty$
23.             TotDist($a$) = TotDist($a$) + $D(a, z)$
24.         $C_c(a) = \frac{1}{\text{TotDist}(a)}$
25.     while PriorityQueue ≠ ∅
26.         $a$ ← extractMin(PriorityQueue)
27.         for each $c \in$ Pred($a$) such that $W(c, a) + D(a, z) < D(c, z)$
28.             TotDist($c$) = $\frac{1}{C_c(c)}$
29.             if $D(c, z) \neq \infty$
30.                 TotDist($c$) = TotDist($c$) - $D(c, z)$
31.             $D(c, z) \leftarrow W(c, a) + D(a, z)$
32.             if $D(c, z) \neq \infty$
33.                 TotDist($c$) = TotDist($c$) + $D(c, z)$
34.             $C_c(c) = \frac{1}{\text{TotDist}(c)}$
35.             if $c \in$ PriorityQueue
36.                 DecreaseKey (PriorityQueue, $c$, $D(c, z)$)
37.             else
38.                 Insert (PriorityQueue, $c$, $D(c, z)$)
39. return AffectedVertices
```

D. Comments on Algorithmic Complexity

In this section, the complexities of the proposed incremental closeness algorithms are discussed. For incremental algorithms, there are different perspectives on how to evaluate their complexities. It has been demonstrated that, in the worst case, no incremental algorithm can perform asymptotically better than the algorithm that computes everything from scratch [15]. Hence, worst-case upper bound time complexity is usually not descriptive enough to explain the performance difference of an incremental algorithm from an algorithm solving the same problem from scratch.

For incremental algorithms, a preferred way of discussing their computational complexity is through the sum of the sizes of changes in the input (e.g. the modified graph/network) and output (e.g. modified distance and closeness centrality values). Next, we discuss the computational complexities of the *INSERTEDGEGROWING* and the *DELETEEDGESHRINKING* algorithms in terms of the changes in the input and output.

The *INSERTEDGEGROWING* algorithm calls the *INSERTUPDATEGROWING* for every AffectedSink and AffectedSource node. The *INSERTUPDATEGROWING* essentially performs a traversal in the neighborhood of every AffectedSink and AffectedSource, respectively. Hence, the complexity of each of these operations is on the order of $O(||\text{Affected}||)$ where ||Affected|| is used to denote the sum of the number of edges and the nodes in the subgraph formed by AffectedSource and AffectedSink nodes' neighborhoods.

Similar complexity analysis can be performed for *DELETEEDGESHRINKING*. Similar to the *INSERTEDGEGROWING*, *DELETEEDGESHRINKING* invokes *DELETEUPDATESHRINKING*for every AffectedSink and AffectedSource node. However, *DELETEUPDATESHRINKING* is more complicated than *INSERTUPDATEGROWING*. The *DELETEUPDATESHRINKING* has two distinct phases with different algorithmic complexities. Phase-2 makes use of a priority queue, whose time complexity must be taken into account separately.

Line-13 of Algorithm 4 checks the existence of the shortest paths between a predecessor (e.g. *x*) and a successor (e.g. *y*) of node *u*. This makes the time complexity of Phase-1 to be limited by $O(||\text{Affected}||_2)$ where the subscript 2 denotes the size of two hop neighborhood of all affected nodes. The set of affected nodes is given by (AffectedSink ∪ AffectedSource). The complexity of Phase-2 is dominated by the complexity of priority queue, denoted by $O(||\text{Affected}|| \log ||\text{Affected}||)$.

Since all the changes that are made to compute closeness centrality are of O(1) time complexity, computing closeness centrality along with the dynamic maintenance of the shortest paths does not increase the overall time complexity of Ramalingam and Reps algorithm. Similar to the Ramalingam and Reps algorithm, the memory requirement is quadratic.

As one final note, the algorithms presented in this section are the modified versions of the dynamic shortest path algorithms proposed in [13] to incorporate the computation of closeness centrality. Both in Algorithm-2 and Algorithm-4, closeness centrality of a node *x* is updated *only* when the shortest distance from node *x* to another node is updated. Hence, accurate maintenance of closeness values depends on the accurate maintenance of shortest distances, whose correctness was proved in [13]. The reader is referred to [13] for more details on the proof of correctness regarding the shortest path updates.

IV. DATASETS AND RESULTS

The goal of this paper is to draw attention to the use of incremental algorithm design in social network analysis methods. In particular, in this paper, we show how to design an incremental algorithm for closeness centrality; and, we explore how much speedup we obtain due to the use of this incremental algorithm on different types of synthetic and real-life networks that are used by social network researchers. Hence, our performance evaluations primarily show how much performance improvement can be achieved over the most commonly used way of computing the all-pairs shortest paths in a network as well as closeness centralities as a by-product of it (e.g. Dijsktra's algorithm).

A. Coding and Computing Environment

The proposed incremental algorithms were coded as an extension to GraphStream [16]. Performance results were measured on a machine with a quad-core 3.20 Ghz Intel Xeon CPU and 256 GB RAM.

B.Performance Results with Synthetic Networks

To understand the performance of the proposed incremental closeness algorithm, we have designed experiments with networks that are generated using different graph generation algorithms and network sizes. To understand the impact of topology, we ran a number of experiments on synthetic networks using three different topologies, while keeping the number of nodes and the average degree fixed. In our experiments, we use three different topologies: preferential attachment networks [17], Erdos-Renyi networks [18], and small-world networks [19]. We use networks with 1000, 3000, and 5000 nodes and set the average degree to 6. The average degree of a network is a measure that compares the number of edges against the number of nodes in the network. It is computed as $2.|E(G)| / |N(G)|$ as each edge contributes to the degree of both nodes it is connecting. For small world networks, the rewiring probability is 0.5.

To measure the performance of the incremental closeness algorithm for growing network updates, we generate the synthetic networks described above 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 Dijkstra's algorithm. For instance, if it takes 5 seconds to complete a set of updates using incremental algorithms and 30 seconds to complete the same set using Dijkstra's algorithm, we conclude that the incremental algorithm is 6x faster than Dijkstra's algorithm on average. The values presented in Table 1 reflect speedup values obtained this way. Table 2 shows the percentage of total number of nodes that are affected.

Similar experiments have been designed for measuring the performance of the proposed incremental closeness algorithm under shrinking network updates and the respective performance values are reported in Table 3. In the experiments performed for shrinking network updates, we start with the full network and incrementally remove the same set of edges used in the experiments whose performance values are reported in Table 1. Hence, we also have a way of comparing how different types of network updates affect the performance. The performance values in Table 1 and Table 3 describe the speedup obtained by the incremental closeness algorithm over computing closeness via repeated invocations of Dijkstra's algorithm averaged across 100 updates on the network. Considering the results presented in Table 1 – 4, several observations are in order. Firstly, the performance improvement obtained over traditional computation methods is less with shrinking network updates than the growing network updates although the number of affected nodes is lower on average for the shrinking update types. Since *DELETEUPDATESHRINKING* maintains a priority queue while *INSERTUPDATEGROWING* does not, the overall algorithmic complexity and the actual execution time are higher in shrinking network updates.

Table 1 - Performance improvements obtained on networks with different topologies/sizes (Using InsertEdgeGrowing and InsertUpdateGrowing over repeated invocations of Dijkstra's algorithm).

#(Nodes)	Preferential Attachment	Erdos-Renyi	Small World
1000	900x	123.07 x	288.97 x
3000	16732.48 x	515.35 x	1093.86 x
5000	47738.81 x	890.56 x	2228.91 x

Table 2 - Percentage of affected nodes in incremental network updates: union of nodes in AffectedSinks and AffectedSources.

#(Nodes)	Preferential Attachment	Erdos-Renyi	Small World
1000	3.79%	60.86%	28.38%
3000	2.13%	69.02%	30.69%
5000	1.33%	70.66%	27.26%

Table 3 - Performance improvements obtained on networks with different topologies/sizes (Using DeleteEdgeShrinking and DeleteUpdateShrinking over repeated invocations of Dijkstra's algorithm).

#(Nodes)	Preferential Attachment	Erdos-Renyi	Small World
1000	467.70 x	58.37 x	121.75 x
3000	2852.33 x	169.83 x	290.16 x
5000	10150.29 x	304.44 x	586.36 x

Table 4 - Percentage of affected nodes in decremental network updates: union of nodes in AffectedSinks and AffectedSources.

#(Nodes)	PreferentialAttachment	Erdos-Renyi	Small World
1000	2.93%	43.71%	22.45%
3000	1.51%	41.31%	25.71%
5000	0.97%	41.21%	22.91%

Table 5 - Network statistics. There are additional metrics that are measured but not listed. For all the networks laid out in this table Network Fragmentation is 0, and Avg. Deg. is 6.

Topology	Size	Min Deg	Max Deg	Degree StdDev	Diameter	Avg Path Len	Clustering Coeff
Pref. Attach.	1000	3	89	6.822	10	3.45	0.014
Pref. Attach.	3000	3	233	8.064	14	4.126	0.007
Pref. Attach.	5000	3	212	8.251	16	4.442	0.005
Erdos-Renyi	1000	1	14	2.498	15	6.305	0.003
Erdos-Renyi	3000	2	13	1.572	14	7.086	0.001
Erdos-Renyi	5000	2	11	1.362	14	7.492	0.001
Small World	1000	3	12	1.545	33	7.612	0.044
Small World	3000	3	12	1.526	55	10.333	0.039
Small World	5000	3	4	1.502	71	11.934	0.039

Secondly, incremental closeness algorithm performs best with networks that are generated following preferential attachment network generation model both for growing and shrinking updates. Comparing the speedup obtained on different networks, speedup obtained using the incremental closeness algorithm increases with the increased network size. It is also observed that other parameters shown in Table 5 such as network diameter and characteristic path length are inversely related with the performance obtained. For instance, in networks generated using a preferential attachment generation model, characteristic path length and diameter are lower compared to other topologies. When the paths in a network are short, an update on the shortest paths cannot propagate very far, resulting in quick return from the update and a very limited number of affected nodes (e.g. less than 5% in the case of preferential attachment networks). To elaborate, when the shortest paths in a network are not long, there are fewer nodes that lie on the shortest paths (i.e. fewer nodes are affected) and the overall depth of the shortest path tree is shorter, which also results in fewer updates when there is need for reconstruction of the shortest paths in the network.

Building Large Networks Incrementally: One important point about larger networks is that it takes too long for them to compute measures and cope with the dynamism. And, the performance improvements that were obtained on networks with short average/characteristic path length are substantial (See performance results obtained on preferential attachment networks presented in Table 3). One key use of incremental algorithms for larger networks is that they can substitute for the traditional closeness algorithms for computing centralities for the very large network even after all the updates in the network are final. In other words, incremental closeness algorithms support both on-the-fly computation and faster computation of the metric on large, static networks.

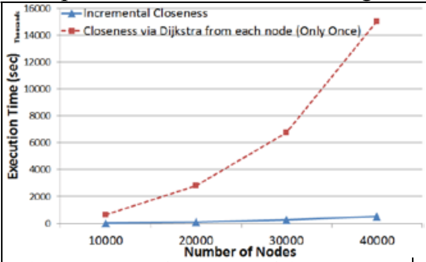

Figure 2 - Performance of incremental closeness computation on large preferential attachment networks. The Dijkstra's algorithm executed only once after all updates completed.

Table 6 - Performance improvements for the results in Figure 2.

|N|	|E|	Performance
10,000	30,000	14.78
20,000	60,000	24.17
30,000	90,000	26.28
40,000	120,000	29.53

We support this claim by the results presented in Figure 2, in which traditional closeness centrality computation is executed *only* once instead of modeling the growth of the network. In other words, we pretend to build the network incrementally by invoking the *INSERTEDGEGROWING* once for each edge in the network and compare its performance against computing closeness centrality by running Dijkstra *only* once from each node in the network on the final version of the network. Since many real-life, large-scale networks exhibit scale-free behavior and the best performance improvements with incremental closeness centrality are obtained on preferential attachment networks. The performance benefits in this case are primarily governed by the network structure, which contains relatively short shortest paths compared to other network topologies. In the results presented in Figure 2 and Table 6, we use four different preferential attachment networks where the number of nodes is varied between 10,000 and 40,000 with a step size of 10,000 and we set the average degree to 6.

C. *Performance Results with Real-Life Networks*

Next, we evaluate the performance of incremental closeness algorithm on a variety of real-life networks that are of different scale and have different topological features. The real life networks used in our evaluations in this section are prepared as weighted networks where the cost of an edge is inversely proportional to the strength of the relationship it is modeling. We obtain the edge costs by consolidating multiple updates for the same pair of nodes in a single edge. For instance, if a communication or interaction from node x to y has been recorded twice up to a certain point, then the edge $x \rightarrow y$ has the cost of 1/2. When a new, third communication or interaction is observed from node x to y, then the cost of the edge $x \rightarrow y$ is updated to be 1/3.

Next, we report our performance results and explain them in line with the topological features of the networks described above. For the real life network, in order to test how the two algorithms (incremental closeness algorithm vs. traditional closeness computation using Dijkstra's algorithm) perform with the growing network updates, we start with an earlier version of the network that has all the information expect the last 100 updates. Similarly, to obtain performance results with shrinking network updates, we start with the final version of the network and remove the same set of 100 edges that we have used in our runs with growing network updates.

Table 7 - Performance improvements of incremental closeness centrality over computing closeness centrality using Dijkstra's algorithm. Information on the affected portion of the network is also provided.

Network	Growing Speedup	Growing Affected%	Shrinking Speedup	Shrinking Affected%
SocioPatterns [20]	10.68	38.57	39.36	5.92
OnlineForum [21]	241.65	13.84	122.34	4.74
P2P [22]	173505.1	0.026	19694.11	0.026
HEPCo-authorship [23]	8101.75	9.99	3133.75	8.41

Table 8 - Topological features of real-life networks. Corresponding performance results are presented in Table 8.

	N	E	Avg Deg	Max Deg	Std. Dev. Deg	Diameter	Avg Path Len.	Clus Coef
SocioPatterns	113	4392	38.8	98	18.3	3	1.656	0.53
OnlineForum	1897	20290	21.4	339	35.6	8	3.196	0.08
HEPCoauthor	7507	38804	5.16	64	6.14	15	5.742	0.45
P2P Comm.	6843	7572	2.21	2185	38.3	3	1.248	0

In general, similar to the results obtained on synthetic networks, performance benefits of the proposed incremental closeness algorithm increase with the increasing network size and the growing network updates return higher benefits than the corresponding shrinking network updates. However, how much performance improvement can be obtained is also a factor of the changes made to the network, the portion of the network that is affected as well as the structure of the shortest paths in the network.

For instance, in P2P file transfer network, there are very few nodes that serve files for download to the other users, and the majority of the network consist of users that do not share files and are only there for downloading file that are of interest for them. Hence, the shortest paths in this network are very short. The average shortest path length is 1.24, which is slightly more than a single hop, which reflects as an enormous speedup that is obtained over the non-incremental algorithm. So, the majority of the edges are on the shortest paths for the nodes they are connecting. Hence, the speedup that can be obtained on such a network in the case of shrinking networks is lower, as it is also shown in Table 7. Because most of the time we remove an edge $\{x \rightarrow y\}$, we actually make a change on the shortest paths, and the algorithm probes all other neighbors of node x to see if it can find another path to y and which one is the shortest if any. Given that the network has hubs with very high degree centrality (max degree = 2185), probing for the new shortest path might take longer when an edge is removed than it takes when an edge is inserted.

Another interesting observation comes from the SocioPatterns network. This network is a small network and it reflects the interactions between the attendees of a conference. This group of people constitutes a relatively close-knit group, with high transitivity and a global clustering coefficient of 0.534. Transitivity refers to the probability of two nodes i and k being connected given that there exist an edge (i, j) and (j, k), and this is a kind of behavior one would expect to observe in conference-like environment. When we insert new edges in the SocioPatterns network a substantial portion of the network (38.57%) is affected. However, when we start removing edges, the network starts becoming partitioned into several disconnected components, and the portion affected by the shrinking network updates is substantially lower (5.92%). Therefore, although the *DELETEEDGESHRINKING* has higher complexity compared to the *INSERTEDGEGROWING* algorithm, the performance improvement obtained on shrinking network updates becomes higher in the SocioPatterns network.

Finally, the experiments are run separately for the growing and shrinking network updates to measure the performance across different update types and to quantitatively measure the performance differences. However, the algorithms cover the most generic case where network updates can be issued in any order.

V. CONCLUSIONS AND FUTURE WORK

This paper proposes incremental algorithms for computing closeness centrality in dynamic social networks. In general, the incremental algorithm proposed in this paper increases the speed with which closeness centrality can be calculated for all nodes in a network. The performance gain increases with the size of the network. The ability to calculate closeness centrality incrementally means that we can use these metrics to quickly identify over-time change and to set up alerts. A second use of this metric is to use it on a large static network. By "pretending" that the network is being built incrementally we can apply this algorithm and calculate closeness more quickly than we can using the non-incremental algorithm. In conclusion, incremental algorithm design offers the potential to allow dynamic social network analysis to be applied to real time data, and to much larger datasets than would have been possible using traditional centrality metric computations.

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.

VII. REFERENCES

[1] P. Bonacich, "Power and centrality: A family of measures," *American Journal of Sociology,* vol. 92, no. 5, pp. 1170--1182, March 1987.

[2] G. Sabidussi, "The centrality index of a graph," *Psychometrika,* vol. 31, no. 4, pp. 581--603, 1966.

[3] L. C. Freeman, "A Set of Measures of Centrality based on Betweenness," *Sociometry,* pp. 35-41, 1977.

[4] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik,* vol. 1, no. 1, pp. 269--271, 11 June 1959.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, 2nd Edition ed., Cambridge, MA: MIT Press, 2001, pp. 595-601.

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

[7] S. Nanda and D. Kotz, "Social Network Analysis Plugin (SNAP) for Mesh Networks," in *Wireless Communications and Networking Conference (WCNC)*, 2011.

[8] U. Brandes, P. Kenis and D. Wagner, "Communicating Centrality in Policy Network Drawings," *Transactions on Visualization and Computer Graphics,* vol. 9, no. 2, pp. 241-253, 2003.

[9] R. Yang and L. Zhuhadar, "Extensions of closeness centrality?," in *Proceedings of the 49th Annual Southeast Regional Conference*, Kennesaw, GA, 2011.

[10] K. Okamoto, W. Chen and X. Y. Li, "Ranking of closeness centrality for large-scale social networks," in *Proceedings of the 2nd International Frontiers of Algorithmics Workshop (FAW)*, Changsha, China, 2008.

[11] D. Eppstein and J. Wang, "Fast approximation of centrality," in *Proceedings of the twelfth annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Washington, D.C., United States, 2001.

[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.

[13] G. Ramalingam and T. Reps, "On the Computational Complexity of Incremental Algorithms," Madison, 1991.

[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, "GraphStream," 2010. [Online]. Available: http://graphstream-project.org/. [Accessed 3 February 2012].

[17] A. 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. W. a. S. Strogatz, "Collective Dynamics of 'Small-World' Networks," *Nature,* vol. 393, 1998.

[20] SocioPattern Project, "Hypertext 2009 Dynamic Contact Network Dataset," 2009. [Online]. Available: http://www.sociopatterns.org/datasets/hypertext-2009-dynamic-contact-network/.

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

[22] UMass Amherst, "Dataset: Can-o-sleep," 21 May 2003. [Online]. Available: http://kdl.cs.umass.edu/proximity/index.html. [Accessed 21 April 2012].

[23] 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.