Soc. Netw. Anal. Min. (2014) 4:220
DOI 10.1007/s13278-014-0220-6

ORIGINAL ARTICLE

Efficient algorithms for incremental all pairs shortest paths, closeness and betweenness in social network analysis

Sushant S. Khopkar · Rakesh Nagi · Alexander G. Nikolaev · Vaibhav Bhembre

Received: 10 March 2014 / Revised: 20 June 2014 / Accepted: 10 July 2014 / Published online: 3 August 2014

Abstract One of the biggest challenges in today's social network analysis (SNA) is handling dynamic data. Real-world social networks evolve with time, forcing their corresponding graph representations to dynamically update by addition or deletion of edges/nodes. Consequently, a researcher is often interested in fast recomputation of important SNA metrics pertaining to a network. Recomputations of SNA metrics are expensive. Use of dynamic algorithms has been found as a solution to this problem. For calculating closeness and betweenness centrality metrics, computations of all pairs shortest paths (APSP) are needed. Thus, to compute these SNA metrics dynamically, APSP are needed to be computed dynamically. This paper presents fast incremental updating algorithms along with the time complexity results for APSP, closeness centrality and betweenness centrality, considering two distinct cases: edge addition and node addition. The following time complexity results are presented: (1) The incremental APSP algorithm runs in $O(n^2)$ time ($\Omega(n^2)$ is the theoretical lower bound of the APSP problem), (2) The incremental closeness algorithm that runs in $O(n^2)$ time, and (3) The incremental betweenness algorithm runs in $O(nm + n^2 \log n)$ time. Here, m is the number of edges and n is the number of nodes in the network. Though the time complexity of the presented incremental betweenness algorithm is no better than its static counterpart (Brandes, J Math Sociol 25(2):163–177, 2001), the experimental comparisons demonstrate that the former performs better than the latter. All the presented methods are applicable to weighted, directed or undirected graphs with non-negative real-valued edge weights. An alternate version of the incremental APSP algorithm is presented in the Appendix section. It is demonstrated that this version works better on large graphs.

Keywords Incremental graph algorithms · Centrality metrics · APSP · Social network analysis

1 Introduction

A social group or community is often represented by analysts as a graph or a network comprised of nodes and edges. Nodes (sometimes also referred to as vertices) commonly represent people, enterprise departments, organizations, etc., while edges (also referred to as links) connecting the nodes represent relationships between the nodes.

Various social network analysis (SNA) parameters, or metrics, have been designed to study characteristics of networks, with centrality metrics among the most widely used ones. Centrality metrics help to identify network nodes that are more important or central compared to other nodes (Knoke and Yang 2008; Bavelas 1948). Centrality has been used to study social influence in inter-organizational networks (Laumann and Pappi 1976), assess political power (Burt 1995), compare employment opportunities (Mark 1973) and quantify web page popularity (Kleinberg 1999).

S. S. Khopkar (✉) · A. G. Nikolaev · V. Bhembre
Department of Industrial and Systems Engineering, State University of New York at Buffalo, Buffalo, NY 14260, USA
e-mail: skhopkar@buffalo.edu

A. G. Nikolaev
e-mail: anikolae@buffalo.edu

R. Nagi
Department of Industrial and Enterprise Systems Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
e-mail: nagi@illinois.edu

Closeness centrality and betweenness centrality (Freeman 1977) are two of the most significant and widely used SNA metrics. Closeness centrality can be regarded as a measure of how fast the spread of information can occur starting from a given node to all other nodes in the network (Newman 2005), while betweenness centrality is a measure of the extent, to which a node participates in or controls information transfer over the network, based on the number of shortest paths between network nodes on which a selected node lies.

Closeness centrality is calculated as the inverse of the sum of the shortest path distances between a node 'i' and the remaining '$n-1$' nodes in a network of size n.

Thus, the node i's closeness centrality, $C_C(i)$ is given by:

$$C_C(i) = \frac{1}{\sum_{j=1, i\neq j}^{n} d_{ij}} \quad (1)$$

where, d_{ij} is the distance between nodes i and j taken along a shortest path. Note that, this definition of closeness centrality is applicable only to those nodes that form a connected graph. An alternate definition of closeness centrality is proposed in Opsahl et al. (2010), which is applicable to disconnected graphs as well.

According to the original definition, the betweenness centrality value for a fixed node 'i' in a given network is found as follows (Freeman 1979). Consider every pair of nodes, 'j' and 'k', in the network. Let σ_{jk} denote the number of shortest paths between nodes 'j' and 'k', and let $\sigma_{jk}(i)$ denote the number of shortest paths between nodes 'j' and 'k' that involve node 'i'. Then, betweenness centrality of node 'i' is given by:

$$C_B(i) = \sum_{j,k(j\neq i, k\neq i)} \frac{\sigma_{jk}(i)}{\sigma_{jk}}. \quad (2)$$

It can be observed from Eqs. (1) and (2) that computations of both, closeness and betweenness centrality metrics need values of all pairs shortest paths (APSP) first. Algorithms for computing APSP, and in turn closeness and betweenness centrality are well developed. But, those are static algorithms. If a graph changes too frequently by additions of new nodes and edges, then recomputing APSP distances and centrality metrics is computationally expensive. The fastest known betweenness centrality algorithm executed for all the nodes in a weighted static graph requires $O(nm + n^2 \log n)$ time, where m is the number of edges and n is the number of nodes in the network (Brandes 2001). It is referred as "Brandes' algorithm".

Dynamic graph algorithms overcome this difficulty by avoiding recomputations. They are defined as algorithms "that maintain some property of a changing graph more efficiently than recomputation from scratch after change" (Eppstein et al. 1997).

A dynamic graph algorithm modifies a known data set solution to reflect changes in the data set of the previous state. It does not apply the algorithm which was used for the original data set. But once the dynamic graph algorithm is applied to the original data set, the same algorithm is repeated for each state of data formed by the change in it. That is, if the solution for the original data set is computed at a time t_0 and if the data set is changed at time t_1, then the changes in the solution would be reflected at time t_2 without recomputing the entire solution from scratch and without applying the same algorithm used before time t_0. Further, if the data set is again changed at time t_3, then these changes would get reflected at time t_4 by applying the dynamic algorithm which was used after time t_1.

Dynamic graph algorithms are further classified as 'partially dynamic algorithms' and 'fully dynamic algorithms'.

1. *Partially dynamic algorithms:* A partially dynamic algorithm is the one in which only one type of update is involved. This update could either be an insertion or deletion of nodes and/or edges. Partially dynamic algorithms involving insertion updates are called incremental algorithms; while those which involve deletion are called decremental algorithms.
2. *Fully dynamic algorithms:* Fully dynamic algorithms involve insertion as well as deletion of nodes and/or edges. This is composed of two largely independent sub-algorithms. One is for node and/or edge addition and the other is for node and/or edge deletion purpose.

This paper presents partially dynamic algorithms for APSP distances, closeness centrality and betweenness centrality, and which can handle a node addition and an edge addition operations on general directed and undirected graphs with real-valued edge weights. Since these algorithms handle addition operations, they are also called as incremental algorithms.

2 Literature review

Dynamic shortest path algorithms update the information of shortest paths in the network. These algorithms have applications in many areas including transportation networks, operating systems, information systems, VLSI (Very-large-scale integration) design, CAD (computer aided design) and traffic optimization.

The dynamic maintenance of shortest paths has a significantly long history. The first articles on it were published over 47 years ago (Loubal 1967; Murchland 1967; Rodionov 1968). Many shortest path algorithms have been

proposed in later years (King 2002; Ramalingam and Reps 1996a, b).

Many dynamic algorithms have some types of restrictions on them. Ausiello et al. (1991) proposed an edge insertion algorithm for directed graphs having positive integer weights less than 'C'. Algorithm proposed by Lin and Chang (1991) handles edge insertion operations, but all edges need to be of equal length. Algorithm proposed by Westbrook and Tarjan (1992) works on undirected graphs. Henzinger et al. (1997) designed a fully dynamic algorithm for APSP on planar graphs with integer weights. King (2002) proposed a fully dynamic algorithm that works on directed graphs having edge weight less than 'C'. Demetrescu and Italiano (2001) proposed a fully dynamic algorithm on general directed graphs with real weights with a restriction that an edge can take 'S' different values and the performance of their algorithm was dependent on 'S'.

Our algorithms are partially dynamic algorithm which can handle a node addition and an edge addition operations on general directed and undirected graphs with real-valued edge weights. There are no restrictions on the type of graph, maximum weight an edge can have and on number of different values an edge can take.

To the best of our knowledge, latest and the most general fully dynamic algorithm for directed graphs with non-negative real-valued edge weights that support any sequence of operations is designed by Demetrescu and Italiano (2004). It takes $O(n^2\log^3 n)$ amortized time per update and unit worst-case time per distance query. Thorup (2004) has improved this theoretical bound to $O(n^2(\log n + \log^2(\bar{m}/n)))$ amortized time per update. Here m is the number of edges and $\bar{m}$ is $m + n$.

In the same paper (Demetrescu and Italiano 2004), an incremental algorithm with running time complexity of $O(n^2)$ is presented under the name of 'Decrease-only' algorithm. This is the most recent and efficient incremental algorithm having minimal restrictions, that we are aware of. We compare our incremental algorithms with the incremental algorithm in Demetrescu and Italiano (2004), which we will refer to as ' DI incremental algorithm' in this paper. Recently, researches have started designing dynamic algorithms to update centrality metrics. Green et al. (2012) proposed an algorithm that updates betweenness centrality values upon addition of an edge. However, this algorithm works only on unweighted graphs. The time complexity of this algorithm is the same as that of Brandes' algorithm (Brandes 2001), but its space complexity is higher than Brandes' algorithm by a factor of n (number of nodes). Our algorithms, however, have the same time and space complexity as that of Brandes' algorithm, and they can handle weighted graphs having non-negative real-valued edge weights. Lee et al. (2012) proposed the QUBE framework to update betweenness centrality values of nodes upon addition or deletion of an edge. It identifies nodes whose betweenness values can be changed and efficiently recomputes those values. Due to inherent nature of this algorithm, it only performs significantly well on graphs with a tree-like structure having many small biconnected components. In Lee et al. (2012), the algorithm is demonstrated on undirected graphs and the complexity analysis is not provided. On the other hand, the algorithms proposed in this paper work on directed graphs as well. We also propose an incremental betweenness algorithm for addition of a node case, which is a novel contribution. Several metrics similar to betweenness centrality have recently been proposed for weighted graphs with real-valued edge weights (Brandes 2001; Opsahl et al. 2010). The algorithms presented in this paper can be easily adapted to handle dynamic recomputations of those metrics as well.

Section 3 describes the proposed incremental APSP algorithms, which involve a node addition and an edge addition algorithm with time complexity analysis. Section 4 explains how an incremental closeness algorithm can be designed with the help of the proposed incremental APSP node addition algorithm. Section 5 explores how the betweenness centrality values in a given graph can be updated for two of those types, viz., 'addition of an edge' and 'addition of a node', and presents a dynamic betweenness update algorithms. Section 6 provides experimental comparisons of the incremental APSP algorithms with the best known algorithm so far (Demetrescu and Italiano 2004) and of the proposed incremental betweenness algorithm with its best known static counterpart (Brandes 2001). The paper is concluded in Sect. 7 and future research directions have been discussed. A modified version of incremental APSP algorithm is provided in the Appendix section, which works better on large graphs.

3 Incremental APSP algorithms

3.1 Incremental APSP algorithm for node addition

This algorithm quickly updates all pairs shortest paths (APSP), following an addition of a new node to a network. The node addition implies that the edges connecting it to existing network nodes are added as well. This is a partially dynamic graph algorithm and it works on general directed and undirected graphs having real-valued edge weights. The algorithm is based on the concept that, if any shortest path passes through the newly added node in the graph, then it must first pass through at least one of its neighbors, which were already present in the original graph (see Fig. 1). The algorithm leverages its performance by taking

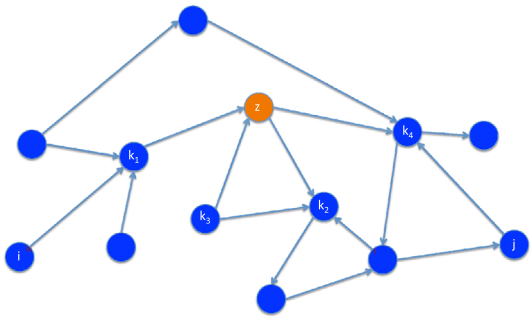

Fig. 1 Newly added node z shown with neighbors k_1, k_2, k_3 and k_4

help of the knowledge of all pairs shortest paths in the original graph to calculate the new shortest paths. The newly found shortest paths are then compared with the old ones to obtain the updated shortest paths.

1. *Calculate lengths of new paths to the new node:* Add a new node 'z' in the original graph of n nodes. Identify its neighbors. Let T_1 be the set of all incoming neighbors of z. There can be at most n incoming neighbors. Let T_2 be the set of all outgoing neighbors of z. There can be at most n outgoing neighbors. Let t_1 be the cardinality of set T_1 and t_2 be the cardinality of set T_2. Let k^{in} denote an incoming neighbor of the newly added node z (e.g., k_1 or k_3 in Fig. 1). Let $d_{ik^{\text{in}}}$ denote the length of the shortest path between nodes i and k^{in}. Note that this shortest path length is already available, having been calculated for the old network. Let $d_{iz}^{k^{\text{in}}}$ denote the length of the path from node i to node z that passes through k^{in}. Let $w_{k^{\text{in}},z}$ denote the weight of the edge between nodes k^{in} and z. Then, one has

$$d_{iz}^{k^{\text{in}}} = d_{ik^{\text{in}}} + w_{k^{\text{in}},z}.$$

Since multiple such nodes as k^{in} can exist in a network (e.g., in the presented example one has the case $k^{\text{in}} = k_1$ and the case $k^{\text{in}} = k_3$), this step is repeated for all incoming neighbors of the newly added node, i.e.: for i = 1 to n, for k^{in} = 1 to T_1,

$$d_{iz}^{k^{\text{in}}} = d_{ik^{\text{in}}} + w_{k^{\text{in}},z}. \tag{3}$$

2. *Calculate shortest path distances to the new node:* In this step, the shortest paths from each node in the original graph to the newly added node 'z' are located through one of its neighbors. This can be done by taking the minimum over all distances $d_{iz}^{k^{\text{in}}}$, for each i. We denote it by $\min_{k^{\text{in}}\in T_1}\{d_{iz}^{k^{\text{in}}}\}$. Note that $\min_{k^{\text{in}}\in T_1}\{d_{iz}^{k^{\text{in}}}\}$ is the shortest path distance between node i and node z that passes through one of the incoming neighbors of z.

3. *Calculate lengths of new paths from the new node:* Similarly, let k^{out} denote an outgoing neighbor of node z (e.g., in the presented example in Fig. 1, k^{out} could be k_2 or k_4). Let $d_{k^{\text{out}}j}$ denote the shortest distance between node k^{out} and node j. Note that this shortest path distance had already been calculated in the original (old) network. Let $d_{zj}^{k^{\text{out}}}$ denote the length of the path from node z to node j that passes through k^{out}. Then, one has

$$d_{zj}^{k^{\text{out}}} = w_{z,k^{\text{out}}} + d_{k^{\text{out}}j}.$$

Since multiple such nodes as k^{out} can exist in a network (e.g., in the presented example in Fig. 1, one has the case $k^{\text{out}} = k_2$ and the case $k^{\text{out}} = k_4$), this step is repeated for all outgoing neighbors of the newly added node, i.e., For j = 1 to n, for k^{out} = 1 to T_2,

$$d_{zj}^{k^{\text{out}}} = w_{z,k^{\text{out}}} + d_{k^{\text{out}}j}. \tag{4}$$

4. *Calculate shortest path distances from the new node:* In this step, the shortest paths from the newly added node 'z' to each node in the original graph are located through one of its neighbors. This can be done by taking the minimum over all distances $d_{zj}^{k^{\text{out}}}$, for each j. We denote it by $\min_{k^{\text{out}}\in T_2}\{d_{zj}^{k^{\text{out}}}\}$. Note that $\min_{k^{\text{out}}\in T_2}\{d_{zj}^{k^{\text{out}}}\}$ is the shortest path distance between node z and node j that passes through one of the outgoing neighbors of z.

5. *Update APSP distances:* Combination of $\min_{k^{\text{in}}\in T_1}\{d_{iz}^{k^{\text{in}}}\}$ and $\min_{k^{\text{out}}\in T_2}\{d_{zj}^{k^{\text{out}}}\}$ for any pair of nodes (i,j) gives rise to a newly formed shortest path from 'i' to 'j'. Hence, for any node pair (i,j), if the sum of the lengths of the shortest paths between node i and node z and between node z and node j is smaller than the original shortest path length between i and j (d_{ij}), replace the old shortest path distance with the new one. This step is repeated for each node pair (i,j) in the original graph. For $i = 1$ to n, for $j = 1$ to n $(i \neq j)$,

$$\text{If } \left(\min_{k^{\text{in}}\in T_1}\{d_{iz}^{k^{\text{in}}}\} + \min_{k^{\text{out}}\in T_2}\{d_{zj}^{k^{\text{out}}}\} < d_{ij}\right),$$
$$d_{ij} = \min_{k^{\text{in}}\in T_1}\{d_{iz}^{k^{\text{in}}}\} + \min_{k^{\text{out}}\in T_2}\{d_{zj}^{k^{\text{out}}}\}. \tag{5}$$

It can be observed that $\min_{k^{\text{in}}\in\{t_1\}}[d_{iz}^{k^{\text{in}}}]$ is the shortest path distance from any node i to z and $\min_{k^{\text{out}}\in\{t_2\}}[d_{zj}^{k^{\text{out}}}]$ is the shortest path distance from z to any node j. In this way, all shortest path distances can be updated. Algorithm 1 below shows the pseudocode of the algorithm.

Algorithm 1 Implementation of the Incremental APSP Algorithm for Node Addition

w_{ij} is edge-weight(i, j), $(i, j) \in$ E, E is a set of edges.
L_{ij} is the length of a path from node i to node j.
d_{ij} is the current shortest path length from node i to node j.
T_1 is a set of incoming neighbor nodes of the newly added node z.
T_2 is a set of outgoing neighbor nodes of the newly added node z.
$\min_{k^{in} \in T_1} \{d_{iz}^{k^{in}}\}$ is the minimum path length between i and z, $\forall i \in V$ // V is a set of all nodes.
$\min_{k^{out} \in T_2} \{d_{zj}^{k^{out}}\}$ is the minimum path length between z and j, $\forall j \in V$
$\min_{k^{in} \in T_1} \{d_{iz}^{k^{in}}\} \leftarrow [\,]$;
$\min_{k^{out} \in T_2} \{d_{zj}^{k^{out}}\} \leftarrow [\,]$;
for each $v \in V$ do
    for each $k^{in} \in T_1$ do
        if $v \neq k^{in}$ then do
            $L_{vz} \leftarrow d_{vk^{in}} + w_{k^{in}z}$; // Calculate path lengths from all nodes to node z.
            if $L_{vz} < \min_{k^{in} \in T_1} \{d_{iz}^{k^{in}}\}$ then do // $v = i$ in this case
                $\min_{k^{in} \in T_1} \{d_{iz}^{k^{in}}\} \leftarrow L_{vz}$; // Update minimum path length between i and z.
            end if
        end if
    end for
    for each $k_{out} \in T_2$ do
        if $v \neq k_{out}$ then do
            $L_{zv} \leftarrow w_{zk_{out}} + d_{k_{out}v}$; // Calculate path lengths from node z to all other nodes.
            if $L_{zv} < \min_{k^{out} \in T_2} \{d_{zj}^{k^{out}}\}$ then do // $v = j$ in this case
                $\min_{k^{out} \in T_2} \{d_{zj}^{k^{out}}\} \leftarrow L_{zv}$; //Update minimum path length between z and j.
            end if
        end if
    end for
end for
for each i in $\min_{k^{in} \in T_1} \{d_{iz}^{k^{in}}\}$
    for each j in $\min_{k^{out} \in T_2} \{d_{zj}^{k^{out}}\} (j \neq i)$
        if $\min_{k^{in} \in T_1} \{d_{iz}^{k^{in}}\} + \min_{k^{out} \in T_2} \{d_{zj}^{k^{out}}\} < d_{ij}$ then do
            $d_{ij} \leftarrow \min_{k^{in} \in T_1} \{d_{iz}^{k^{in}}\} + \min_{k^{out} \in T_2} \{d_{zj}^{k^{out}}\}$; // Compare the old shortest path with the new one and update it if the new shortest path is shorter.
        end if
    end for
end for
for each i in $\min_{k^{in} \in T_1} \{d_{iz}^{k^{in}}\}$
    $d_{iz} \leftarrow \min_{k^{in} \in T_1} \{d_{iz}^{k^{in}}\}$;
end for
for each j in $\min_{k^{out} \in T_2} \{d_{zj}^{k^{out}}\}$
    $d_{zj} \leftarrow \min_{k^{out} \in T_2} \{d_{zj}^{k^{out}}\}$;
end for

The algorithm described above updates the APSP distances. Once this is done, it becomes fairly straightforward to update the actual paths. If the condition in Eq. (5) is satisfied, the old shortest paths are discarded and replaced by the new shortest paths. Each new shortest path then passes through node z. As we have information about the actual paths between i to k^{in} and between k^{out} to j, the new shortest paths can be constructed by inserting the subpath '$k^{\text{in}} - z - k^{\text{out}}$' between them. Note that node k^{in} is the incoming neighbor of node z that lies on the shortest path between node i and node z, while k^{out} is the outgoing neighbor of node z that lies on the shortest path between node z and node j.

If the old shortest path distance is equal to the newly formed shortest path distance, then the new shortest paths are added to the list of old shortest paths for each pair of nodes.

3.2 Incremental APSP algorithm for edge addition

The incremental APSP algorithm for edge addition is based on the same principle as that of the incremental APSP algorithm for node addition. However, there are few differences. This section presents the incremental APSP algorithm for edge addition.

Assume that a new edge is added between a pair of nodes (k_1, k_2) in a given network.

Case 1 ($w_{k_1,k_2} > d_{k_1k_2}$): If the old shortest path distance between nodes k_1 and k_2 is smaller than the weight of the newly added edge, then the shortest path(s) between nodes k_1 and k_2 will not change. Also, the shortest paths between other nodes that were using shortest path(s) between nodes k_1 and k_2 as their subpath(s), will continue to use it (them), and hence there will be no change in the shortest paths between any pair of nodes in the network.

Case 2 ($w_{k_1,k_2} < d_{k_1k_2}$): If the weight of the newly added edge is less than the shortest path distance between nodes k_1 and k_2, then the shortest path distance between nodes k_1 and k_2 will change and other shortest path distances in the network may also change.

Let the newly added edge be denoted by $e(k_1, k_2)$ and its weight be denoted by w_{k_1,k_2}. Let the old shortest path distance between nodes k_1 and k_2 be denoted by $d_{k_1k_2}$, and assume that $w_{k_1,k_2} < d_{k_1k_2}$. Then, the APSP in the network can be updated as follows:

1. *Calculate* $d_{k_1k_2}$: The new shortest path between nodes k_1 and k_2 will be $e(k_1, k_2)$ and the shortest path distance will be $w_{k_1k_2}$, i.e., an update occurs:
$$d_{k_1k_2} = w_{k_1,k_2}. \quad (6)$$

2. *Calculate other shortest path distances:* For a pair of nodes (i,j), let d_{ik_1} denote the shortest path distance from node i to node k_1 and let d_{k_2j} denote the shortest path distance from k_2 to j. Check if
$$d_{ik_1} + w_{k_1,k_2} + d_{k_2j} < d_{ij}, \quad (7)$$
for every pair of nodes (i,j). If the above condition is true, we update the shortest path distance between nodes i and j as
$$d_{ij} = d_{ik_1} + w_{k_1,k_2} + d_{k_2j}. \quad (8)$$

Update shortest paths: The updated shortest path(s) between nodes i and j is (are) obtained by combining the old shortest path(s) between i and k_1, edge between k_1 and k_2 and the old shortest path(s) between k_2 and j.

Case 3 ($w_{k_1,k_2} = d_{k_1k_2}$): If the weight of the newly added edge is equal to the shortest path distance between nodes k_1 and k_2, then no shortest path distance will change, however, new alternative shortest paths will be formed between some pairs of nodes in the network.

Update shortest paths: In this case, one has $w_{k_1,k_2} = d_{k_1k_2}$. Consider a node pair (i,j). If m old shortest paths from i to j were using the shortest path from k_1 to k_2 as a subpath, then additional m shortest paths will be formed from i to j, with the old subpath from k_1 to k_2 replaced by the newly added edge.

3.3 Time complexity analysis

The time required to compute $d_{iz}^{k^{\text{in}}}$ for all nodes is $O(nt_1)$ (see Eq. 3). In the worst case $t_1 = n$, hence the time required becomes $O(n^2)$. Similarly, the time required to compute $d_{zj}^{k^{\text{out}}}$ for all nodes is $O(nt_2)$ (see Eq. 4). In the worst case $t_2 = n$, hence the time required becomes $O(n^2)$. In the last step of the algorithm, each newly found shortest path is compared against the corresponding old shortest paths (see Eq. 5). This takes $O(n^2)$. Thus, the running time complexity of the algorithm is $O(n^2)$.

It can be shown in a similar manner that the time complexity of the incremental APSP algorithm for edge addition is $O(n^2)$.

As mentioned in Demetrescu and Italiano (2004), this is the theoretical lower bound of the problem. A small illustration for explanation of this lower bound is as follows:

Consider an undirected graph of eight nodes in a form of an octagon as shown in Fig. 2a. Each edge in the graph has a weight of three units. Now, as shown in Fig. 2b, if a new node is added at the center of this graph, with eight new edges having weight of one unit each and connecting to all eight nodes in the original graph, then all the shortest paths present in the original graph change (Start passing through the new node.).

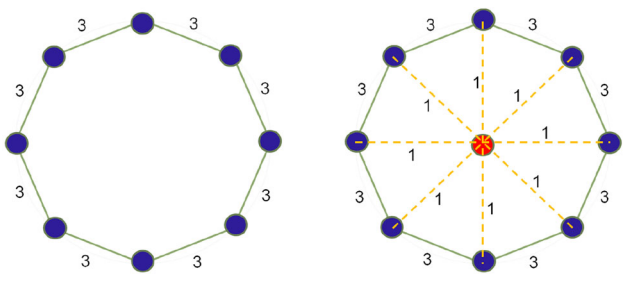

Fig. 2 An Example demonstrating theoretical lower bound of the APSP problem

Since in a graph of n nodes, each of the $n \times (n-1)$ shortest paths can get changed, there must be $n \times (n-1)$ updating operations in the distance matrix of shortest paths and hence the theoretical complexity is $\Omega(n^2)$.

4 Incremental algorithm for closeness centrality

This section discusses one of the applications of the incremental APSP algorithms in the field of social network analysis. Computations of all pairs shortest paths in a network are necessary step in calculating many important network measures or parameters in social network analysis. Closeness centrality is one of those metrics.

Out of various different centrality metrics, closeness centrality in particular can be regarded as a measure of how fast the spread of information can occur starting from a given node to all other nodes in the network (Newman 2005).

Closeness centrality is calculated as the inverse of the sum of the shortest path distances between a node 'i' and the remaining '$n-1$' nodes in a network of size n. Refer to Eq. (1) for mathematical definition of closeness centrality.

Here, we show that the closeness centrality can be computed dynamically upon addition of new nodes and new edges in a graph using the presented incremental APSP algorithms.

Once the shortest paths have been updated between all pairs of nodes in the new network (formed by the addition of a node or an edge) by either incremental APSP algorithm for node addition or by incremental APSP algorithm for edge addition, calculating closeness centralities as per Eq. (1) is a fairly easy task. It can be done in $O(n)$ additional time.

1. *Add shortest path distances:* In the distance matrix of shortest paths, each row represents shortest path distances originating from a particular node to each remaining node in the network. Sum all the elements in a row of the distance matrix. Follow the same procedure for each row in the matrix. This will give us sum of all the shortest path distances starting from a particular node, to all the nodes in the network. For $i = 1$ to $n+1$

$$d_i^{\text{sum}} = \sum_{j=1, i \neq j}^{n+1} d_{ij}.$$

This sum will give us $n+1$ different values (of n nodes in the original network and the newly added node); each value corresponds to each individual node in the network.

2. *Calculate closeness centralities* Take inverse of each of these $n+1$ values to get updated values of closeness centralities of all the nodes in the network.

$$C_{\text{C}}(i) = \frac{1}{\sum_{j=1, i \neq j}^{n+1} d_{ij}}, \quad \forall i.$$

Algorithm 2 shows the pseudocode of the algorithm.

Algorithm 2 Implementation of the Incremental Closeness Centrality

```
1: C_C[v] ← 0, v ∈ V;
2: S[v] ← 0, v ∈ V;
3: Compute D[i][j] from the incremental (node or edge addition) algorithm, i, j ∈ V and
   D[i][j] contains shortest path between 'i' and 'j'.
4: for i ∈ V do
5:     for j ∈ V do
6:         S[i] ← S[i] + D[i][j];
7:     end for
8: end for
9: C_C[i] ← 1/S[i];
```

5 Incremental algorithms for betweenness centrality

Betweenness centrality is another important and widely used centrality metric in social network analysis. It is a measure of the extent, to which a node participates in or controls information transfer over the network, based on the number of shortest paths between network nodes on which a selected node lies.

Unlike closeness centrality, which can be computed only with the help of shortest path distances, computation of betweenness centrality for all nodes requires the number of shortest paths between each pair of nodes and predecessors of each node on all shortest paths (if Brandes (2001)' algorithm is used), in addition to APSP distances. Below, we explain how to incrementally update betweenness centrality values for 'addition of a node' and 'addition of an edge' case. Refer to Eq. (2) for mathematical definition of betweenness centrality. Here, we assume that APSP have been updated using the presented incremental APSP algorithm.

5.1 Incremental betweenness algorithm for edge addition

With similar logic explained in Sect. 3.2, betweenness centrality values for nodes in the network will be updated whenever some shortest paths change, i.e., in Case 2, when $w_{k_1,k_2} < d_{k_1k_2}$, and Case 3, when $w_{k_1,k_2} = d_{k_1k_2}$. Note that Case 1, when $w_{k_1,k_2} > d_{k_1k_2}$ is not necessary to consider, because original shortest paths do not change.

Case 2: ($w_{k_1k_2} < d_{k_1k_2}$):

Brandes (2001) algorithm is the fastest known algorithm to compute betweenness centrality values of all nodes in a static graph. The running time complexity of this algorithm is $O(nm)$ for an unweighted graph and $O(nm + n^2\log n)$ for a weighted graph. There are two main steps in the Brandes' algorithm. The first step of the Brandes' algorithm involves calculating all shortest paths and the number of shortest paths between all node pairs by running Dijkstra's algorithm n times, as explained in step 1 ('Calculate number of shortest paths') of the algorithm below. The second step of the Brandes' algorithm involves accumulation of pair dependencies to get betweenness centrality values, as explained in step 4 ('Update betweenness centralities') of the algorithm below.

The betweenness centrality values are updated as follows:

1. *Calculate number of shortest paths:* Assume that Brandes' algorithm (Brandes 2001) has already been executed to compute betweenness centrality values for all the nodes in the old network. The first step of the Brandes' algorithm (Brandes 2001) is to run the Dijkstra's algorithm n times, treating each node as the source node once, to obtain APSP in a network. Apart from computing APSP, it is possible to count the number of shortest paths between every pair of nodes in the network with the same running time complexity. This procedure of counting number of shortest paths between every pair of nodes is described below: Observe the network in Fig. 3a. Suppose that Dijkstra's algorithm is applied to calculate a shortest path between source node 0 and node 5. Node 5 is the neighbor of node 1. So, scanning from node 1, the algorithm will assign label (1, 3) to node 5. The meaning of this label is that the current best shortest path from node 0 to node 5 has distance 3 and it passes through node 1, termed a predecessor. As the algorithm has already calculated a shortest path between node 0 and node 1, the entire shortest path from node 0 to node 5 can be reconstructed using the predecessor labels.
   Now, when the algorithm scans from node 2, it discovers that the shortest path from node 0 to node 5 passing through node 2 also has length 3. It is an alternative path in that the algorithm has already discovered a path of length 3 from node 0 to node 5 (via node 1), so the original Dijkstra's algorithm would not change the label of node 5. However, for betweenness centrality computations, one needs to keep track of the number of all the alternative paths, and hence, expanding a single node 5 label into a set of labels: (1, 3) and (2, 3). These labels store the information that two currently best shortest paths (of distance 3) have been discovered from node 0 to node 5. (Note that it is not necessary to store 2 different labels for node 5 here. Instead, a separate parameter in a single label indicating the number of shortest paths discovered can be created. Separate labels have been created in the Fig. 3a for ease of understanding). Next, observe the network in Fig. 3b, where the algorithm has advanced to begin scanning from node 3. Here, it discovers an even shorter path from node 0 to node 5, of length 2. Thus, all the previously set labels at node 5 (count of number of shortest paths) are removed, with a new label assigned, (3, 2). Then, scanning from node 4, the algorithm will discover another path of distance 2 from node 0 to node 5, and node 5 will expand its label set by the addition of label (4, 2). When the algorithm terminates, two shortest paths of distance 2 will be returned. Refer to this described algorithm as the modified version of Dijkstra's algorithm. The worst-case time complexity of the modified Dijkstra's algorithm executed on an entire network is the same as the worst-case time complexity of running standard Dijkstra's algorithm n times, which is $O(nm + n^2\log n)$.
   Note that the information about the betweenness centrality values, the number of shortest paths between all pairs of nodes and the actual shortest path distances are computed from scratch only for the original network, with the Brandes' algorithm executed only once.
2. *Update number of shortest paths:* For each pair of nodes in the network, it suffices to track whether the

shortest path distance between the pair is improved as a result of emergence of a new path or paths. For each node pair (i,j), whose shortest path distance must be updated, the updated number of shortest paths is calculated as follows. If there are m shortest paths between node i and node k_1, and n shortest paths between nodes k_2 and j, then the total number of new shortest paths between node i and j is found as the product of m and n.

3. *Update predecessors:* When Brandes' algorithm is applied to the original network, the actual nodes lying on each shortest path can be identified by storing predecessors. When a shortest path for every origin-destination pair of nodes (i,j) is updated, predecessors of node j can be updated as follows. Let $p\text{Set}_{ij}$ denote a set of predecessors of node j obtained by traversal from node i as the source node. If the shortest path distance between nodes i and j is reduced, the predecessors of node j (only) are changed in the current iteration, in the following manner. All the existing predecessors of node j are removed from set $p\text{Set}_{ij}$, and then, this set is filled with all the elements of the set $p\text{Set}_{k_2j}$. This is because, even though the original shortest paths from node i to node j are changed, the original shortest paths between node k_2 and node j have now become subpaths of a new shortest path between node i and node j. Hence, the predecessors of node j when i is the source node will now be the same as the predecessors of node j when node k_2 is the source node. Thus, nodes in $p\text{Set}_{k_2j}$ will be the predecessors of node j on a shortest path (or paths) starting from node i. If $j = k_2$, then node k_1 is added to $p\text{Set}_{ij}$. If a new shortest path of the same length is formed between node i and node j, then we do not remove nodes from $p\text{Set}_{ij}$. However, we add all nodes from $p\text{Set}_{k_2j}$ to $pSet_{ij}$. If $j = k_2$, then node k_1 is added to $p\text{Set}_{ij}$. This step is repeated for all node pairs for which, new shortest paths of same or lesser length have been formed between them by addition of the new edge.
4. *Update betweenness centralities:* To update the betweenness centrality values, the second step of the Brandes' algorithm can be used. In this step, the betweenness centrality value is computed by summing the so-called pair dependencies of all pairs on that node.

$$C_B(v) = \sum_{s \neq v \neq t \in V} \delta_{st}(v). \tag{9}$$

Here, V is a set of nodes in the graph and $\delta_{st}(v)$ is calculated as,

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}. \tag{10}$$

Here, σ_{st} is the total number of shortest paths between nodes s and t, while $\sigma_{st}(v)$ is the total number of shortest paths between nodes s and t that pass through node v. Brandes (2001) then defines a dependency of node s on a single node v as,

$$\delta_{s\cdot}(v) = \sum_{t \in V} \delta_{st}(v). \tag{11}$$

and discovers the following recursive relation for general directed acyclic graphs:

$$\delta_{s\cdot}(v) = \sum_{w : v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_{s\cdot}(w)). \tag{12}$$

Here, node v is a predecessor of node w and $P_s(w)$ is a set of predecessors of node w when s is the source node.

5. Once the count of shortest paths and predecessors on affected shortest paths is updated as explained above, the betweenness centrality values can be computed using Eqs. (9), (10), (11) and (12). Refer to Brandes (2001) for its detailed explanation.

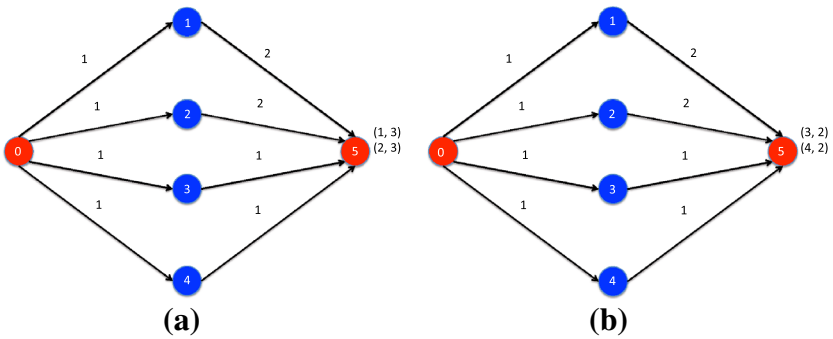

Fig. 3 Expanded label sets in the modified version of Dijkstra's algorithm

Case 3: ($w_{k_1k_2} = d_{k_1k_2}$):
The procedure to update the betweenness centrality is exactly the same as explained for Case 2 (when $w_{k_1,k_2} < d_{k_1k_2}$). The only difference is, while updating the number of shortest paths, we add the new number of shortest paths to the original number of shortest paths between each node pair, instead of deleting original number of shortest paths. Same is the case for updating predecessors. Consider node pair (i,j). If the shortest path between this node pair has changed, while updating predecessors of node j, we do not empty the predecessor set of node j when node i is the source node. But, we add predecessors of node j when the corresponding k_2 node is the source node into the same set.

5.2 Incremental betweenness algorithm for node addition

Once the APSP have been updated, the betweenness centralities are updated similarly to the update detailed for the 'addition of an edge' case. However, certain revisions in the procedure are required. In terms of updating the number of shortest paths and predecessors, those differences are as follows.

Updating number of shortest paths: First, we need to update the number of shortest paths from node i to node z and from node z to node j, for all i and j. If there are s_1 shortest paths from node i to node k^{in}, then there will be s_1 shortest paths from node i to node z, as there is a single edge between node k^{in} and node z. If multiple alternatives for k^{in} exist, then we add the number of multiple corresponding paths (of the form $i \rightarrow k^{\text{in}} \rightarrow z$) to the number of shortest paths from node i to node z. The same logic applies to the shortest paths from node z to node j (of the form $z \rightarrow k^{\text{out}} \rightarrow j$).

Now, if a shorter path is found between nodes i and j upon the addition of node z, then if there are m shortest paths from node i to node z and n shortest paths from node z to node j, then the new number of shortest paths between nodes i and j is found as $m \times n$.

If the new paths from node i to node j are just new alternate shortest paths (i.e., they are of the same length as the paths existing in the original network), then this number, $m \times n$, is added to the old count of shortest paths between node i and node j.

Updating predecessors: Recall that $p\text{Set}_{\text{ij}}$ denotes a set of predecessors of node j obtained by traversal from node i as the source node. When node z is the source node, add all elements of $p\text{Set}_{\text{k}^{\text{out}}\text{j}}$ to the set $p\text{Set}_{\text{zj}}$. There can be more than one k^{out} (k_2 and k_4). Repeat it for all such k^{out}. This step will update predecessors of all nodes, when node z (new node) is the source node.

When node z is the target node, for each source node i, add all corresponding k^{in} nodes to the predecessors' set $p\text{Set}_{\text{iz}}$.

If the shortest path distance between nodes i and j is reduced, the predecessors of the node j only are changed in the current iteration, in the following manner. All the elements are removed from set $p\text{Set}_{\text{ij}}$, and all elements of set $p\text{Set}_{\text{zj}}$ are added to set $p\text{Set}_{\text{ij}}$. This is because, even though the original shortest paths are changed from node i to node j, the shortest paths in the original network between node k^{out} and node j have now become subpaths of a new shortest path between node i and node j. Hence, the predecessors of node j when i is the source node will now be the same as the predecessors of node j when node z is the source node.

If a new shortest path of the same length is formed between node i and node j, then we do not remove predecessors of node j from set $p\text{Set}_{\text{ij}}$. However, we add all nodes from set $p\text{Set}_{\text{zj}}$ to set $p\text{Set}_{\text{ij}}$.

This step is repeated for all node pairs for which new shortest paths of same or lesser length have been formed because of the addition of the new node.

Algorithm 3 shows a pseudocode of the algorithm which updates APSP distances, number of shortest paths between each node pair and predecessors of all nodes when each node (including new node) is the source node.

Algorithm 3 Implementation of Updating Number of SPs and Predecessors for Node Addition

// Updating Number of Shortest Paths

σ_{ij} is the number of shortest paths from node i to node j.
$\sigma_{iz} \leftarrow 0$, $\sigma_{zj} \leftarrow 0$, $\sigma_{zz} \leftarrow 1$
for each i in $\underset{k^{in} \in T_1}{min} \{d_{iz}^{k^{in}}\}$
    for each k^{in} in $\underset{k^{in} \in T_1}{min} \{d_{iz}^{k^{in}}\}$
        $\sigma_{iz} \leftarrow \sigma_{iz} + \sigma_{ik^{in}}$ // Update number of SPs from i to z
    end for
end for
for each j in $\underset{k^{out} \in T_2}{min} \{d_{zj}^{k^{out}}\}$
    for each k^{out} in $\underset{k^{out} \in T_2}{min} \{d_{zj}^{k^{out}}\}$
        $\sigma_{zj} \leftarrow \sigma_{zj} + \sigma_{k^{out}j}$ // Update number of SPs from z to j
    end for
end for
for each i in $\underset{k^{in} \in T_1}{min} \{d_{iz}^{k^{in}}\}$
    for each j in $\underset{k^{out} \in T_2}{min} \{d_{zj}^{k^{out}}\} (j \neq i)$
        if $\underset{k^{in} \in T_1}{min} \{d_{iz}^{k^{in}}\} + \underset{k^{out} \in T_2}{min} \{d_{zj}^{k^{out}}\} < d_{ij}$ then do
            $\sigma_{ij} \leftarrow \sigma_{iz} \times \sigma_{zj}$ // Update number of SPs from i to j
        end if
        else if $\underset{k^{in} \in T_1}{min} \{d_{iz}^{k^{in}}\} + \underset{k^{out} \in T_2}{min} \{d_{zj}^{k^{out}}\} = d_{ij}$ then do
            $\sigma_{ij} \leftarrow \sigma_{ij} + (\sigma_{iz} \times \sigma_{zj})$ // Update number of SPs from i to j
        end if
    end for
end for

// Updating Predecessors

pSet_{ij} is a set of predecessors of j when i is the source.
predMap $\leftarrow$ Map(i, Map(j, pSet_{ij})) // This map is filled when Brande's algorithm is run on original graph.
for each j in $\underset{k^{out} \in T_2}{min} \{d_{zj}^{k^{out}}\}$
    for each k^{out} in $\underset{k^{out} \in T_2}{min} \{d_{zj}^{k^{out}}\}$
        $\text{pSet}_{zj} \leftarrow \text{pSet}_{zj}$.addAll(predMap.get($k^{out}$).get($j$)) //Update predecessors when z is the source node.
    end for
end for
for each i in $\underset{k^{in} \in T_1}{min} \{d_{iz}^{k^{in}}\}$
    for each k^{in} in $\underset{k^{in} \in T_1}{min} \{d_{iz}^{k^{in}}\}$
        $\text{pSet}_{iz} \leftarrow \text{pSet}_{iz}$.add($k^{in}$) //Update predecessors when z is the source node.
    end for
    predMap.get(i).put(z, pSet_{iz})
end for
for each i in $\underset{k^{in} \in T_1}{min} \{d_{iz}^{k^{in}}\}$
    for each j in $\underset{k^{out} \in T_2}{min} \{d_{zj}^{k^{out}}\} (j \neq i)$
        if $\underset{k^{in} \in T_1}{min} \{d_{iz}^{k^{in}}\} + \underset{k^{out} \in T_2}{min} \{d_{zj}^{k^{out}}\} < d_{ij}$ then do
            predMap.get(i).get(j).remove()
            $\text{pSet}_{ij} \leftarrow$ predMap.get(z).get(j)
            predMap.get(i).put(j, pSet_{ij})
        end if
        else if $\underset{k^{in} \in T_1}{min} \{d_{iz}^{k^{in}}\} + \underset{k^{out} \in T_2}{min} \{d_{zj}^{k^{out}}\} = d_{ij}$ then do
            $\text{pSet}_{ij} \leftarrow \text{pSet}_{ij}$ + predMap.get(z).get(j)
            predMap.get(i).put(j, pSet_{ij})
    end for
end for

5.3 Complexity analysis

Consider any node s as the source node. As shown in Brandes (2001), traverse the nodes in non-increasing order of their distance from s and accumulate dependencies according to Eq. (12). We need to store a dependency per node and list of predecessors. There is at most one node per edge in any of these lists. Hence, the dependencies of s on all other vertices can be computed in $O(m)$ time and $O(n + m)$ space. When this procedure is followed for all nodes, treating them as source nodes, the time complexity for calculating dependencies becomes $O(nm)$. Also, the time complexity of arranging the nodes in non-decreasing order for each source node considering the updated shortest path distances is $O(n^2 \log n)$. Hence, the total time complexity to update betweenness centrality values of all nodes in a network becomes $O(nm + n^2 \log n)$. Though this complexity is the same as that of Brandes (2001) algorithm, as we shall see in Sect. 6, the presented incremental betweenness algorithm runs faster.

6 Experimental comparison

For testing the effectiveness of the presented algorithms empirically, the incremental APSP algorithm for node addition and its modified version (see Sect. 8.1) is tested against the incremental algorithm proposed in Demetrescu and Italiano (2004). We refer to this algorithm as 'DI incremental algorithm'. The 'DI incremental algorithm' combines a single sink computation to the new node and a single source computation from the new node to find a new shortest path passing through the new node. The newly found shortest path distance is then compared against the corresponding old shortest path distance. This last step is similar to the incremental algorithms presented in this paper.

Though the theoretical complexity of 'DI incremental algorithm' and of incremental APSP algorithm for node addition is the same $O(n^2)$, the incremental APSP algorithm for node addition takes the maximum advantage of already computed APSP in the original network, instead of performing single source and single sink shortest path operations. The empirical comparison presented below demonstrates the difference in their performances.

A realistic application of the incremental APSP algorithms is in the field of social network analysis. Hence, for comparison of incremental algorithms three types of social graph generators are used, viz., Random graph generator, Small World generator and Preferential Attachment generator. These generators are taken from GraphStream project (Dutot et al. 2007), which is a Java-based library. The Random graph generator tries to generate nodes with random connections, with nodes having on an average a given degree. The average degree of 10 is considered for generating random graphs. The Small World generator is based on Watts and Strogatz (1998)'s model. It generates a ring of n nodes where each node is connected to its k nearest neighbors in the ring ($k/2$ on each side, which means k must be even). Each edge is then "rewired' with probability β, i.e, one end of the edge is detached and then attached to a randomly selected non-neighbor node from the ring. Graphs generated by this generator have $k = 30$ and $\beta = 0.01$. The Preferential Attachment generator generates Scale-free graphs using the preferential attachment rule given by Barabási and Albert (1999). Using the three generators, various graphs of sizes ranging from 1,000 nodes to 20,000 nodes are generated for the purpose of testing the incremental APSP algorithms. Weights ranging from 1 to 10 are randomly assigned to all the edges.

When a new member joins a social group, the number of connections he/she makes in an initial period varies. This initial period could be the first day, first week or first month of joining. These initial connections of the newly added node to nodes in the network are considered as neighbors of the newly added node. Marlow (2009), of Facebook data team, shows that a core network of a person is very small relative to his/her total friends' network. Here, we make a mild assumption that, when a person joins a network, his/her neighbors in the initial period are the nodes of his/her core network. Hence, when a new node is added to a network its number of incoming and outgoing neighbors are assumed to be two randomly selected numbers between 0 and 10. Once the two random numbers are generated, those many neighbors are randomly selected from the original graph.

Figures 4, 5 and 6 show the comparison of the 'DI incremental algorithm' (Demetrescu and Italiano 2004) with the 'Incremental APSP algorithm for node addition' presented in this paper and with its modified version (The 'Modified incremental APSP algorithm for node addition') on graphs of sizes ranging from 1,000 to 20,000 nodes.

In Figs. 4, 5 and 6, 'x-axis' represents the number of nodes in the graph and its values range from 1,000 to 20,000, while 'y-axis' represents the time in milliseconds required to update the shortest paths between all pairs of nodes in the graph by each algorithm.

Figures 4, 5 and 6 clearly show that both the algorithms presented in this paper perform better than the 'DI incremental algorithm'. For a graph of 20,000 nodes, the 'Modified Incremental APSP algorithm for node addition' runs approximately 10–20 times faster than the 'DI incremental algorithm'. Also, the difference of time required for computation of all pairs shortest path, between the two

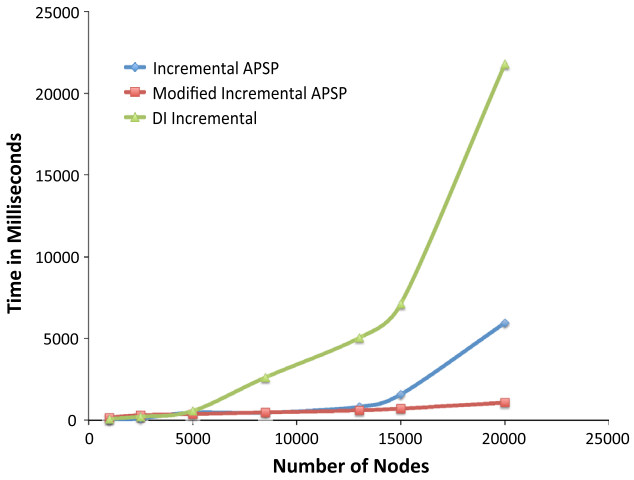

Fig. 4 Comparison of incremental algorithms on random graphs

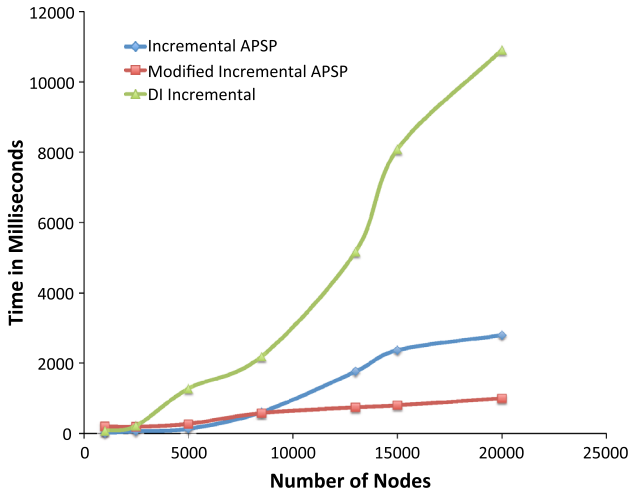

Fig. 5 Comparison of incremental algorithms on small world graphs

algorithms, increases with the increase in number of nodes (see Table 1 for the values). The 'Incremental APSP Algorithm for Node Addition' performs better than the 'Modified Incremental APSP Algorithm for Node Addition' when the graph size is small. However, as the graph size increases, the 'Modified Incremental APSP Algorithm for Node Addition' starts performing better than the 'Incremental APSP Algorithm for Node Addition' algorithm and the performance gap increases with the graph size. Thus, a good strategy would be to use the 'Modified Incremental APSP Algorithm for Node Addition' if the graph size is beyond 10,000 nodes. It can be also observed from the nature of the performance curves of the three incremental algorithms that their performance is robust across all the types of graphs considered.

Though the incremental APSP algorithms are designed by considering 'Social Networks' as their application area,

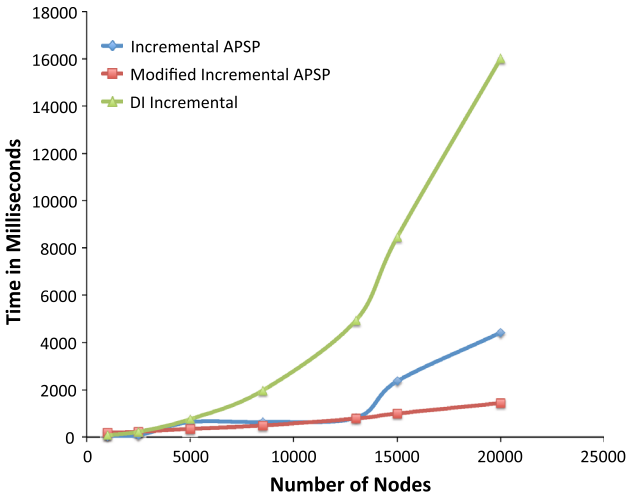

Fig. 6 Comparison of incremental algorithms on preferential attachment graphs

they can be used on any types of graphs. Hence, the newly added node can have large number of incoming and outgoing neighbors. To see whether the performance of the three incremental algorithm can get affected by the number of neighbors of the newly added node, we have tested the three algorithms on the three types of graphs (random, small world and preferential attachment), each with graph sizes 5,000 and 15,000 nodes, with the newly added node having 90–100 neighbors. As shown in Table 2, It is found that the results are consistent with the tests when the number of incoming and outgoing neighbors varies between 0 and 10.

The benchmarks for all the three incremental APSP algorithms were measured on a 8-core compute node with 2.27 GHz clock rate and 24 GB RAM. The algorithms were implemented in Java programming language using 'javac (1.7.0_11)' compiler.

As the designed incremental betweenness algorithm has the same worst case complexity as the fastest known static algorithm (Brandes 2001), it was interesting to compare the two algorithms experimentally.

Synthetic graphs (base graphs) are generated by the graph generator "RTG" (Akoglu and Faloutsos 2009), which claims to adhere to all 11 static and dynamic laws that realistic graphs are known to obey. The two algorithms are tested on graphs with sizes ranging from 101 to 1,764. All the base graphs and their corresponding increments (a new node with incoming and outgoing edges) are generated as follows.

A graph of $n+1$ nodes is generated using the RTG generator. A node with a total degree (outgoing plus incoming) of approximately 10 is removed from the graph with its attached edges: it is then treated as an increment. The remaining graph of n nodes is treated as the base

Table 1 Comparison of Incremental Algorithms

Number of nodes	Time in milliseconds								
	Random graphs			Small world graphs			Preferential attachment graphs		
	A	B	C	A	B	C	A	B	C
1,000	17.25	168.75	76.75	16	199	72.75	37.75	185.5	78.25
2,500	110.25	312.5	216.25	64.25	187.75	221	75.5	227.75	225.5
5,000	467.25	370	566.25	132	272.5	1,275.25	633	347.75	759
8,500	466	478.25	2,618.75	607.25	579.75	2,188	638.25	489.25	1,971.25
13,000	818.75	600.5	5,042.5	1,761.5	741.5	5,171.75	812	781.25	4,932.25
15,000	1,578	710.5	7,116.75	2,366.75	799.75	8,081.25	2,365.25	995.25	8,460.75
20,000	5,952.25	1,070.5	21,794.25	2,798.25	999.5	10,912	4,420.25	1,439.75	16,018.5

A incremental APSP algorithm for node addition, *B* modified incremental APSP algorithm for node addition, *C* DI incremental algorithm

Table 2 Comparison of incremental algorithms when the new node has large number of neighbors

Number of nodes	Time in milliseconds								
	Random graphs			Small world graphs			Preferential attachment graphs		
	A	B	C	A	B	C	A	B	C
5,000	561.25	436.5	766.75	105	301.25	1,293.5	298	351.5	942
15,000	2,675	918.5	6,032	2,629.25	741.25	5,703.25	2,783.5	990.75	6,235.25

A incremental APSP algorithm for node addition, *B* modified incremental APSP algorithm for node addition, *C* DI incremental algorithm

graph. Both the algorithms have been run 30 times on all the graphs. Figure 7 shows a box plot of those readings, while Table 3 reports the average readings. It can be observed from both Fig. 7 and Table 3 that the presented incremental betweenness algorithm performs better than Brandes' betweenness algorithm.

As the total number of edges in a graph affects the performance of the two betweenness algorithms, it is important to test these algorithms on graphs having varying densities. Hence, the two algorithms are also tested on a set of graphs generated by a Snijder's stochastic actor-based model for network dynamics (Snijders et al. 2010). This model represents network dynamics as being driven by many different tendencies, which are micro-mechanisms in social networks formations, and which may operate simultaneously. Some examples of such tendencies are outdegree (tendency to form ties in general), reciprocity, transitivity ("friend of a friend is my friend"), and homophily (choice of network ties based on similarity of attributes). A stochastic actor-based model allows to test a hypothesis about these tendencies and to estimate parameters, which demonstrates the strengths of selected tendencies in the given network. Snijder's stochastic actor-based model is briefly described in the Sect. 8.2. Using this model, 60 different graphs are generated with their node sizes ranging from 100 to 1,000. Figure 8 shows the results of the comparison and Table 4 shows the actual values. These are the average readings of 30 runs on each of the 60 graphs. It is evident that the incremental betweenness algorithm outperforms Brandes' betweenness algorithm. The benchmarks for the two betweenness algorithms were measured on a personal computer with 2.3 GHz Intel Core i5 processor and 4 GB 1333 MHz DDR3 memory. The algorithms were implemented in Java programming language using 'javac (1.6.0_37)' compiler.

Table 3 Comparison of the static and dynamic betweenness algorithms on graphs generated by RTG generator

Number of nodes	Number of edges	Time in milliseconds	
		Brandes' algorithm	Incremental algorithm
101	277	110	74
452	1,944	799	568
588	1,730	1,031	668
699	2,645	1,342	750
987	3,715	1,888	1,031
1,574	5,577	2,944	1,928
1,764	6,280	3,998	2,428

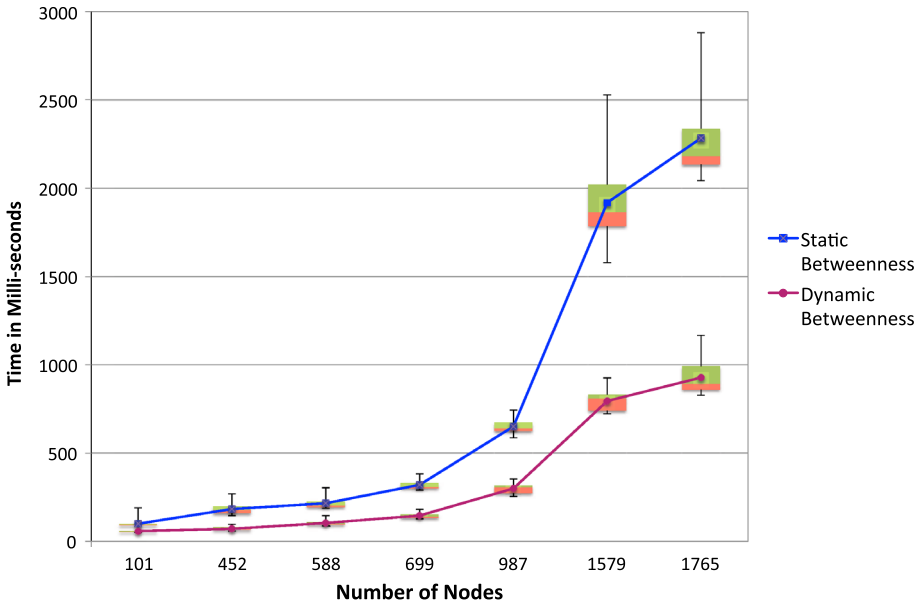

Fig. 7 Experimental comparison of the static and dynamic betweenness algorithms on graphs generated by RTG generator

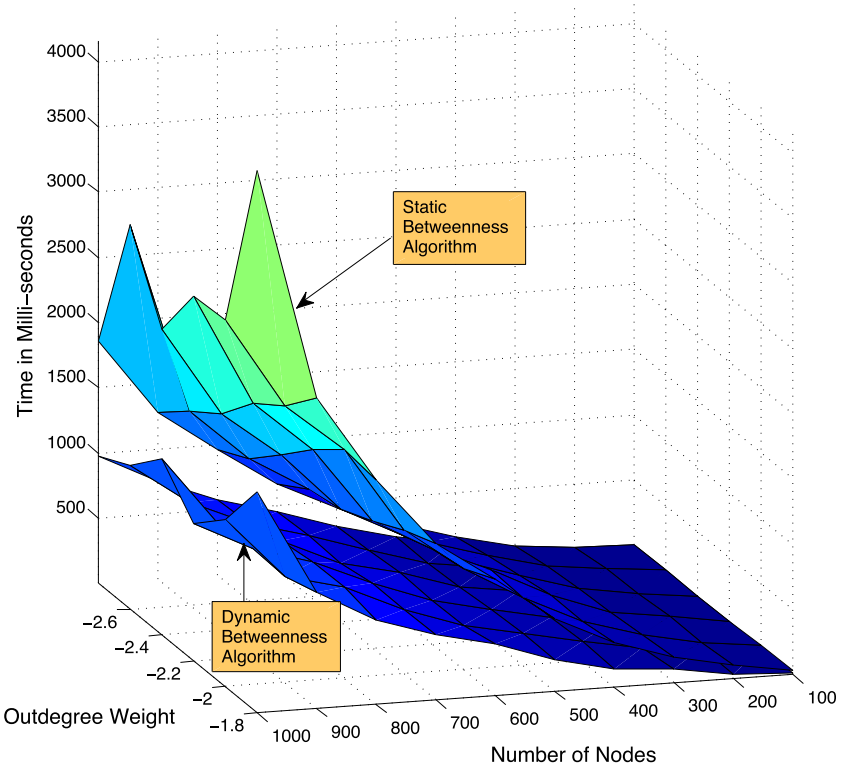

Fig. 8 Experimental comparison of the static and dynamic betweenness algorithms on graphs generated by a stochastic actor-based model Snijders et al. (2010)

7 Conclusion and future research directions

Social networks are dynamic in nature. Social network analysis involves computations of metrics, whose values need to be updated frequently. To save computational time, dynamic network algorithms are needed. Closeness centrality and betweenness centrality are two such important metrics. The dynamic all pairs shortest path algorithms for the edge addition and node addition cases (incremental algorithms) are prerequisites for dynamic closeness and betweenness centrality.

Table 4 Comparison of the static and dynamic betweenness algorithms on graphs generated by a stochastic actor-based model (Snijders et al. 2010)

Number of nodes	Outdegree weight	Time in milliseconds		Number of nodes	Outdegree weight	Time in milliseconds	
		Brandes'	Incremental			Brandes'	Incremental
100	−2.8	3	2	600	−2.8	454	305
100	−2.6	5	4	600	−2.6	560	324
100	−2.4	7	5	600	−2.4	641	351
100	−2.2	13	9	600	−2.2	729	339
100	−2	24	9	600	−2	774	341
100	−1.8	35	11	600	−1.8	895	398
200	−2.8	14	13	700	−2.8	668	437
200	−2.6	50	24	700	−2.6	821	469
200	−2.4	62	29	700	−2.4	869	452
200	−2.2	80	47	700	−2.2	978	503
200	−2	85	39	700	−2	1,103	496
200	−1.8	102	31	700	−1.8	1,207	507
300	−2.8	54	54	800	−2.8	961	580
300	−2.6	103	65	800	−2.6	1,093	594
300	−2.4	134	76	800	−2.4	1,314	660
300	−2.2	148	69	800	−2.2	1,559	697
300	−2	177	77	800	−2	1,753	701
300	−1.8	202	110	800	−1.8	1,723	646
400	−2.8	161	116	900	−2.8	1,279	751
400	−2.6	216	127	900	−2.6	1,489	789
400	−2.4	264	140	900	−2.4	1,662	795
400	−2.2	307	141	900	−2.2	1,942	830
400	−2	303	159	900	−2	2,119	811
400	−1.8	346	138	900	−1.8	2,386	893
500	−2.8	296	194	1,000	−2.8	1,855	973
500	−2.6	355	195	1,000	−2.6	2,953	1,099
500	−2.4	413	208	1,000	−2.4	2,345	1,349
500	−2.2	537	235	1,000	−2.2	2,794	1,052
500	−2	524	249	1,000	−2	2,819	1,285
500	−1.8	558	249	1,000	−1.8	4,159	1,697

This paper presents an incremental APSP algorithm for node addition which runs in $O(n^2)$, which is also a theoretical lower bound of the APSP problem. A modified version of the incremental APSP algorithm for node addition is also presented in the appendix section, which empirically performs better than the incremental APSP algorithm on large graphs (size $> \sim 10,000$ nodes). An incremental APSP algorithm for edge addition is also presented, which has $O(n^2)$ as its run time complexity.

An incremental closeness algorithm and incremental betweenness algorithms for node and edge addition cases are presented in the paper. These algorithms are based on the designed incremental APSP algorithms. The time complexity to update betweenness centrality values of all nodes in a network becomes $O(nm + n^2 \log n)$. Though this complexity is same as that of Brandes' algorithm (Brandes 2001), the presented experimental comparisons show that the incremental betweenness algorithm runs much faster in practice.

The paper also consists of an experimental comparison of the presented incremental APSP node addition algorithms with the latest known incremental algorithm (Demetrescu and Italiano 2004) designed for APSP ("DI incremental algorithm"). Results indicate that both the versions of incremental algorithms presented in this paper perform significantly better than the 'DI incremental algorithm'.

In many graph representations of real-world scenarios, nodes and edges get deleted from a graph. In such situations, decremental APSP algorithms are needed to update the shortest paths. The best known decremental APSP algorithm runs in $O(n^2 \log n)$ amortized time. Design of more efficient (theoretically and empirically) decremental algorithms for APSP, and for centrality metrics is a potential direction for future research.

Finally, the computational test design presented in this paper using Snijder's actor-based stochastic model opens a new way to study how the betweenness values of nodes in graphs are affected by various parameter values in the model.

Acknowledgments This work was supported by the National Science Foundation via grant number ICES-1216082. This support is gratefully acknowledged.

Appendix

The modified incremental APSP algorithm

This section presents a modified version of the designed incremental APSP algorithm. In the incremental algorithm for node addition, the shortest paths passing through the new node z are compared with all the n^2 shortest paths in the original network, in the last step of the algorithm. In this modified version, we try to avoid some of these comparisons with the aim to improve the empirical performance of the algorithm.

A new shortest path distance between a pair of nodes (i,j) and passing through the newly added z has two parts; $\min_{k^{\text{in}} \in T_1}\{d_{iz}^{k^{\text{in}}}\}$ and $\min_{k^{\text{out}} \in T_2}\{d_{zj}^{k^{\text{out}}}\}$. Now, if any of these two parts is greater than the original shortest path distance, d_{ij}, i.e., if $\min_{k^{\text{in}} \in T_1}\{d_{iz}^{k^{\text{in}}}\} > d_{ij}$ or $\min_{k^{\text{out}} \in T_2}\{d_{zj}^{k^{\text{out}}}\} > d_{ij}$, then the new shortest path distance of a path passing through z will not be shorter than the original shortest path distance d_{ij} and in this way further computations can be avoided. This idea provides a foundation to the concept of the modified version of the incremental algorithm. For this algorithm, we sort all rows and columns of the original distance matrix, which takes $O(n^2 \log n)$ time. However, this is only one time investment, made at first addition of a new node, for original distance matrix. For each consecutive addition of a node, new insertion of shortest path distance into the distance matrix takes only $O(\log n)$ time and hence the method turns out to be useful, as it saves computational efforts in the long run.

1. *Sort shortest path distances starting from each node:* Let $d_{i\cdot}$ be the set of the shortest paths that start from node i and end at any other node. $d_{i\cdot}$ represents a single row in the distance matrix of shortest paths. Sort this row in the descending order according to path lengths. Remove all elements in this row starting from the element of the highest path length, that satisfy the condition $d_{i\cdot} > \min_{k^{\text{in}} \in T_1}\{d_{iz}^{k^{\text{in}}}\}$ and store in a new matrix. Follow this procedure for all i nodes.
2. *Sort shortest path distances ending at each node:* Similarly, let $d_{\cdot j}$ be the set of shortest paths ending at node j. $d_{\cdot j}$ represents a single column in the distance matrix of shortest paths. Sort this column in the descending order according to the path lengths from a copy of the distance matrix. Remove all elements in this column starting from the element of the highest path length, that satisfy the condition $d_{\cdot j} > \min_{k^{\text{out}} \in T_2}\{d_{zj}^{k^{\text{out}}}\}$ and store in the same new matrix. Repeat this procedure for all columns.
3. *Update shortest path distances:* Now, in this new matrix, the old shortest path distances that are selected by row as well as column operations will try to overlap in the same element of the matrix. Select only those paths for the final comparison of checking if $\min_{k^{\text{in}} \in T_1}\{d_{iz}^{k^{\text{in}}}\} + \min_{k^{\text{out}} \in T_2}\{d_{zj}^{k^{\text{out}}}\} < d_{ij}$. All non-overlapping elements do not qualify for this comparison. In this way, empirical performance can be further improved.

In the worst case when all d_{ij} are greater than both $\min_{k^{\text{in}} \in T_1}\{d_{iz}^{k^{\text{in}}}\}$ and $\min_{k^{\text{out}} \in T_2}\{d_{zj}^{k^{\text{out}}}\}$ each time, this algorithm also performs n^2 comparisons and thus does not prove to be an improvement over the original version of our algorithm. However, usually the number comparisons are almost always smaller than n^2 and hence this algorithm can prove to be empirically effective (see also Sect. 4). Algorithm 4 below shows the pseudocode of the algorithm.

Algorithm 4 Implementation of the Modified Incremental APSP Algorithm for Node Addition

d_{ij} is the current shortest path length from node i to node j.
$\min_{k^{in} \in T_1} \{d_{iz}^{k^{in}}\}$ is the minimum path length between i and z, $\forall i \in V$ // V is a set of all nodes.
$\min_{k^{out} \in T_2} \{d_{zj}^{k^{out}}\}$ is the minimum path length between z and j, $\forall j \in V$
S_{ij} is a set of start and end node pairs in the distance matrix that correspond to shortest path lengths.
rowMap ← sortedMap(d_{ij}, S_{ij}) $i \in V, j \in V$; //Shortest paths starting from node i are sorted in a descending order.
colMap ← sortedMap(d_{ij}, S_{ij}) $j \in V, i \in V$; //Shortest paths terminating at node j are sorted in a descending order.
T_{ij} ← [ ][ ]; //T_{ij} is a matrix that is used to store locations of qualified shortest paths.
for each $i \in \min_{k^{in} \in T_1} \{d_{iz}^{k^{in}}\}$
    tmap ← new sortedMap();
    tmap ← rowMap.head($\min_{k^{in} \in T_1} \{d_{iz}^{k^{in}}\}$); //Extract all values from *rowMap* $\geq \min_{k^{in} \in T_1} \{d_{iz}^{k^{in}}\}$
    for each key in tmap
        val ← tmap(key);
        for each node $t \in val$
            $T[i][t] \leftarrow -1$; // If $\min_{k^{in} \in T_1} \{d_{iz}^{k^{in}}\}$ is smaller than the old shortest path, mark that path for further comparison.
        end for
    end for
end for
for each $j \in \min_{k^{out} \in T_2} \{d_{zj}^{k^{out}}\}$
    tmap ← new sortedMap();
    tmap ← colMap.head($\min_{k^{out} \in T_2} \{d_{zj}^{k^{out}}\}$); //Extract all values from *colMap* $\geq \min_{k^{out} \in T_2} \{d_{zj}^{k^{out}}\}$
    for each key in tmap
        val ← tmap(key);
        for each node $t \in val$
            if ($T[t][j] == -1$) then do // If $\min_{k^{out} \in T_2} \{d_{zj}^{k^{out}}\}$ is also smaller than the old shortest path, then select that path for a final comparison.
                if ($\min_{k^{in} \in T_1} \{d_{tz}^{k^{in}}\} + \min_{k^{out} \in T_2} \{d_{zj}^{k^{out}}\} < d_{t.j}$)
                    colMap.remove(key);
                    rowMap.remove(key);
                    $d_{t.j} \leftarrow \min_{k^{in} \in T_1} \{d_{tz}^{k^{in}}\} + \min_{k^{out} \in T_2} \{d_{zj}^{k^{out}}\}$;
                    colMap.put($d_{t.j}$, val);
                    rowMap.put($d_{t.j}$, val);
                end if
            end if
        end for
    end for
end for

Snijder's stochastic actor-based model for network dynamics

This section briefly describes Snijder's stochastic actor-based model for network dynamics, which has been used to generate a set of graphs to test the two betweenness algorithms.

A set of longitudinal observed networks is an input to such a model. They are assumed to be outcomes of a Markov process. These models are "actor-based" models because it is assumed that actors in a network control their outgoing ties. Given an opportunity, an actor can form a tie, delete one of its existing ties, or do nothing. An opportunity to make such a change is probabilistically given to an actor. This opportunity may depend on the network position of the actors (e.g., centrality) and on actor covariates (e.g., age and sex). The probabilities of an actor's choices depend on the so-called objective function.

The objective function expresses how likely it is for the actor to change her/his network in a particular way. On average, each actor "tries to" move into a direction of higher values of her/his objective function, subject to constraints of the current network structure, the changes made by other actors in the network, and subject to random influence. The objective function is assumed to be a linear combination of effects.

$$f_i(\beta, x) = \sum_k \beta_k s_{ki}(x). \quad (13)$$

Here, $f_i(\beta, x)$ is the value of the objective function for actor i depending on the state x of the network. The function $s_{ki}(x)$ are effects, chosen based on subject-matter knowledge and correspond to the tendencies of the actors. The weights β_k are the statistical parameters. The effects represent aspects of the network as viewed from the point of view of actor i. If β_k equals 0, the corresponding effect plays no role in the network dynamics; if β_k is positive, then there will be a higher probability of moving into directions where the corresponding effect is higher, and the converse is true of β_k is negative.

In the model that is used to generated graphs for testing the two betweenness algorithms, we have considered outdegree, reciprocity and transitive triplets as the effects in the model. Hence, the objective function of actor i in our model would be,

$$f_i(\beta, x) = \beta_1 s_{1i}(x) + \beta_2 s_{2i}(x) + \beta_3 s_{3i}(x). \quad (14)$$

Here, $s_{1i}(x) = \sum_j x_{ij}$, the outdegree effect; $s_{2i}(x) = \sum_j x_{ij}x_{ji}$, the reciprocity effect; and $s_{3i}(x) = \sum_{j,h} x_{ih}x_{ij}x_{jh}$, the transitive triplets effect.

The parameters of such a model are estimated by a maximum likelihood method. Once the parameters are estimated, we can simulate different networks having same effects and same strength. For the purpose of generating graphs, an R script mentioned in Ripley et al. (2011) is used. It is assumed that the model parameters are already estimated. 60 different graphs are generated with their node sizes ranging from 100 to 1000 and outdegree weights(β_1) ranging from -2.8 to -1.8, keeping all other parameter values fixed (rate = 2, reciprocity parameter$(\beta_2) = 2$, transitive triplets parameter$(\beta_3) = 0.3$). These values are decided as per guidance given in the script. Note that, the outdegree weights have negative values. This parameter generally has negative value in large networks to hinder link formations between nodes that are remotely connected to each other. A node with a total degree (outgoing plus incoming) of approximately 10 is removed from each generated graph with its attached edges; it is then treated as an increment.

References

Akoglu L, Faloutsos C (2009) RTG: a recursive realistic graph generator using random typing. Mach Learn Knowl Discov Databases 13–28

Ausiello G, Italiano G, Nanni U (1991) Incremental algorithms for minimal length paths* 1. J Algorithms 12(4):615–638

Barabási AL, Albert R (1999) Emergence of scaling in random networks. Science 286(5439):509–512

Bavelas A (1948) A mathematical model for group structures. Hum Organ 7(3):16–30

Brandes U (2001) A faster algorithm for betweenness centrality. J Math Sociol 25(2):163–177

Burt R (1995) Structural holes: the social structure of competition. Harvard University Press, USA

Demetrescu C, Italiano G (2001) Fully dynamic all pairs shortest paths with real edge weights. In: Proceedings of the 42nd IEEE symposium on foundations of computer science, IEEE, New York, pp 260–267

Demetrescu C, Italiano G (2004) A new approach to dynamic all pairs shortest paths. J ACM (JACM) 51(6):968–992

Dutot A, Guinand F, Olivier D, Pigné Y et al (2007) Graphstream: a tool for bridging the gap between complex systems and dynamic graphs. In: Emergent properties in natural and artificial complex systems. Satellite conference within the 4th European conference on complex systems (ECCS'2007)

Eppstein D, Galil Z, Italiano G, Nissenzweig A (1997) Sparsification—a technique for speeding up dynamic graph algorithms. J ACM (JACM) 44(5):669–696

Freeman L (1977) A set of measures of centrality based on betweenness. Sociometry 35–41

Freeman L (1979) Centrality in social networks conceptual clarification. Soc Netw 1(3):215–239

Green O, McColl R, Bader DA (2012) A fast algorithm for streaming betweenness centrality. In: Proceeding of the 2012 international conference on privacy, security, risk and trust (PASSAT), 2012 international confernece on social computing (SocialCom), IEEE, New York, pp 11–20

Henzinger M, Klein P, Rao S, Subramanian S (1997) Faster shortest-path algorithms for planar graphs* 1,* 2. J Comput Syst Sci 55(1):3–23

King V (2002) Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In: Proceeding of the 40th annual symposium on foundations of computer science, 1999, IEEE, New York, pp 81–89

Kleinberg J (1999) Authoritative sources in a hyperlinked environment. J ACM (JACM) 46(5):604–632

Knoke D, Yang S (2008) Social network analysis. Sage Publications Inc., New York

Laumann E, Pappi F (1976) Networks of collective action: a perspective on community influence systems. Academic Press, New York

Lee MJ, Lee J, Park JY, Choi RH, Chung CW (2012) Qube: a quick algorithm for updating betweenness centrality. In: Proceedings of the 21st international conference on World Wide Web. ACM, New York, pp 351–360

Lin C, Chang R (1991) On the dynamic shortest path problem. J Inform Process 13(4):470–476

Loubal P (1967) A network evaluation procedure. Highw Res Board 205:96–109

Mark S (1973) The strength of weak ties1. Am J Sociol 78(6):1360–1380

Marlow C (2009) Maintained relationships on facebook. Retriev Febr 15:2010

Murchland J (1967) The effect of increasing or decreasing the length of a single arc on all shortest distances in a graph. Technical report. London Business School, Transport Network Theory Unit

Newman M (2005) A measure of betweenness centrality based on random walks. Soc Netw 27(1):39–54

Opsahl T, Agneessens F, Skvoretz J (2010) Node centrality in weighted networks: generalizing degree and shortest paths. Soc Netw 32(3):245–251

Ramalingam G, Reps T (1996a) An incremental algorithm for a generalization of the shortest-path problem. J Algorithms 21(2):267–305

Ramalingam G, Reps T (1996b) On the computational complexity of dynamic graph problems. Theor Comput Sci 158(1–2):233–277

Ripley RM, Snijders TA, Preciado P (2011) Manual for rsiena. University of Oxford, Department of Statistics, Nuffield College

Rodionov V (1968) The parametric problem of shortest distances. USSR Comput Math Math Phys 8(5):336–343

Snijders TA, Van de Bunt GG, Steglich CE (2010) Introduction to stochastic actor-based models for network dynamics. Soc Netw 32(1):44–60

Thorup M (2004) Fully-dynamic all-pairs shortest paths: faster and allowing negative cycles. Algorithm theory-SWAT 2004, pp 384–396

Watts DJ, Strogatz SH (1998) Collective dynamics of small-world networks. Nature 393(6684):440–442

Westbrook J, Tarjan R (1992) Maintaining bridge-connected and biconnected components on-line. Algorithmica 7(1):433–464