Incremental Centrality Algorithms for Dynamic Network Analysis

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Miray Kas

B.S., Computer Engineering, Bilkent University, Turkey
M.S., Computer Engineering, Bilkent University, Turkey
M.S., Electrical and Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

August 2013

To my Parents & Matta

ACKNOWLEDGEMENTS

I would like to express my appreciation and thanks to my advisors Prof. L. Richard Carley and Prof. Kathleen M. Carley for making this thesis possible and for their guidance and support. I am very grateful for their very positive, supportive, and transparent approach. Having worked with several professors, I feel happy and honored to get my Ph.D. from them.

I would also like to thank my committee members Prof. Jurgen Pfeffer and Prof. Huan Liu (Arizona State) for their support and guidance. I am also very grateful to Matthew Wachs for his help and support in every step of my Ph.D. and for the encouragement he gave me to complete my degree. Last but not least, I would like to thank CASOS members for insightful discussions and feedback they gave me at various stages of my studies.

This research is conducted in collaboration with the center for Computational Analysis of Social and Organizational Systems (CASOS) under the Institute for Software Research within the School of Computer Science (SCS) at Carnegie Mellon University (CMU). Financial support was in part provided by the Defense Threat Reduction Agency (DTRA) under the grant number HDTRA11010102. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Threat Reduction Agency (DTRA) or the U.S. government.

ABSTRACT

Incremental Social Centrality Algorithms for Dynamic Networks

Miray Kas
Electrical and Computer Engineering
Carnegie Mellon University

The increasing availability of online resources such as digital libraries and social networking websites has led to an upsurge of interest in the analysis of social networks. To date, a wealth of social centrality measures have been designed for determining the importance of nodes in a social network from different aspects. A significant number of social centrality metrics depend on the shortest paths in the network, usually requiring solving the all-pairs shortest path problem. However, most of these metrics were designed for static snapshots of 20-30 node networks. Computing centrality metrics in dynamically changing, large networks is almost unfeasibly costly, especially if it involves repeatedly calculating centralities from scratch for each incremental change.

This thesis proposes incremental algorithms for the two of the most popular shortest-path based social centrality metrics (e.g. closeness centrality and betweenness centrality) to avoid computations from scratch by performing early-pruning and achieving substantial performance improvements in dynamically changing networks. It explores the computational time savings and the memory requirements as the realistic social networks being analyzed scale to very large sizes. The key idea is to start with the old output of the algorithm and to modify/update only the affected values such that the changes in the network (e.g. edge/node insertions/deletions/modifications) are reflected in the centrality values as well. The approximate versions of incremental closeness and betweenness centralities through *k*-hop bounded computations are also designed where the shortest paths included in the computations should be less than or equal to *k*; forcing the centrality computations to remain within a *k*-hop subgraph of a node instead of the entire graph. The performance results obtained via experiments on a wide variety of synthetic and real-life dynamic networks suggest substantial improvements over the state of the art.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 INTRODUCTION

1.1 PROBLEM DOMAIN AND RESEARCH FIELD

Social network analysis has been considered to be an important tool for finding the structural holes in organizational structures, for analyzing the patterns of information dissemination, or for identifying the key actors that can influence/control others; all of which are very useful for answering several key business or organizational questions.

As a result, determining the most influential, well connected, and prominent nodes in networks has been an important research area and, to date, several social centrality metrics have been designed. Some of these metrics are based on the number of immediate connections nodes have (e.g. degree centrality, eigenvector centrality) while others are based on the shortest communication paths in the network (e.g. closeness centrality, betweenness centrality).

This dissertation examines networks that change dynamically and require reanalyzing centralities every time a network update is issued. Since updating degree-based centralities for dynamically changing connections can be trivially achieved by maintaining a number of counters, they are not considered to be as challenging as updating the centrality metrics based on dynamic shortest paths. Overall, the goal of this dissertation is to propose and evaluate the effectiveness of incremental algorithms for closeness and betweenness centralities, while handling various network types (e.g. directed/undirected binary/positively weighted) and network updates (e.g. node/edge addition/removal, edge cost increase/decrease).

1.2 THESIS STATEMENT AND SUPPORT

This dissertation explores and confirms the effectiveness of incremental algorithm design for computing the shortest path based social centrality metrics such as closeness centrality and betweenness centrality in dynamic networks. In particular:

“Computations of the shortest path based centrality metrics such as betweenness centrality and closeness centrality are important and costly dynamic network analysis problems, and the solutions to these problems can greatly benefit from incremental algorithms which build on results from previous runs to perform early pruning and enable substantial performance improvements.”

This thesis statement is supported experimentally by measuring the performances of the proposed incremental algorithms against their non-incremental counterparts, using both synthetic and real-life networks. In support of this thesis statement, several algorithms are developed and analyzed, which are further listed in detail in the summary of contributions.

1.3 KEY IDEA AND APPROACH

This thesis employs an *incremental algorithm* design approach in order to handle the special needs of rapidly changing, dynamic networks. An incremental algorithm uses information from earlier computations such that the changes in the input (e.g. network structure) are reflected on the output values (e.g. centrality values) as well.

An incremental algorithm is different from its static counterpart that performs all computations from scratch. At one point, an initial run is performed by a static algorithm that performs the desired computation from scratch. Then, the incremental algorithm is used in subsequent runs to handle changes in the input (e.g. network). The benefit of an incremental algorithm is that by being able to build on prior computations, an incremental algorithm is able to perform early pruning and update *only* the affected parts of the network while avoiding recomputations to the extent possible. The algorithms proposed in this dissertation are designed to handle various network updates such as node/edge insertions, node/edge deletions, and edge cost modifications in the case of weighted networks. Out of these network update types, node/edge insertions and edge cost decreases are grouped as growing network updates, while node/edge deletions and edge cost increases are grouped as shrinking network updates, which are further explained in Chapter 3.

1.4 BROADER IMPACT AND INTENDED USES

The study of networks, including social networks, biological networks, information networks, and many others has been a major topic in scientific research. Traditionally, social networks have been studied in social sciences such as sociology, psychology, and business administration [1]. The general features of these classical studies are that they are often restricted to small networks, and often consider the networks as static graphs, whose nodes represent individuals and whose links represent the social interactions among these individuals. Although it has been researched for almost sixty years, social network analysis has witnessed uses in other remote fields in the last two decades or so. This is primarily due to two reasons: abundance of data and potentially broader impact. First, as more datasets become available online, the interest in social networks grows rapidly, especially in the area of longitudinal analysis of data accruing over time. Second, many of the obtained results are not only the solutions for problems in the field of social networks, but they are also applicable to other fields such as biological networks [2], statistical physics, physical/technological networking [3] [4], bibliometrics/scientometrics [5], or even remote fields like urban planning [6] and nuclear capabilities assessment [7].

Among the classical and most commonly used techniques of social network analysis is the centrality analysis, which measures the relative importance of entities in a social structure that is modeled as a group of entities (represented as nodes) connected through links to model certain relationships or interactions. Most centrality metrics consider the immediate connections of the entities modeled or the features of the overall structure or network topology. This makes social network analysis techniques applicable to any concept or problem where the entities to be examined can be represented as nodes in a network that models their relationships or interactions. This is also the reason why social network analysis has such a demonstrated breadth of applicability in its use.

Betweenness centrality and closeness centrality are metrics to measure the relative importance of nodes in a network, considering the overall structure (topology) of the network. This makes them applicable to various networks designed to model very different entities. The incremental centrality algorithms proposed in this dissertation are intended to be applied on over-time dynamic networks, regardless of the structure or the set of entities the network models, applicable to many other fields such as biological networks (protein-protein interaction networks), physical/technological networking (e.g. wireless networking, P2P file sharing, the US power grid), bibliometrics/scientometrics (e.g. citation, coauthorship, collaboration analysis, expertise analysis), urban planning, or transportation networks.

Unlike the early examples of datasets used in social network analysis, most of the abovementioned examples of over-time dynamic networks are obtained through automated data collection methods. Hence, they have the potential to be substantially larger (i.e. on the order of several thousands) than the early social network datasets used by traditional sociologists, which were mostly less than a hundred nodes. However, current technology also allows us to collect network data that would model millions and billions of users from the Internet. Since the incremental centrality algorithms build on the results from earlier runs, they usually need a large amount of data to be in the memory so that they can fetch and use related pieces of data quickly. Larger memory requirements restrain the use of incremental centrality algorithms to networks that are on the order of several thousands and tens of thousands of nodes and edges, but not necessarily to networks whose node counts are on the order of millions; unless they are combined with approximation algorithms. Hence, the datasets used in this dissertation are on the scale of thousands of nodes. Approximations required for applicability to larger networks through extensions for *k*-centrality metrics and potential application areas are further discussed in Chapter 6.

1.5 WHAT CAN OVER-TIME CENTRALITY METRICS SHOW?

The most straightforward application area for the incremental centrality metrics is the computation of over-time centrality values for dynamically changing datasets. This section investigates what kind of additional information over-time centrality metrics are able to provide through a detailed examination of the over-time centrality values for the SocioPatterns social netwrok. SocioPatterns is a publicly available, dynamic social network dataset that models the interactions and face-to-face communication in real time among 113 attendees during the ACM Hypertext 2009 conference for 2.5 days [8].

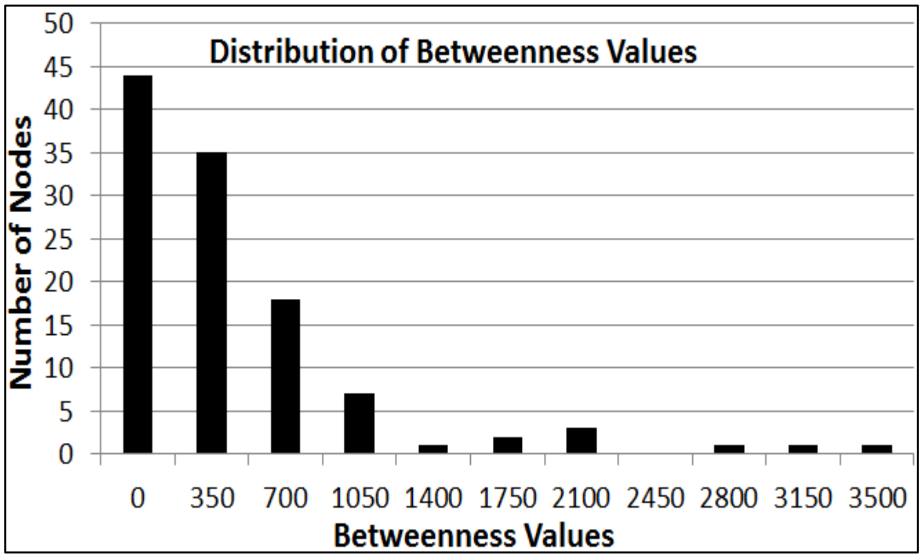

Figure 1 - Distribution of betweenness centrality values on SocioPatterns dataset.

Figure 1 presents a histogram for the distribution of betweenness centrality values and Table 1 presents the basic statistics for the aggregate, static snapshot of the network that is formed after all the real-time updates during the conference are issued. The distribution presented in Figure 1 shows that a very high number of nodes in the network have zero betweenness (e.g. 44 nodes out of 113 nodes). Such behavior is in line with the previous findings in the literature. For instance, [9] shows that the distributions of the betweenness values in many real-life, scale-free networks follow power laws where many nodes have zero betweenness centrality while only a few nodes have very high values of betweenness centrality.

Table 1 - Basic statistic on betweenness centrality values obtained on SocioPatterns dataset.

Statistics	**Value**
Mean	346.32
Median	28
Std. Err.	60.16
Std. Dev.	639.51
Sample Variance	408967.81
Kurtosis	9.68
Skewness	2.94
Min. Value	0
Max. Value	3570

Considering the data presented in Figure 1, two interesting research questions are as follows: ‘Are all the attendees with zero betweenness value unimportant for the community?’ or ‘Can they really have the same level of importance?’. In an attempt to find answers to such questions, in Figure 2 and Figure 3, the evolutions of betweenness centrality values for two groups of attendees are traced: (*i*) top three nodes with highest betweenness centrality values, (*ii*) the nodes that have zero betweenness at the static, aggregate snapshot of the network. The results presented in Figure 2 and Figure 3 indicate that temporal patterns of betweenness centrality values reveal interesting information and provide supplementary data for further ranking of nodes in a network.

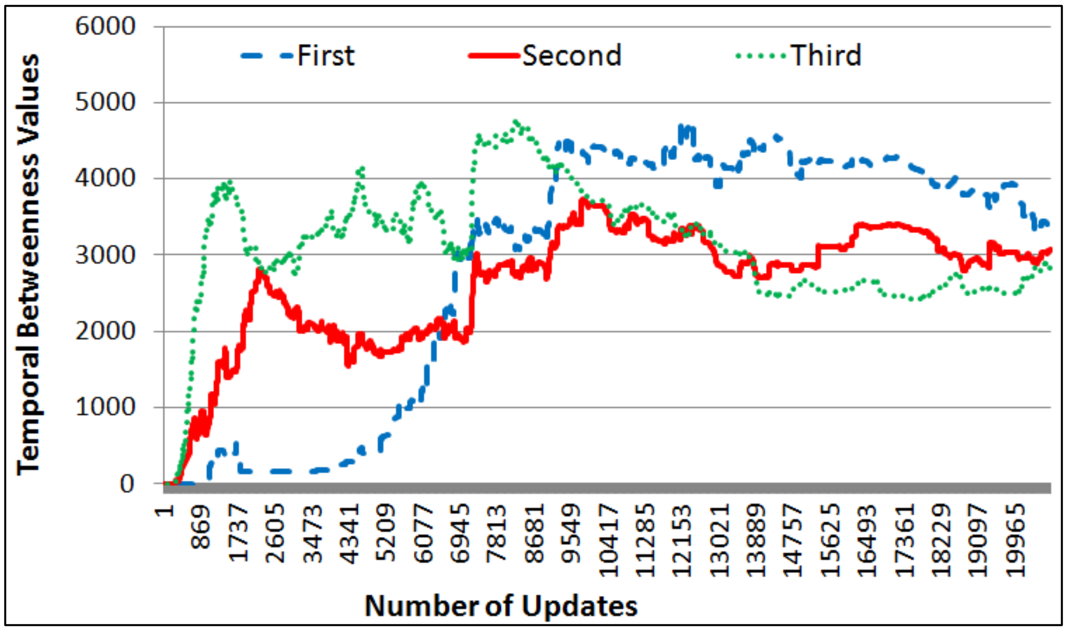

Figure 2 - Over-time betweenness centrality values for the top-three nodes in Socio Patterns dataset.

Figure 2 presents the overtime betweenness centrality values for three attendees that are ranked as top-three in terms of betweenness centrality. Considering the results presented in Figure 2, it can be observed that the rankings of the top three attendees are not very stable, especially in the first half of the network lifetime, when the network was still in the growth phase. Up-to-date analysis for the over-time centrality values might enable phase shifts, detection of sudden jumps, similar to the jump observed around 7813th update for the *First* attendee, or detection of declines similar to the decline that start around 9549th update for the *Third* attendee.

Figure 3 investigates the second group of attendees and presents the temporal behaviors of the betweenness centrality values of four attendees that have zero betweenness centrality at the final snapshot of the network. Out of 44 attendees that have zero betweenness in the final snapshot, only 25 of them have zero betweenness across the entire network lifetime, and 19 of them had non-zero betweenness values for some time during the conference. Since drawing 19 nodes' temporal betweenness behavior is visually not comprehensible, Figure 3 presents information for only four nodes.

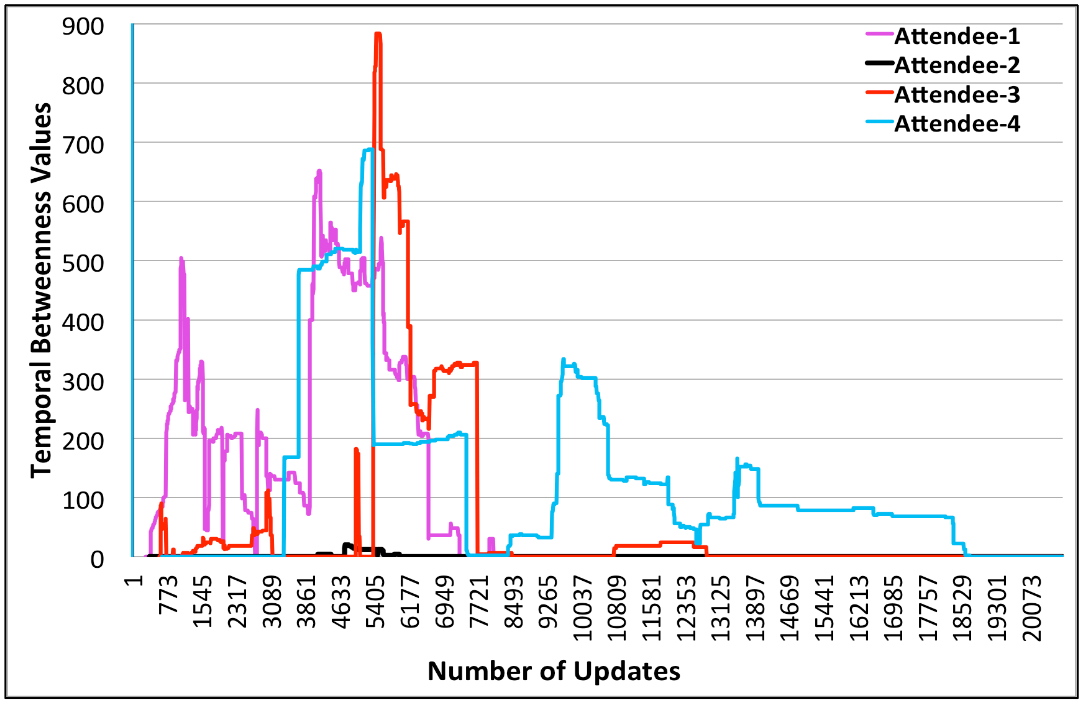

Figure 3 - Over-time betweenness centrality values for four selected nodes that have zero betweenness centrality in the final, aggregated snapshot of the SocioPatterns social network.

Figure 3 depicts multiple attendees whose betweenness centralities are non-zero during various intervals, although they appear to have zero betweenness centrality at the final snapshot of the network due to formation of new, shorter paths over time. As attendees keep interacting, new communication paths are formed through introduction of people to one another. In such cases, new introductions may lead to new, shorter paths and an attendee that used to be a part of the shortest paths may not necessarily be on the shortest paths any more. However, such attendees are more central to the community than the attendees whose betweenness centralities remain as zero for the entire lifetime of the network. For instance, the attendee shown in *Attendee-1* (drawn in pink) is only active in the first half of the network's lifetime, which indicates *Attendee-1* as a prominent person in the field, who did not attend the second half of the conference while the other attendees continued communicating with one another. As another example, all other attendees shown in Figure 3 are more central than *Attendee-2*, who had a very low non-zero betweenness centrality for a very short time period, around the 3861st – 6117th updates. Similarly, all attendees investigated in detail in Figure 3 are more central than all other nodes that have zero betweenness centrality for the entire lifetime of the network.

Considering only the betweenness values obtained at the final, static snapshot of a network, it is not possible to distinguish the entities that have the same final centrality values, potentially generating misleading results for dynamic networks that change over time. Hence, understanding the temporal behavior of centrality values is important, especially when there is need for further differentiation of entities' importance. This makes the incremental algorithms proposed in this dissertation valuable especially for cases when it is very costly to compute the over-time behavior of centrality values.

1.6 Thesis Organization

The first part of this thesis focuses on the problem of costly computation of the shortest path based social centrality metrics in dynamically changing networks and proposes incremental algorithms for computing closeness and betweenness centrality in less computational time while handling the different network types and updates. To make the algorithms proposed in this dissertation more generalizable to even larger networks, the second part of this dissertation covers *k*-centrality extensions and potential applications of incremental centrality algorithms to clustering (e.g. modified Girvan-Newman algorithm for finding groups) and wireless network analysis. The third and final part of this thesis reports experiments on a wide variety of synthetic and real-life dynamic networks to evaluate the behavior of algorithms on different network topologies and on networks with different sizes and densities.

Towards this end, the remainder of this dissertation is structured as follows. Chapter 2 provides background information and reviews the related work available in the current literature. Chapter 3 discusses the algorithmic framework including the network types and updates supported by the algorithms developed in this dissertation. In Chapter 4 and Chapter 5, the incremental closeness and the incremental betweenness algorithms are described, respectively. Chapter 6 discusses the *k*-centrality extensions and potential application of incremental centrality metrics on clustering and wireless networks vulnerability analysis and channel access scheduling. Chapter 7 describes the details of the synthetic and real-life networks used in the performance evaluations. Chapter 8 reports the performance results obtained. Finally, Chapter 9 concludes the dissertation by highlighting the key findings and potential directions for future research.

1.7 Dynamic Network Datasets Used

In this dissertation, both synthetic and real-life datasets are used to measure the performance of incremental centrality algorithms. For synthetic networks, we use a variety of topologies generated using preferential attachment [10], Erdos-Renyi [11], and small-world [12] network generation mechanisms.

Real life networks to be examined in more detail are as listed follows:

- Socio Patterns Dataset [8]
- Facebook-like Online Student Forum Network [13] [14]
- High-Energy Physics Scientific Co-Authorship Networks [15]
- User-to-User Twitter network about news about sanctions on Iran [16]
- Peer-2-Peer (P2P) File Sharing Network [17]

1.8 SUMMARY OF CONTRIBUTIONS

As mentioned above, the first part of this thesis designs incremental algorithms for closeness centrality and betweenness centrality to accommodate the needs of dynamically changing, streaming social networks. Hence, the contributions of the first part of this dissertation are the design, implementation, and the analysis of the algorithmic complexity of four incremental centrality computation algorithms in which the accuracies of the centrality values are perfectly maintained (i.e., no approximations are employed) across multiple updates:

- Incremental closeness centrality algorithm (Growing network updates Chapter 4.3)
- Incremental closeness centrality algorithm (Shrinking network updates Chapter 4.4)
- Incremental betweenness centrality algorithm (Growing network updates Chapter 5.3)
- Incremental betweenness centrality algorithm (Shrinking network updates Chapter 5.4)

The second part of this dissertation focuses on potential extensions and applications of the above mentioned algorithms. For the extensions, the plan is to adapt working principles of *k*-centralities into incremental centrality algorithms listed above and propose a new set of algorithms that can be applied to very large networks. The *k*-centralities approximate and speed up shortest path based centralities by only considering the first *k*-hop neighborhoods of nodes, instead of considering the shortest paths across the entire network [18]. As far as the applications go, one particular application is to demonstrate the use of streaming social centrality metrics in designing faster grouping (clustering) algorithms and analyzing over-time behavior and evolution of social agent and content based networks. The contributions for this part include the design of the following additional algorithms:

- Incremental *k*-closeness centrality algorithm (Chapter 6.2)
- Incremental *k*-betweenness centrality algorithm (Chapter 6.3)
- Incremental edge betweenness centrality that handles shrinking network updates and incorporation of incremental edge betweenness centrality into the Girvan-Newman clustering algorithm [19]. (Chapter 6.5.1)

CHAPTER 2 BACKGROUND AND RELATED WORK

This chapter reviews research available in the current literature that is closely related to the work presented in this dissertation. First, background information on social centrality metrics is presented, followed by a review of research done on closeness and betweenness centrality metrics. Then, background information and literature review on incremental algorithms, dynamic all-pairs shortest path algorithms, and clustering algorithms are presented.

2.1 Social Network Analysis and Centrality Metrics

A social network (SN) is a social structure consisting of a group of people that are connected by various relationships such as friendship, family ties or common interests and beliefs. Social networks are traditionally modeled and analyzed as graphs where the social actors (*i.e.,* people) are represented as nodes while the relationships between the people are represented by the links drawn between the nodes on these social network graphs.

Research on social networks in the past 60 years or so has led to a wealth of findings about the structure and evolution of these networks and a host of metrics and tools for assessing, forecasting, and visualizing network behavior more generally. The three major mathematical underpinnings of social network methods are graph theory, statistical and probability theory, and algebraic models [1]. Historically, much of the relevant work in social networks analysis (SNA) was applied to assessing the behavioral patterns and social interactions among human beings using graph theoretic metrics. SNA centrality measures focus on finding the key actors in a social network. To date, hundreds of social centrality metrics have been designed and discussed in the literature for evaluating the prominence/importance of the actors in the network from different aspects [20] [21] [1] [22] [23] [24].

However, a significant number of publications analyzing social networks consider only a handful of metrics: degree centrality, eigenvector centrality [25], closeness centrality [26], and betweenness centrality [27] [28]. Out of these four metrics, the former two focus on the connections of nodes while the latter two focus on the overall topology and the shortest path in the networks. Table 2 lists the meanings and common usages of these key metrics. Depending on the research question at hand, one centrality metric might become more important than the others because each metric provides insights into different aspects of the networks and has different implications and usages.

Table 2 - Common centrality metrics used in social network analysis.

Measure	**Definition**	**Common Usage**
Degree Centrality	Node with most connections	Identifying sources for intel
Betweenness Centrality	Connection between disconnected groups	Reducing/controlling/influencing activity by disconnecting groups
Closeness Centrality	Node that is closest to all other nodes	Information spreading and rapid access to information
Eigenvector Centrality	Nodes connected to most highly connected nodes	Identifying who can mobilize others in the network

Degree based metrics consider the number of connections a node has. In binary networks, *degree centrality* of a node is simply defined as the number of its connections while in weighted networks this definition is extended to include the total weight of the immediate connections a node has. Another degree based centrality measure, *spectral centrality*, is a recursively-calculated metric, which defines an actor/node as prominent if another prominent actor points to it. *Eigenvector centrality* is another metric used for key actor identification, which defines the centrality value of a node to be proportional to the sum of centrality values of all its neighbors. In other words, it is used for finding the node that is most connected to other highly connected nodes, indicating a stronger capital.

Unlike degree-based metrics, geodesic distance based metrics focus on the network topology, the connections, and the distances between the nodes. *Closeness centrality* of a node evaluates its information propagation efficiency. For any *node-X*, closeness centrality is defined as the inverse of the sum of distances from *node-X* to all other actors in the network. *Betweenness centrality* is defined as the number/fraction of the best (shortest) paths that pass through a *node-X*. For instance, in a clustered network, a node that is high in betweenness is likely to be a node that connects two clusters. Another centrality metric derived from betweenness centrality is *bridging centrality*. Bridging centrality of a *node-X* is calculated by multiplying its betweenness centrality value by a bridging coefficient such that it indicates how well the *node-X* is positioned among nodes with high degree centralities.

The initial design point for all of these centrality metrics, including betweenness, was static snapshots of small networks (e.g. 20-30 nodes) [27]. And the limiting algorithmic complexities and computation times of centrality measures were not a significant problem for such small, static networks. However, restricting the representation of social networks to static snapshots has been shown to result in substantial information loss in several studies, especially when the dynamic nature of social relationships is of interest [29].

Currently, there is an increasing trend toward dynamic network analysis (DNA). Dynamic network analysis is useful for analyzing social networks that evolve over time and serves as a response to the concerns about the limitations of analyses performed on static snapshots of social networks. However, keeping the costly-to-compute centrality metrics up to date in dynamic networks that rapidly change/grow due to several thousands of updates becomes almost infeasibly costly from a computation time perspective. This is because many important centrality metrics such as betweenness require re-solving the all-pairs shortest path problem with every update/change made to the network. Expensive computation times inhibit many social network researchers from analyzing over-time variations of centrality values on time-variant networks. To maintain solutions to costly problems on continually changing networks, we propose using *incremental algorithms* that update the solution to a problem after an incremental change is made to the network structure [30]. Next, we discuss some of the research on dynamic network analysis, then review the literature on the centrality metrics for which incremental algorithms are discussed in this dissertation in more detail.

2.1.1 Background on Dynamic Network Analysis

Relations in social networks are often dynamic and changing over time. Longitudinal networks are studied by researchers in several fields (e.g. sociologists, anthropologists, computer scientists, statistical physicists, etc.) to understand network evolution, belief formation, friendship formation, diffusion of innovations, acquisition and dissipation of power, spread of deviant behavior, emergence and evolution of communities etc. [31]. One traditional way of collecting longitudinal network data for analysis is to repeat the same information collection method (e.g. survey) over the same community at different points in time which results in two or more snapshots of the same network. Early research on longitudinal network analysis dates back to 1960's. One of the first studies in longitudinal networks is that of Newcomb's where he studied the evolution of friendship in US college fraternity, and published his research in 1961 [32]. Another famous example from 1969 is the Sampson's monastery [33], where Sampson stayed with a group of monks as an experimenter on vision and collected data at three different points in time to capture changes within the group over time. With recording/archiving of more data, it is also possible to have detailed information certain events (e.g. birth/death) for several time periods (e.g. generations).

Recently, automated online data collection techniques have brought a different dimension to dynamic network analysis by enabling collecting over-time data that has more than a handful of snapshots. In the ideal case, we would prefer to collect and analyze streaming network data where each update is recorded with a timestamp as soon as it takes place, which are the types of datasets that are under investigation in this dissertation.

While data collection is one aspect of dynamic network analysis that is different than conventional social network analysis, the analyses that can be performed, information to be extracted or inferred, and the scope of questions one can find answers to are different as well. Dynamic network analysis provides a means for seeking answers to research questions from a broader perspective, through time. A few, questions that one can find answers to using dynamic network analysis over social network analysis can be listed as follows:

- How do different contact sequences and interactions among the agents in a dynamic network influence dynamical processes?
- How do the distributions of links in dynamic networks change over time?
- How to control diffusion, spreading, contact, information, knowledge, money, and resource flow processes for networks over time?
- How do the influence, prestige, and prominence of agents in a network change over time?

The last one of these questions can be answered by investigation of centralities over time. This dissertation primarily designs new incremental algorithms for computing two of the most commonly used shortest path based centrality measures (e.g. betweenness centrality and closeness centrality), which are discussed in detail in the following sections.

2.1.2 Literature Review on Closeness Centrality

Closeness centrality is a commonly used social centrality metric and it can have different uses/meanings in different contexts. Closeness centrality is defined as the inverse of the sum of distance from a node *x* to all other nodes in the network. Closeness centrality of a social actor describes the 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 certain node to all other nodes in a network. Hence, it can also be used as a performance measure in technological networks [4]. For instance in [3], a channel access scheduling scheme (a resource allocation mechanism) is proposed where the nodes with higher closeness centralities are assigned a higher number of slots. In such a mechanism, the nodes that are "close" to all other nodes in the network (i.e. the nodes with high closeness centrality) get to deliver the majority of the messages, thus the end-to-end throughput the end users in the network experience is improved. Closeness centrality has also been used as a malware containment strategy in cellular mobile networks [34]. Nodes with high closeness centrality are used to dispatch rapid patches so that cellular resource consumption and associated costs due to malware spread are reduced.

As another example, in [35], 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 are expected to have high closeness centrality for the network to function effectively.

An interesting observation about closeness centrality, that comes from research on citation networks, is that the papers that are high in closeness centrality are usually relatively recent papers, which have rapid access to information in other papers. Since citation networks are directed and acyclic, older papers cannot cite newer papers. In other words, the older papers do not have paths to access more recent papers while they are accessible by other, newer papers. Therefore, the distance from a newer to older paper is defined (e.g. a real number) while the distance from an older to a newer paper is undefined (e.g. set to infinity). Hence, closeness values of newer papers will be considerably higher than those of older papers and closeness rankings have been shown to be useful for identifying more recent papers when the citation network is provided without publication dates [15].

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 others papers that employ closeness centrality 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 [36]. There has also been research on new methods to select top-*k* nodes in terms of closeness in large-scale networks [37], algorithms for approximation of closeness [38], the robustness of closeness centrality in terms of accuracy in the case of missing data [39], and incremental algorithms for network management and analysis based on closeness centrality [40].

2.1.3 Literature Review on Betweenness Centrality

In the field of social network analysis, betweenness centrality is one of the most commonly used metrics. The original argument and the algorithm for calculating betweenness centrality were introduced by Freeman [27]. Currently, the majority of the implementations for betweenness centrality use Brandes' algorithm or a variant of it [28]. Betweenness is critical in part because actors high in betweenness have social power to get things done [25], in scientific networks they are likely to be interdisciplinary [41] and have high mobility [15], and the removal of high betweenness nodes is likely to cripple network capability more than the removal of other nodes; e.g., nodes with high degree centrality [42]. Betweenness is often highly correlated with other network metrics; however, the degree of correlation depends on the network topology such that their correlation is lower in fractal networks [43]. Betweenness, however, is a fragile metric [44] that changes easily as the network structure changes [45].

Therefore, the number of studies that focus on betweenness is significant. A group of studies on betweenness centrality focus on its variants. One such work is that of Everett and Borgatti's where the authors discuss betweenness centrality of the ego in an ego network [46]. Another work that specifically focuses on the variants of betweenness centrality is [47]. In [47], Brandes discusses several ways of generalizing betweenness centrality including scaling of values with respect to length, inclusion of end-points in the path, computation of edge betweenness, group betweenness, and many others.

Another set of variants of betweenness centrality focus on incorporating over-time information into the definition of betweenness for dynamically changing graphs. To give a few examples, [48] proposes a model that reduces time-varying interactions in a network to directed flows on a static network to compute traditional, static centrality measures. On the other hand, [49] focuses on the general class of path-based metrics and introduces a new way of computing centrality measures by integrating time in the form of memory retention (i.e., aging/decaying of information over time) and accounting for the length of retention as well. The authors of [50] have a broader perspective, and propose temporal closeness and betweenness centrality metrics, which incorporate time into the computation of these metrics based on the temporal network model that is previously proposed in [51]. The temporal network idea in these two papers is based on reachability concept which emphasizes that not all nodes are reachable to one another right from the start as an aggregate form of the network suggests; they become reachable over time with the inclusion of new edges. In [52], the authors focus specifically on betweenness centrality and propose three different methods of incorporating time into the computation of betweenness centrality via explicit time-based ordering of edges. Similarly, [53] also uses time ordering to analyze centralities of nodes on complete worldwide airport transportation network, which results in design of new centrality measures (e.g. time-dependent transfer centrality and time-dependent connection centrality) that incorporate sequence of event arrivals and time orderings in centrality measures.

In contrast to these studies we do not change or extend the definition of betweenness; we rather focus on faster computation methods for the original betweenness metric in dynamically evolving networks. Another very recent study focuses on speeding up the exact computation of betweenness centrality using two different heuristics [54]. The first heuristic is used for identifying and contracting the structurally equivalent nodes that have the same centrality and the same contribution to the centrality of others. The second heuristic used focuses on partitioning the network into several smaller components and computing betweenness centrality on them. Although [54] focuses on speeding up betweenness computation, the suggested heuristics are geared towards handling static network snapshots; and does not focus on maintaining betweenness centrality across several network updates as it is done in this dissertation. In [55], the authors present a streaming betweenness algorithm for the fast computation of betweenness centrality in the case of edge insertions for binary networks. The main idea of [55] is based on modifying the breadth-first search tree that is built as a part of the shortest path search process in binary networks when computing betweenness centrality. In addition to binary networks, the algorithms proposed in this dissertation cover weighted networks while providing support for a wide variety of network update types including edge/node insertions and removals as well as edge cost modifications.

Another approach that is closely related to our work is QuBE that focuses on quickly updating betweenness centralities without computing all-pairs shortest paths in the network [56]. The idea of QuBE depends on estimating the nodes whose betweenness values might change due to an update in a network and quickly restoring them while avoiding computation of all-pairs shortest paths. Further comparisons against the QuBE algorithm are provided in Chapter 8.4.2.2.

Other than the studies discussed so far, there are also other studies that investigate how betweenness centrality changes with time varying networks instead of proposing a new metric or computation method for it. For instance, in [45], the authors investigate how betweenness values change when the density and the structure of the network change dynamically.

Betweenness is also commonly discussed within the concept of vulnerability of networks to different attacks since it is useful for identifying the nodes that might partition the network when they are removed. One such work, [42], notes that removal of the nodes with the highest betweenness values (i.e. remove the node with the highest betweenness value, recalculate betweenness values on the remaining network) is more harmful than the betweenness or degree centrality values computed on the initial, complete form of the network.

There are also other papers on betweenness centrality discussing its properties from different perspectives. For instance, [41] investigates the meaning of betweenness centrality in the context of scientific networks and suggests that betweenness centrality is actually a good indicator of interdisciplinarity for authors that publish in different fields. In [15], it is further discussed that in scientific networks betweenness centrality can also indicate high-mobility for authors that have co-authors from different affiliations they have visited; connecting/bridging different communities. Apart from these studies, betweenness centrality has also become a topic of interest for more mathematical or statistical physics based research such as [43], where the authors examine the correlation between degree and betweenness centrality and show that the correlation between these centrality measures is weaker in fractal models compared to non-fractal models.

Another line of research involves examining the studies done on the approximation of betweenness centrality. For instance, [57] approximates betweenness centrality by performing adaptive sampling that selects a number of source (seed) nodes to calculate the shortest paths from. As another example, [58], discusses scaling the betweenness contributions of nodes' with respect to their distance from the sampled source (seed) nodes. In [59], a new metric called *k*-path centrality is defined which relies on the assumption that communication messages travel across *k* paths at most and a randomized algorithm for its computation is proposed.

Approximation techniques are important for providing important tradeoffs between performance and accuracy and they are very important for larger networks. Hence, approximation algorithms for closeness and betweenness centralities are examined further in Chapter 6.

There are a number of techniques proposed for parallelization of betweenness centrality [60] [61] [62] [63] [64]. Out of these papers, [60] by Bader and Madduri is the one that proposed the first parallel implementations of betweenness centrality along with other parallel algorithms for the commonly used social network analysis techniques and centrality metrics. In [61], Madduri et al. propose a lock-free parallel algorithm for betweenness centrality and evaluate their algorithm on massive complex networks. The authors of [61] also provide optimized implementations designed for the massively multithreaded Cray XMT system with the Threadstorm processor. In [62], the authors propose a parallel algorithm for computing betweenness centrality and the novelty of their algorithm lies within the handling of access conflicts for a CREW PRAM algorithm. CREW PRAM stands for Concurrent Read Exclusive Write (CREW) Parallel Random Access Machine (PRAM) where simultaneous reads of the same memory cell are allowed while only one process can write to an individual memory cell. The authors of [63] note the problem with the additional space requirements of the betweenness algorithm which requires quadratic space when parallelization is performed over the vertex set of the standard betweenness algorithm (Brandes' algorithm [28]) and proposes a solution to reduce the space consumption of the parallel algorithm to $O(|V| + |E|)$. The solution proposed in [63] is designed to work with Concurrent Read Concurrent Write (CRCW) PRAM systems where simultaneous reads and writes on the same memory cell are allowed. More recently, there is also work on computing betweenness centrality on systems such as IBM Cyclops64 [64] and GPUs [65]. There are also papers on specific applications of parallelized betweenness centrality computations on complex networks such as the US power grid and resilience and response in the case of an emergency [66].

2.2 Incremental Algorithms

An incremental algorithm is an algorithm that updates the solution to a problem after an incremental change is made on its input [30]. Incremental algorithms arrive at solutions for computationally complex problems in an efficient manner without recomputing everything from scratch by preserving substantial information from prior computations. They present a tradeoff between incremental computation time and additional storage requirements for intermediate results.

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 a non-incremental algorithm that performs the desired computation from scratch. Despite their limited use in social network analysis so far, incremental algorithms have been used in several different areas: computer aided design, artificial intelligence, programming languages, database maintenance and data analysis, networked systems, and graph based computations. Next, I briefly review their uses in these above-mentioned areas to give a better understanding of what incremental algorithms mean to researchers from different fields.

Incremental computation and optimization is very important for **VLSI CAD tool** development [67] [68]. For instance, assume that a researcher is building a circuit and changing the layout structure by adding new connections between two different blocks, which in turn changes the shortest paths and the associated wiring costs and delays at every step. In such a case, incremental algorithms, rather than computing the shortest paths from scratch, are used to provide reduced computation time for a smoother human-computer interaction for the end user. Circuit-related incremental algorithms have been the topic of several research studies [68] [69] [70].

In the early days of **artificial intelligence**, incremental algorithms have been used for truth maintenance systems [71] [72]. A truth maintenance system (TMS) is a problem solver subsystem for reasoning programs and it makes inferences based on the reasons and beliefs that are recorded and maintained in the system. A TMS also allows agents to choose their actions based on their beliefs and assumptions and to revise those beliefs and assumptions when the assumptions held by the system contradict the discoveries made [71]. More recently, incremental algorithms are increasingly used for decision-making systems, especially in the field of robotics. To name a couple, various incremental algorithms have been proposed for deployment (i.e. using information on what has already been deployed when deploying a new instance) [73], or updating the solution after an incremental change is made on the training sample on learning procedures [74] [75].

In the field of **programming languages** research, language processing algorithms for interactive of programming tools/systems, expression evaluation via function caching (memoization) [76], high-level programming language design [77], and the design of compilers that re-compile the code after an incremental change or provide loop optimizations especially benefit from incremental algorithm design as they need to handle changes in the background triggered by small changes in the input [78]. The POPL'93 paper, [79], provides a thorough categorized bibliography on incremental computation examples covering the literature until the date of its publication going beyond the field of programming languages. As an example to more recent studies in this field, in [80], the authors build on work on dynamic dependence graphs (e.g. [81], [82]) and discuss combining memorization with the computation of dynamic dependence graphs and present experimental performance results.

In the field **of database management and data analysis**, there are a number of problems that are discussed that need and benefit from solutions based on incremental algorithms. Algorithms for computing database queries incrementally are called incremental view maintenance techniques and such techniques have been studied in the context of propagation of changes in both relational [83] and nested data collections [84] [85] [86], mostly focusing on object additions and deletions, leaving attribute modifications out. Later, incremental computation of complex queries has been studied for object-oriented database (OODB) query languages that allow construction of composite objects and nested collections [87] [88] [89].

Other examples of incremental algorithms emerge from **networked systems** such as dynamic routing over the Internet [90], and analysis/monitoring of network anomalies or user pattern where new data arrives continuously. Analysis/monitoring of such data is usually performed in a sliding window fashion where a full recomputation is needed across the entire window every time the window slides. Sliding window analysis techniques can be improved by using incremental algorithms that efficiently update the output by reusing the old sub-computations, adding new data, and dropping the oldest data. In a recent study, [91], the authors discuss the performance of the incremental sliding-window analysis including URL propagation on Twitter and the 95th percentile distance between users using the data collected by Glasnost measurement servers [92].

There are also many other incremental algorithm studies done in the field of **graph theory and algorithms**, some of them date back to 1970s [93]. More recent studies done in this field focus on computing solutions for graph-based problems in dynamically growing graphs. One such example is [94] where the authors propose an incremental algorithm for graph pattern matching with a focus on subgraph isomorphism that computes changes to the matches in response to the updates on large-scale graphs. Although pattern matching on graphs is one of the topics that are actively researched these days, I should point out that this is an active topic for almost thirty to forty years [95] and the use of incremental algorithms in this context has been discussed since early 1980s [96]. Some of the other graph problems incremental algorithms are used include online updating of minimum spanning trees [97] [98], graph connectivity [99] [100], and maintenance of transitive closures and transitivity [101].

Although incremental algorithms are commonly used in other research fields, they are relatively under-explored in the field of **social network analysis**. This dissertation targets the problem of rapidly computing closeness and betweenness centralities of nodes for each snapshot of a dynamically updated network. In our case, the incremental algorithms use information from earlier computations such that the changes in the input (e.g. network structure) are reflected on the output values (e.g. centrality values) to handle various network updates such as edge cost modifications (in the case of weighted networks), node/edge insertions, and node/edge deletions. The benefit of an incremental algorithm is that by being able to build on prior computations, an incremental algorithm is able to perform early pruning and update *only* the affected parts of the network while avoiding recomputation to the extent possible.

2.3 DYNAMIC ALL-PAIRS SHORTEST PATH ALGORITHMS

The second set of studies related to our research is the set of studies on dynamic shortest path computations. As noted earlier, the computations of closeness and betweenness centrality are tightly coupled with solving the all-pairs shortest paths problem. In the literature, there are many different techniques proposed for solving the all-pairs shortest paths problem dynamically [102] [103] [104] [105]. However, some of these techniques come with a number of restrictions. For instance, [103] solves the all-pairs shortest paths problem in networks that have positive integer edge costs that are less than a certain number, b, which is a serious limitation for networks whose edges are positive real valued. Another algorithm, the algorithm of Demetrescu and Italiano [102], depend on the notions of locally shortest paths and locally historical paths. A path is a locally shortest path if the paths obtained by deleting its first or last edge on the path are both shortest paths. According to this notion, empty paths and edges are considered to be locally shortest paths. While a shortest path is a locally shortest path by definition, a locally shortest path may not necessarily be a shortest path. A locally historic path is a path that has been identified as a shortest path at some point and has not been modified since then. The main idea is to maintain dynamically the set of locally historical paths including the shortest paths and the locally shortest paths as special cases.

In this dissertation, the Ramalingam and Reps dynamic all-pairs shortest path algorithm [104] is used as a basic building block for a number of reasons. First, the Ramalingam and Reps algorithm provides an easy to understand framework and clearly distinguish between different network update types (e.g. growing versus shrinking network updates). Second, the Ramalingam and Reps algorithm is one of the most commonly used dynamic all-pairs shortest paths algorithms, and has been deployed practical industrial applications. For instance, the dynamic shortest path algorithms developed by Ramalingam and Reps have been used by AT&T to improve the performance of Open Shortest Path First (OSPF) [106], the most commonly used intra-domain routing protocol on the Internet [107]. Third, it has good performance compared to other dynamic all-pair shortest path algorithms available in the literature. In 2006, Demetrescu and Italiano have published a study that performs an experimental analysis of the dynamic all-pairs shortest paths algorithms available in the literature [108]. As it has also been pointed out in Demetrescu and Italiano's paper [108], the Ramalingam and Reps algorithm performs quite well on sparse, real-life networks and the compute times of Ramalingam and Reps' algorithm and Demetrescu and Italiano's algorithm are quite close.

The underlying computing platform plays a role in determining which dynamic all-pairs shortest path algorithm performs better. In experiments done with real life networks (presented in [108]), the Ramalingam and Reps algorithm has the lowest or one of the lowest compute times among all the dynamic all-pairs shortest paths algorithms compared in that paper. The authors state that Demetrescu and Italiano's algorithm's performance gets better with the increased cache size while the Reps and Ramalingam algorithm is likely to become faster as the number of nodes increases because Demetrescu and Italiano's algorithm maintains more global structures and requires more memory while the Ramalingam and Reps algorithm requires less space and exhibits better locality in the memory access pattern. Since supporting an increasing number of nodes is important for dynamically growing social networks and it has overall good performance, we have decided to use the Ramalingam and Reps dynamic all-pairs shortest path algorithm as a building block in the design of incremental centrality algorithms.

2.3.1 Ramalingam and Reps Dynamic All Pairs Shortest Paths Algorithm

The incremental centrality algorithms proposed in this dissertation build on the Ramalingam and Reps dynamic all pairs shortest path algorithm presented in [104]. Hence, the working principles and the underlying key ideas of the Ramalingam and Reps algorithm are briefly reviewed here. In [104], the authors start by posing the dynamic single sink shortest path problem where the shortest path tree into a single sink (destination) node is maintained across incremental updates. In this context, the shortest path tree represents the shortest paths from every possible source node into the sink node, and the sink node is a distinguished node v in a network G, the shortest paths to which are calculated. A source node is a node that is the origin/starting point of a path. The working principles of the incremental single sink shortest path algorithm is similar those of the Dijkstra's algorithm [109]. Thus, the edges in the studied networks are expected to have positive edge costs.

In the following sections of their technical report [104], the authors extend the ideas and the algorithms designed for the single source shortest path problem into a solution for the dynamic all-pairs shortest path problem. Both single-sink and all-pairs shortest path algorithms depend on identifying the set of nodes that are affected by an incremental network update. In the all-pairs shortest paths algorithms, the sets of affected sink and affected source nodes are identified and the rest of the computations are performed on these reduced subsets, which are mostly much smaller than the entire network size. The incremental all-pairs shortest paths algorithm designed by Ramalingam and Reps have two distinct pieces: (*i*) deleting an edge and (*ii*) inserting an edge, and a separate algorithm is designed for each case. The algorithms for edge deletion are the *DELETEEDGE* (Algorithm-1) and the *DELETEUPDATE* (Algorithm-2) while the algorithms for edge insertion are the *INSERTEDGE* (Algorithm-3) and the *INSERTUPDATE* (Algorithm-4). The entry point for execution is the *INSERTEDGE* algorithm when a new edge is inserted and the *DELETEEDGE* algorithm when an existing edge is removed.

To process edge deletions, at a very high level, the idea is to look for affected nodes whose distances to a particular sink node have now increased and to update those distances accordingly. The proposed all-pairs shortest path algorithm idea depends on the solution for the single-sink shortest path problem: the same graph is traversed twice, once in forward direction, once in backward direction. When the edges are traversed in a backward fashion the sink nodes that are identified to be affected would actually be the affected source nodes if the graph were to be processed in a forward fashion in all directions. This is an attempt to restrict the traversal complexity, which may become unbounded if it were not forced stick with the shortest paths only, either processed forward or backward. Once the affected sink nodes are identified, then the algorithm attempts to find a new path to every sink node starting from the modified or removed edges head and tail nodes. This part of the problem is handled by the *DELETEUPDATE* algorithm (Algorithm-2).

```
Algorithm–1: DELETEEDGE (G, src, dest, c)
1. $C(src, dest) \leftarrow c$;  $\tilde{C}(dest, src) \leftarrow c$
2. AffectedSinks    ←  DELETEUPDATE $(\tilde{G}, dest, src, src)$
3. AffectedSources  ←  DELETEUPDATE $(G, src, dest, dest)$
4. for $s \in$ AffectedSinks
5.     DELETEUPDATE $(G, src, dest, s)$
6. for $s \in$ AffectedSources
7.     DELETEUPDATE $(\tilde{G}, dest, src, s)$
```

```
Algorithm–2: DELETEUPDATE (G, src, dest, z)
1.  AffectedVertices ← $\emptyset$
2.  if there does not exist $x \in$ Succ($src$) such that SP($src, x, z$)
3.          Workset ← {$src$};
4.          while Workset $\neq \emptyset$
5.              $u$ ← pop (Workset)
6.              Add $u$ to AffectedVertices
7.              for $x \in$ Pred($u$) such that SP($x, u, z$)
8.                  if (all $y \in$ Succ($x$) such that SP($x, y, z$) and $y \in$ AffectedVertices)
9.                      push $x$ into Workset
10.     PriorityQueue ← $\emptyset$
11.     for $a \in$ AffectedVertices
12.         $minDst \leftarrow \min(\{C(a,b) + D(b,z) \mid \{a \to b\} \in E(N) \;\&\; b \notin \text{AffectedVertices}\}, \{\infty\})$
13.         $D(a, z) \leftarrow minDst$
14.         if $D(a, z) \neq \infty$
15.             Insert (PriorityQueue, $a$, $D(a, z)$)
16.     while PriorityQueue $\neq \emptyset$
17.         $a$ ← extractMin(PriorityQueue)
18.         for $c \in$ Pred($a$) such that $C(c, a) + D(a, z) < D(c, z)$
19.             $D(c, z) \leftarrow C(c, a) + D(a, z)$
20.             if $c \in$ PriorityQueue
21.                 DecreaseKey (PriorityQueue, $c$, $D(c, z)$)
22.             else
23.                 Insert (PriorityQueue, $c$, $D(c, z)$)
24. return AffectedVertices
```

Algorithm–3: *INSERTEDGE* (*G, src, dest, c*)

```
1. $C(src, dest) \leftarrow c$;  $\tilde{C}(dest, src) \leftarrow c$
2. AffectedSinks    $\leftarrow$ INSERTUPDATE $(\tilde{G}, dest, src, src)$
3. AffectedSources  $\leftarrow$ INSERTUPDATE $(G, src, dest, dest)$
4. for $s \in$ AffectedSinks
5.     INSERTUPDATE $(G, src, dest, s)$
6. for $s \in$ AffectedSources
7.     INSERTUPDATE $(\tilde{G}, dest, src, s)$
```

Algorithm–4: *INSERTUPDATE* (*G, src, dest, z*)

```
1.  Workset $\leftarrow \{src \rightarrow dest\}$
2.  VisitedVertices   $\leftarrow src$
3.  AffectedVertices  $\leftarrow \emptyset$
4.  while Workset $\neq \emptyset$
5.      $\{x \rightarrow y\} \leftarrow$ pop (Workset)
6.      if $C(x, y) + D(y, z) < D(x, z)$
7.          Add $x$ to AffectedVertices
8.          $D(x, z) \leftarrow C(x, y) + D(y, z)$
9.          for $u \in$ Pred$(x)$
10.             if $SP(u, x, src)$ and $u \notin$ VisitedVertices
11.                 push $\{u \rightarrow x\}$ into Workset
12.                 Insert $u$ into VisitedVertices
13. return AffectedVertices
```

2.4 CLUSTERING ALGORITHMS

This dissertation investigates clustering algorithms as one potential application area for the proposed incremental centrality algorithms. Clustering is defined as the process of organizing objects into groups whose members are similar in some way [110] [111]. Clustering is a very widely studied research area, and to date, several clustering algorithms have been proposed, which can be classified in various ways [112].

One plausible way of classifying clustering algorithms is to consider how different groups are formed: partitional or hierarchical.

Partitional clustering algorithms (e.g. *k*-means, DBSCAN [113]) produce unnested grouping, which simply divides data into several groups based on a clustering criterion. These kinds of algorithms usually try to minimize a cost function or optimality criteria where a cost is associated to each cluster instance. Some of these algorithms are probabilistic where the input is a number of observations from a set of *k* unknown distributions. Each data point is assumed to belong to a single distribution and several efficient Expectation-Maximization schemes exist to perform the optimization iteratively [114]. Due to its simplicity, the *k*-means algorithm is one of the first clustering algorithms that were proposed. In the *k*-means algorithm, each point is assigned to one of the *k* clusters initially. Then, the center of each cluster is replaced by the mean of that cluster. These two steps are repeated until convergence. A point is assigned to a cluster that is close to it in terms of the distance function that is being used (e.g. Euclidian distance). Although the idea is simple, the outcome is very much dependent on the distance function (Euclidian distance or some other specially defined function), and can get stuck in local minima, which is an effect of the sensitivity to the initial assignments. In addition, obtaining meaningful clusters with such algorithms usually involves assumptions and/or prior knowledge about the data to assign correct parameters for the algorithm of choice [115].

On the other hand, hierarchical clustering algorithms produce dendrograms by splitting or merging groups based on similarity criteria. A dendrogram is a hierarchical tree formed of nested series of partitions. Hierarchical clustering algorithms can be further grouped as agglomerative and divisive clustering algorithms. Agglomerative clustering algorithms form clusters by merging subgroups (e.g. single link clustering [116], group-averaging clustering, complete link clustering [117] [118]). In agglomerative clustering, initially, each object is a unique cluster on its own and the closest clusters are successively merged in a bottom-up fashion until a single cluster remains and the output is usually represented in the form of a dendrogram. In contrast, divisive clustering algorithms form a dendrogram by splitting larger groups into smaller groups in a top-down fashion (e.g. Girvan-Newman [19] [119], DIVCLUST-T [120], DHSCAN [121]). The divisive clustering algorithms start with one, all-inclusive cluster and keep splitting until only singleton clusters of individual points remain. The differences between different divisive algorithms stem from how the algorithms decide to split the clusters at each step and how the actual splitting is done [122].

In addition to classifying them as partitional or hierarchical, it is also possible to categorize clustering algorithms in other ways. One plausible way of classifying clustering algorithms is to classify them based on their outputs. The output of a clustering algorithm can be (*i*) hard (strict) or (*ii*) fuzzy (overlapping) clusters. In strict partitioning algorithms, an object is assigned to only one cluster [123]. In fuzzy clustering, objects have varying degrees of membership in different groups and can be members of several of different clusters to different extents. Hence, an object is allowed to belong to multiple clusters with certain degrees of membership [124]. Given the additional computations required for calculating the degrees of memberships for each object, fuzzy clustering algorithms are computationally more complex and incur higher overhead compared to their hard (strict) clustering counterparts. Fuzzy clustering algorithms are particularly useful when the boundaries among different clusters in the data are not well defined and separated. Fuzzy clustering algorithms also enable the discovery of potentially more sophisticated relationships among objects and the clusters they belong to.

To date, hundreds of clustering algorithms have been proposed. Herein, we only briefly review how this huge body of work can be categorized at a very high level. The references regarding the literature on clustering algorithms have been collected in a number of survey papers and books [111] [112] [115] [122] [123] [125].

One algorithm of interest for this dissertation is the Girvan-Newman clustering algorithm [19]. The Girvan-Newman clustering algorithm is a hierarchical, divisive clustering algorithm that removes the edges in a graph based on the betweenness values of the edges. Chapter 6.5.1 discusses the details of the Girvan-Newman clustering algorithm and the proposed variations of it available in the literature, along with the modifications proposed to make it work with the incremental centrality algorithms proposed in this dissertation.

CHAPTER 3 ALGORITHMIC FRAMEWORK AND NOTATION

3.1 NETWORK TYPES

The algorithms proposed in this dissertation are designed to work on single-mode, dynamic networks with positive edge weights/costs, including directed/undirected and binary/weighted networks. The algorithms proposed in this dissertation have similar working principles to those underlying Dijkstra's algorithm so they are not designed to handle negative edge costs.

Undirected (bidirectional) networks model relationships that are mutually maintained. Undirected (bidirectional) networks can be considered as a special case of directed networks. Undirected networks can be represented as directed networks where the edge $\{x-y\}$ is represented using two directed edges $\{x \rightarrow y\}$ and $\{y \rightarrow x\}$.

Similarly, binary networks can also be represented as a special case of weighted networks where the existing edges' weights/costs are always equal to 1. Therefore, directed, weighted networks provide the most generalized coverage of different network types. Thus, throughout this dissertation, the pseudocodes of algorithms designed to handle positively weighted, directed networks are presented.

3.2 NETWORK UPDATE TYPES

For each centrality metric examined in this dissertation, two sets of algorithms will be designed to handle two separate classes of network updates. Broadly, network updates can be classified into two: (*i*) growing network updates, and (*ii*) shrinking network updates. These two classes of network updates interact differently with the structure of the shortest paths in the network. Growing network updates may result in shorter paths or new shortest paths of equivalent length while shrinking network updates may result in longer paths. Since they interact differently with the structure of the shortest paths in a network, there is need for designing two sub-algorithms for each centrality metric; one for each class of network updates. For both classes of network updates, the proposed algorithms focus on unit changes, handling each modification in the network one at a time.

3.2.1 Growing Network Updates

The first group of updates, the growing network updates, includes (*i*) inserting a new node, (*ii*) inserting a new edge, and (*iii*) decreasing the cost of an existing edge. We call them 'growing network updates' because networks usually grow by new agents/actors joining the network (e.g. inserting a new node), new relationships or interactions observed among agents in a network (e.g. inserting a new edge), or existing relationships becoming stronger due to increased communication/interaction levels.

The growing network updates may result in new shorter paths or additional shortest paths of equal length. Consider the following scenario. Assume that a new edge $\{x \rightarrow y\}$ is incrementally inserted into network G. However, the node x did not exist before this update. Then, this update will result in the insertion of a new node x and the discovery of a number of shortest path that start from node x and reach out to the rest of the network through node *y*. Or, consider a different scenario. An edge $\{u \rightarrow z\}$ is inserted where both nodes *u* and *z* existed before the update and node *u* was able to reach to node *z* following a number of edges. Now, with the insertion of the edge $\{u \rightarrow z\}$, there might be new paths that are shorter, using the edge that has just been inserted. Or, it may result in the formation of additional shortest paths without changing the shortest distance, just increasing the redundancy in terms of the number of available shortest paths. However, it is not necessarily guaranteed that growing network updates will result in new shortest paths. They may not have any effect on the shortest paths in the network and do not propagate far in the network. However, they *cannot* result in any longer paths.

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

3.2.2 Shrinking Network Updates

Similar reasoning is also applicable for the second class of updates, the shrinking network updates. Shrinking network updates include (*i*) deleting an existing node, (*ii*) deleting an existing edge, and (*iii*) increasing the cost of an existing edge. We call them 'shrinking network updates' because they are usually observed due to existing actors/agents departing from the network or broken/weakening relationships among agents.

The shrinking network updates may result in longer paths or have no effect on the construction of the shortest paths in the network. However, they *cannot* result in any shorter paths. For instance, when a node x is removed from the network, the shortest paths to and from node x should be removed as well. Assume another scenario. There is an edge that has a very high edge cost and it is not used by any of the shortest paths in the network. When this edge is deleted, then there are no changes in the shortest paths. However, when an edge $\{x \rightarrow y\}$ that lies on the shortest paths is removed or its cost increased, then we may need to look for new shortest paths (which will have to be longer than what we had before the network update) or eliminate the shortest paths that pass through $\{x \rightarrow y\}$.

Handling the deletion of a node with several edges reduces to several edge deletions to handle the deletion of each edge emanating from and/or entering into the deleted node. Similarly, deleting an existing edge can be represented as a special case of the network update that increases the cost of an edge. This is because deleting an existing edge corresponds to increasing the cost of an edge from a real, positive value to infinity in the adjacency matrix. Hence, a single algorithm will be sufficient to cover all three sub-types of shrinking network updates.

3.3 NOTATION

This section discusses the terminology and notation used in this dissertation. A directed network G consists of a set of nodes $V(G)$ and edges $E(G)$ where n is the number of nodes, and m is the number of edges in the network. The number of nodes and edges can also be represented as $|V(G)|$ and $|E(G)|$, respectively. $\{x \rightarrow y\} \in E(G)$ represents an edge directed from node x to node y, where $x \in V(G)$ is a predecessor of y, and $y \in V(G)$ is a successor of x. $Pred(x)$ is used to denote all predecessors of node x in the network G. $P_x(y)$ denotes the set of predecessors of node y on the shortest paths from node x. In other words, $P_x(y)$ is a subset of $Pred(y)$; $P_x(y) \subseteq Pred(y)$. $\tilde{G}$ 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.

In weighted networks, each edge e in the network has a weight $W(e)$ associated with the edge, which might denote the strength of the relationship between two actors in a social network. However, the weight of an edge should not be confused with the cost of the edge, $C(e)$. In this framework, the weight and cost of edges are interpreted as the inverse of one another such that having a stronger, closer relationship (i.e. higher weight) is inversely proportional to the effort to communicate (i.e. lower communication cost).

To be more precise, if the weight of an edge e is 2 ($W(e) = 2$), then, the cost of traversing the edge e is 0.5 ($C(e) = 0.5$). We assume that $C(e) > 0$ for $e \in E(G)$. The length of a path *Path* is defined as the sum of the costs of the edges on *Path*. The distance from node x to node y is the length of the minimum length path from node x to node y, which is also called the shortest path.

The algorithms proposed in this dissertation maintain additional information. $D(x, y)$ denotes the shortest distance while $\sigma(x, y)$ denotes the number of distinct shortest paths from node x to node y. C_B and C_C are vectors of length n, holding the betweenness and closeness centrality value of each node in the network, respectively. B_E is a vector of length m that holds the edge betweenness values of the edges in the network.

$I(a, z)$ denotes the set of intermediate nodes on the shortest paths from node a to node z. Similarly, $I_E(a, z)$ denotes the set of edges that are on the shortest paths from node a to node z. $SP(x, y, z)$ is true if the edge $\{x \rightarrow y\} \in E(G)$ is on a shortest path from node x to node z, satisfying the two conditions: (*i*) there is a path from node x to node z (i.e. the distance from node x to node z is $D(x, z) \neq \infty$) and (*ii*) $D(x, z) = C(x, y) + D(y, z)$. SP is false otherwise [104].

Finally, as a part of the related work, single source shortest path based algorithms, the Dijkstra's algorithm [109] and the Brandes' betweenness algorithm [28], are also discussed. In these algorithms, single source shortest distance information is stored in vector d. The $d[v]$ holds the shortest distance to the destination node v where the source node is implicit. Similarly, $P[v]$ indicates the predecessors on the shortest paths to the destination node v where the source node is again implicit.

CHAPTER 4 INCREMENTAL CLOSENESS ALGORITHM

This chapter presents the design of an incremental algorithm for closeness centrality 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 chapter for two reasons. The definition of closeness centrality depends entirely on the shortest distances across all pairs of nodes in the network. 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 chapter are generalizable to other metrics with shortest path computation as their core speed limitation. Most other shortest path based centrality metrics are not as generic as closeness centrality and require additional information such as the number of shortest paths between nodes, the predecessors and/or successors on these shortest paths. Therefore, we start incremental algorithm design for social centrality measures with the closeness centrality.

4.1 DEFINITION & COMPUTATION OF CLOSENESS CENTRALITY

Closeness centrality $C_c(x)$ of node x is defined as the inverse of the sum of the distances between x and all other nodes in the network as formulated in Eq(1). In Eq(1), $D(x, y)$ denotes the shortest distance from node x to node y.

$$C_c(x) = \frac{1}{\sum_{x \neq y} D(x,y)} \qquad \text{Eq(1)}$$

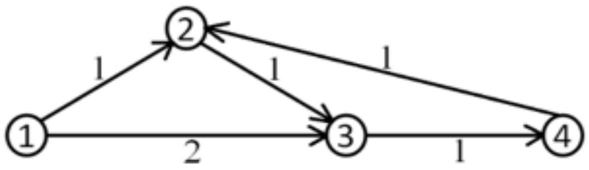

Figure 4 - Example network.

Figure 4 depicts a sample network and we walk through the computation of the closeness centrality, C_C, for each node in the network as laid out in Table 3. Each edge is label with its cost. In Table 3, each row represents a node *x*, and each column in the table represents a node to which node *x*'s shortest distance is calculated. The rightmost column, 'Total', represents the sum of all the defined distances from node *x* to all other nodes in the network.

Table 3 - Computation of closeness centralities for the network depicted in Figure 4.

Node \ **Distance To**	**1**	**2**	**3**	**4**	**Total**
1	0	1	2	3	6
2	∞	0	1	2	3
3	∞	2	0	1	3
4	∞	1	2	0	3

The closeness centrality, C_C, of each node is calculated by inverting the total value at its corresponding row: $C_C(1) = 1/6$; $C_C(2) = 1/3$; $C_C(3) = 1/3$; and $C_C(4) = 1/3$. Computation of closeness centrality can be performed by running an all-pair shortest paths algorithm (e.g. Floyd-Warshall algorithm [126]), which directly results in $O(n^3)$ time complexity.

Algorithm–5: *DIJKSTRA'S SINGLE-SOURCE SHORTEST PATH ALGORITHM* (G, *src*)
Input: A network $G(V, E)$ and source node *src*.
Output: Shortest path distances from *src* to all nodes n $V(G)$ are computed.

```
1. for $v \in V(G)$
2.     $d[v] \leftarrow \infty$
3.     $P[v] \leftarrow$ empty list
4. $d[s] \leftarrow 0$
5. $Q \leftarrow w \in V(G)$
6. while $Q \neq \emptyset$
7.     dequeue $u \leftarrow Q$
8.     if $d[u] = \infty$
9.         break;
10.    for neighbor $v$ of $u$
11.        $alt \leftarrow d[u] + C(u, v)$
12.        if $alt < d[v]$
13.            $d[v] \leftarrow alt$
14.            $P[v] \leftarrow u$
15.            decrease-key $v$ in $Q$
```

Closeness centrality is traditionally best computed by running a single-source shortest path algorithm (e.g. the Dijkstra's algorithm [109]) using each node as the source node once. At each iteration, the distances found are summed up to obtain the total distance from a given source node, and the total distance is then inverted to obtain the closeness value of the source node. In directed networks, the Dijkstra's algorithm [109] 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 [127]. When Dijsktra's algorithm is invoked using every node in the network once as the source node to compute all-pairs shortest paths, the overall complexity is $O(mn + n^2\log n)$. The algorithmic complexities of both the Floyd-Warshall algorithm and the Dijkstra's algorithm are sufficiently high that they are very costly to re-execute for keeping centrality values up-to-date in dynamically changing, large networks every time the network is updated. Hence, this dissertation proposes incremental centrality algorithms to address this problem and avoid the cost of recomputing all the shortest paths from scratch for keeping centrality values up to date at every network update.

4.2 Incremental Closeness Algorithm: High-Level Overview

To compute closeness centrality incrementally for streaming, dynamically changing social networks, the incremental all-pairs shortest paths algorithm proposed by Ramalingam and Reps [104] [105] is extended such that closeness values are incrementally updated in line with the changing shortest path distances in the network.

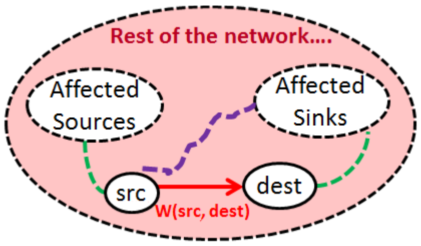

Figure 5 - Abstract figure describing that the entire network is not affected by an update or modification done on the network.

At each network update, there is only a subset of the network that is affected; the entire network is not affected. Assume that there are two nodes called src and $dest$ in a network, and an edge from src to $dest$, $\{src \rightarrow dest\}$, is inserted as shown in the abstract Figure 5.

In such a scenario, there will be a subset of nodes that are affected by this change. For instance, with the insertion of the edge $\{src \rightarrow dest\}$, the paths from node src to node $dest$ and to some of the *AffectedSinks* might now be using a new path that passes through the newly inserted edge $\{src \rightarrow dest\}$. Similarly, there will be a number of AffectedSource nodes that will now access the node $dest$ and/or some of the AffectedSink nodes using the newly inserted edge $\{src \rightarrow dest\}$. In most real life networks, we usually do not encounter edges that lie on all the shortest paths between all pairs of nodes and that are being continuously updated. Therefore, there is usually a fairly large portion of the network that is not necessarily affected by a network update.

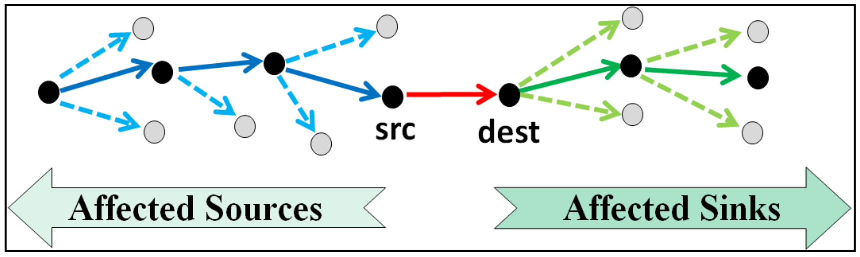

Figure 6 - Abstract figure describing how affected sink and source nodes are selected and how early pruning is done.

Before moving on to the low level details of how the shortest path distances and closeness centrality values are updated, I describe (i) how affected sink and source nodes are identified and (ii) how early pruning of shortest paths is performed at a high level.

Consider the case when the edge $\{src \rightarrow dest\}$ is inserted to a network as depicted in Figure 6. Updating the shortest paths and closeness centralities starts with the inserted edge. The edges that are on the shortest paths are followed to ensure propagation of the shortest path updates as far as needed. In Figure 6, the edges that are drawn with solid lines are the edges on the shortest paths, and they are considered for subsequent processing later in the execution to see if the update should propagate any further. The edges that are drawn with dashed lines are not on the shortest paths. The gray nodes that are accessible from those edges represent the nodes are not affected by the incoming network update; hence, they are pruned early from recomputation. The set of black nodes, which are either the head or the tail of one of the shortest path edges, form the set of the affected sink and source nodes.

The following sections (Chapter 4.3 and Chapter 4.4) describe how to find the affected set of nodes, and how to update the closeness centralities while updating the shortest paths to/from the affected nodes.

Finally, for generalization purposes, Figure 6 depicts a directed graph. However, an undirected graph can simply be handled by converting each undirected edge $\{a — b\}$ into two edges in opposite directions $\{a \rightarrow b\}$ and $\{b \rightarrow a\}$.

4.3 INCREMENTAL CLOSENESS ALGORITHM: GROWING NETWORK UPDATES

When a growing network update is issued (e.g. insertion of a new node/edge or edge cost decrease), the incremental computation of closeness centrality is handled by two sub-algorithms: *INSERTCLOSENESS* and *INSERTUPDATECLOSENESS.* The main entry point for execution is the *INSERTCLOSENESS* algorithm.

4.3.1 INSERTCLOSENESS

The *INSERTCLOSENESS* algorithm invokes the *INSERTUPDATECLOSENESS* algorithm 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. *INSERTCLOSENESS* first invokes *INSERTUPDATECLOSENESS* 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 *INSERTCLOSENESS*). Then, *INSERTUPDATECLOSENESS* is invoked for each AffectedSink and AffectedSource (Lines 4–7 of *INSERTCLOSENESS*) 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–6: *INSERTCLOSENESS* ($G, src, dest, c$)

Input: A network $G(V, E)$ for which shortest distances between all nodes (D) and closeness values of all nodes (C_c) are pre-computed. A newly inserted or modified edge $\{src \to dest\}$, with a cost of c where $c < C(src, dest)$.

Output: Network $G(V, E)$ updated with the edge $\{src \to dest\}$. The shortest path distances between all nodes (D) and closeness values of all nodes (C_c) updated.

```
1. $C(src, dest) \leftarrow c$;   $\tilde{C}(dest, src) \leftarrow c$
2. AffectedSinks    $\leftarrow$ INSERTUPDATECLOSENESS ($\tilde{G}, dest, src, src$)
3. AffectedSources  $\leftarrow$ INSERTUPDATECLOSENESS ($G, src, dest, dest$)
4. for $s \in$ AffectedSinks
5.     INSERTUPDATECLOSENESS ($G, src, dest, s$)
6. for $s \in$ AffectedSources
7.     INSERTUPDATECLOSENESS ($\tilde{G}, dest, src, s$)
```

4.3.2 INSERTUPDATECLOSENESS

In the *INSERTUPDATECLOSENESS* algorithm, 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 needed 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 of nodes are already computed for a given network. In the case of a network update, we only need to update the closeness centrality 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 the *INSERTUPDATECLOSENESS* algorithm. If this condition holds, it means that there is now a path from node x to node z which is strictly shorter than the previously known shortest path from x to node z and this new path uses the edge $\{x \rightarrow y\}$. Hence, the distance from node x to node z (i.e. $D(x, z)$) should be updated to hold the cost of this newly discovered path (i.e. $C(x, y) + D(y, z)$) (Line–14 of *INSERTUPDATECLOSENESS*).

Before the previous value of $D(x, z)$ is overridden with the new value, Lines 8–13 of the *INSERTUPDATECLOSENESS* algorithm handle the accurate maintenance of closeness centrality. To update closeness centrality 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 node x to node z had a contribution to the closeness centrality of node x. In this case, it is necessary to first subtract the previously known shortest distance from node x to node z (e.g. $D(x, z)$), and then add the new shortest distance (e.g. $C(x, y) + D(y, z)$) to the sum of distances to all other nodes from node x. Otherwise, nothing is subtracted, only the new shortest distance is added to the sum of distances from node *x*. Finally, closeness centrality of node *x* is obtained by inverting the total distance (*INSERTUPDATECLOSENESS*, Line 13).

Algorithm–7: *INSERTUPDATECLOSENESS* (*G*, *src*, *dest*, *z*)

Input: A network $G(V, E)$, a newly inserted or modified edge $\{src \rightarrow dest\}$, and an affected node z.

Output: The shortest distances (D) to the affected node z are updated, and closeness values (C_c) of the sources of the updated distances are also updated.

```
1.  Workset ← $\{src \rightarrow dest\}$
2.  VisitedVertices   ← $src$
3.  AffectedVertices  ← $\emptyset$
4.  while Workset $\neq \emptyset$
5.      $\{x \rightarrow y\}$ ← pop (Workset)
6.      if $C(x, y) + D(y, z) < D(x, z)$
7.          Add $x$ to AffectedVertices
8.          TotDist($x$) ← $\frac{1}{C_c(x)}$
9.          if $D(x, z) \neq \infty$
10.             TotDist($x$) ← TotDist($x$) – $D(x, z) + C(x, y) + D(y, z)$
11.         else
12.             TotDist($x$) ← TotDist($x$) + $C(x, y) + D(y, z)$
13.         $C_c(x) \leftarrow \frac{1}{\text{TotDist}(x)}$
14.         $D(x, z) \leftarrow C(x, y) + D(y, z)$
15.         for $u \in$ Pred($x$)
16.             if $SP(u, x, src)$ and $u \notin$ VisitedVertices
17.                 push $\{u \rightarrow x\}$ into Workset
18.                 Insert $u$ into VisitedVertices
19. return AffectedVertices
```

The final part of the *INSERTUPDATECLOSENESS* algorithm (Lines 15–17) performs checks for subsequent processing. This part of the algorithm also prunes the portions of the network that are not affected by the changes in the shortest paths. For each edge $\{u \rightarrow x\}$ entering into the affected node x, it is checked to see if the edge $\{u \rightarrow x\}$ is on the inspected shortest paths. If *SP* returns true, and if the other end of the edge (node *u*) is not in the list of already processed nodes, the edge $\{u \rightarrow x\}$ is inserted in the set of edges that would need subsequent processing.

4.4 Incremental Closeness Algorithm: Shrinking Network Updates

Next, the algorithms for incremental maintenance of closeness values in the case of shrinking network updates (e.g. node/edge deletions or edge cost increases) are presented. Following an approach similar to that of Chapter 4.3, shrinking network updates are handled by two sub-algorithms: *DeleteCloseness* (Algorithm–3) and *DeleteUpdateCloseness* (Algorithm–4).

4.4.1 DeleteCloseness

The *DeleteCloseness* (Algorithm–8) follows a very similar logic to that of the *InsertCloseness* (Algorithm–6).

Algorithm–8: *DeleteCloseness* (*G*, *src*, *dest*, *c*)

Input: Network $G(V, E)$ for which shortest distances between all nodes (D) and closeness values of all nodes (C_c) are pre-computed. A modified edge $\{src \rightarrow dest\}$ with a cost of c, where $c > C(src, dest)$.

Output: Network $G(V, E)$ updated with the changes on the edge $\{src \rightarrow dest\}$. The shortest distances between all nodes (D) and closeness values of all nodes (C_c) are also updated.

```
1. $C(src, dest) \leftarrow c$;  $\tilde{C}(dest, src) \leftarrow c$
2. AffectedSinks    $\leftarrow$ DeleteUpdateCloseness $(\tilde{G}, dest, src, src)$
3. AffectedSources  $\leftarrow$ DeleteUpdateCloseness $(G, src, dest, dest)$
4. for $s \in$ AffectedSinks
5.     DeleteUpdateCloseness $(G, src, dest, s)$
6. for $s \in$ AffectedSources
7.     DeleteUpdateCloseness $(\tilde{G}, dest, src, s)$
```

After updating the edge cost matrix for the modified/deleted edge (Line-1 of Algorithm–8), the *DeleteUpdateCloseness* algorithm (Algorithm–9) is invoked several times: first, to identify affected sink and source nodes and then, to process each affected sink and source node separately.

4.4.2 DELETEUPDATECLOSENESS

The *DELETEUPDATECLOSENESS* algorithm (Algorithm–9) has two distinct phases. The first phase of the *DELETEUPDATECLOSENESS* algorithm is between Lines 1–14 while the second phase is between Lines 15–39. The first phase of the algorithm identifies the set of affected nodes. In this case, affected nodes are the nodes whose shortest distances to node *z* (the third parameter of the algorithm) have increased. The shortest path distance from node *x* to node *z* may only increase if the network update is made on an edge which used to lie on all the shortest paths from node *x* to *z* and all the available shortest paths from node *x* to node *z* pass through the modified/deleted edge (i.e. when there is no alternative shortest paths which would still be shorter). The check for this condition is performed in Lines 12–13 of the *DELETEUPDATECLOSENESS* algorithm.

The second phase of the *DELETEUPDATECLOSENESS* algorithm determines the new shortest path distance from all affected nodes identified in the first phase of the algorithm to node *z* (the third parameter of the algorithm). The new shortest path distances to node *z* are kept in a min-key priority queue where the priority of a node corresponds to its distance to node *z*.

Nodes' closeness centralities are also updated in the second phase of the *DELETEUPDATECLOSENESS* algorithm. To be more precise, closeness centrality values are updated whenever a *D* value is changed (Lines 18–25 and Lines 29–35). The idea behind the closeness updates is similar to the idea described in Chapter 4.3. However, in this case additional checks are performed to avoid updating closeness values with a distance that was just set to infinity. Because, shrinking network updates might disconnect two nodes that were previously connected, causing the distance between them to be set as infinity. Line 19 of Algorithm–9 checks if node *z* was previously reachable from node *a* (e.g. $D(a, z) \neq \infty$). If it was reachable before, then it means the distance from node *a* to node *z* had a contribution to the closeness centrality of node *x*. In this case, it is necessary to first subtract the previously known shortest distance from node *a* to node *z* (e.g. $D(a, z)$). Next, we update $D(a, z)$ with the new shortest path distance. Since the shortest paths in the network have changed due to a shrinking network update, it might have resulted in disconnecting the two nodes *a* and *z*, making node *z* unreachable from node *a*. In such a case, there would be no *b* nodes that would satisfy the condition on Line 17, and the value of *minDst* has to be chosen as infinity. Before, updating the total distance from node *a* and the closeness centrality of node *a*, we again need to check whether the current distance from node *a* to *z*, $D(a, z)$, is infinity (Line 22 of *DELETEUPDATECLOSENESS*) and if it would contribute to the closeness centrality value of node *a* (Line 25 of *DELETEUPDATECLOSENESS*).

In Line 24, node *a* is inserted into the priority queue that holds the distances to node *z* in ascending fashion. A node is inserted into this priority queue only if its shortest distance to node *z* is changed and if it is still able to reach to node *z*.

Algorithm–9: *DELETEUPDATECLOSENESS* (*G*, *src*, *dest*, *z*)
Input: Network $G(V, E)$, deleted or modified edge $\{src \rightarrow dest\}$, and an affected node z.
Output: The shortest distances (D) to the affected node z are updated, and closeness values (C_c) of the sources of the updated distances are also updated.

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

The final part of the *DELETEUPDATECLOSENESS* algorithm processes this priority queue to see if the shortest distances can be further updated. In Line 17, if no b nodes that would satisfy the update condition were found, then the shortest distance of the node a to node z was forced to be infinity. Starting with Line 26 of the *DELETEUPDATECLOSENESS* algorithm, the shortest paths that are properly discovered earlier in the algorithm are examined to see if shorter paths that would use them as their sub-paths can be discovered.

The process of searching for new shortest paths is carried out in the form of ripples expanding outwards. For each node a in the priority queue, we check its predecessor nodes ($c \in \mathrm{Pred}(a)$) and see if the newly discovered shortest distance from node a to node z is useful in finding a shorter path from node c to node z. When $D(c, z)$ is successfully updated to a smaller value, the closeness value of node c is also updated in Lines 29–35, similar to the operations performed earlier in Lines 18–25. In addition, the priority queue is also updated to hold the new distance of node c to node z as there might be some other nodes that use $D(c, z)$ as their sub-paths to reach node z. If node c is already an element of the priority queue its priority is updated with the new shortest distance, otherwise it is inserted into the priority queue with $D(c, z)$ as its priority.

4.5 COMMENTS ON ALGORITHMIC COMPLEXITY

4.5.1 Run Time Analysis

In this final section of Chapter 4, the algorithmic complexities of the algorithms proposed in Chapter 4.3 and Chapter 4.4 are discussed. There are different perspectives on how to evaluate the complexities of incremental algorithms. In some cases, it has been demonstrated that, in the worst case, no incremental algorithm can perform asymptotically better than the algorithm that computes everything from scratch [128] [129]. Because, in the worst case, everything will be updated and the incremental algorithm will not be able to perform any early pruning. Hence, the worst-case upper bound time complexity is usually not descriptive enough to discuss the performance benefits of an incremental algorithm observed in practice over the performance of a non-incremental algorithm solving the same problem from scratch.

Such research has led to discussions on what is the best way to evaluate the complexities of incremental algorithms. Complexity analysis for incremental algorithms usually incorporates the complexity of changes for expressing the time complexity of the incremental function. For incremental algorithms, a preferred way of discussing their computational complexity is through the sum of the sizes of the input (e.g. the modified graph/network) and the output (e.g. modified distance and closeness centrality values). Next, we discuss the computational complexities of the *InsertCloseness* and the *DeleteCloseness* algorithms in terms of the changes made in the input and output.

The *InsertCloseness* algorithm calls the *InsertUpdateCloseness* algorithm for every AffectedSink and AffectedSource node. The *InsertUpdateCloseness* algorithm essentially performs a traversal in the neighborhood of every AffectedSink and AffectedSource node, 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 the AffectedSource and AffectedSink nodes' neighborhoods.

Similar complexity analysis can be performed for the *DELETECLOSENESS* algorithm. The *DELETECLOSENESS* algorithm invokes the *DELETEUPDATECLOSENESS* algorithm for every AffectedSink and AffectedSource node. However, the *DELETEUPDATECLOSENESS* algorithm is more complicated than the *INSERTUPDATECLOSENESS* algorithm. The *DELETEUPDATECLOSENESS* algorithm has two distinct phases with different algorithmic complexities. Phase–1 continues until Line–14 and Phase–2 starts with Line–15. Phase–2 makes use of a priority queue, whose time complexity must be taken into account separately.

In Phase–1, the *SP* conditions in Lines 12–13 of the *DELETEUPDATECLOSENESS* algorithm check 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 $O(||\text{Affected}||_{2,z})$ where the subscript 2 denotes the size of two-hop neighborhood of all affected nodes and *z* refers to the last parameter of the algorithm (i.e. the node to which the shortest path distances are updated). The complexity of Phase–2 is dominated by the complexity of priority queue, which is denoted by $O(|\text{Affected}_z| \log |\text{Affected}_z|)$. Hence, the overall time complexity of the *DELETECLOSENESS* algorithm is bounded by $O(||\text{Affected}||_2 + |\text{Affected}| \log |\text{Affected}|)$ where the set of affected nodes is given by the combination of AffectedSink and AffectedSource nodes.

In terms of time complexity, an advantage of the *INSERTUPDATECLOSENESS* algorithm over the *DELETEUPDATECLOSENESS* algorithm is that the

InsertUpdateCloseness algorithm does not maintain a priority queue. Since the growing network updates result in potentially shorter paths, and the shortest distance that is previously known is the minimum of all, the information we would need is directly available without needing to maintain a min-first priority queue. All the changes that are made to update closeness centrality incrementally are of $O(1)$ time complexity. Therefore, computing closeness centrality along with the dynamic maintenance of the shortest paths does not increase the overall time complexity.

4.5.2 Memory Consumption and Overhead Analysis

Next, we discuss the theoretical scaling argument of how the memory usage scales with the problem size for the incremental closeness centrality algorithm. For solving the single-source shortest path problem, the Dijkstra's algorithm has $\mathrm{O}(m+n)$ space complexity. This space requirement increases to $\mathrm{O}(mn+n^2)$ when the Dijsktra's single-source shortest path algorithm is run once from each node in the network to compute and maintain the shortest paths across all node pairs.

The incremental closeness algorithm maintains the original graph and its transpose (reverse) simultaneously. This duality is of critical importance for achieving a bounded incremental update algorithm. Hence, the memory requirement increases to $2m+2n$ for graph representation (i.e. the representation of nodes and edges). The duality is an algorithmic property inherited from the incremental all-pairs shortest paths algorithm designed by Ramalingam and Reps. The incremental closeness centrality algorithm also needs an additional memory space of $\mathrm{O}(n)$ for storing the closeness centrality value of each node. However, the closeness attributes or other attributes do not need to be maintained for the transpose network; they only need to be maintained for the original version of the graph. Hence, the closeness centrality values need to be stored only once in the memory.

The space the incremental closeness centrality algorithm needs to hold the shortest distance values across all pairs of nodes is O(n^2) in the worst case (i.e. O(n^2) distance matrix). To be more precise, for each node, there are $(n - 1)$ other nodes in the network it can reach out to. Hence, the precise upper bound for the number of node pairs for whose distances are defined is $n(n - 1) = n^2$, which is represented as O(n^2) in the worst case in the Big-O notation. We do not need to hold the information for $D(i, i), i \in G$ as $D(i, i)$ is always equal to 0 by default. However, a lot of real life networks are very sparse and the distance matrix does not even need to be stored as an O(n^2) matrix; the sparse representations are preferable. In sparse networks, not every node is reachable from every other node in the network. More precisely, the data structure that holds the shortest distance information has *Conn*(*G*) entries where *Conn*(*G*) represents the number of node pairs in the network that have a finite shortest distance defined. The only piece of information that needs to be maintained for the transpose of the graph is the shortest distances defined on that version of the graph. This adds another *Conn*(*G*) entries to the memory consumption.

Overall, the space complexity of the incremental closeness centrality algorithm is ($2n + 2m + n + 2$*Conn*(*G*)) which reduces to O($n^2 +m$). The space complexity of O($n^2 +m$)

can also be represented as O(n^2) given that the worst case value of m (i.e. the maximum possible number of edges) is O(n^2) as well.

In addition to the amount of data that needs to be stored permanently across several different iterations of the incremental closeness centrality algorithm, another aspect of the memory consumption is the overhead: the amount of data that is temporarily stored during the execution of a single incremental network update, and not maintained across different updates.

There are two sub-algorithms that handle the growing network updates: the *INSERTCLOSENESS* algorithm and the *INSERTUPDATECLOSENESS* algorithm. In the *INSERTUPDATECLOSENESS* algorithm, lists of AffectedVertices and VisitedVertices are maintained as well as a working set of edges (e.g. Workset). Both the AffectedVertices and VisitedVertices might be of order O(n) while the Workset might be of order O(m) in the worst case. Hence, the memory overhead per iteration in the worst case is ($2n + m$), which can be represented as O($n + m$). These data structures are only maintained during the execution of the *INSERTUPDATECLOSENESS* algorithm and cleared once the execution of the algorithm is complete. The *INSERTCLOSENESS* algorithm maintains two sets of nodes: AffectedSinks and AffectedSources, which are of order O(n) each. These two objects are stored during the entire lifetime of an incremental update and cleared once the network update completes successfully.

Similar memory overhead analysis can be performed for temporarily stored data for the algorithms that handle the shrinking network updates. There are two sub-algorithms that handle the shrinking network updates: the *DELETECLOSENESS* algorithm and the *DELETEUPDATECLOSENESS* algorithm. In the *DELETEUPDATECLOSENESS* algorithm, lists of AffectedVertices and VisitedVertices are maintained and a set working nodes (e.g. Workset). The AffectedVertices, VisitedVertices, and Workset are of order O(n) in the worst case. Hence, the memory overhead per iteration in the worst case is $3n$, which can be represented as O(n). These data structures are only maintained during the execution of the *DELETEUPDATECLOSENESS* algorithm and cleared once the execution of the algorithm is complete. Similar to the *INSERTCLOSENESS* algorithm, the *DELETECLOSENESS* algorithm maintains two sets of nodes: AffectedSinks and AffectedSources, which are of order O(n) each. These two objects are stored during the entire lifetime of an incremental update and cleared once the network update completes successfully.

As the analyses done on the performance and memory usage suggest, different network features have different effects on both the performance and memory consumption. For instance, if a network has several small, disconnected components, then the network's diameter and average shortest path length are likely to be smaller than they would be otherwise. In addition, *Conn*(G) is also likely to be smaller, which is one of the main values that make up the memory consumption of the incremental centrality algorithms. Another effect this would have is that the number of nodes that are affected by the incremental network update is likely to be smaller, which reflects as increased performance as the percentage of the affected nodes decreases.

Another feature that might affect the level of connectivity, *Conn*(G), which represents the number of node pairs that are connected, is whether a network is directed or undirected (bidirectional). When a network is undirected, if a node *x* can reach to another node *y*, then node *y* can reach to node *x* as well. However, this is not necessarily true when the edges in a network are directed. Hence, when a network is undirected (bidirectional), *Conn*(*G*) is likely to be higher, which reflects as a potential increase in the memory consumption and the percentage of nodes affected by an incremental network update as well as a potential decrease in the performance speedup that can be obtained.

4.5.3 Comments on Accuracy

The algorithms presented in this chapter are the modified versions of the dynamic shortest path algorithms proposed in [104] to incorporate the *accurate* computation of closeness centrality. In other words, the incremental closeness centrality algorithms presented in this chapter has no loss in the accuracy of closeness centrality values. Both in the *INSERTUPDATECLOSENESS* algorithm and the *DELETEUPDATECLOSENESS* algorithm, the closeness centrality of a node *x* is updated *only* when the shortest distance from *x* to another node is updated. Hence, accurate maintenance of closeness centrality values depends strictly on the accurate maintenance of the shortest distances, whose correctness was proved in [104]. The reader is referred to [104] for more details on the proof of correctness regarding the shortest path updates.

One important point to watch out while implementing these incremental algorithms is how the floating-point numbers are handled. If the intention is to match the results produced by the non-incremental algorithms that compute closeness centrality, then both the incremental and the non-incremental closeness centrality algorithms should handle the floating-point numbers using the same epsilon value, ε. Two real numbers a and b are considered equal if $|a - b| < \varepsilon$, where $|a - b|$ denotes the absolute value of $(a - b)$. The conditions that test whether a path is a shortest path compare the length of the path against the shortest distance to look for equality. Such conditional checks lie at the heart of both the incremental and non-incremental algorithms and govern how the rest of the network update propagates in the network and how the rest of the shortest path trees are built. Hence, the conditional checks that test the equality/inequality of the shortest path distances should be supported by an epsilon range. The epsilon value used in our implementations was 10^{-7}.

For verification, every time the incremental algorithm was run, the standard non-incremental algorithm was also run and the results compared. Once we took the same epsilon into account in both, the results of both the incremental algorithm and the standard algorithm were identical in every case. Further comments on verification can be found in Chapter 8.7.

CHAPTER 5 INCREMENTAL BETWEENNESS

This chapter presents the design of an incremental algorithm for betweenness centrality that handles various types of network updates including addition, removal, and modification of nodes and edges. First, we provide the formulation of betweenness centrality for weighted, directed networks and discuss traditional algorithms for computing betweenness centrality. Then, the proposed incremental betweenness algorithm is presented, along with the details of the sub-algorithms designed FOR handling growing and shrinking network updates. Finally, the algorithmic complexities and generalization to other shortest path based problems are discussed.

5.1 DEFINITION & COMPUTATION OF BETWEENNESS CENTRALITY

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

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

The value of $C_B(i)$ depends on the number of nodes in the network G. To calculate the betweenness centrality of node i, $C_B(i)$ is then normalized by the number of possible node pairs that do not involve i: $\text{Betweenness}(i) = C_B(i)/((n-1)(n-2))$

As a centrality metric based on the shortest paths between nodes, betweenness centrality requires information on the number of shortest paths between any node i and j where $i \neq j$, and the intermediate nodes on each shortest path.

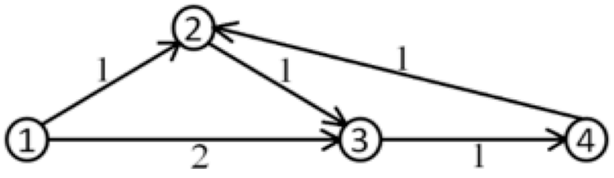

Figure 7 - Example network.

Figure 7 depicts a sample network with edge costs as indicated on the edges. We walk through the computation of the C_B value of each node in the network as laid out in Table 4. Each column in Table 4 represents a node pair (i, j) where $i \neq j$, while rows represent the nodes in the network.

Table 4 - Computation of betweenness centralities for the network depicted in Figure 7.

	1,2	**1,3**	**1,4**	**2,1**	**2,3**	**2,4**	**3,1**	**3,2**	**3,4**	**4,1**	**4,2**	**4,3**
1	--	--	--	--	0	0	--	0	0	--	0	0
2	--	$1/2$	$1/2$	--	--	--	0	--	0	0	--	1
3	0	--	1	0	--	1	--	--	--	0	0	--
4	0	0	--	0	0	--	0	1	--	--	--	--

Consider the computation of $C_B(2)$. From node 1 to node 3, there are two shortest paths: one of them follows the edge $\{1 \rightarrow 3\}$; the other one follows the edges $\{1 \rightarrow 2\}$, $\{2 \rightarrow 3\}$. Out of these two paths, node 2 appears on only one of them. Hence, the contribution of pair $(1, 3)$ to $C_B(2)$ is 1/2. This notion is called 'pair dependency'. In this case, the dependency of the node pair (1, 3), $\delta_{1,3}$, on node 2 is $\delta_{1,3}(2) = 0.5$. Next, consider the computation of $C_B(3)$. When the shortest paths from node 1 to 4 are considered, there are again two different shortest paths: $\{1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 4\}$ and $\{1 \rightarrow 3, 3 \rightarrow 4\}$. However, node 3 appears on both paths. Therefore, the contribution of pair (1, 4) to $C_B(3)$ is 2/2 = 1. The C_B value of each node is calculated as the sum of values at its corresponding row: $C_B(1) = 0$; $C_B(2) = 2$; $C_B(3) = 2$; $C_B(4) = 1$. The normalized betweenness centrality value of each node is then calculated by dividing C_B values by (3 × 2) = 6, which is equal to the total number of pairs that do not involve the node whose betweenness value is computed.

5.2 TRADITIONAL ALGORITHMS FOR COMPUTING BETWEENNESS CENTRALITY

In its most naïve form, betweenness can be computed by augmenting an all-pairs shortest paths algorithm (e.g. Floyd-Warshall [126]) with path counting. Having obtained the information on the number of shortest paths between each node pair and the intermediate nodes on these shortest paths, a $O(n^3)$ nested for loop (e.g. $O(n)$ invocations of a $O(n^2)$ loop to sum up dependencies for each node) is then run to compute the $C_B(i)$ of each node i, where n is the number of nodes in the network.

In 2001, Brandes published his research on faster computation of betweenness centrality which yields $O(nm + n^2 \log n)$ performance for a weighted network where n is the number of nodes in the network and m is the number of edges [28]. Brandes' algorithm has become the most commonly used implementation of betweenness centrality.

```
Algorithm–10: BRANDES' BETWEENNESS ALGORITHM (G)
Input: A network $G(V, E)$.
Output: Betweenness values of all nodes $n$ $V(G)$ are computed.
1. $C_B[\,] \leftarrow 0, v \in V(G)$
2. for $s \in V(G)$
3.     $S \leftarrow$ empty stack
4.     $P[w] \leftarrow$ empty list, $w \in V(G)$
5.     $\sigma[t] \leftarrow 0,\ t \in V(G); \sigma[s] \leftarrow 1$
6.     $d[t] \leftarrow -1, t \in V(G); d[s] \leftarrow 0$
7.     $Q \leftarrow$ empty queue
8.     enqueue $s$ into $Q$
9.     while $Q \neq \emptyset$
10.        dequeue $v \leftarrow Q$
11.        push $v$ into $S$
12.        for neighbor $w$ of $v$
13.            if $d[w] < 0$
14.                enqueue $w$ into $Q$
15.                $d[w] \leftarrow d[v] + 1$
16.            if $d[w] = d[v] + 1$
17.                $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$
18.                append $v$ into $P[w]$
19. $\delta[v] \leftarrow 0, v \in V(G)$
20. while $S \neq \emptyset$
21.    pop $w \leftarrow S$
22.    for $v \in P[w]$
23.        $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]} .(1 + \delta[w])$
24.    if $w \neq s$
25.        $C_B[w] \leftarrow C_B[w] + \delta[w]$
```

The idea of Brandes' algorithm is to exploit the sparseness of large, real life networks, hence to avoid some of the superfluous work done in $O(n^3)$ algorithms. To exploit sparseness, path counting is performed by running the Dijkstra's single-source shortest path algorithm [109] once for each node, which also naturally discovers all the shortest paths from the selected source node and the predecessors on these paths. Thus, the Brandes' algorithm benefits from avoiding the computation of unrelated pair dependencies in the network by only focusing on the nodes in the predecessor lists on the shortest paths discovered by the Dijsktra's shortest path algorithm.

The pseudocode for the Brandes' betweenness algorithm provided in Algorithm–10 is the original version of the algorithm as discussed in his paper [28]. This version of the algorithm is designed for computing betweenness centrality in unweighted graphs. This is why in Lines 15 and 16 of Algorithm–10, we use increments of distance by 1, instead of using the cost of the corresponding edges between w and v. To make it work for weighted networks, $d\ [v] + 1$ should be replaced with $d\ [v] + C(v, w)$ where $d\ [v]$ denotes the distance of from the source node s to node v and $C(v, w)$ denotes the cost of the edge $\{v \rightarrow w\}$. In addition, this version of the algorithm makes it more suitable for undirected edges because in Line 12, all neighbors of node v are considered without considering the edge orientation. In directed networks, only the successors of node v should be considered: 'for *neighbor* w of v' should be replaced with 'for $w \in \text{Succ}(v)$'. However, the most important difference in converting the Brandes' algorithm presented above for the weighted, directed networks comes from the implementation of the queue Q, which is implemented as a higher complexity, min-key priority queue data structure to handle the differences in distances introduced by the differences in edge weights. The current distance of a node from the current source node is used as its priority for the priority queue.

5.3 Incremental Betweenness Algorithm: Growing Network Updates

This chapter discusses the details of the incremental betweenness algorithm for handling growing network updates (e.g. edge/node insertion or edge cost decrease). There are four sub-algorithms we make use of to incrementally maintain betweenness centrality values: *InsertBetweenness*, *InsertUpdateBetweenness*, *ReduceBetweenness*, and *IncreaseBetweenness*. Next, each sub-algorithm is discussed respectively.

5.3.1 InsertBetweenness

In the event of a supported network update (e.g. node/edge insertion or edge cost decrease), the entry point of execution is the *InsertBetweenness* algorithm. The *InsertBetweenness* algorithm calls the *InsertUpdateBetweenness* algorithm several times; first, to find the complete set of affected sink and source nodes, then, to update the shortest paths to/from each affected sink/source node.

The data structures initialized in Line 2 of the *InsertBetweenness* algorithm are initialized as data structures that are visible to all the algorithms used for handling the growing network updates.

The D_{old} and σ_{old} are implemented as hash maps whose keys are composed of the related $<x, y>$ node identifiers. The D_{old} holds the original $D(x, y)$ values and the σ_{old} values hold the original $\sigma(x, y)$ values before the network update has been issued. The data structure used for *trackLost* is also a hash map whose keys are again constructed as

2-tuples composed of the identifiers of the related nodes <*x*, *y*>. The values held in *trackLost* are also implemented as hash set, holding the identifiers of the nodes that were previously intermediates on the shortest paths from node *x* to node *y*, and that are not on the shortest paths any more. Finally, to avoid reprocessing the same pair of nodes several times, the *PairsDone* holds a set of node pairs <*x*, *y*> that are already processed.

Algorithm–11: *INSERTBETWEENNESS* (*G, src, dest, c*)

Input: Network $G(V, E)$ for which shortest distances between all nodes (D) and betweenness values of all nodes are pre-computed. A newly inserted or modified edge $\{src \rightarrow dest\}$ with a cost of c where $c < C(src, dest)$.

Output: Network $G(V, E)$ updated with the edge $\{src \rightarrow dest\}$. The shortest path distances (D), the number of shortest paths (σ), the predecessors on the shortest paths (P), and the betweenness values of all nodes are updated.

```
1. $C(src, dest) \leftarrow c$;   $\tilde{C}(dest, src) \leftarrow c$
2. $\sigma_{old} \leftarrow [\,]$; $D_{old} \leftarrow [\,]$; $trackLost \leftarrow [\,]$; $PairsDone \leftarrow [\,]$
3. AffectedSinks    ← INSERTUPDATEBETWEENNESS ($\tilde{G}$, dest, src, src, PairsDone)
4. AffectedSources ← INSERTUPDATEBETWEENNESS ($G$, src, dest, dest, PairsDone)
5. for $s \in$ AffectedSinks
6.     INSERTUPDATEBETWEENNESS ($G$, src, dest, s)
7. for $s \in$ AffectedSources
8.     INSERTUPDATEBETWEENNESS ($\tilde{G}$, dest, src, s)
9. INCREASEBETWEENNESS( )
```

In the *INSERTBETWEENNESS* algorithm, the *INSERTUPDATEBETWEENNESS* algorithm is initially invoked twice (Lines 3–4 of *INSERTBETWEENNESS*) to determine the sets of affected sinks and affected sources; passing once the source, once the destination of the inserted edge as a parameter to it. The *INSERTUPDATEBETWEENNESS* algorithm is then invoked for each affected sink and source node (Lines 5–8 of *INSERTBETWEENNESS*) to modify the information required for accurate maintenance of betweenness values. The modified information includes the shortest distances (D), the number of distinct shortest paths (σ), and the predecessors on these shortest paths (P). After all the shortest distances, predecessors, and the shortest path counts are updated accurately for all the affected nodes, betweenness values of the intermediate nodes that lie on the affected paths are adjusted (Line 9 of *INSERTBETWEENNESS*). The second line of the *INSERTBETWEENNESS* algorithm initializes auxiliary data that are only used during the current update, and are not maintained across different updates.

5.3.2 INSERTUPDATEBETWEENNESS

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

(i) If a strictly shorter path is found, the shortest path distance is updated. The predecessors and the number of shortest paths are cleared. Betweenness values for the intermediate nodes on the cleared paths are also reduced opportunistically. (Lines 7–14 of *INSERTUPDATEBETWEENNESS*).

(ii) If the shortest paths have changed in any way (number or length), the predecessors and the number of shortest paths are adjusted accordingly (Lines 15–25 of *INSERTUPDATEBETWEENNESS*).

(iii) Propagation of the update across the network is continued if appropriate (Lines 26–29 of *INSERTUPDATEBETWEENNESS*).

Consider the case where the edge $\{x \rightarrow y\}$ is updated and we examine the impact of this update on the network. In the first phase of the *INSERTUPDATEBETWEENNESS* algorithm, assume that there is now a new shortest path from node x to node z passing through the edge $\{x \rightarrow y\}$, which is strictly shorter than the previously known shortest path(s) from node x to node z (Lines 7–14 of *INSERTUPDATEBETWEENNESS*).

In this case, since a strictly shorter path from node x to node z is found, the previously known shortest paths are not the shortest paths from node x to node z anymore. Hence, the previously known number of shortest paths and the predecessors on the shortest paths from node x to node z should be cleared (Line 11 of *INSERTUPDATEBETWEENNESS*). Before we clear the number of shortest paths (σ (x, z)) and update the distance from node x to node z (D (x, z)) to be equal to the new distance (*alt*), we temporarily record their values in σ_{old} (x, z) and D_{old} (x, z) in Line 9 of the *INSERTUPDATEBETWEENNESS* algorithm. In addition, the betweenness values of the old predecessors because these intermediates do not have any contribution from (x, z) pair anymore are reduced as well (Line 10 of *INSERTUPDATEBETWEENNESS*). This operation should be performed before clearing the number of shortest paths and the predecessors on the shortest paths from node x to node z, because once they are cleared, all the previously known information about them is lost. Attempting to retrieve σ_{old} and D_{old} values returns the temporarily stored values if they are stored and returns the current σ and D values otherwise. The algorithm for the reduction of betweenness centrality values (*REDUCEBETWEENNESS*) will be explained later in Chapter 5.3.3.

At the beginning of the first phase, we check if it is the first time a strictly shorter path is found from node *x* to node *z* by checking if σ_{old} contains any information on the pair (*x*, *z*) (Line 8 of *INSERTUPDATEBETWEENNESS*). We check σ_{old} because a change in betweenness centrality values is required if the number of shortest paths from node *x* to node *z* change even if the shortest distance from node *x* to node *z* does not necessarily change. For every updated pair, we record the original number of shortest paths from node *x* to node *z* known before the update started only once. This check ensures that original information is not overridden by invalid, temporary information because during the time the network update propagates the algorithm might go through states that temporarily contain invalid information.

Additional temporary information is kept to ensure accurate reduction of betweenness centrality values for the nodes that were once on the shortest paths before the network is updated. This is because such nodes are not on the shortest paths anymore and cannot be reached by following the shortest paths in the network. Hence, information should be kept about them temporarily until the network update is complete.

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

AffectedVertices should be expanded to include the nodes that have new alternative shortest path(s) to/from node z passing through the modified edge $\{src \rightarrow dest\}$.

In the second phase of the *INSERTUPDATEBETWEENNESS* algorithm (Lines 15–25), we check if the new shortest distance from node x to node z is now equal to the cost of the alternative shortest path passing through the edge $\{x \rightarrow y\}$ (Line 15). If they are equal, then we need to update the number of shortest paths from x to z and the predecessors on these shortest paths. This part of the algorithm is completely new and is not covered in the original Ramalingam and Reps algorithm, which focuses on finding strictly shorter paths and excludes the shortest paths of equivalent length.

When updating the number of shortest paths from node x to node z, we increase the number of shortest paths only by the number of shortest paths that are newly formed due to the incremental update made on the network. Since this is an incremental algorithm, the old set of shortest paths is already accounted for; we should not double count them. To obtain the number of *newly* formed shortest paths from node x to node z, only the number of shortest paths that use the modified or inserted edge $\{src \rightarrow dest\}$ should be counted. The number of newly discovered paths is calculated as $\sigma(x, src) * 1 * \sigma(dest, z)$, and then added to the $\sigma(x, z)$ to calculate the total number of shortest paths from x to z. From *src* to *dest*, there may be other shortest paths that might already be counted in. Hence, to avoid double counting, we only consider the modified edge, which is represented with the '1' in the above given formulation (Line 21 of *INSERTUPDATEBETWEENNESS*).

Algorithm–12: *INSERTUPDATEBETWEENNESS* (*G*, *src*, *dest*, *z*, *PairsDone*)

Input: Network $G(V, E)$, a newly inserted or modified edge $\{src \rightarrow dest\}$, and an affected node z.

Output: The shortest distances to z. The number of shortest paths and the predecessors on these shortest paths are also updated.

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

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

It should be noted that the entry condition for the second phase of the *INSERTUPDATEBETWEENNESS* algorithm (Line 15) is not a condition that is tied to the if block between Lines 7–14. Once the condition in Line 7 ($alt < D(x, z)$) is satisfied (i.e. a strictly shorter path is found), the value of $D(x, z)$ is updated in Line 9 to be equal to the newly found alternative distance *alt*. Therefore, the condition in Line 15 of the *INSERTUPDATEBETWEENNESS* algorithm ($alt = D(x, z)$) is satisfied for all cases that originally satisfied the conditional check at Line 7 ($alt < D(x, z)$).

The conditional check at Line 15 ($alt = D(x, z)$) covers additional cases where there are newly formed alternative shortest paths whose lengths are equal to what is already known. Such cases would not satisfy the condition on Line 7 which checks for strictly shorter paths, but would still satisfy condition at Line 15 ($alt = D(x, z)$). This is a part of the incremental betweenness algorithm that is not handled by the original Ramalingam and Reps algorithm at all, and it is required for accurate maintenance of betweenness centrality values. Since the second phase of the *INSERTUPDATEBETWEENNESS* algorithm is satisfied by both strictly shorter paths and the shortest paths of equal length, in Line 25, we mark node x as one of the AffectedVertices whose predecessors should be further checked in the third phase of the algorithm to understand if the incremental network update has a wider impact on the network. In order not to reprocess the same pair of nodes multiple times –which would inflate the number of shortest paths between them otherwise–, we process only the node pairs that are not already processed and inserted in *PairsDone* (Line 16 and Line 24 of *INSERTUPDATEBETWEENNESS*)

The third and final phase of the *INSERTUPDATEBETWEENNESS* algorithm (Lines 26–29) is responsible for propagating updates further when required and pruning the parts of the network that are not affected by the changes. An update in the network propagates in the form of ripples that expand outwards as much as required starting at the modified or inserted edge in the center. For each of the edges to/from the affected node x, it is checked to see if they are on the shortest paths. In addition, a set of VisitedVertices is held to ensure that no node is processed more than once due to the potential existence of multiple paths that lead to the same node following different routes. Assume that node u is a predecessor of node x. With SP(u, x, z), we test if the edge $\{u \rightarrow x\}$ is on the shortest path(s) from node u to node z. If SP is true, and if the other end of the edge (node u) is not in the list of already processed nodes, the edge $\{u \rightarrow x\}$ is inserted in the set of edges that would need inspection for subsequent processing. In this case, the edge $\{u \rightarrow x\}$ would carry the network update to the next ripple level and it is identified as an edge that needs further processing. Therefore, the edge $\{u \rightarrow x\}$ is inserted into Workset in Line 28 of the *INSERTUPDATEBETWEENNESS* algorithm for subsequent processing.

5.3.3 REDUCEBETWEENNESS

Within the *INSERTUPDATEBETWEENNESS* algorithm, the *REDUCEBETWEENNESS* algorithm is invoked when the shortest path(s) from a node x to another node z change. For each node pair whose shortest paths change, the *REDUCEBETWEENNESS* algorithm is invoked only once and it opportunistically reduces the betweenness centrality values of the intermediate nodes on the old set of shortest paths from node x to node z that no longer qualify to be the shortest paths. To be able to construct the shortest paths, we only store predecessors; we do not store the entire set of intermediate nodes on these paths, as it would be prohibitively memory intensive.

The shortest paths from node x to node z are constructed on demand by following the predecessors when the betweenness centrality values of the intermediate nodes on the shortest path(s) from x to z are updated. However, since these shortest paths are constructed on demand, there might be some parts of the shortest paths that might have already been updated before the network update propagation reaches the shortest paths/distance from x to z. In such cases, there will be some intermediate nodes that are already cleared and that are not reachable anymore when we follow the predecessor nodes to adjust the betweenness centrality values of the intermediates nodes on the old set of shortest paths. The intermediate nodes that are deleted from the shortest paths are temporarily stored in *trackLost* until the network update is complete.

Algorithm–13: *REDUCEBETWEENNESS* (*x*, *z*)
Input: A source node x and a destination node z whose intermediates on the shortest paths from node x to node z will be cleared.
Output: The betweenness values of the original set of intermediate nodes on the shortest paths from x to z are reduced.

```
1.  if $\sigma_{old}(x, z) = 0$
2.    return;
3.  $Known \leftarrow \emptyset$ //A set of nodes
4.  $Stack \leftarrow \emptyset$
5.  for $n \in P_x(z)$
6.    if $D(x, z) \neq D_{old}(x, n) + D_{old}(n, z)$
7.      continue;
8.    else if $x \neq n$ & $n \neq z$
9.      $C_B(n) \leftarrow C_B(n) - (\sigma_{old}(x, n) * \sigma_{old}(n, z) / \sigma_{old}(x, z))$
10.     Add $<x, z, n>$ to $trackLost$
11.   Add $n$ to $Stack$
12.   Add $n$ to $Known$
13. while $Stack \neq \emptyset$
14.   $p \leftarrow$ pop ($Stack$)
15.   Add $p$ to $Known$
16.   for $n \in P_x(p)$
17.     if $D(x, z) \neq D_{old}(x, n) + D_{old}(n, z)$
18.       continue;
19.     else if $x \neq n$ & $n \neq z$ & $n \notin Known$
20.       $C_B(n) \leftarrow C_B(n) - (\sigma_{old}(x, n) * \sigma_{old}(n, z) / \sigma_{old}(x, z))$
21.       Add $<x, z, n>$ to $trackLost$
22.       Add $n$ to $Stack$
23.       Add $n$ to $Known$
24. $AlreadyDone \leftarrow (Known \cup x)$
25. if $D(v, z) = D_{old}(x, v) + D_{old}(v, z)$ s.t. $v_1, v_2 \in Known$ and $<v_1, v_2, v> \in trackLost$
26.   if $v \notin AlreadyDone$
27.     $C_B(v) \leftarrow C_B(v) - (\sigma_{old}(x, v) * \sigma_{old}(v, z) / \sigma_{old}(x, z))$
28.     Add $v$ to $AlreadyDone$
29.     Add $<x, z, v>$ to $trackLost$
```

In the first phase of the *REDUCEBETWEENNESS* algorithm (Lines 1–23), first, the betweenness value of each intermediate node v that still appears as an intermediate from node x to node z is reduced and the contribution of the node pair (x, z) is removed from the betweenness value of node v. In Lines 5–12, the immediate predecessors of node z on the shortest paths from node x ($P_x(z)$) are processed, and in Lines 13–23, the intermediates that are found on the paths from node x to node z are processed.

When tracing intermediates on the shortest paths to construct the shortest paths on demand, we start with the destination node. The first batch of nodes we find as the intermediates on the shortest paths from node x to node z are the predecessors of node z: $P_x(z)$. Then for each node $n \in P_x(z)$, we find the next level of precessors on the shortest paths from node x to node n: $P_x(n)$. The search for a shortest paths ends when we hit the source node x itself. This way, the full path from node x to node z is constructed on demand. Hence, when are about to update the betweenness centrality of an intermediate node n, it should be checked that node n is an intermediate node, it is neither the source, nor the destination node.

In the second phase of the *REDUCEBETWEENNESS* algorithm (Lines 24–29), the intermediate nodes that are currently unreachable but originally belonged to the shortest paths from node x to node z are processed. Their betweenness values are reduced as required as well (Lines 27).

An intermediate node n might appear on multiple shortest paths from the same source and destination pair and its betweenness value should not be reduced every time it is encountered, it should be reduced once for a single source and destination pair (e.g. x and z). Therefore, a set of intermediate nodes that are already processed is temporarily stored to avoid processing such nodes several times.

5.3.4 INCREASEBETWEENNESS

The *INCREASEBETWEENNESS* algorithm is the final step of the *INSERTBETWEENNESS* algorithm. The *INCREASEBETWEENNESS* algorithm is responsible from finalizing the adjustment of betweenness centrality values. By the time the *INCREASEBETWEENNESS* is called from the *INSERTBETWEENNESS* algorithm (Line 10 of *INSERTBETWEENNESS*), all the shortest paths, the number of distinct shortest paths, and the predecessors affected by the incremental network update are accurately adjusted, and ready for being used for updating the betweenness centrality values. By calling the *REDUCEBETWEENNESS* algorithm, we have also reduced the betweenness centrality values of the intermediate nodes on the invalidated shortest paths. Therefore, the only remaining action is to update the betweenness centrality values for the new set of intermediate nodes on the updated shortest paths that reach from the affected source nodes to the affected sink nodes.

During a network update, a node pair (x, z) is added to the σ_{old} set if their shortest paths change in any way (length or number). For each node pair (x, z) that is recorded in the σ_{old} set, there are a number of intermediate nodes that lie on the shortest paths from node x to node z. For each intermediate node n that lies on the shortest paths from node x to node z, we first compute how many of these paths pass through node n. Out of all the shortest paths from node x to node z, $(\sigma(x, n) * \sigma(n, z))$ of them pass through node n. Then, we increase the betweenness centrality value of node n by $(\sigma(x, n) * \sigma(n, z) / \sigma(x, z))$; the fraction of shortest paths from node x to node z that pass through node n over the total number of shortest paths from node x to node z (Line 8 and 16). Similar to the *REDUCEBETWEENNESS* algorithm, again a set of already processed intermediates (e.g.

Known) is stored temporarily. With the execution of *INCREASEBETWEENNESS*, incremental update of betweenness centrality values is complete.

Algorithm–14: *INCREASEBETWEENNESS* (σ_{old})
Input: The set of affected node pairs whose shortest paths have changed.
Output: The betweenness values of the current intermediate nodes on the affected paths are increased in accordance with the fraction of shortest paths they are on.

```
1.  for $(src, dest) \in \sigma_{old}$
2.    $Known \leftarrow \emptyset$
3.    $Stack \leftarrow \emptyset$
4.    for $n \in P_{src}(dest)$
5.      Add $n$ to $Stack$
6.      Add $n$ to $Known$
7.      if $src \neq n$ & $n \neq dest$
8.        $C_B(n) = C_B(n) + (\sigma(src, n) * \sigma(n, dest) / \sigma(src, dest))$
9.    while $Stack \neq \emptyset$
10.     $n \leftarrow$ pop ($Stack$)
11.     Add $n$ to $Known$
12.     for $p \in P_{src}(n)$
13.       if $p \neq src$ & $p \neq dest$ & $p \notin Known$
14.         Add $p$ to $Stack$
15.         Add $p$ to $Known$
16.         $C_B(p) = C_B(p) + (\sigma(src, p) * \sigma(p, dest) / \sigma(src, dest))$
```

5.4 INCREMENTAL BETWEENNESS ALGORITHM: SHRINKING NETWORK UPDATES

Next, the details of the part of the incremental betweenness algorithm that handles the shrinking network updates (e.g. edge/node deletion or edge cost increase) are discussed. There are five sub-algorithms that are used to achieve accurate maintenance and incremental update of betweenness centrality values: *DELETEBETWEENNESS*, *DELETEUPDATEBETWEENNESS*, *CLEARBETWEENNESS*, *ADJUSTNPS*, and *ADJUSTBETWEENNESS*. In the rest of this section, each sub-algorithm is discussed, respectively.

5.4.1 DELETEBETWEENNESS

When a shrinking network update is issued (e.g. edge/node deletion or edge cost increase), the entry point of execution is the *DELETEBETWEENNESS* algorithm. The *DELETEBETWEENNESS* algorithm calls the *DELETEUPDATEBETWEENNESS* algorithm several times; first, to find the complete set of affected sink and source nodes, then, to update the shortest paths to/from each affected sink/source node.

The *DELETEBETWEENNESS* algorithm resembles the *INSERTBETWEENNESS* algorithm. However, it has minor differences. The most important difference is the call for an algorithm called *ADJUSTNPS*, which is used to calculate the number of shortest paths accurately. After all the shortest distances and the predecessors on the shortest paths are updated accurately, the call to the *ADJUSTNPS* algorithm (Line 9 of *DELETEBETWEENNESS*) finalizes the computation of the number of shortest paths and the call to the *ADJUSTBETWEENNESS* algorithm (Line 10 of *DELETEBETWEENNESS*) completes the computation of betweenness centrality values.

The data structures initialized in Line 2 of the *DELETEBETWEENNESS* algorithm are initialized as data structures that are visible to all the algorithms used for handling the shrinking network updates. The data structures initialized in Line 2 of the *DELETEBETWEENNESS* algorithm have similar structures and usages similar to those in the *INSERTBETWEENNESS* algorithm. The D_{old} and σ_{old} are implemented as hash maps. The keys for the data structure D_{old} are composed of the related $<x, y>$ node identifiers, which holds the original $D(x, y)$ value before the network update has been issued. Similarly, the keys for the data structure σ_{old} are composed of the related $<x, y>$ node identifiers, which holds the original $\sigma(x, y)$ value before the network update has been issued. The data structure used for *trackLost* is also a hash map whose keys are again constructed as the identifiers of the related nodes $<x, y>$. The values held in *trackLost* are implemented as hash set, holding the identifiers of the nodes that were previously intermediates on the shortest paths from node x to node y, and that are not on the shortest paths any more.

Algorithm–15: *DELETEBETWEENNESS* (*G, src, dest, c*)
Input: Network $G(V, E)$ for which betweenness values of all nodes (C_B) are pre-computed. A modified edge $\{src \rightarrow dest\}$ with a cost of c, where $c > C(src, dest)$. The cost parameter c is optional. If it is not provided in the argument list, it is set to ∞ by default and the edge is deleted.
Output: Network $G(V, E)$ updated with the changes on the edge $\{src \rightarrow dest\}$. The shortest distances between all nodes (D), the number of shortest paths, the predecessors on the shortest paths and betweenness values of all nodes (C_B) are also updated.

```
1.  $C(src, dest) \leftarrow c$;  $\tilde{C}(dest, src) \leftarrow c$
2.  $\sigma_{old} \leftarrow [\,]$; $D_{old} \leftarrow [\,]$; $trackLost \leftarrow [\,]$
3.  AffectedSinks    ← DELETEUPDATEBETWEENNESS ($\tilde{G}, dest, src, src$)
4.  AffectedSources  ← DELETEUPDATEBETWEENNESS ($G, src, dest, dest$)
5.  for $s \in$ AffectedSinks
6.      DELETEUPDATEBETWEENNESS ($G, src, dest, s$)
7.  for $s \in$ AffectedSources
8.      DELETEUPDATEBETWEENNESS ($\tilde{G}, dest, src, s$)
9.  ADJUSTNPS ( )
10. ADJUSTBETWEENNESS ( )
```

5.4.2 DELETEUPDATEBETWEENNESS

The core of the incremental betweenness algorithm for handling the shrinking network updates is the *DELETEUPDATEBETWEENNESS* algorithm.

Similar to the *DeleteUpdateCloseness* algorithm, there are two distinct phases of the *DeleteUpdateBetweenness* algorithm. The first phase of the *DeleteUpdateBetweenness* algorithm is between Lines 2 – 7 while the second phase is between Lines 8 – 40. The first phase of the algorithm identifies the set of affected nodes. For betweenness centrality, the affected nodes are the nodes whose shortest paths to node *z* have changed in terms of number or length. The shortest paths from a node *x* to another node *z* may change only if the deleted/modified edge is an edge that used to lie on the shortest path(s) from node *x* to node *z*. Such nodes are inserted into the AffectedVertices set for further processing to find the new shortest paths from each node *x* to node *z*.

The second phase of the *DeleteUpdateBetweenness* algorithm determines the new shortest path distances from all affected nodes to node *z* as well as the predecessors on the shortest paths. In the second phase of the *DeleteUpdateBetweenness* algorithm, the betweenness centrality values are also opportunistically updated for the node pairs whose previously known shortest paths are invalidated.

In Line – 11, one of the AffectedVertices, node *a*, is removed from the AffectedVertices for finding the new shortest path(s) from it to node *z*. If this is the first time, the shortest paths from node *a* to node *z* are examined, we insert the node pair <*a*, *z*> into σ_{old} and D_{old} to keep a record of their previously known shortest distance and shortest path counts before any update is made on them. In Line – 14, by setting *myMin* to infinity, we start with the assumption that we do not have a shortest path from node *a* to node *z* anymore. Since a shrinking network update might result in disconnecting the two nodes *a* and *z* and making node *z* unreachable from node *a*. In order for node *a* to reach to node *z*, there needs to be at least one immediate neighbor of node *a* that connects node *a* to node *z*. Therefore, each successor *b* of node *a* is examined one by one to see which node or nodes provide the shortest path(s) to node *z* and a running min is kept to identify the minimum shortest path distance. If we are unable to find a new shortest path that would pass from at least one of the successor nodes of node *a*, then there would be no node *b* that would satisfy the condition on Line 16, and the value of *myMin* has to be chosen as infinity. This process of discovering the new shortest path(s) from node *a* to node *z* *incrementally* builds on the previous knowledge on the shortest paths as we would need to know the shortest path distance from each successor node *b* to node *z*. Every time we find a node *b* that satisfies one of the 'shorter path' (Line 19) or 'equivalent to the shortest path(s)' (Line 21) conditions, the predecessors on the shortest paths from node *a* to node *z* need to be updated. When a new path from node *a* to node *z* that is strictly shorter than the currently known shortest paths is found (Line 19), the set of predecessors on the shortest paths from node *a* to node *z* is cleared (Line 18), and the set of predecessors from node *b* to node *z* is inserted instead. However, if node *b* is equal to node *z* in Line 21 (i.e. node *a* can reach to node *z* in one hop and the shortest path is actually the edge from node *a* to node *z*), only node *a* is inserted into the set of predecessors on the shortest paths from node *a* to node *z*.

After all the successor nodes are examined, in Line 22, the shortest path distance from node *a* to node *z*, *D*(*a*, *z*), is updated to hold the value of the running min, *myMin*. In Line 24, node *a* is inserted into the priority queue that holds the distances to node *z* in ascending fashion. A node is inserted into this priority queue only if its shortest distance to node z is changed and if it is still able to reach to node z with the possibility of discovering a shorter path from node a to node z as the update continues to propagate in the network.

The rest of the *DELETEUPDATEBETWEENNESS* algorithm, Lines 25 – 40, processes this priority queue to see if the shortest distances can be further updated. In the first part of the algorithm, if there is no node b that would satisfy the condition in Line 16 or Line 20, then no shortest paths from node a to node z were found and the shortest distance from node a to node z is forced to be infinity. Starting with Line 25 of the *DELETEUPDATEBETWEENNESS* algorithm, the shortest paths that are properly discovered earlier in the algorithm are examined to see if new shortest paths that are even shorter can be discovered. The impact of an incremental network update propagates in the form of ripples expanding outward from the modification point of the update. Hence, it may be possible to find shorter paths in the outer levels of these update ripples that use shorter paths from earlier levels.

For each node a inserted in the priority queue Q_inc, we check the predecessors of node a ($c \in \text{Pred}(a)$) and see if the newly discovered shortest distance from node a to node z is useful in finding a shorter path from node c to node z. For each node c, we check if the path from node c to node z which uses the edge $\{c \rightarrow a\}$ is shorter than or equal to the currently known shortest paths from node c to node z.

Algorithm–16: *DELETEUPDATEBETWEENNESS* (*G*, *src*, *dest*, *z*)

Input: Network $G(V, E)$, deleted or modified edge $\{src \rightarrow dest\}$, and an affected node z.
Output: The shortest distances (D) to the affected node z and the predecessors are updated.

```
1.  AffectedVertices ← $\emptyset$
2.  Workset ←{$src$};
3.  while Workset $\neq \emptyset$
4.    $u$ ←pop (Workset)
5.    if $u \notin$ AffectedVertices then add $u$ to AffectedVertices
6.    for $x \in$ Pred($u$) such that SP($x, u, z$)
7.      add $x$ into Workset
8.  AffVert ← AffectedVertices.copy()
9.  Q_inc ← $\emptyset$
10. while AffVert $\neq \emptyset$
11.     $a$ ← extractMin(AffVert)
12.     if($< a, z > \notin \sigma_{old}$ )  then add $< a, z, \sigma(a, z) >$ into $\sigma_{old}$ and CLEARBETWEENNESS ($a, z$)
13.     if ($< a, z > \notin D_{old}$) then add $< a, z, D(a, z) >$ into $D_{old}$
14.     myMin ← $\infty$
15.     for $b \in$ Succ($a$)
16.        if ($C(a, b) + D(b, z) <$ myMin)
17.          myMin← $C(a, b) + D(b, z)$
18.          $P_a(z) \leftarrow \emptyset$
19.          if $b$ = z ? Append $a$ to $P_a(z)$: Append $P_b(z)$ to $P_a(z)$
20.        else if ($C(a, b) + D(b, z)$ = myMin & myMin $\neq \infty$ & $b \notin$ AffectedVertices)
21.          if $b$ = z ? Append $a$ to $P_a(z)$: Append $P_b(z)$ to $P_a(z)$
22.     $D(a, z)$ ← myMin
23.     if myMin $\neq \infty$ & $a \notin$ Q_inc
24.        add $a$ into Q_inc
25. while Q_inc $\neq \emptyset$
26.     $a$ ← extractMin(Q_inc)
27.     for $c \in$ Pred($a$)
28.        if ($C(c, a) + D(a, z) < D(c, z)$ & $c \notin$ AffectedVertices)
29.          if($< c, z > \notin \sigma_{old}$ )  then add $< c, z, \sigma(c, z) >$ into $\sigma_{old}$ and CLEARBETWEENNESS ($c, z$)
30.          if ($< c, z > \notin D_{old}$) then add $< c, z, D(c, z) >$ into $D_{old}$
31.          $P_c(z) \leftarrow \emptyset$
32.          if $a$ = z ? Append $c$ to $P_c(z)$: Append $P_a(z)$ to $P_c(z)$
33.          $D(c, z) \leftarrow C(c, a) + D(a, z)$
34.          if $c \notin$ Q_inc
35.            add $c$ into Q_inc
36.        else if ($C(c, a) + D(a, z) = D(c, z)$ & $c \notin$ AffectedVertices)
37.          if($< c, z > \notin \sigma_{old}$ )  then add $< c, z, \sigma(c, z) >$ into $\sigma_{old}$ and CLEARBETWEENNESS ($c, z$)
38.          if $a$ = z ? Append $c$ to $P_c(z)$: Append $P_a(z)$ to $P_c(z)$
39.          if $c \notin$ Q_inc
40.            add $c$ into Q_inc
41. return AffectedVertices
```

When a shorter path from node c to node z is found (Line 28), we also check if the node c is not in the list of affected nodes (AffVert). If node c is not an element of AffVert, we check if the node pair $< c, z >$ has been updated before during the propagation of the current network update. If it is the first time the shortest paths from node c to node z are to be updated, to keep track of the original values known before the network update started, the previously known shortest distance from node c to node z and the shortest path count from node c to node z are inserted into D_{old} and σ_{old}, respectively. In addition, the betweenness centrality values of the predecessors on the shortest paths from node c to node z are reduced as required by the *CLEARBETWEENNESS* algorithm in Line 29 of the *DELETEUPDATEBETWEENNESS* algorithm, and then cleared in Line 31. Lines 32 of the *DELETEUPDATEBETWEENNESS* algorithm updates the predecessors on the shortest paths from node c to node z to include the predecessors on the shortest paths from node c to node z that pass through node a while Line 33 updates $D(c, z)$, the shortest distance from node c to node z, to be the new shortest distance discovered through node a. Following a similar reasoning to the first part of the algorithm, in Lines 34 – 35, node c is inserted into the priority queue Q_inc to see if there are any shortest paths that might use the shortest paths from node c to node z as their subpaths to reach to node z.

When an equivalent shortest path from node c to node z is found (Line 36) and node c is not an element of AffVert, we go through steps that are similar to those we go through when a new shorter path is found. First, we check if the node pair $< c, z >$ has been updated before during the propagation of the current network update. If it is the first time the shortest paths from node c to node z are to be updated, to keep track of the original values, the previously known shortest path count from node c to node z is inserted into σ_{old}. In addition, the betweenness values of the predecessors on the shortest paths from node c to node z are reduced as required by calling the *ClearBetweenness* algorithm in Line 37 of the *DeleteUpdateBetweenness* algorithm. However, different from the case when a strictly shorter path is found, we do not clear the previously known predecessors on the shortest paths from node c to node z. Following a similar reasoning to the earlier parts of the algorithm, in Lines 39 – 40, node c is inserted into the priority queue Q_inc to see if there are any shortest paths that might use the shortest paths from node c to node z as their subpaths to reach to node z.

5.4.3 ClearBetweenness

The next algorithm we discuss is the *ClearBetweenness* algorithm which is invoked from the *DeleteUpdateBetweenness* algorithm when a shortest path is invalidated. The betweenness values of the intermediates on the invalidated path(s) should be reduced before their relationships with the previously known shortest paths become intractable. In other words, the main functionality of the *ClearBetweenness* algorithm is to reduce the betweenness values of the intermediates that lie on the old set of shortest paths from node a to node z, where nodes a and z are given as the input parameters to the algorithm. The *ClearBetweenness* algorithm is invoked only once for each node pair <a, z> when it is first discovered that the shortest path(s) from node a to node z need to be updated during the propagation of the current shrinking network update.

The first part of the *ClearBetweenness* algorithm (Lines 1–7) processes the old set of intermediate nodes that are currently accessible when the shortest path(s) from node *a* to node *z* are traced. The betweenness centrality value of each intermediate node *v* is reduced by the contribution of the shortest path(s) from node *a* to node *z* (Line 6) and inserted into *trackLost*. Before their betweenness centrality values are modified, we do check if the intermediate node *v* belongs to the original set of intermediates that existed before the network update or if it is one of the new intermediates that became accessible due to currently partially updated shortest paths from node *a* to node *z*. If node *v* is a new intermediate node, it is skipped without further processing (Line 4). An intermediate node *v* satisfies the following equality if it is an old intermediate:

$$D_{old}(a,v) + D_{old}(v,z) \overset{?}{=} D_{old}(a,z).$$

Other than the nodes that are still accessible by following the currently known predecessors, there might be some other nodes that were once on the shortest paths from node *a* to node *z*, but cannot be accessed now because some of the pointers are broken when the current shrinking network update started propagating in the network. Such nodes are processed in the second half of the *ClearBetweenness* algorithm (Lines 8–15). These nodes are inserted into *trackLost* before they are cleared later in the *DeleteUpdateBetweenness* algorithm when the predecessors of the invalidated shortest paths are cleared. Lines 9–15 of the *ClearBetweenness* algorithm go through the entries of *trackLost* to find the previously existing intermediates that cannot be found otherwise.

Such intermediates are usually lost as a part of a subpath that is lost before the current network update propagated this far.

Algorithm–17: *CLEARBETWEENNESS* (a, z)

Input: A source node a and a destination node z. The old set of shortest paths from a to z need to be invalidated.

Output: Betweenness values of the intermediates on the previously known shortest paths are reduced. The previous intermediates whose connections need to be broken are maintained in *trackLost*.

```
1.  if $\sigma(a, z) \neq 0$
2.      for $v \in I(a, z)$
3.          if $(D_{old}(a, v) + D_{old}(v, z) \neq D_{old}(a, z))$
4.              continue
5.          else if $(a \neq v$ & $v \neq z)$
6.              $C_B(v) \leftarrow C_B(v) - (\sigma_{old}(a, v) * \sigma_{old}(v, z) / \sigma_{old}(a, z))$
7.              add $<<a, z>, v>$ into $trackLost$
8.  $AlreadyDone \leftarrow I(a, z)$
9.  for $(<x, y> \in trackLost$.KeySet())
10.     if $(D_{old}(a, x) + D_{old}(x, y) + D_{old}(y, z) = D_{old}(a, z))$
11.         for $interm \in trackLost<x, y>$
12.             if $(interm \notin AlreadyDone$ & $D_{old}(a, interm) + D_{old}(interm, z) = D_{old}(a, z))$
13.                 $C_B(interm) \leftarrow C_B(interm) - (\sigma_{old}(a, interm) * \sigma_{old}(interm, z) / \sigma_{old}(x, z))$
14.                 add $<<a, z>, interm>$ into $trackLost$
15.                 add $interm$ into $AlreadyDone$
```

Assume that there is a path in the following form: a....→…x…. →.... *interm* …→….y…→…..z. By the time the propagation of the shrinking network update reaches the level of the shortest paths from node a to node z, the subpath(s) from node x to y might have already been updated and the node *interm* may not necessarily be an intermediate on the shortest path(s) from node x to node y anymore. Hence, we cannot find the node *interm* as an intermediate for the shortest path(s) from node a to node z although the node pair $<a, z>$ still has a contribution on the betweenness centrality value of the node *interm*. This is why we keep track of such cases in a separate data structure (e.g. *trackLost*) and the betweenness values of such nodes are reduced later as required

(Line 13 of *CLEARBETWEENNESS*). In this example, we had the information that node *interm* was an intermediate on the shortest path(s) from node x to node y. However, we did not have direct information about node *interm* being an intermediate from node a to node z. In Line 14 of the *CLEARBETWEENNESS* algorithm, we add the information that *interm* is a lost intermediate of the shortest path(s) from node a to node z by inserting a tuple of $<<a, z>$, *interm*$>$ into *trackLost*. To avoid processing the same node multiple times, we keep track of the intermediates that are already processed in the *AlreadyDone* set.

5.4.4 ADJUSTNPS

The next algorithm that is discussed is the *ADJUSTNPS* algorithm. The *ADJUSTNPS* algorithm is called from the *DELETEBETWEENNESS* algorithm after all the shortest path distances and the predecessors on the shortest paths are updated accurately. The main functionality of the *ADJUSTNPS* algorithm is to accurately update the number of shortest for all modified shortest paths, either by length or number.

The *ADJUSTNPS* algorithm loops through the list of node pairs $< a, z >$ whose shortest paths changed in terms of length or number. Such node pairs are stored in σ_{old} during the course of a shrinking network update propagation (Line 1).

There are two corner cases for the number of shortest paths. If the nodes a and z are the same, then $\sigma(a, z) = 1$ by definition. If the distance from node a to node z is undefined (e.g. ∞), then there is no shortest path from a to z, and $\sigma(a, z) = 0$. Lines 2 – 5 of the *ADJUSTNPS* algorithm handle these two conditions.

Algorithm–18: *ADJUSTNPS* ( )
Input: The list of source and destination nodes for the set of affected shortest paths.
Output: The numbers of shortest paths for all affected paths are updated.

```
1.  for $(a, z) \in \sigma_{old}$
2.      if $(a = z)$
3.          $\sigma(a, z) \leftarrow 1$
4.      else if $(D(a, z) = \infty)$
5.          $\sigma(a, z) \leftarrow 0$
6.      else
7.          $Known \leftarrow \emptyset$;        $Stack \leftarrow \emptyset$
8.          for $n \in P_a(z)$
9.              push $n$ to $Stack$
10.             add $n$ to $Known$
11.         $counter \leftarrow 0$
12.         while $Stack \neq \emptyset$
13.             $n \leftarrow$ pop ($Stack$)
14.             add $n$ to $Known$
15.             if $(n = a)$
16.                 $counter \leftarrow counter + 1$
17.             for $p \in P_a(n)$
18.                 if $(p \neq a \ \& \ p \neq z)$
19.                     push $p$ to $Stack$
20.                 if $(p = a)$
21.                     $counter \leftarrow counter + 1$
22.         $\sigma(a, z) \leftarrow counter$
```

The rest of the algorithm is the core of the *ADJUSTNPS* algorithm and handles the common case. The key idea is the following. If, in a network, it is possible to reach from node a to node z via multiple paths, then each path can be reconstructed separately following the predecessors on the shortest paths. Hence, starting with the destination node z, we push each of the predecessors on a stack at every level until the source node a is reached. The number of times we hit the source node a is equal to the number of distinct routes one can take, which is represented as the number of shortest paths from node a to node z. Every time an intermediate node is processed, we check if we hit node a. If it is true, then the counter is incremented (Lines 15 – 16; Lines 20 – 21).

5.4.5 ADJUSTBETWEENNESS

The final step required for completing a shrinking network update is the *ADJUSTBETWEENNESS* algorithm. It loops through the list of node pairs $< a, z >$ where the shortest paths from node a to node z changed in terms of length or number. For every node pair, the shortest path distances, the intermediates on the shortest paths, and the number of shortest paths are already computed before the *ADJUSTBETWEENNESS* algorithm is called. The *ADJUSTBETWEENNESS* algorithm increments the betweenness value of each intermediate node n by the fraction of the shortest paths from node a to node z that it lies on. With this step, the incremental update of betweenness centralities for the shrinking network updates is complete.

Algorithm–19: *ADJUSTBETWEENNESS* ( )
Input: The list of node pairs between which the shortest paths are updated.
Output: Betweenness values of the intermediates on the affected shortest paths are increased by the fraction of shortest paths they lie on.

```
1. for $(a, z) \in \sigma_{old}$
2.     for $n \in I(a, z)$
3.         if $n \neq a$ & $n \neq z$
4.             $C_B(n) \leftarrow C_B(n) + (\sigma(a, n) * \sigma(n, z) / \sigma(a, z))$
```

5.5 Discussion on Algorithmic Complexity

5.5.1 Run Time Analysis

Next, we discuss the time complexities of the algorithms proposed earlier in Chapter 5.3 and Chapter 5.4, respectively.

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

The overall run-time complexity of the *InsertUpdateBetweenness* algorithm is dominated by the complexity of the priority queue Workset. The Workset is used to track all the affected nodes as the shrinking network update keeps propagating. The

INSERTUPDATEBETWEENNESS algorithm essentially performs a traversal in the neighborhood of every AffectedSink and AffectedSource, respectively. The work performed inside the while loop is O(||Affected|| log ||Affected||) + I^2) where ||Affected|| is used to denote the sum of the number of the edges and the nodes in the subgraph formed by AffectedSource and AffectedSink nodes' neighborhoods. Finally, the *INSERTBETWEENNESS* algorithm invokes the *INSERTUPDATEBETWEENNESS* algorithm for each AffectedSink and AffectedSource node once, followed by a call for the *INCREASEBETWEENNESS* algorithm, yielding O((|AffectedSink|+|AffectedSource|) ||Affected|| log ||Affected||) + I^2 + $|\sigma_{old}|\ I$) time complexity overall.

For handling the shrinking network updates, the *DELETEBETWEENNESS* algorithm is the entry point for execution. It has multiple invocations of *DELETEUPDATEBETWEENNESS* and at the end; it also calls the *ADJUSTNPS* and *ADJUSTBETWEENNESS* algorithms. The *DELETEUPDATEBETWEENNESS* algorithm calls the *CLEARBETWEENNESS* algorithm, which as a result, has a contribution to the overall time complexity of the *DELETEUPDATEBETWEENNESS* algorithm. Hence, the *ADJUSTNPS*, the *ADJUSTBETWEENNESS,* and the *CLEARBETWEENNESS* algorithms' time complexities are independent of the other algorithms' time complexities.

For the *ADJUSTBETWEENNESS* algorithm, the overall complexity of the procedure is O($|\sigma_{old}|\ I$) where I represents the total number of intermediates processed for all node pairs listed in σ_{old}.

For the *ADJUSTNPS* algorithm, the time complexity analysis resembles that of the *INCREASEBETWEENNESS* algorithm presented in Chapter 5.3. The *ADJUSTNPS* algorithm runs a for-loop for σ_{old} many iterations. Inside this outer for loop, there is one for loop and one while loop. These two loops should be considered in combination because the intermediate nodes on the shortest paths from the source to the destination are handled by one or the other loop and the distinction is irrelevant. The complexity of the bodies of these loops are O(1), and they are executed once for each intermediate node. Therefore, the overall complexity of the procedure is $O(|\sigma_{old}|\ I)$ where I represents the total number of intermediates processed for all node pairs $<a, z>$ listed in σ_{old}.

Similarly, the time complexity analysis of the *CLEARBETWEENNESS* algorithm is in line with the time complexity analysis of the *REDUCEBETWEENNESS* algorithm presented in Chapter 5.3. The run time of the *CLEARBETWEENNESS* algorithm is dominated by the if-block at the end (Lines 9 – 15 of *CLEARBETWEENNESS*). This block performs a search over the map of all known intermediate nodes on the shortest paths from node a to node z (nodes a and z are the two parameters given as input to the algorithm) and uses two intermediates at a time to form the key to the map. Hence, and its complexity is $O(I(a, z)^2)$ where $I(a, z)$ represents the number of intermediates on the on the shortest paths from node a to node z.

The time complexity analysis for the *DELETEUPDATEBETWEENNESS* algorithm can take the time complexity analysis of the *DELETEUPDATECLOSENESS* algorithm, presented in Chapter 4.4.2. However, since the *DELETEUPDATECLOSENESS* algorithm makes calls to the *CLEARBETWEENNESS* algorithm, its overall time complexity is different than that of the *DELETEUPDATECLOSENESS* algorithm.

In the *DELETEUPDATEBETWEENNESS* algorithm, the time complexity of the second phase of the algorithm dominates the time complexity of the first phase of the algorithm. The time complexity of the second phase of the algorithm is governed by the time complexity of the AffVert and Q_inc priority queues maintained by the algorithm. The Q_inc priority queue is a subset of the AffVert as only certain elements are added into it conditionally in Line 24 of the *DELETEUPDATEBETWEENNESS* algorithm. The operations performed on the while block starting on Line – 10 and the while starting on Line – 25 are similar in terms of time complexity. However, in the worst case, all the elements in the AffVert are added into Q_inc. In the first while block the successors of each element, and in the second while block, the predecessors of each element is checked, which results in probing of the two hop neighborhood of the affected vertices with a time complexity of $O(||\text{AffectedVertices}||_{2,z})$ where the subscript 2 denotes the size of two-hop neighborhood of all affected nodes and *z* refers to the last parameter of the algorithm (i.e. the node to which the shortest path distances are updated). The insertions into and extractions from a priority queue are on the order of log*n* for a priority queue of size *n*. Hence, these operations will take $O(|\text{AffectedVertices}| \log|\text{AffectedVertices}|)$ time in total. The *CLEARBETWEENNESS* algorithm, in the worst case, might be called for all the elements in the AffectedVertices, covering all possible AffectedSource – AffectedSink node pairs, which would result in $O(I^2)$ time complexity where I represents the total number of intermediates processed for all node pairs listed in σ_{old}. Hence, the overall time complexity of the *DELETEUPDATEBETWEENNESS* algorithm is $O(||\text{Affected}||_{2,z} + |\text{Affected}| (\log |\text{Affected}| + I^2) + I^2 + |\sigma_{old}|\ I)$ where the set of affected nodes (Affected) is given by the combination of AffectedSink and AffectedSource nodes. Although the complexity of each sub-algorithm can be precisely computed in terms of the changes made, in terms of the upper bound worst-case time complexity, the incremental betweenness algorithm does not do better than the Brandes' betweenness algorithm. This is because, in the worst case, the size of the Affected set can be as large as the entire node set of the network, and σ_{old} can be on the order of O(n^2).

5.5.2 Memory Consumption and Overhead Analysis

Next, we discuss the theoretical scaling argument of how the memory usage scales with the problem size for the incremental betweenness centrality algorithm. The Brandes' betweenness algorithm takes O(n + m) space. The Brandes' betweenness algorithm finds the shortest paths from a certain source node and can reuse that space while discovering the shortest path tree from another node. However, the incremental betweenness centrality algorithm would not override such data as it needs the information on all-pairs shortest paths and the predecessors readily available for issuing the incremental network updates quickly. Hence, the incremental betweenness algorithm has larger space requirements as discussed in detail below.

Similar to the incremental closeness algorithm, the incremental betweenness centrality algorithm maintains the original graph and its transpose (reverse) simultaneously. This duality is of critical importance for achieving a bounded incremental update algorithm. Hence, the memory requirement increases to O($2m$+$2n$) for graph representation (i.e. the representation of nodes and edges). The duality is an algorithmic property inherited from the incremental all-pairs shortest paths algorithm designed by Ramalingam and Reps.

The incremental betweenness centrality algorithm also needs an additional memory space of O(n) for storing the betweenness centrality value of each node. However, the betweenness attributes or other attributes do not need to be maintained for the transpose network; they only need to be maintained for the original version of the graph. Hence, the betweenness centrality values need to be stored only once in the memory. The betweenness centrality is very costly to compute and needs several pieces of additional information by definition. Betweenness centrality requires the number of shortest paths and the predecessors on the shortest paths, which reflect as additional memory space required by the betweenness algorithms.

Similar to the space the incremental closeness centrality algorithm needs to hold the shortest distance values across all pairs of nodes, the space required for storing the shortest path distances for the betweenness centrality is O(n^2) in the worst case (i.e. O(n^2) distance matrix). To be more precise, for each node, there are $(n-1)$ other nodes in the network it can reach out to. Hence, the precise upper bound for the number of node pairs for whose distances are defined is $n(n-1) = n^2$, which is represented as O(n^2) in the worst case in the Big-O notation. We do not need to hold the information for $D(i,i), i \in G$ as $D(i,i)$ is always equal to 0 by default. However, a lot of real life networks are very sparse and the distance matrix does not even need to be stored as an O(n^2) matrix; the sparse representations are more preferable. In sparse networks, not every node is reachable from every other node in the network. More precisely, the data structure that holds the shortest distance information has $Conn(G)$ entries where $Conn(G)$ represents the number of node pairs in the network that have a finite shortest distance defined. This analysis is valid for the number of shortest paths in the network. For the number of shortest paths, we do not need to hold the information for $\sigma(i, i), i \in G$ as $\sigma(i, i)$ is always equal to 1 by default. In total, (2 * $Conn(G)$) entries are required to hold the shortest distances and the number of shortest paths across all node pairs. The only piece of information that needs to be maintained for the transpose of the graph is the shortest distances defined on that version of the graph. This adds another $Conn(G)$ entries to the memory consumption.

Another piece of information the computation of betweenness centrality needs is the predecessor nodes on the shortest paths (i.e. the information on the intermediate nodes on a path) so that it can find the fraction of the shortest paths a node lies on. This information only needs to be maintained for the original version of the graph. The shortest path tree stemming from a node i can at most have n nodes and m edges in it. This is the case where a node can access all other nodes in a network, and all the edges in the network are used as parts of the shortest paths stemming from node i. The shortest path tree stemming from a single node takes $O(n + m)$ space. The incremental betweenness centrality algorithm needs to maintain the predecessors on the shortest paths from all nodes, which takes $O(n^2 + nm)$ space in the worst case. However, in most cases, the information required for holding the shortest path trees requires less space. On average, the number of predecessors on the shortest paths can be calculated as the multiplication of the average shortest path length (in terms of number of hops), the average number of shortest paths, and $Conn(G)$. Considering the information presented on the network topologies in Chapter 7, for instance, the average shortest path lengths for networks with 1000-5000 nodes remain in the range of 3.45 – 20.26, mostly staying less than 13.0, which are values that are much smaller than the total number of nodes in the network.

Overall, the worst-case space complexity of the incremental betweenness centrality algorithm is $(2n + 2m + n + 3Conn(G) + n^2 + nm)$ which is represented as $O(n + m + Conn(G) + n^2 + nm)$ in the Big-O notation. This space complexity can also be represented as $O(n^2 + nm)$ given that (*i*) the worst case value of $Conn(G)$ is $O(n^2)$ and (*ii*) both n and m are of lower degree from n^2 and nm.

In addition to the amount of data that needs to be stored permanently across several different iterations of the incremental betweenness centrality algorithm, another aspect of the memory consumption is the overhead: the amount of data that is temporarily stored during the execution of a single incremental network update, and not maintained across different updates.

There are four sub-algorithms that handle the growing network updates: the *INSERTBETWEENNESS,* the *INSERTUPDATEBETWEEENNESS,* the *REDUCEBETWEENNESS,* and the *INCREASEBETWEENNESS* algorithms.

The *INSERTBETWEENNESS* algorithm maintains four data structures that are initialized at the beginning of the *INSERTBETWEENNESS* algorithm, created once, and accessed by all other sub-algorithms of the incremental betweenness centrality that handle the growing network updates: σ_{old}, D_{old}, $trackLost$, and $PairsDone$. The σ_{old}, D_{old}, and *PairsDone* are O(*Conn*(*G*)) in the worst case, however, in reality they are much smaller. For their sizes, |AffectedSinks| × |AffectedSources| is a more realistic upper bound. In addition, the D_{old} is likely to be smaller than σ_{old} because σ_{old} contains a key for all pairs of nodes whose number of shortest paths or distances change while the D_{old} contains only the pairs of nodes whose shortest distances change. The *trackLost* data structure holds the information for the predecessors that were on the invalidated shortest paths. If there are too many changes in the network, this data structure can potentially become large. Its size can be described as O(*I*) where *I* represents the total number of intermediates processed for all node pairs listed in σ_{old}.

In addition, the *INSERTBETWEENNESS* algorithm maintains two sets of nodes: AffectedSinks and AffectedSources, which are of order O(*n*) each. These two objects are stored during the entire lifetime of an incremental update and cleared once the network update completes successfully.

In the *INSERTUPDATEBETWEENNESS* algorithm, lists of AffectedVertices and VisitedVertices are maintained as well as a working set of edges (e.g. Workset). Both the AffectedVertices and VisitedVertices might be of order O(*n*) while the Workset might be of order O(*m*) in the worst case. Hence, the memory overhead per iteration in the worst case is (2*n* + *m*), which can be represented as O(*n* + *m*). These data structures are only maintained during the execution of the *INSERTUPDATEBETWEENNESS* algorithm and cleared once the execution of the algorithm is complete.

The *REDUCEBETWEENNESS* algorithm utilizes information from $\sigma_{old,}$ $D_{old,}$ *trackLost*, and *PairsDone,* and grows *trackLost* as needed, which are already discussed as a part of the *InsertBetweenness* algorithm. There are two data structures *Known* and *Stack*, and they are used together: an item popped out of *Stack* is inserted into *Known*. The total size of *Known* and *Stack* is on the order of O(n). However, the maximum size of these data structures is the number of intermediates on the shortest paths from node x to node z, which are the two input parameters given to the algorithm. On average, the number of predecessors on the shortest paths from node x to node z can be calculated as the multiplication of the average shortest path length (in terms of number of hops) and the average number of shortest paths.

Finally, the *IncreaseBetweenness* algorithm utilizes information from σ_{old} and its execution is very similar to that of the *ReduceBetweenness* algorithm. Hence, their analysis will be similar as well. For each node pair <a, b> in σ_{old}, there are two data structures *Known* and *Stack* that are used together: an item popped out of *Stack* is inserted into *Known*. The total size of *Known* and *Stack* is on the order of O(n). However, the maximum size of these data structures is the number of intermediates on the shortest paths from node a to node b, which are the two input parameters given to the algorithm. On average, the number of predecessors on the shortest paths from node a to node b can be calculated as the multiplication of the average shortest path length (in terms of number of hops) and the average number of shortest paths.

Overall, the data structures initialized at the beginning of the *InsertBetweenness* algorithm are maintained during the entire lifetime of an incremental network update, which can be stated as the memory overhead of the incremental betweenness centrality algorithm. The total upper bound for the space these data structures consume is

((3*|AffectedSinks| × |AffectedSources|) + *I*), which can be represented as O(|AffectedSinks| × |AffectedSources|) + *I*) in the Big-O notation.

Similar memory overhead analysis can be performed for temporarily stored data for the algorithms that handle the shrinking network updates. The analysis done for the components of the incremental betweenness centrality algorithm that handle the growing network updates mostly holds for the sub-algorithm that handle the shrinking network updates. There are five sub-algorithms that handle the shrinking network updates: the *DELETEBETWEENNESS*, the *DELETEUPDATEBETWEEENNESS*, the *CLEARBETWEENNESS*, the *ADJUSTNPS*, and the *ADJUSTBETWEENNESS* algorithms. The analysis done for the *DELETEBETWEENNESS* algorithm is valid for the *DELETEBETWEENNESS* algorithm as well. Similarly, the analysis done for the *INCREASEBETWEENNESS* algorithm holds for the *ADJUSTBETWEENNESS* algorithm. The *CLEARBETWEENNESS* and the *ADJUSTNPS*, have working principles and analyses that follow the same approach as the *REDUCEBETWEENNESS* algorithm. Hence, they are not discussed separately here.

In the *DELETEUPDATEBETWEENNESS* algorithm, lists of AffectedVertices and VisitedVertices are maintained and a set working nodes (e.g. Workset). The AffectedVertices, VisitedVertices, and Workset are of order O(n) while in the worst case. Hence, the memory overhead per iteration in the worst case is 3n, which can be represented as O(n). These data structures are only maintained during the execution of the *DELETEUPDATEBETWEENNESS* algorithm and cleared once the execution of the algorithm is complete.

As the analyses done on the performance and memory use suggest, different network features have different effects on both the performance and memory consumption. For instance, betweenness centrality needs the information on the predecessors that lie on the shortest paths, which are actually the nodes on a shortest path tree. If a network has low diameter and average shortest path length, this causes the average and maximum depth of shortest path trees stemming from the each node in the network to be short and the tree might potentially contain fewer nodes. Or, if a network has several small, disconnected components, if a network its diameter and average shortest path length are likely to be smaller than they would be otherwise. In addition, *Conn*(*G*) is also likely to be smaller, which is one of the main values that make up the memory consumption of the incremental centrality algorithms. Another effect this would have is that the number of nodes that are affected by the incremental network update is likely to be smaller, which reflects as increased performance as the percentage of the affected nodes decreases.

Another feature that might affect the level of connectivity, *Conn*(*G*), which represents the number of node pairs that are connected, is whether a network is directed or undirected (bidirectional). When a network is undirected, if a node x can reach to another node y, then node y can reach to node x as well. However, this is not necessarily true when the edges in a network are directed. Hence, when a network is undirected (bidirectional), *Conn*(*G*) is likely to be higher, which reflects as a potential increase in the memory consumption and the percentage of nodes affected by an incremental network update as well as a potential decrease in the performance speedup that can be obtained.

5.5.3 Comments on Accuracy

The algorithms presented in this chapter build on the Ramalingam and Reps dynamic shortest path algorithms proposed in [104] to incorporate the *accurate* computation of betweenness centrality. In other words, the incremental betweenness centrality algorithms presented in this chapter are designed to incur no loss in the accuracy of betweenness centrality values.

A discussion that is similar to the discussion on the implementation of the incremental closeness centrality also holds for the incremental betweenness centrality implementation. It is important to handle the floating-point numbers with caution. In order to ensure that the results produced by the incremental betweenness centrality algorithm match the results produced by the non-incremental betweenness algorithms, both the incremental and the non-incremental betweenness centrality algorithms should handle the floating-point numbers using the same epsilon value, ε. Two real numbers a and b are considered equal if $|a - b| < \varepsilon$, where $|a - b|$ denotes the absolute value of $(a - b)$. The conditions that test whether a path is a shortest path compare the length of the path against the shortest distance to look for equality. Such conditional checks lie at the heart of both the incremental and non-incremental algorithms and govern how the rest of the network update propagates in the network and how the rest of the shortest path trees are built. Hence, the conditional checks that test the equality/inequality of the shortest path distances should be supported by an epsilon range. The epsilon value used in our implementations was 10^{-7}.

For verification, every time the incremental algorithm was run, the standard non-incremental algorithm (e.g. the Brandes' algorithm) was also run and the results compared. Once we took the same epsilon into account in both, the results of both the incremental algorithm and the standard algorithm were identical in every case. Further comments on verification can be found in Chapter 8.7.

5.6 Discussion on Generalization to Other Centrality Metrics and Shortest-Path based Problems

As described earlier, most social centrality metrics can be loosely classified into two types: (*i*) social centrality metrics based on shortest-paths, and (*ii*) social centrality metrics based on degree of nodes. The incremental centrality algorithms proposed in this dissertation dynamically maintain the all-pairs shortest path information and auxiliary data related to the centrality metrics. The proposed incremental betweenness algorithm performs accurate maintenance of dynamic all-pair shortest paths along with the shortest path counts and predecessor tracking. With modest extensions, the proposed algorithms can be generalized to calculate other shortest-path based social centrality metrics such as stress centrality [20], and flow-betweenness centrality [130].

There are other problem domains that benefit from information on the number of shortest paths or predecessor tracking. Redundant path planning for fault tolerant networks (whether they be communication or transportation networks), and analyzing optimal game state transitions in artificial intelligence or game theory are among such examples. In transportation/communication networks, the number of shortest paths represents the number of redundant but optimal routes between a pair of nodes, and predecessors represent routing points. In game state analysis, the number of shortest paths to a winning state represents the number of optimal winning strategies not foreclosed by an opponent's moves, and predecessors indicate intermediate states as the game progresses.

CHAPTER 6 EXTENSIONS AND APPLICATIONS

The first part of this chapter describes extensions of incremental centrality metrics to provide support for their approximations. The approximations discussed in this chapter are in the form of *k*-centrality metrics where the shortest paths in a network are calculated within *k*-hops of each node. In the second part of this chapter, a number of potential applications (i.e. use cases) of the incremental centrality metrics are described including clustering, vulnerability analysis, and resource allocation.

6.1 WHAT IS *K*-CENTRALITY?

The standard shortest path based centrality metrics consider all the shortest paths in a network topology irrespective of the lengths of the shortest paths. When computing local approximations for the shortest path based centrality metrics, one alternative is to consider the shortest paths within a bounded distance. Consider the following scenario where a social agent in a social network needs help to solve a problem. The social agent is more likely to contact one of its friends, a friend of a friend, maybe someone those people can introduce; but most likely it will not search for help beyond the first couple of hops of its connections. In most cases, social agents are not even aware of the set of contacts they can reach out to if they were to follow communication paths along multiple hops. Such social behavior motivates the use of local approximations of the shortest paths bounded within first *k*-hops instead of using global metrics that are computed across the entire network topology. This is also in line with the other findings available in the current literature, which discuss that very long connections are not considered to be realistic for friendship-like networks.

We name the approximations of the betweenness and closeness centrality that are computed using the shortest paths within *k*-hops as *k*-betweenness and *k*-closeness, respectively.

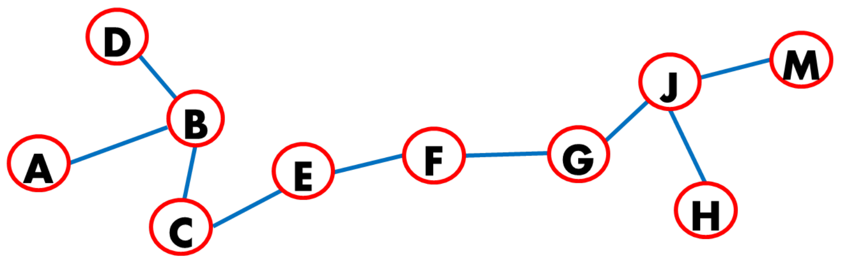

Figure 8 - Example network for *k*-centrality discussion.

Consider the example network depicted in Figure 8. Figure 8 represents a binary, undirected (bidirectional) network where all existing edges' costs and weights are equal to 1. Assume that we need to compute *k*-closeness and *k*-betweenness for $k = 3$. In this case, the shortest paths are computed within the first three hops of each node. The shortest paths whose total costs are more than three hops are dropped from the computations. Hence, when $k = 3$, not all the shortest paths across the entire network are included in the computations of centrality metrics.

Next, the computations of *k*-closeness and *k*-betweenness for $k = 3$ are described using node-*B* in Figure 8 as an example.

To compute the *k*-closeness centrality of node-*B* for $k = 3$, the nodes that are reachable from node-*B* in less than or equal to three hops are considered. In this case, the following nodes contribute to the closeness centrality of node-*B*: A, D, C, E, and F.

To compute the *k*-betweenness centrality of node-*B*, the node pairs that contribute to the *k*-betweenness centrality of node-*B* are as follows:

$<A, D> \quad <A, C> \quad <C, D> \quad <A, E> \quad <D, E>$

The node pairs listed above are the only node pairs that can reach to one another in less than or equal to three hops and have node-*B* as an intermediate node on their shortest paths. For instance, the shortest path between node-*A* and node-*F* is not included even though node-*F* is within three hops of node-*B*. This is because it takes four hops in total to reach from node-*A* to node-*F* and vice versa. Such paths would not be included in the final computations of *k*-betweenness centrality for $k = 3$ where the computations are restricted to include the shortest paths that are three hops or less. The shortest path between node-*B* and node-*F* is also excluded from the computation of betweenness centrality for node-*B* as node-*B* is an end node, not an intermediate node.

Similar to the examples discussed above, the *k*-centrality metrics traditionally focus on the number of hops, where the networks are treated as binary networks, and the edge costs are ignored. Using the edge weights becomes especially tricky (or even misleading) when the edge costs can be fractional, using values that are less than 1. In such networks, setting the *k* value to a commonly used, small value such as $k = 2$ or $k = 3$ might become confusing and problematic; potentially leading to unintended results. This is because, the cost of a shortest path composed of several hundreds of edges with edge cost, say, 0.0001 would still satisfy the requirement that the shortest path distance should be less than or equal to k. In such a case, the attempt to obtain an approximate value for the centrality metrics would result in almost no approximation. Therefore, although we provide generic pseudocodes that would work with both weighted and binary networks, we recommend using k-centralities with the binary versions of networks.

As one last point, the longest possible shortest path in a network is observed in the form of a chain, which can have at most $n - 1$ hops. When $k = n - 1$, the k-centrality version of a shortest path based centrality metric corresponds to the standard version of that shortest path based centrality.

In the current literature, there exist papers that specifically focus on k-centralities (e.g. [18]). In [18], the authors combine the k-centrality computation for closeness and betweenness centrality metrics in a single algorithmic framework and comment on the details like normalization, implementation, and algorithmic complexity. As one example, closeness centrality C_c is normalized by the maximum possible closeness value $C_C^{max} = {}^{1}/_{(n-1)}$, which belongs to the central node in a star-topology network of n nodes. In [18], a normalization criterion is proposed for k-closeness centrality. In the case of k-closeness, the number of nodes that can reach within k hops, n_k, tends to be smaller than the case where the number of hops on the shortest paths is not bounded. For k-closeness, $C_C^{max,k}$ is adjusted as $C_C^{max,k} = {}^{1}/_{n_k}$ and the raw k-closeness value is normalized by ${}^{n_k}/_{(n-1)}$. Similar criteria are discussed for k-betweenness as well.

In the context of *k*-centralities, there is more work on *k*-betweenness centrality than there is on *k*-closeness centrality. Other studies that focus on *k*-betweenness centrality and not mentioned so far in this dissertation in detail include [131], [132], [133], and [134].

In [131], the authors deviate from the traditional interpretation of *k*-betweenness centrality and define *k*-betweenness as a centrality metric that captures the additional information provided by paths whose length is within *k* units of the shortest path length. In other words, the authors extend and generalize the definition of betweenness centrality to include additional, non-shortest paths according to an input parameter *k* and present a new parallel algorithm to calculate this variant of betweenness centrality. In [132], the authors from the same group as [131], build on the work presented in [131] and discuss its application on social network data (e.g. Twitter).

In [133], the authors focus on two methods for approximating betweenness centrality. One approximation method that is discussed in [133] specifically targets scale-free networks and their topological properties. In scale-free networks, incoming new nodes prefer attaching to the nodes with a high number of connections, which play the role of "hubs". The authors observed that centrality rankings tend not to change dramatically unless the node a new edge emanates from is one of the central nodes. This brings along a way of enabling lazy updates in dynamic networks while providing fairly accurate centrality rankings. The second approximation method discussed in [133] is *k*-betweenness centrality. In [134], the authors integrate these approximation methods into an Internet Deployed P2P Reputation System and conclude that it is easier to approximate betweenness centralities in scale-free networks than it is in random networks due to their topological properties. One of the real life datasets used in the evaluations in this dissertation is also a real life P2P dataset.

6.2 *k*-Closeness

This section discusses the algorithms designed for the incremental computation of *k*-closeness centrality. The algorithms that are discussed in this section, Chapter 6.2.1 and Chapter 6.2.2, are variants of the algorithms discussed in Chapter 4.3 and Chapter 4.4, respectively. For completeness reasons, the pseudocodes that are modified from the incremental closeness centrality algorithm for the incremental *k*-closeness centrality algorithm are provided. The changes that are specifically made for the incremental computation of *k*-closeness centrality are pointed out as well.

As mentioned in earlier chapters, the most commonly used way of computing closeness centrality is to compute the all pairs shortest paths by running a single source shortest path algorithm such as the Dijkstra's algorithm using every node in the network as the source node. Hence, before we start our discussion for converting the incremental closeness centrality algorithm into incremental *k*-closeness centrality algorithm, we briefly discuss how to convert the Dijkstra's algorithm to work with *k*-hop shortest path bounds.

6.2.1 Dijkstra's *k*-hop Shortest Path

Algorithm–20 provides the algorithm for *k*-hop single source shortest path algorithm (e.g. the Dijkstra's algorithm). To ensure that only the shortest paths that are less than or equal to *k* are processed, Line – 12 of Algorithm–20 is modified to include a condition which checks if the newly found shortest path is less than or equal to *k*. The modification done is highlighted with bold font in Line – 12.

```
Algorithm–20: SINGLE-SOURCE K-SHORTEST PATH (DIJKSTRA'S) ALGORITHM (G, src)
Input: A network $G(V, E)$ and source node $src$.
Output: Shortest path distances from $src$ to all nodes $n$ $V(G)$ are computed.
1. for $v \in V(G)$
2.     $d[v] \leftarrow \infty$
3.     $P[v] \leftarrow$ empty list
4. $d[s] \leftarrow 0$
5. $Q \leftarrow w \in V(G)$
6. while $Q \neq \emptyset$
7.     dequeue $u \leftarrow Q$
8.     if $d[u] = \infty$
9.         break;
10.    for neighbor $v$ of $u$
11.        $alt \leftarrow d[u] + C[u, v]$
12.        if $alt < d[v]$ && $alt \leq k$
13.            $d[v] \leftarrow alt$
14.            $P[v] \leftarrow u$
15.            decrease-key $v$ in $Q$
```

6.2.2 Incremental *k*-Closeness Centrality: Growing Network Updates

The changes required for converting incremental closeness centrality into incremental *k*-closeness centrality are primarily about bounding the discovery of the shortest paths to remain within *k* hops. Although the incremental closeness centrality algorithm is composed of two sub algorithms, *INSERTCLOSENESS* and

InsertUpdateCloseness, the changes required for *k*-closeness centrality are made on the *InsertUpdateCloseness* algorithm only.

Algorithm–21: *InsertUpdateKCloseness* (*G*, *src*, *dest*, *z*)

Input: A network $G(V, E)$, a newly inserted or modified edge $\{src \rightarrow dest\}$, and an affected node z.

Output: The shortest distances (D) to the affected node z are updated, and closeness values (C_c) of the sources of the updated distances are also updated.

```
1.  Workset ←{$src \rightarrow dest$}
2.  VisitedVertices    ←$src$
3.  AffectedVertices  ← ∅
4.  while Workset ≠ ∅
5.      {$x \rightarrow y$} ← pop (Workset)
6.      if $C(x, y) + D(y, z) < D(x, z)$ && $\boldsymbol{C(x, y) + D(y, z) \leq k}$
7.          Add $x$ to AffectedVertices
8.          TotDist($x$) ← $\frac{1}{C_c(x)}$
9.          if $D(x, z) \neq \infty$
10.             TotDist($x$) ← TotDist($x$) – $D(x, z) + C(x, y) + D(y, z)$
11.         else
12.             TotDist($x$) ← TotDist($x$) + $C(x, y) + D(y, z)$
13.         $C_c(x) \leftarrow \frac{1}{\text{TotDist}(x)}$
14.         $D(x, z) \leftarrow C(x, y) + D(y, z)$
15.         for $u \in$ Pred($x$)
16.             if $SP(u, x, src)$ and $u \notin$ VisitedVertices
17.                 push {$u \rightarrow x$} into Workset
18.                 Insert $u$ into VisitedVertices
19.   return AffectedVertices
```

Algorithm–21 provides the related pseudocode for the *InsertUpdateKCloseness* algorithm. The change required for converting incremental closeness centrality into incremental *k*-closeness centrality to handle growing network updates is only at Line – 6 of Algorithm–21. In Line – 6 of the *InsertUpdateCloseness* algorithm, we check for changes in the shortest path distances to see if there are any newly formed shorter paths. In Line – 6 of the *InsertUpdateKCloseness* algorithm, this check is performed along with an additional check to ensure that only the shortest paths that are less than or equal to *k* hops are included in the computations. The remainder of the *INSERTUPDATEKCLOSENESS* algorithm is the same as the *INSERTUPDATECLOSENESS* algorithm.

6.2.3 Incremental *k*-Closeness Centrality: Shrinking Network Updates

Next, we discuss the changes required for converting the part of the incremental closeness algorithm that handles shrinking network updates to work with *k*-hop bounded shortest paths. Similar to the modifications required to handle growing network updates, to handle shrinking network updates in incremental *k*-closeness centrality, we modify only the *DELETEUPDATECLOSENESS* algorithm. Algorithm–22 presents the pseudocode for *DELETEUPDATEKCLOSENESS* algorithm. To arrive at the *DELETEUPDATEKCLOSENESS* algorithm, there are two changes that need to be made on the *DELETEUPDATECLOSENESS* algorithm: Line – 17 and Line – 28. These two lines are the points where a new shortest path is discovered after an edge on a shortest path has been removed or modified and there is now the possibility of finding a new shortest path. In both of the modified lines, additional conditional checks are inserted to ensure that the sum of the cost of the examined edge and the cost of the rest of the path remains within *k* hops if this shortest path is to be included in the rest of the computations.

Algorithm–22: *DELETEUPDATEKCLOSENESS* (*G*, *src*, *dest*, *z*)

Input: Network $G(V, E)$, deleted or modified edge $\{src \rightarrow dest\}$, and an affected node z.
Output: The shortest distances (D) to the affected node z and the closeness values of the sources of the updated distances are updated.

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

The *InsertCloseness* and the *DeleteCloseness* algorithms remain unchanged except for calling the *InsertUpdateKCloseness* and the *DeleteUpdateKCloseness* algorithms instead of the *InsertUpdateCloseness* and the *DeleteUpdateCloseness* algorithms, respectively. Hence, their pseudocodes are not included in this chapter.

6.3 *k*-Betweenness

This section discusses the algorithms designed for the incremental computation of *k*-betweenness centrality. The term "*k*-betweenness centrality" was introduced by Borgatti and Everett [23]. The *k*-betweenness centrality of a node is defined as the sum of the dependencies of pairs that are at most *k* hops apart from one another [23]. In [47], Brandes provides the pseudocode for computing *k*-betweenness centrality, which is presented as a modification of the original Brandes' algorithm. Chapter 6.3.1 presents the *k*-betweenness algorithm as discussed by Brandes in [47].

The algorithms that are discussed in Chapter 6.3.2 and Chapter 6.3.3 are variations of the algorithms discussed in Chapter 5.3 and Chapter 5.4, respectively. For completeness reasons, we provide the full pseudocode for the modified incremental *k*-betweenness centrality algorithms and point out the specific changes made for the incremental computation of *k*-betweenness centrality.

6.3.1 Brandes' *k*-Betweenness Algorithm

In [47], Brandes provides the pseudocode for computing *k*-betweenness centrality by modifying his original algorithm that is published in [28]. The only change made to the original betweenness algorithm of Brandes is the inclusion of the check for the distance being less than or equal to k. In Line – 15 of the Brandes' k-betweenness algorithm (Algorithm–23), we need to check that the distance to node w is less than k before we insert it into the queue for further processing as a part of the path discovery. Since this check is enforced for all nodes and all paths, the shortest paths are enforced to be bounded by k-hops.

Algorithm–23: *BRANDES' K-BETWEENNESS ALGORITHM* (G)
Input: A network $G(V, E)$.
Output: k-betweenness values of all nodes n $V(G)$ are computed.

```
1. $C_B [\,] \leftarrow 0, v \in V(G)$
2. for $s \in V(G)$
3.     $S \leftarrow$ empty stack
4.     $P[w] \leftarrow$ empty list, $w \in V(G)$
5.     $\sigma [t] \leftarrow 0,\ t \in V(G); \sigma [s] \leftarrow 1$
6.     $d [t] \leftarrow -1, t \in V(G); d [s] \leftarrow 0$
7.     $Q \leftarrow$ empty queue
8.     enqueue $s$ into $Q$
9.     while $Q \neq \emptyset$
10.        dequeue $v \leftarrow Q$
11.        push $v$ into $S$
12.        for neighbor $w$ of $v$
13.            if $d [w] < 0$
14.                $d [w] \leftarrow d [v] + 1$
15.                if $d [w] \leq k$
16.                    enqueue $w$ into $Q$
17.            if $d [w] = d [v] + 1$
18.                $\sigma [w] \leftarrow \sigma [w] + \sigma [v]$
19.                append $v$ into $P[w]$
20. $\delta [v] \leftarrow 0, v \in V(G)$
21. while $S \neq \emptyset$
22.     pop $w \leftarrow S$
23.     for $v \in P[w]$
24.         $\delta [v] \leftarrow \delta [v] + \frac{\sigma [v]}{\sigma [w]} .(1 + \delta [w])$
25.     if $w \neq s$
26.         $C_B [w] \leftarrow C_B [w] + \delta [w]$
```

6.3.2 Incremental *k*-Betweenness Centrality: Growing Network Updates

Next, we discuss the changes required for converting the part of the incremental betweenness centrality algorithm that handles growing network updates to work with *k*-hop bounded shortest paths. As observed in the *k*-closeness centrality algorithms as well as the Brandes' *k*-betweenness algorithm, the modifications required to limit the shortest paths to be less than or equal to *k* hops are done as a part of the path discovery process. Since the path discovery is handled as a part of the *INSERTUPDATEBETWEENNESS* algorithm, we provide the pseudocode for the modified version of the *INSERTUPDATEBETWEENNESS* algorithm as the *INSERTUPDATEKBETWEENNESS* algorithm. The *REDUCEBETWEENNESS* and the *INCREASEBETWEENNESS* algorithms remain the same. Thus, their pseudocodes are not included again in this chapter. The *INSERTBETWEENNESS* algorithm needs to invoke the *INSERTUPDATEKBETWEENNESS* algorithm instead of the *INSERTUPDATEBETWEENNESS* algorithm to compute the *k*-betweenness centrality incrementally. Since this is a trivial change, the pseudocode for the modified version of the *INSERTBETWEENNESS* algorithm is not included in this chapter.

There are two conditional checks inserted into the *INSERTUPDATEBETWEENNESS* algorithm to arrive at the *INSERTUPDATEKBETWEENNESS* algorithm: Line – 7 and Line – 15 of the *INSERTUPDATEKBETWEENNESS* algorithm presented below. Originally, Line – 7 checks if the path with the alternative distance *alt* (i.e. the distance alt is obtained by summing up the cost of the edge under examination and the cost of the rest of the shortest path to the affected node *z*) is shorter than the previously known shortest path(s).

Algorithm–24: *INSERTUPDATEKBETWEENNESS* (*G, src, dest, z, PairsDone*)
Input: Network G, newly inserted/modified edge $\{src \rightarrow dest\}$, an affected node z.
Output: The shortest distances to z. The number of shortest paths and the predecessors on these shortest paths are also updated.

```
1.  Workset ←{$src \rightarrow dest$} //Min-key priority queue w.r.t the distance of the head of the edge to $src$
2.  VisitedVertices ← $src$
3.  AffectedVertices ← ∅
4.  while Workset ≠ ∅
5.    {$x \rightarrow y$} ← pop (Workset)
6.    $alt \leftarrow C(x, y) + D(y, z)$
7.    if $alt < D(x, z)$ & $alt \leq k$
8.      if $<x, z> \notin \sigma_{old}$
9.        $D_{old}(x, z) \leftarrow D(x, z)$;    $\sigma_{old}(x, z) \leftarrow \sigma(x, z)$
10.       REDUCEBETWEENNESS $(x, z)$
11.       $\sigma(x, z) \leftarrow 0$; $P_x(z) \leftarrow \emptyset$
12.     if $[u, z] \in PairsDone$
13.       Remove $[x, z]$ from $PairsDone$
14.     $D(x, z) \leftarrow alt$
15.   if $alt = D(x, z)$ & $D(x, z) \neq \infty$ & $alt \leq k$
16.     if $[x, z] \notin PairsDone$
17.       if $<x, z> \notin \sigma_{old}$
18.         REDUCEBETWEENNESS $(x, z)$
19.       if $\sigma(x, z) \neq 0$
20.         $\sigma_{old}(x, z) \leftarrow \sigma(x, z)$
21.       $\sigma(x, z) \leftarrow \sigma(x, z) + (\sigma(x, src) * 1 * \sigma(dest, z))$
22.       Append $x$ to $P_x(y)$
23.       Append $P_y(z)$ to $P_x(z)$
24.     Insert $[x, z]$ into $PairsDone$
25.     Insert $x$ into AffectedVertices
26.     for $u \in$ Pred $(x)$ //sorted w.r.t. edge costs in ascending order
27.       if SP $(u, x, src)$ and $u \notin$ VisitedVertices
28.         push {$u \rightarrow x$} into Workset
29.         Insert $u$ into VisitedVertices
30. return Affected Vertices
```

Similarly, Line – 15 checks if the alternative distance *alt* is equal to the currently known shortest path length to see if it is a newly found shortest path. Both of these lines are entry conditions for two cases of the path discovery phase: (*i*) the discovery of strictly shorter paths and (*ii*) the discovery of the shortest paths of equal length. In the *INSERTUPDATEKBETWEENNESS* algorithm, these conditions are supplemented by additional (***alt* ≤ *k***) checks to ensure that the newly found shortest paths also obey the restriction for the shortest paths to be less than or equal to *k* hops.

6.3.3 Incremental *k*-Betweenness Centrality: Shrinking Network Updates

Next, we discuss the changes required for converting the part of the incremental betweenness centrality algorithm that handles shrinking network updates to work with *k*-hop bounded shortest paths. The *CLEARBETWEENNESS*, *ADJUSTNPS*, and *ADJUSTBETWEENNESS* algorithms remain unchanged while the *DELETEBETWEENNESS* algorithm has straightforward modifications where the invocations of the *DELETEUPDATEBETWEENNESS* algorithm are replaced by the invocations of the *DELETEUPDATEKBETWEENNESS* algorithm. Hence, only the pseudocode for the *DELETEUPDATEKBETWEENNESS* algorithm (Algorithm–25) is provided in this chapter.

In the *DELETEUPDATEKBETWEENNESS* algorithm, there are four different points where restricting conditions for the shortest paths to be less than or equal to *k* hops are inserted: Line – 16, Line – 20, Line – 28, and Line – 36. The changes in Line – 16 and Line – 20 of the *DELETEUPDATEKBETWEENNESS* algorithm are a part of the path discovery process after a line has been removed or modified. Line – 16 checks for strictly shorter paths while Line – 20 checks for the shortest paths of equivalent length.

Algorithm–25: ***DELETEUPDATEKBETWEENNESS* (G, src, dest, z)**

Input: Network $G(V,E)$, deleted or modified edge $\{src \rightarrow dest\}$, and an affected node z.

Output: The shortest distances (D) to the affected node z and the predecessors on these shortest paths are updated. The k-betweenness values of the intermediates on the invalid paths are reduced.

```
1.  AffectedVertices ← $\emptyset$
2.  Workset ←{$src$};
3.  while Workset $\neq \emptyset$
4.    $u$ ←pop (Workset)
5.    if $u \notin$ AffectedVertices then add $u$ to AffectedVertices
6.    for $x \in$ Pred($u$) such that SP($x, u, z$)
7.      add $x$ into Workset
8.  AffVert ← AffectedVertices.copy()
9.  Q_inc ← $\emptyset$
10. while AffVert $\neq \emptyset$
11.     $a$ ← extractMin(AffVert)
12.     if ($< a, z > \notin \sigma_{old}$) then add $< a, z, \sigma(a, z) >$ into $\sigma_{old}$ and CLEARBETWEENNESS ($a, z$)
13.     if ($< a, z > \notin D_{old}$) then add $< a, z, D(a, z) >$ into $D_{old}$
14.     myMin ← $\infty$
15.     for $b \in$ Succ($a$)
16.        if ($C(a,b) + D(b,z) <$ myMin **&** $(\boldsymbol{C(a,b) + D(b,z)) \leq k}$)
17.          myMin← $C(a,b) + D(b,z)$
18.          $P_a(z)$ ← $\emptyset$
19.          if $b$ = z ? Append $a$ to $P_a(z)$: Append $P_b(z)$ to $P_a(z)$
20.        else if ($C(a,b) + D(b,z)$ = myMin & myMin $\neq \infty$ & $b \notin$ AffectedVertices **& myMin** $\leq \boldsymbol{k}$)
21.          if $b$ = z ? Append $a$ to $P_a(z)$: Append $P_b(z)$ to $P_a(z)$
22.     $D(a, z)$ ← myMin
23.     if myMin $\neq \infty$ & $a \notin$ Q_inc
24.        add $a$ into Q_inc
25. while Q_inc $\neq \emptyset$
26.     $a$ ← extractMin(Q_inc)
27.     for $c \in$ Pred($a$)
28.        if ($C(c,a) + D(a,z) < D(c,z)$ & $c \notin$ AffectedVertices **&** $\boldsymbol{C(c,a) + D(a,z) \leq k}$)
29.          if($< c, z > \notin \sigma_{old}$) then add $< c, z, \sigma(c, z) >$ into $\sigma_{old}$ and CLEARBETWEENNESS ($c, z$)
30.          if ($< c, z > \notin D_{old}$) then add $< c, z, D(c, z) >$ into $D_{old}$
31.          $P_c(z)$ ← $\emptyset$
32.          if $a$ = z ? Append $c$ to $P_c(z)$: Append $P_a(z)$ to $P_c(z)$
33.          $D(c,z)$ ← $C(c,a) + D(a,z)$
34.          if $c \notin$ Q_inc
35.            add $c$ into Q_inc
36.        else if ($C(c,a) + D(a,z) = D(c,z)$ & $c \notin$ AffectedVertices **&** $\boldsymbol{C(c,a) + D(a,z) \leq k}$)
37.          if($< c, z > \notin \sigma_{old}$) then add $< c, z, \sigma(c, z) >$ into $\sigma_{old}$ and CLEARBETWEENNESS ($c, z$)
38.          if $a$ = z ? Append $c$ to $P_c(z)$: Append $P_a(z)$ to $P_c(z)$
39.          if $c \notin$ Q_inc
40.            add $c$ into Q_inc
41. return AffectedVertices
```

The changes in the Line – 28 and Line – 36 are a part of the second phase of the shortest path discovery process where new shortest paths are searched for using the other shortest paths that are discovered in the first phase. Line – 28 checks for strictly shorter paths while Line – 36 checks for the shortest paths of equivalent length.

Chapter 8.3.4 and Chapter 8.4.3 discuss the performance improvements of the incremental *k*-centrality algorithms in detail for synthetic and real-life networks, respectively. Consider the following performance result example. For a 5000-node Erdos-Renyi network, the *k*-betweenness centrality based on the Brandes' algorithm runs 64.1 times faster than the original Brandes' betweenness algorithm for $k = 2$. Then, with the incorporation of the incremental approach, the incremental *k*-betweenness algorithm runs 64049 times faster than the *k*-betweenness algorithm based on the Brandes' algorithm. Such a significant performance improvement comes from the fact that the number of affected nodes for the incremental *k*-betweenness centrality is very small. After updating a small number of shortest paths in a restricted area, the incremental *k*-betweenness algorithm terminates. However, the non-incremental version of the *k*-betweenness algorithm needs to compute the shortest paths from every single one of 5000 nodes even if the shortest paths are restricted to *k* hops. Hence, the centrality algorithms benefit significantly from the incremental algorithm design, especially when the shortest paths are restricted to remain within *k* hops, forcing the portion of the network affected by the incremental network update to be small. The results presented in Chapter 8.3.4 and Chapter 8.4.3 focus on demonstrating the performance improvements observed due to incorporation of the incremental algorithm design. Therefore, the results presented in

Chapter 8.3.4 and Chapter 8.4.3 compare the performance benefits of the incremental *k*-centrality algorithms over the *k*-betweenness algorithm based on the Brandes' betweenness algorithm and the *k*-closeness algorithm based on the Dijsktra's shortest path algorithm. The performance benefits that only come from the *k*-hop approximations of the global centrality measures computed across the entire network have been demonstrated earlier in the literature in different contexts in a number of papers including [18], [135], and [136]. The findings of Pfeffer and Carley in [18] suggest that *k*-centralities have substantial improvements in small-world type of networks where the networks are highly clustered and there are relatively long paths. On the contrary, in [18], the authors show that preferential attachment networks benefit less from *k*-hop approximations of centralities due to relatively shorter average path distances. Both of these results are inline with our findings as further detailed in Chapter 8.3.4.

The memory improvements primarily stem from restricting the depth of the shortest path trees built from each source node to remain less than or equal to *k*. Unless restricted to remain within the first *k* hops, the depths of the shortest path trees might go as high as the diameter of the network, which can be very long especially in small-world type of networks, as shown in the network properties in Table 6, Table 7, and Table 8 in Chapter 7.1. This is especially helpful for the computation of betweenness centrality, which requires information on the intermediate nodes that lie on the shortest paths in a network.

6.4 DISCUSSION ON THE ANALYSIS OF *K*-CENTRALITIES

The runtime and memory consumption analyses done for the incremental closeness centrality algorithm (Chapter 4.5) and the incremental betweenness centrality algorithm (Chapter 5.5) are valid for the incremental *k*-closeness and the incremental *k*-betweenness algorithms as well. However, a lot of the complexities are expressed in terms of the sizes of affected sink and source node sets, the number of shortest paths, and the average shortest path length.

When the shortest paths in a network are bounded to remain within *k* hops, the percentages of affected nodes drop substantially as discussed later in the results section in Chapter 8. For instance, the percentages of affected nodes drop from 0.89% - 68.31% range to 0.32%-3.47% range for the incremental betweenness centrality when $k = 3$ is used as the bounding parameter.

In addition, some of the nodes that were connected by long, multi-hop paths before are not necessarily connected anymore when the paths connecting them cannot be any longer than *k* hops. This reflects as a reduction in $Conn(G)$ which is used to denote the number of node pairs that are connected across the entire network topology.

Moreover, the depth of a shortest path tree stemming from any node in the network cannot be any longer than *k* hops. However, when the shortest paths are not bounded to remain within *k* hops, the maximum depth of a shortest path tree stemming from a node is equal to the diameter of the network. Considering the network statistics presented in Table 6 and Table 7 in Chapter 7 for the controlled experiments with the synthetic networks, the average shortest path lengths for synthetic network with 1000-5000 nodes are in the range of 3.45-20.26 hops while the diameters of the same networks are in the range of 10-102 hops. Although both the average shortest path length and the diameter values are much lower than $(n-1)$(i.e. the theoretical worst case upper bound for the length of a shortest path in a network), they are still considerably longer than 2 or 3 hops, which would be the maximum when $k = 2$ or $k = 3$ is used as the bounding parameter; offering significant opportunity for improvement via the use of k-centralities in terms of memory consumption and speedup for very large problem sizes.

6.5 EXAMPLE APPLICATIONS OF INCREMENTAL CENTRALITY ALGORITHMS

This section examines a number of potential use cases for the incremental centrality algorithms in different contexts. First, we discuss how clustering algorithms can benefit from the incremental computation of the betweenness centrality. Then, we discuss the use of incremental centrality algorithms in real life, physical networks (e.g. wireless mesh networks). Rather than focusing on how fast incremental centrality metrics can be computed on such topologies, we focus on how they can be used in real-life wireless networks.

6.5.1 Example Use Case in Clustering (Community Detection)

In this dissertation, a modified version of the Girvan-Newman algorithm is investigated based on the proposed incremental betweenness algorithm. The Girvan-Newman clustering algorithm is a hierarchical, divisive graph-clustering algorithm that uses edge betweenness and generates a dendrogram as output [19] [119]. It successively partitions the graph based on edge betweenness centrality. Edge betweenness is a variant of regular vertex betweenness where an edge's betweenness depends on the fraction of the shortest paths using that edge across all pairs of nodes. The main steps of the Girvan-Newman clustering algorithm are as follows [19]:

1. Calculate the edge-betweenness for all edges in the network.
2. Remove the edge with the highest edge-betweenness.
3. Recalculate the edge-betweenness of all the edges for the remaining topology.
4. Repeat from the 2nd step until no edges remain on the graph.

The Girvan-Newman clustering algorithm has been widely used and a number of variants of it have been proposed so far. This algorithm has been used for several types of networks, especially in the context of biological networks and social networks [137].

In [136] and independently later in [135] the use of *k*-betweenness in clustering has been investigated and modifications of the Girvan-Newman clustering algorithm have been discussed. In [136], the *k*-hop limited version of the local flow betweenness is used while in [135], the *k*-hop limited version of the classical betweenness is used. Both in [136] and [135], the main arguments were speeding up the execution time and focusing on local groups so that a change in a far apart group does not necessarily affect clustering of another group.

The authors of [136] focus on validating the groupings formed incorporating the *k*-hop limited version of the local flow in the Girvan-Newman clustering algorithm. The proposed clustering algorithm is validated using a corpus of instant messages that called MLog and a larger IBM intranet, global dataset. The resulting clusters were evaluated by the appropriate executive subject-matter experts of IBM through surveys and feedback and found to reflect the business relations quite accurately.

On the other hand, in [135], Gregory investigates the performance improvements that can be achieved by using the *k*-hop limited approximation of the betweenness centrality instead of using the regular (i.e. full or global) betweenness centrality to discover communities in a network. In a 1133-node real-life network, the performance improvement achieved over the Girvan-Newman clustering algorithm is 312x when $k = 2$ while the performance improvement increases up to 1325x when the network size is 10680 nodes in terms of total execution time.

The modifications done on the Girvan-Newman clustering algorithm also include performing fuzzy (overlapping) clustering [138]. Steve Gregory, the author of [138], later found that the fuzzy clustering algorithm he proposed based on the Girvan-Newman algorithm is very slow and, in [139], he extends his previous work ([138]) to propose a faster overlapping algorithm that combines the use of *k*-betweenness and fuzzy grouping on top of the Girvan-Newman clustering algorithm. Steve Gregory finds that a lot of the real-life networks are indeed too large for the clustering algorithms that repeated use the global version of the betweenness centrality and provides performance results using only $k = 3$ and $k = 2$ instead. The author also observes that using $k = 3$ or $k = 2$ makes more of a performance difference in larger networks (7000+ nodes) than in smaller networks (300-2000 nodes). In addition, the author also discusses that using the *k*-hop limited version of the betweenness centrality in clustering provides fairly accurate groupings in most cases. However, its accuracy decreases with respect to using the full (global) betweenness centrality when the diameters of the groups are larger than the k parameter value selected.

The main disadvantage of the Girvan-Newman clustering algorithm is its time complexity. This is also the reason why several variants of the Girvan-Newman clustering algorithm have been proposed to use k-betweenness to make it faster. The time complexity of the Girvan-Newman algorithm is $O(m^2n)$, where m is the number of edges and n is the number of nodes in the network. The Girvan-Newman algorithm loops through the main steps of the algorithm m times and Step-3 takes $O(mn)$ time for binary network. If the costs of the edges are taken into account, then Step-3 takes $O(mn + n^2\log n)$; yielding $O(m^2n + mn^2\log n)$ overall time complexity for the Girvan-Newman algorithm. However, in the worst case, the maximum possible number of edges, m, is $O(n^2)$. Therefore, the overall time complexity of the Girvan-Newman clustering algorithm can also be stated as $O(n^5)$.

Due to the high complexity of the Girvan-Newman clustering algorithm, it is not always feasible to use it in larger networks. In addition to its high time complexity, it is also an iterative algorithm, repeatedly recomputing the betweenness values from scratch over and over again. These characteristics suggest that the Girvan-Newman clustering algorithm can benefit from an incremental retrofit that incorporates the incremental betweenness algorithm proposed in Chapter 5.4.

The rest of this section first discusses how to convert the incremental betweenness algorithm proposed in Chapter 5.4 to have a different accounting so that the betweenness values are calculated for the edges instead of the nodes. Then, a modified version of the Girvan-Newman algorithm is presented which incorporates the incremental edge betweenness algorithm in its main steps.

6.5.1.1 Incremental Edge Betweenness with Shrinking Network Updates

The Girvan-Newman clustering algorithm iteratively removes the edges from a network and recalculates the edge betweenness values for the remaining edges. Hence, to incorporate the incremental algorithm design into the Girvan-Newman clustering algorithm, modifying only the parts of the incremental betweenness centrality algorithm that handle the shrinking network updates for the computation of edge betweenness centrality is sufficient.

In particular, to make the incremental betweenness centrality work with the Girvan-Newman clustering algorithm, the betweenness centrality values should be maintained for each edge in the network instead of each node in the network. In short, the only difference between the incremental node betweenness and the incremental edge betweenness algorithms is how the accounting for the betweenness centrality values is done. For edge betweenness centrality, a betweenness value should be stored for each edge in the network instead of each node in the network. In addition, when the shortest paths are updated due to an incremental network update, instead of adjusting the betweenness values of all the intermediate nodes, the edge betweenness values of all the edges on the updated shortest paths should be adjusted.

As explained earlier in Chapter 5.4, the part of the incremental betweenness algorithm that handles shrinking network updates consists of the following sub-algorithms: *DELETEBETWEENNESS, DELETEUPDATEBETWEENNESS*, *CLEARBETWEENNESS*, *ADJUSTNPS*, and *ADJUSTBETWEENNESS*.

Out of these five sub-algorithms, the *ADJUSTNPS* algorithm remains the same, with no modifications. The *ADJUSTNPS* algorithm is responsible from adjusting the number of shortest paths between node pairs, and it does not have anything to do with how the accounting for the betweenness values is done. The *DELETEBETWEENNESS* and *DELETEUPDATEBETWEENNESS* algorithms remain primarily the same except for a couple of lines where the edge betweenness versions of the *CLEARBETWEENNESS* and the *ADJUSTBETWEENNESS* algorithms are invoked. The pseudocodes for the modified *DELETEBETWEENNESS* and the *DELETEUPDATEBETWEENNESS* algorithms are also provided for completeness reasons. To avoid confusion, the edge betweenness versions of these two algorithms are renamed as the *DELETEEDGEBETWEENNESS* and the *DELETEUPDATEEDGEBETWEENNESS*, respectively.

The *CLEARBETWEENNESS* and the *ADJUSTBETWEENNESS* algorithms are the main parts of the incremental betweenness centrality algorithm where the betweenness centrality values are modified. The pseudocodes for the modified *CLEARBETWEENNESS* and *ADJUSTBETWEENNESS* algorithms are also presented and the changes relevant for the edge betweenness computation are discussed in detail. To avoid confusion, the versions of these algorithms that are designed for the edge betweenness computation are also renamed as *CLEAREDGEBETWEENNESS* and *ADJUSTEDGEBETWEENNESS*, respectively.

6.5.1.1.1 DELETEEDGEBETWEENNESS ALGORITHM

The *DELETEEDGEBETWEENNESS* algorithm (Algorithm-26) is a modification of the *DELETEBETWEENNESS* algorithm (Algorithm-15) presented in Chapter 5.4.1. The difference of the *DELETEEDGEBETWEENNESS* algorithm from the *DELETEBETWEENNESS* algorithm is that the *DELETEEDGEBETWEENNESS* algorithm invokes the edge betweenness versions of the algorithms that were initially presented in Chapter 5.4 for the incremental betweenness centrality at the node level.

Algorithm-26: *DELETEEDGEBETWEENNESS* (*G*, *src*, *dest*, *c*)

Input: Network $G(V, E)$ for which betweenness values of all edges (B_E) are pre-computed. A modified edge $\{src \rightarrow dest\}$ with a cost of c, where $c > C(src, dest)$. The cost parameter c is optional. If it is not provided in the argument list, it is set to ∞ by default and the edge is deleted.

Output: Network $G(V, E)$ updated with the changes on the edge $\{src \rightarrow dest\}$. The shortest distances between all nodes (D), the number of shortest paths, the predecessors on the shortest paths and betweenness values of all edges (B_E) are also updated.

```
1.  $C(src, dest) \leftarrow c$;  $\tilde{C}(dest, src) \leftarrow c$
2.  $\sigma_{old} \leftarrow [\,]$; $D_{old} \leftarrow [\,]$; $trackLost \leftarrow [\,]$
3.  AffectedSinks    ← DELETEUPDATEEDGEBETWEENNESS ($\tilde{G}$, $dest$, $src$, $src$)
4.  AffectedSources  ← DELETEUPDATEEDGEBETWEENNESS ($G$, $src$, $dest$, $dest$)
5.  for $s \in$ AffectedSinks
6.      DELETEUPDATEEDGEBETWEENNESS ($G$, $src$, $dest$, $s$)
7.  for $s \in$ AffectedSources
8.      DELETEUPDATEEDGEBETWEENNESS ($\tilde{G}$, $dest$, $src$, $s$)
9.  ADJUSTNPS ( )
10. ADJUSTEDGEBETWEENNESS ( )
```

6.5.1.1.2 DeleteUpdateEdgeBetweenness Algorithm

The *DeleteUpdateEdgeBetweenness* algorithm (Algorithm-27) is a version of

Algorithm-27: *DeleteUpdateEdgeBetweenness* (*G*, *src*, *dest*, *z*)

```
1.  AffectedVertices ← ∅
2.  Workset ←{$src$};
3.  while Workset ≠ ∅
4.    $u$ ←pop (Workset)
5.    if $u \notin$ AffectedVertices then add $u$ to AffectedVertices
6.    for $x \in$ Pred($u$) such that SP($x, u, z$)
7.      add $x$ into Workset
8.  AffVert ← AffectedVertices.copy()
9.  Q_inc ← ∅
10. while AffVert ≠ ∅
11.     $a$ ← extractMin(AffVert)
12.     if($< a, z > \notin \sigma_{old}$) then add $< a, z, \sigma(a, z) >$ into $\sigma_{old}$ and ClearEdgeBetweenness ($a, z$)
13.     if ($< a, z > \notin D_{old}$) then add $< a, z, D(a, z) >$ into $D_{old}$
14.     myMin ← ∞
15.     for $b \in$ Succ($a$)
16.        if ($C(a, b) + D(b, z) <$ myMin)
17.          myMin← $C(a, b) + D(b, z)$
18.          $P_a(z)$ ← ∅
19.          if $b$ = z ? Append $a$ to $P_a(z)$: Append $P_b(z)$ to $P_a(z)$
20.        else if ($C(a, b) + D(b, z)$ = myMin & myMin ≠ ∞ & $b \notin$ AffectedVertices)
21.          if $b$ = z ? Append $a$ to $P_a(z)$: Append $P_b(z)$ to $P_a(z)$
22.     $D(a, z)$ ← myMin
23.     if myMin ≠ ∞ & $a \notin$ Q_inc
24.        add $a$ into Q_inc
25. while Q_inc ≠ ∅
26.     $a$ ← extractMin(Q_inc)
27.     for $c \in$ Pred($a$)
28.        if ($C(c, a) + D(a, z) < D(c, z)$ & $c \notin$ AffectedVertices)
29.          if($< c, z > \notin \sigma_{old}$) then add $< c, z, \sigma(c, z) >$ into $\sigma_{old}$ and ClearEdgeBetweenness ($c, z$)
30.          if ($< c, z > \notin D_{old}$) then add $< c, z, D(c, z) >$ into $D_{old}$
31.          $P_c(z)$ ← ∅
32.          if $a$ = z ? Append $c$ to $P_c(z)$: Append $P_a(z)$ to $P_c(z)$
33.          $D(c, z) \leftarrow C(c, a) + D(a, z)$
34.          if $c \notin$ Q_inc
35.            add $c$ into Q_inc
36.        else if ($C(c, a) + D(a, z) = D(c, z)$ & $c \notin$ AffectedVertices)
37.          if($< c, z > \notin \sigma_{old}$) then add $< c, z, \sigma(c, z) >$ into $\sigma_{old}$ and ClearEdgeBetweenness ($c, z$)
38.          if $a$ = z ? Append $c$ to $P_c(z)$: Append $P_a(z)$ to $P_c(z)$
39.          if $c \notin$ Q_inc
40.            add $c$ into Q_inc
41. return AffectedVertices
```

the *DeleteUpdateBetweenness* algorithm (Algorithm-16) presented in Chapter 5.4.2. Only three lines are modified to convert the *DeleteUpdateBetweenness* algorithm into the *DeleteUpdateEdgeBetweenness* algorithm: Line 12, Line 29, and Line 37. The changes are marked with bold font.

6.5.1.1.3 ClearEdgeBetweenness Algorithm

Assume that an edge that lies on the shortest path(s) from a node a to another node z is removed and the previously known shortest path became disconnected and it cannot be the shortest path any more. The *ClearEdgeBetweenness* algorithm is responsible from reducing the edge betweenness values of the edges that lie on the previously known shortest paths from node a to node z, which are not the shortest paths any more. The *ClearEdgeBetweenness* algorithm has two phases. The second phase of the algorithm starts at Line 8 of Algorithm–28.

In Line 3 of Algorithm–28, each edge $\{s \rightarrow t\}$ that appears as a part of the currently-known shortest path(s) is an element of the list $I_E(a, z)$. $I_E(a, z)$ denotes the list of edges that form the shortest paths from node a to node z just before the invalidation of the old set of shortest paths from a to z. Line 4 checks if the edge $\{s \rightarrow t\}$ was originally a part of the shortest paths from node a to node z, before any update is made on the network. Since $I_E(a, z)$ returns the current list of the edges that appear to be on the shortest paths from node a to node z, it may contain some newly inserted edges which were not in the original set of edges that constitute the shortest paths from node a to node z before any incremental update is made on the network. If the edge $\{s \rightarrow t\}$ is in the list of original set of the edges that constitute the shortest paths from node a to node z, then its edge betweenness value ($B_E(\{s \rightarrow t\})$) is reduced by the original contribution of the node pair $<a, z>$ to the edge betweenness value of the edge $\{s \rightarrow t\}$.

Algorithm–28: *CLEAREDGEBETWEENNESS* (*a*, *z*)
Input: A source node a and a destination node z. The old set of shortest paths from a to z need to be invalidated.
Output: The edge betweenness values of the edges on the previously known shortest paths are reduced.

```
1.  $AlreadyDone \leftarrow \emptyset$
2.  if $\sigma(a, z) \neq 0$
3.      for each edge $\{s \rightarrow t\} \in I_E(a, z)$
4.          if $(D_{old}(a, s) + C(s, t) + D_{old}(t, z) = D_{old}(a, z))$
5.              add $\{s \rightarrow t\}$ into $AlreadyDone$
6.              $B_E(\{s \rightarrow t\}) \leftarrow B_E(\{s \rightarrow t\}) - (\sigma_{old}(a, s) * 1 * \sigma_{old}(t, z) / \sigma_{old}(a, z))$
7.              add $<<a, z>, \{s \rightarrow t\}>$ into $trackLost$
8.  for $(<x, y> \in trackLost$.KeySet())
9.      if $(D_{old}(a, x) + D_{old}(x, y) + D_{old}(y, z) = D_{old}(a, z))$
10.         for each edge $\{s \rightarrow t\} \in trackLost<x, y>$
11.             if $(D_{old}(a, s) + C(s, t) + D_{old}(t, z) = D_{old}(a, z)$ & $\{s \rightarrow t\} \notin AlreadyDone$
12.                 add $\{s \rightarrow t\}$ into $AlreadyDone$
13.                 $B_E(\{s \rightarrow t\}) \leftarrow B_E(\{s \rightarrow t\}) - (\sigma_{old}(a, s) * 1 * \sigma_{old}(t, z) / \sigma_{old}(a, z))$
14.                 add $<<a, z>, \{s \rightarrow t\}>$ into $trackLost$
```

This contribution is calculated as the fraction of the original set of shortest paths from node a to node z that pass through the edge $\{s \rightarrow t\}$. This fractional value is computed by breaking the shortest paths from node a to node z into several pieces. The old number of shortest paths from node a to node z that pass through the edge $\{s \rightarrow t\}$ is represented as the and computed as the multiplication of the old number of shortest paths from node a to node s and the old number of shortest paths from node t to node z. Then, this value is divided by the total number of shortest paths from node a to node z: $(\sigma_{old}(a, s) * 1 * \sigma_{old}(t, z)) / \sigma_{old}(a, z)$. To ensure that only the edge $\{s \rightarrow t\}$ is considered among all possible shortest paths from node a to node z, the number of shortest paths from node s to node t is replaced with 1.

In the second phase of the *CLEAREDGEBETWEENNESS* algorithm, we look for the edges on the original set of shortest paths that are not visible in the current list of edges that make up the currently known shortest paths. An edge $\{s \rightarrow t\}$ which was originally a part of the shortest paths from node a to node z might be updated and removed when a sub-path from node x to node y is updated: $a \ldots \rightarrow x \ldots \{s \rightarrow\ t\} \ldots \rightarrow\ y \ldots \rightarrow\ z$. In Line 9 of Algorithm–28, we check if the shortest paths from node x to node y were sub-paths of the shortest paths from node a to node z. If the condition in Line 9 is satisfied, then, Line 11 checks if the edge $\{s \rightarrow t\}$ was originally a part of the shortest paths from node a to node z. Note that $\{s \rightarrow t\}$ is different from the edge $\{s \rightarrow t\}$ used in the first phase of the algorithm; it is rather an overloaded use of the notation for an edge. If both conditions hold, then the same of set of operations that was applied in the first phase (Lines 5 – 7) is applied for the edge $\{s \rightarrow t\}$ in Lines 12 – 14 as well: (*i*) inserting $\{s \rightarrow t\}$ into the *AlreadyDone* set, which is a set of edges that are already processed for the <a, z> node pair and the set *AlreadyDone* is not maintained across invocations of this algorithm (*ii*) reducing the edge betweenness value of the edge $\{s \rightarrow t\}$ by the original contribution of the node pair <a, z>, and (*iii*) inserting the edge $\{s \rightarrow t\}$ into the *trackLost* set which is maintained during the entire propagation of a network. Since the edges (e.g. $\{s \rightarrow t\}$) that were on the previously known shortest paths which are not the shortest paths anymore can no longer be reached by following the predecessors, there is need for a mechanism to keep track of such edges. Therefore, the *trackLost* set contains key-value pairs where the key consists of the node pair (e.g <*a*, *z*>) whose shortest paths are about to be invalidated, and the value contains the list of edges on the invalidated paths.

6.5.1.1.4 ADJUSTEDGEBETWEENNESS ALGORITHM

The *ADJUSTEDGEBETWEENNESS* algorithm (Algorithm–29) is a fairly straightforward algorithm. The *ADJUSTEDGEBETWEENNESS* algorithm loops through the set of node pairs <*a*, *z*> that had their shortest paths changed either in terms of number or length (i.e. the node pairs listed in σ_{old}). For each node pair <*a*, *z*>, the shortest paths are constructed on demand by following the predecessors and the edges on these shortest paths are represented as $I_E(a, z)$. Then, the edge betweenness value of each edge $\{s \rightarrow t\}$ in $I_E(a, z)$ is increased by the fraction of shortest paths from node *a* to node *z* that use the edge $\{s \rightarrow t\}$.

Algorithm–29: *ADJUSTEDGEBETWEENNESS* ( )
Input: The list of node pairs between which the shortest paths are updated.
Output: The edge betweenness values of the edges on the affected shortest paths are increased by the fraction of shortest paths they lie on.

```
1. for $(a, z) \in \sigma_{old}$
2.     for each edge $\{s \rightarrow t\} \in I_E(a, z)$
3.         $B_E(\{s \rightarrow t\}) \leftarrow B_E(\{s \rightarrow t\}) + (\sigma(a, s) * 1 * \sigma(t, z) / \sigma(a, z))$
```

6.5.1.1.5 INCREMENTAL EDGE K-BETWEENNESS ALGORITHM

The algorithms discussed earlier in this chapter (Chapter 6.5.1.1.1 - Chapter 6.5.1.1.4) can also be used to calculate the *k*-hop limited version of the edge betweenness centrality.

Algorithm–30: *DELETEUPDATEEDGEBETWEENNESS-K* (*G*, *src*, *dest*, *z*)

```
1.  AffectedVertices ← ∅
2.  Workset ←{src};
3.  while Workset ≠ ∅
4.    u ←pop (Workset)
5.    if u ∉ AffectedVertices then add u to AffectedVertices
6.    for x ∈ Pred(u) such that SP(x, u, z)
7.      add x into Workset
8.  AffVert ← AffectedVertices.copy()
9.  Q_inc ← ∅
10. while AffVert ≠ ∅
11.     a ← extractMin(AffVert)
12.     if(< a, z > ∉ σ_old) then add < a, z, σ(a, z) > into σ_old and CLEAREDGEBETWEENNESS (a, z)
13.     if (< a, z > ∉ D_old) then add < a, z, D(a, z) > into D_old
14.     myMin ← ∞
15.     for b ∈ Succ(a)
16.         if (C(a, b) + D(b, z) < myMin & (C(a, b) + D(b, z)) ≤ k)
17.           myMin← C(a, b) + D(b, z)
18.           P_a(z) ← ∅
19.           if b = z ? Append a to P_a(z): Append P_b(z) to P_a(z)
20.         else if (C(a, b) + D(b, z) = myMin & myMin ≠ ∞ & b ∉ AffectedVertices & myMin ≤ k)
21.           if b = z ? Append a to P_a(z): Append P_b(z) to P_a(z)
22.     D(a, z) ← myMin
23.     if myMin ≠ ∞ & a ∉ Q_inc
24.         add a into Q_inc
25. while Q_inc ≠ ∅
26.     a ← extractMin(Q_inc)
27.     for c ∈ Pred(a)
28.         if (C(c, a) + D(a, z) < D(c, z) & c ∉ AffectedVertices & C(c, a) + D(a, z) ≤ k)
29.           if(< c, z > ∉ σ_old) then add < c, z, σ(c, z) > into σ_old and CLEAREDGEBETWEENNESS (c, z)
30.           if (< c, z > ∉ D_old) then add < c, z, D(c, z) > into D_old
31.           P_c(z) ← ∅
32.           if a = z ? Append c to P_c(z): Append P_a(z) to P_c(z)
33.           D(c, z) ← C(c, a) + D(a, z)
34.           if c ∉ Q_inc
35.             add c into Q_inc
36.         else if (C(c, a) + D(a, z) = D(c, z) & c ∉ AffectedVertices & C(c, a) + D(a, z) ≤ k)
37.           if(< c, z > ∉ σ_old) then add < c, z, σ(c, z) > into σ_old and CLEAREDGEBETWEENNESS (c, z)
38.           if a = z ? Append c to P_c(z): Append P_a(z) to P_c(z)
39.           if c ∉ Q_inc
40.             add c into Q_inc
41. return AffectedVertices
```

This can be achieved by modifying only the *DELETEUPDATEEDGEBETWEENNESS* algorithm to include the conditional checks in the *DELETEUPDATEKBETWEENNESS* algorithm for discovering the shortest paths within *k*-hops. The changes are minor and the modified lines are marked in bold in the *DELETEUPDATEEDGEBETWEENNESS-K* algorithm (Algorithm-30). Since the changes related to the *k*-hop centrality were discussed earlier in Chapter 6.3.3 they are not discussed here again.

6.5.1.2 Incremental *k*-Betweenness Clustering Algorithm

This section discusses an incremental *k*-betweenness clustering algorithm, which is a modified version of the Girvan-Newman clustering algorithm. The proposed incremental *k*-betweenness clustering algorithm differentiates from the original Girvan-Newman clustering algorithm in three ways:

1. The edge betweenness values are calculated incrementally using the incremental *k*-betweenness algorithm for edges that handles the shrinking network updates (Chapter 6.5.1.1.5) instead of using the regular Brandes' *k*-betweenness algorithm (Chapter 6.3.1).
2. Instead of using the global version of the edge betweenness that is computed over the entire network topology, its local approximate version (edge *k*-betweenness) is used.
3. Instead of having to remove all the edges in the network until no edges remain, a certain number of edges to be removed can be passed as a

parameter to the algorithm, providing flexibility for early termination condition.

The idea of using the *k*-hop limited, approximate version of the edge betweenness centrality instead of regular edge betweenness is based on the following reasons.

First, both the vertex betweenness and the edge betweenness are metrics that are calculated at the global level, and they are affected very much by any change in the network. In the prior literature, betweenness has been shown to be to a fragile metric [44] that changes easily as the shortest paths in the network change [45]. Thus, a change in the network topology due to the removal of an edge has implications on the local clustering of a cluster, which may actually be a cluster that is far away from the change made on the network. See Figure 9 for an abstract visual description of the idea.

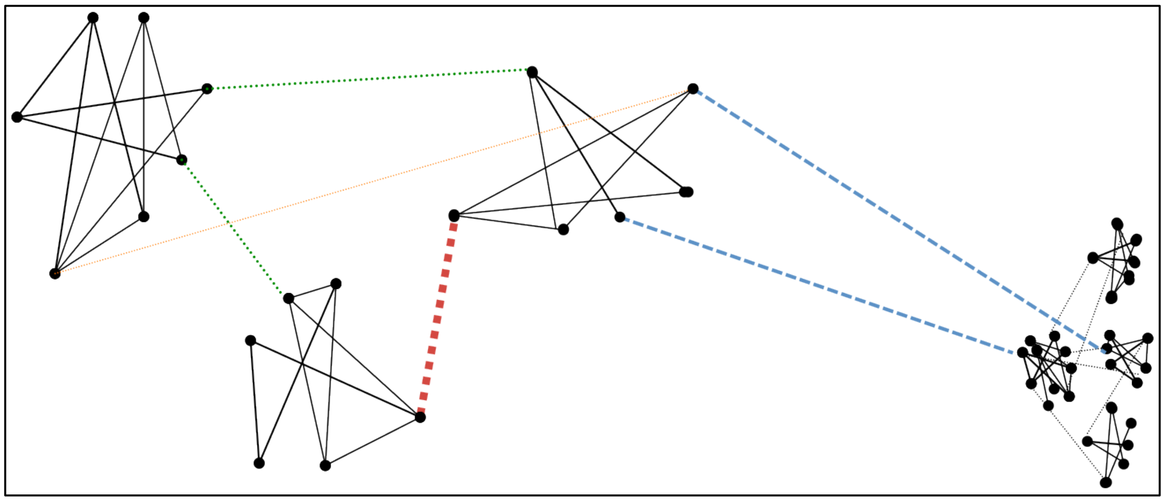
Figure 9 - Far away changes on the network tend to have local implications.

This was also the main motivation argument of the clustering algorithm based on the local flow betweenness proposed in [136]. The results produced by the *k*-hop limited edge betweenness centrality have been successfully validated against real-life organizational datasets in [136].

Second, the memory requirement of the Girvan-Newman clustering is very large. Since the edge betweenness values used for the Girvan-Newman clustering algorithm are maintained per each edge in the network instead of per each node in the network, the memory requirement of the edge betweenness is usually larger than that of the vertex (node) betweenness. Moreover, the Girvan-Newman clustering algorithm generates a dendrogram as an output, which is additionally maintained in the memory as a tree with n nodes, where n is the number of nodes in a network. Since the incremental betweenness algorithms require larger memory than their non-incremental counter parts, this would only exacerbate the memory requirements. Hence, using an approximate version of the betweenness centrality will help reducing the memory requirements while speeding up the computation time as well.

Third, depending on the network structure, removing the edge with the highest edge betweenness might trigger a worst case update where a significant number of shortest paths in the network are updated and all the nodes in the network are inserted into the sets of affected source or affected sink nodes. Such a widespread network update may sweep away the potential benefits of using an incremental algorithm, leaving too little room for speedup while significantly increasing the memory requirements. However, when the changes are forced to remain within a number of hops surrounding each edge, then the network update is prevented from potentially generating a worst-case update, which propagates to the entire network. Hence, using the *k*-hop limited approximation of the edge betweenness centrality for clustering provides a good solution.

The idea of giving the flexibility to remove a certain number of edges from the network stems from the implementation of the Girvan-Newman clustering algorithm in JUNG Java graph library [140]. While providing additional flexibility to stop the iteration of recalculating edge betweenness values at an earlier point, the output produced by the *INCREMENTALBETWEENNESSCLUSTERING* algorithm is exactly the same as the original Girvan-Newman clustering algorithm when the number of edges to be removed is equal to the total number of edges in the network. Removing the edges with the highest edge betweenness centrality values tend to partition the network. Hence, as more edges are removed, the resulting clusters become smaller and more cohesive.

Stopping the iteration of edge removals at an earlier point before all the edges in the network are removed is similar to creating a full dendrogram and cutting it at the level where the iterations have stopped. A dendrogram is usually represented as a tree in the memory with *n* leaf nodes, where *n* is the number of nodes in the network. Since it takes substantial extra memory, it is also possible not to build a dendrogram if it is allowed to stop removing the edges earlier before no edges remain in the network. In such a case, a plausible option for generating output is to run weak component clustering which finds all the weak clusters in a network. A weak cluster is defined as the maximal subgraph where all pairs of nodes in the subgraph are reachable from one another in the underlying undirected subgraph (i.e. regardless of edge orientation).

The *INCREMENTALBETWEENNESSCLUSTERING* algorithm (Algorithm-31) produces the same results as the original Girvan-Newman clustering algorithm if the following three changes are made:

- The call for the *BRANDESEDGEKBETWEENNESS* algorithm on Line–5 of the *INCREMENTALBETWEENNESSCLUSTERING* algorithm is replaced with a call for the *BRANDESEDGEBETWEENNESS* algorithm.
- The call for the *INCREMENTALEDGEKBETWEENNESS* algorithm on Line–7 of the *INCREMENTALBETWEENNESSCLUSTERING* algorithm is replaced with
- The number of edges to be removed is given as |$E(G)$|.

Algorithm–31: *INCREMENTALBETWEENNESSCLUSTERING(G, removeCnt)*
Input: Network $G(V, E)$ and the number of edges to be removed.
Output: Clusters of nodes formed after *removeCnt* many edges are removed.

```
1.  edgeToRemove ← null
2.  maxBtw ← -1
3.  for (i = 0; i < removeCnt; i ++)
4.    if (i = 0)
5.       BRANDESEDGEKBETWEENNESS(G)
6.    else
7.       INCREMENTALEDGEKBETWEENNESS (G, edgeToRemove)
8.    edgeToRemove ← null
9.    maxBtw ← -1
10.   for (Edge e : E(G))
11.      if ($B_E(e)$ > maxBtw)
12.         maxBtw ← $B_E(e)$
13.         edgeToRemove ← e
14. clusterSet ← weakComponentClustering(G)
```

Chapter 8.6 evaluates the performance of the versions of the *INCREMENTALBETWEENNESSCLUSTERING*, and compare it against the original Girvan-Newman clustering algorithm and a modified version of the Girvan-Newman algorithm which uses the *BrandesEdgeKBetweenness*. This way, it is possible to comment on what percentage of the performance improvements comes from the use of approximate edge *k*-betweenness and what percentage comes from the use of incremental algorithm.

6.5.2 Example Use Cases in Wireless Network Analysis

Another use case of the incremental centrality metrics beyond social networks is in physical networks. This section discusses the uses of the incremental betweenness and the incremental closeness centrality computations in network management and resource allocation in wireless mesh networks. A wireless mesh network (WMN) is a multi-hop communication network in which the nodes (routers) are self-organized (i.e., without needing a central coordinator) to form a mesh topology to provide communication over multiple wireless links without requiring an external authority imposing a planned structure. Over the last decade, wireless mesh networking technology has emerged as an important enabling technology to provide better services in wireless networks [3]. A wireless mesh network is composed of mesh routers, gateways, and mesh clients. Gateways are the nodes that have access to the Internet, and the other nodes have to route through multiple hops to get access to the Internet. Mesh clients represent the end users such as mobile devices or laptops.

In our evaluations, we use a real-life wireless mesh network that is provided by the University of California, Santa Barbara (UCSB) MeshNet on-campus WMN deployment. The UCSB MeshNet is a multi-radio 802.11 a/b network consisting of 38 PC-nodes deployed indoors on five floors of a typical office building in the UCSB campus. The data contains 2 sub-networks, each consisting of 19 nodes. In our evaluations, we use one of the 19-node sub-networks.

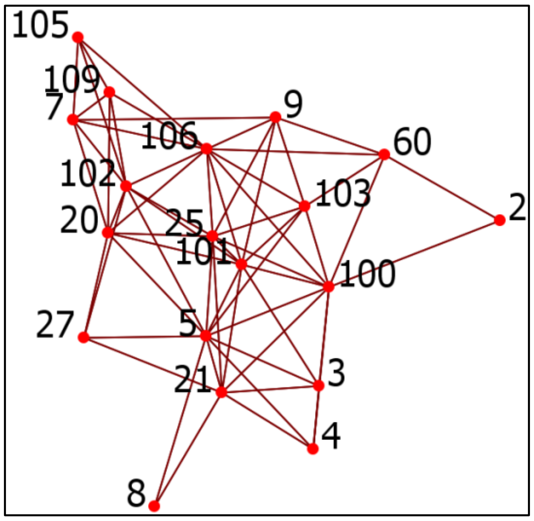

Figure 10 - UCSB Meshnet Topology (19-node subnetwork). All node labels in the figure are preceded with 10.2.1. to form their IP addresses.

Figure 10 depicts the topology of the 19-node sub-network while Table 5 shows the five top-ranked nodes in terms of the degree centrality, the closeness centrality, and the betweenness centrality along with their corresponding normalized centrality values presented in parentheses.

Table 5 - Social centrality rankings of the nodes in UCSB MeshNet.

Degree Centrality	**Closeness Centrality**	**Betweenness Centrality**
10.2.1.5 (0.556)	10.2.1.5 (0.720)	10.2.1.5 (0.176)
10.2.1.106 (0.556)	10.2.1.106 (0.692)	10.2.1.100 (0.153)
10.2.1.100 (0.500)	10.2.1.102 (0.643)	10.2.1.106 (0.146)
10.2.1.102 (0.500)	10.2.1.101 (0.643)	10.2.1.102 (0.107)
10.2.1.101 (0.500)	10.2.1.20 (0.621)	10.2.1.101 (0.086)

6.5.2.1 Wireless Network Management: Vulnerability & Network Security

Social centrality metrics can aid system administrators or automated management systems to better analyze the state of a WMN, and manage it in a more effective manner

[141]. Social centrality metrics can provide answers for questions like: *(i)* Which nodes are more critical from a robustness point of view? *(ii)* Loss of which nodes would have a significant impact on the connectivity of the network? This section investigates the impact of social network analysis on reliability assessment. To discuss how we can use incremental centrality metrics in wireless network analysis and the impact of social centrality metrics on the communication routes (in terms of average number of hops packets travel in the network), we perform coordinated attacks. In other words, we issue shrinking network updates and introduce progressive failures to the central nodes.

Simulated Failure Scenarios: For each centrality metric of interest (e.g. betweenness centrality, closeness centrality, and degree centrality), we incrementally compute the nodes with the highest centrality values by applying shrinking network updates. At each step:

1. Compute the centrality values of the nodes in the network
2. Identify the node with the highest centrality value
3. Remove the node with the highest centrality value
4. Restart from Step-1 to recompute the centrality values of the remaining nodes.

Instead of recomputing centrality values from scratch after each change in the network, we issue the node removals as shrinking network updates in the network, and incrementally compute the centrality values. This makes vulnerability analysis a prefect use case for incremental centrality metrics. For the UCSB Meshnet network, we select up to the first five nodes with the highest centrality.

Technical Details on the Simulations: For each experiment, we simulate the same uniform traffic scenario where every node generates a constant bit rate (CBR) connection to every other node, resulting in $O(n^2)$ connections. All the connections start at the 25th second and end at the 125th second and the CBR rate is fixed at 500 bps for all connections. The simulations last 200 seconds. The amount of total traffic that is generated is kept the same. These experiments follow the methodology in [142], [143].

In the simulations, we used the Optimized Link State Routing protocol (OLSR) as the routing protocol. It is a proactive link state routing protocol where each node stores next-hop destinations for all nodes in the network using the shortest-hop paths. To compute the incremental social centrality metrics, we learn the shortest path related information from the OLSR routing protocol. In the case of node removals, the behavior of OLSR routing protocol is consistent across different runs on the same network topology (e.g., when node x is removed in two different runs, the newly formed shortest paths are consistent across different runs.)

Simulation Results: Figure 11 shows the impact of central nodes' failures on the average number of hops packets traverse to reach their destinations. *Random*, the baseline case, shows the average of 10 experiments where the failing nodes are selected randomly.

Closeness centrality identifies nodes that have rapid access to information by being close to many other nodes on average. On the other hand, betweenness centrality detects nodes that are on the shortest paths for many other nodes, which are usually the nodes that can partition the network. Therefore, in Figure 11, when one or two nodes fail, the impact of betweenness centrality is less than that of closeness centrality because the original topology is relatively well connected and it is not immediately partitionable. However, as the number of failing nodes increases, the residual topologies have longer paths causing the steep increase in the betweenness centrality results.

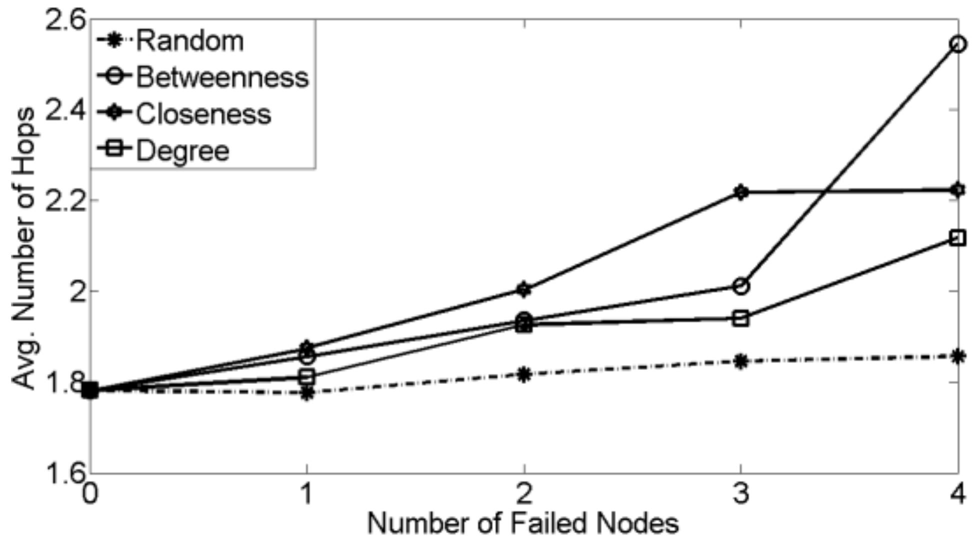

Figure 11 - Increase in average number of hops as top ranked nodes are progressively removed.

Simulations on a larger network: To show that our results are generalizable to larger networks, we perform simulations on a 200-node network as well, and progressively remove up to 40 nodes (20%). The results in Figure 12 justify the relative ranking of centrality metrics to be betweenness, closeness and degree centrality in terms of their importance for network reliability. In other words, the metric that is most effective in degrading network performance in a coordinated attack is betweenness centrality. This behavior is consistent with other work on the literature that assesses the effectiveness of centrality metrics in coordinated attacks [42].

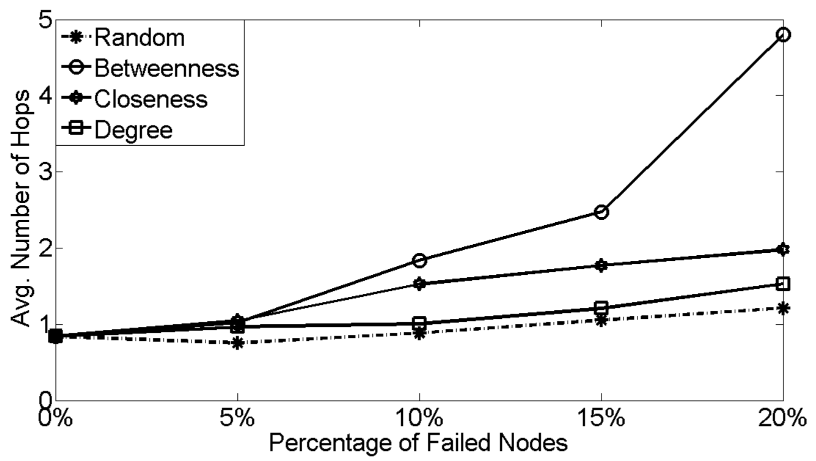

Figure 12 - Increase in average number of hops as top ranked nodes are progressively removed from a 200-node wireless mesh network topology.

6.5.2.2 Resource Allocation: Channel Access Scheduling

Another case study we investigate is the use of dynamic computation of centrality metrics in resource allocation in wireless mesh networks. In particular, we focus on channel access scheduling designed at the MAC layer. The MAC layer protocols that are available in the literature can be broadly classified into two groups: contention based protocols and scheduling based protocols. In contention-based protocols, nodes contend for channel access and collisions are possible. 802.11 MAC protocol [144], which is based on carrier sense multiple access/collision avoidance (CSMA/CA), is one of the most well known examples of contention based MAC protocols. The second group of MAC protocols, the scheduling-based protocols, schedules the access of nodes or links to the channel in advance. TDMA based protocols that operate in discrete slotted time and typically arrange the transmission of the nodes or links in the network based on a schedule constitute examples of scheduling-based protocols [145].

Recent WMN standards such as WiMAX [146] and 802.11s [147] consider Spatial-TDMA (STDMA) based MAC mechanisms and WMNs operate in multi-hop environments. In STDMA based schemes, two nodes that are in non-conflicting parts of the network (i.e. nodes that are located far apart from one another) can be scheduled to transmit simultaneously.

As a part of this case study, we propose an STDMA-based channel access scheduling scheme that uses closeness centrality at its heart. In particular, we investigate how much we can improve the end-to-end throughput at the user level if we use *closeness* centrality to prioritize nodes accessing the wireless medium such that the nodes that are ranked higher in terms of closeness centrality are prioritized over others. We prefer using closeness centrality because, by definition, it is used to describe information propagation efficiency and it is an appropriate metric for optimizing the efficiency of communication networks, including WMNs. In addition, the computation of closeness requires fewer resources compared to other social centrality metrics because Eigenvector centrality has a recursive implementation and betweenness requires additional information on all shortest paths in the network.

Closeness-based Scheduling Scheme: The proposed channel access scheduling algorithm is a lottery based slot assignment where the nodes' closeness values are used as their approximate priorities. The goal is to improve throughput by assigning more slots to more central nodes. Each node generates as many pseudorandom lottery ticket numbers as its closeness value for each time slot. Since the nodes with higher closeness values hold more lottery tickets, their probability for winning the lottery for a particular time slot is higher. To give an example, if a node's closeness value is 10, then it joins the elections with 10 tickets. If another node has closeness value 2, it joins the elections with 2 tickets.

The node that has the highest ticket number for that slot is the winner of the slot and is the node that has the right to transmit and a node with a higher number of tickets has a higher chance of winning. With this kind of scaling, the probability of each node to win a slot will be approximately proportional to its closeness priority. Further details can be found in [3].

Technical Details on the Simulations: The simulations are performed in ns-2.31 using data rates from 650 bits/sec up to 1350 bits/sec. We measure end-to-end delay and end-to-end throughput calculated across all data packets (excluding control packets) generated during the simulations. WMNs have fairly static topologies. Therefore, only the static topologies are simulated. At the routing layer, the Optimized Link State Routing protocol (OLSR) is used as the routing protocol. For the incremental closeness calculations performed in real time, periodical OLSR HELLO packets are broadcast every 2ms. Hence, node mobility and topology changes are accounted for in the closeness centrality computations.

Simulation Results: Next, the performance of the proposed socially aware scheduling scheme is evaluated. For the baseline scheme, a similar framework is used where only the prioritization scheme is modified. In the baseline case, each node generates a random number of lottery tickets, rather than closeness many tickets. In other words, each node has an equal chance of being the winning node that earns transmission right during each time slot. At each time slot the winning node is selected randomly following no particular prioritization. In other words, in the baseline case, nodes generate random weights instead of using their closeness centrality values as their weights.

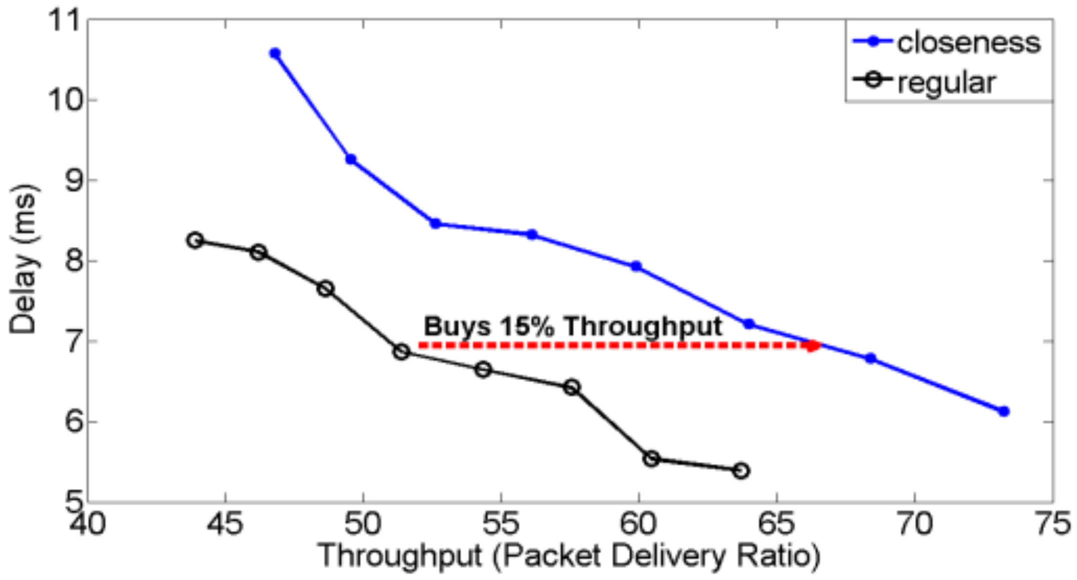

Figure 13 - Delay vs. Throughput (When closeness centrality is used in prioritization, the throughput increases by 15%).

Figure 13 presents the performance results on the tradeoff between delay and throughput, justifying the throughput advantage brought by the use of social centrality metrics. According to the results presented in Figure 13, the packet delivery ratio increases by about 15% for the same amount of delay incurred when closeness centrality is used in prioritization of the packets.

CHAPTER 7 DATASETS

This chapter describes the datasets used for performance evaluation in this dissertation. Both synthetic and real-life networks of different sizes, topologies, and average degrees have been used. Chapter 7.1 describes the used synthetic networks while Chapter 7.2 describes the real life networks.

7.1 SYNTHETIC NETWORKS

This section describes the synthetic networks used in the performance evaluations. There two main goals of using synthetic networks in performance evaluations. First, using a set of networks with controlled topologies enables us to evaluate the impact of the network topology on the performance benefits that can be obtained using the incremental algorithms proposed in this dissertation. Second, by changing the network size and average degree, it is possible to evaluate how the performance benefits of the incremental algorithm scale with the increasing network size and average node degree.

To evaluate the impact of network topology, synthetic networks with three different topologies are used: preferential attachment networks [10], Erdos-Renyi networks [11], and small-world networks [12].

7.1.1 Preferential Attachment Networks

Barabasi and Albert have proposed the preferential attachment graph generation model for generating graphs with heavy-tailed degree distributions, which is observed in many real life networks including the Internet, World Wide Web (WWW), and many social networks [10]. The preferential attachment graph generation model has two phases: growth and preferential attachment [148]. In the growth phase, starting with a small number of nodes (m_o starting nodes), new nodes are added one at a time, with m edges ($m < m_o$) to link the newly added node to the nodes that are already in the network. In the preferential attachment phase, when the incoming nodes are about to choose the nodes to connect to, the current degrees of already existing nodes are taken into account. Nodes that have a higher number of existing connections are more likely to receive more/new connections. In sociology, this phenomenon is called the Matthew effect (or accumulated advantage), which is based on a phrase from Bible and originally used to explain the issues with fame, status, and advantages of economic capital [149]. This phenomenon is also known as 'rich get richer'. Numerical studies indicated that networks generated with this model evolve into a scale-free (scale invariant) state with power-law degree distribution [148]. In such networks, several nodes have very little number of connections (low degrees) while very few nodes act as major hubs with too many connections, resulting in power-law degree distributions.

7.1.2 Erdos-Renyi Networks

In their original 1959 article, Erdos and Renyi describe a random network with *m* edges and *n* nodes where a fixed number of edges are drawn uniformly at random from $n(n - 1) / 2$ possible edges [11]. A similar approach for the generation of random graphs was also discussed by Edgar Gilbert in 1959 where each edge is created *independently* with probability *p*, where $0 < p < 1$ [150]. In other words, *p* represents the fraction of edges that appear in a network out of all possible $n(n - 1) / 2$ edges. Since all edges (relationships) form independently from the others, the networks (graphs) generated with Erdos-Renyi or Gilbert model do not usually match well with the topologies of real-life networks. In real life networks, there are additional interaction mechanisms such as transitivity or introductions that influence the formation of new connections, causing real life networks' topologies to be very different from random graphs. However, due to their ease of analysis, Erdos-Renyi networks are usually included in network studies as a standard way of making comparisons with the state of the art literature.

7.1.3 Small World Networks

In 1998, Watts and Strogatz introduced a model that starts with a regular lattice (ring) of *n* nodes and then modified by rewiring [12]. Small world networks model communities where most nodes are not neighbors of one another and most nodes are confined to their own neighborhoods, however, due to existence of random connection formed via rewiring, most nodes can reach one another in a small number of hops. In the generation process of small world networks, initially, there is a ring of *n* nodes and each node is connected to its $2r$ nearest neighbors. This regular ring is modified by rewiring each edge randomly and independently with a rewiring probability p, where $0 < p < 1$. The topology changes with the increased rewiring probability p, which reflects as increased randomness and reduction in the average distance (i.e. characteristic path length).

7.1.3.1 Selecting a rewiring probability (*p*) value

Small world networks are discussed to model complex real world networks nicely, and there exist studies that study the extent different networks exhibit small world network characteristics. In the paper [151], the authors design a metric called 'small-world-ness', a score that determines how small-world like a network is and assigns scores to them accordingly. The authors also examine 30 different real life networks and found that 27 of them exhibited small-world network properties to varying degrees. Another approach to determine the appropriateness of small-world network model for the real life networks is to fit the small world network model to the real data and extract the parameter values from the model. In [152], the authors study 1000-node neuron networks from brain and the impact of the structure on memory/recall and observe that when the rewiring parameter $p = 0.3$, the performance of the network start converging and the optimal value is obtained when $p = 0.4$.

In [153], the author examines the ethnocentrism phenomenon which refers to the tendency to behave differently towards strangers based only on whether they belong to the in-group or the out-group and how the social agents perform in social networks (i.e.

how much they cooperate or interact with their neighbors). The authors find that the optimal settings are observed in small neighborhoods with few random connections ($p < 0.3$), and when the rewiring probability $p > 0.3$, the networks are greatly reconnected and the social agents perform worse than they perform in regular lattice networks.

Another study, [154], studies the spread of rumor in social networks on an underlying small world structure using three p different values: $p = 0.05$, $p = 0.19$, and $p = 0.3$. The experiments with rumor spread suggest that when the p value is too small ($p = 0.05$), the rumor dies too soon, and when the $p = 0.3$, the behavior of the rumor spread depends on whether or not a shortcut is reached. The authors suggest that they obtain their optimal results when $p = 0.19$.

Another paper from the field of economic research ([155]) studies the self-organizing, Ising-like model of financial markets with underlying small world networks of interacting agents. The authors find that using the rewiring probability $p = 0.3$ can reproduce main stylized facts including price dynamics, evolution of returns; producing the most accurate results. Considering all the examples mentioned above from different fields, the general inclination of the authors is that smaller p values ($p < 0.4$) usually work well in different contexts, especially when $p \leq 0.3$. In our experiments, we use $p = 0.3$.

7.1.4 Directed Cycles

Directed cycles are formed as a variant of small-world networks using the same parameters and the network generation algorithm. The only difference between directed cycles and small world networks is that in directed cycles, the edges are directed while in small world networks the edges are undirected or bidirectional. Therefore, when the edges are directed in one direction only, the resulting network is a big ring of nodes, with random bounds in it. Such networks have been studied in the literature in different contexts before and they appear under various names including 'directed small world networks', 'asymmetric small world networks', 'rings with random bounds', and 'directed cycles' [156] [157] [158] [159]. In the rest of this dissertation, we refer to these networks as 'directed cycles'. Since such networks have directed edges, they tend to form very long shortest paths for the node pairs that cannot benefit from the existence of random shortcuts. Therefore, in such networks the diameter of the network can get very large as we see later in this section, which can form an example for a pathological case for the incremental centrality algorithm forcing them to update shortest paths that are too long. To be able to provide direct comparisons between small world networks and directed cycles, we again use $p = 0.3$ for generating directed cycles.

7.1.5 Synthetic Network Set

In order to understand how the performance benefits of the incremental centrality algorithms scale with the increasing network size and average node degree, two sets of synthetic networks from the abovementioned network topologies are used. 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.

First, for each synthetic network topology, the number of nodes is varied from 1000 to 5000 with a step size of 2000, and the average degree is fixed to 6. For small world networks, the rewiring probability is 0.5. Table 6 lists basic statistical information about the synthetic networks generated with different topology generation models and network sizes.

In the second set of synthetic networks, for each synthetic network topology, the number of nodes is fixed to 3000, and the average number of nodes is varied from 4 to 8 with a step size of 2. Table 7 lists basic statistical information for the networks with varied average number of nodes, similar to the information presented in Table 6.

Table 6 - Network statistics for varying network sizes. There are additional metrics that are measured but not listed. For all the networks in this table minimum betweenness is 0, network fragmentation is 0, and the average degree is 6.

Topology	**Size**	**Max Btw**	**Avg. Btw**	**Std. Dev. Btw**	**Min Deg**	**Max Deg**	**Std.Dev Deg**	**Diameter**	**Char. Path Length**	**Clustering Coefficient**
Pref. Attach.	1000	1953.97	94.37	177.47	3	89	6.822	10	3.45	0.014
Pref. Attach.	3000	5183.26	197.59	434.83	3	233	8.064	14	4.126	0.007
Pref. Attach.	5000	12987.22	292.48	749.01	3	212	8.251	16	4.442	0.005
Erdos-Renyi	1000	25429.36	4777.28	4249.81	1	14	2.498	15	6.305	0.003
Erdos-Renyi	3000	76713.80	18136.74	10087.35	2	13	1.572	14	7.086	0.001
Erdos-Renyi	5000	108061.53	32073.39	16062.96	2	11	1.362	14	7.492	0.001
Small World	1000	7376.179	1809.59	1184.202	3	11	1.292	8	4.623	0.212
Small World	3000	31788.719	6617.851	4378.002	3	11	1.271	9	5.413	0.208
Small World	5000	71757.523	12079.138	7844.011	3	12	1.237	9	5.833	0.216
Directed Cycles	1000	13112.60	3544.12	2379.46	3	11	1.292	38	8.38	0.106
Directed Cycles	3000	75111.21	14986.28	10954.08	3	11	1.271	57	11.37	0.104
Directed Cycles	5000	145539.10	28762.20	21557.20	3	12	1.237	77	12.98	0.108

According to the topological statistics presented in Table 6, in preferential networks, the maximum number of connections nodes can attain (Max Deg.) is substantially higher in preferential attachment networks; resulting in high deviation of node degrees (Std. Dev. Degree). In addition since there are few nodes with very high number of connections, a substantial portion of the nodes in preferential attachment networks are directly connected to such nodes. Hence, the shortest paths in such networks are relatively shorter when compared to the networks that are generated using the other graph generation models. The networks generated by the Barabasi-Albert graph generation model have systematically lower average path lengths than a random network. The average path length for Barabasi-Albert preferential attachment networks increases approximately logarithmically with the increasing network size, following $Avg.Path\ Length \sim \frac{\ln|N(G)|}{\ln\ln|N(G)|}$.

Table 7 - Network statistics for varying average node degrees. There are additional metrics that are measured but not listed. For all the networks in this table minimum betweenness is 0, network fragmentation is 0, and the number of nodes is fixed to 3000 nodes.

Topology	Avg Deg	Max Btw	Avg. Btw	Std.Dev. Btw	Min Deg	Max Deg	Std.Dev Deg	Diameter	Char. Path Length	Clustering Coefficient
Pref. Attach.	4	5673.02	76.60	270.60	2	141	5.841	12	3.89	0.0058
Pref. Attach.	6	5183.26	197.59	434.83	3	233	8.064	14	4.126	0.007
Pref. Attach.	8	11264.55	414.08	855.95	4	238	10.388	15	4.269	0.009
Erdos-Renyi	4	247466.93	18077.70	24650.83	1	11	1.904	30	10.622	0.0009
Erdos-Renyi	6	76713.80	18136.74	10087.35	2	13	1.572	14	7.086	0.0010
Erdos-Renyi	8	77281.40	14015.49	10985.36	1	20	2.879	14	5.907	0.0012
Small World	4	98401.898	9477.128	6988.962	2	11	1.002	13	7.320	0.179
Small World	6	31788.719	6617.851	4378.002	3	11	1.271	9	5.413	0.208
Small World	8	25648.943	5492.811	3194.403	4	15	1.445	7	4.663	0.231
Directed Cycles	4	119074.50	19577.90	19271.70	2	11	1.002	102	20.26	0.090
Directed Cycles	6	75111.21	14986.28	10954.08	3	11	1.271	57	11.37	0.104
Directed Cycles	8	50415.84	10557.80	6886.33	4	15	1.445	39	8.10	0.115

On the other hand, small world networks have higher local clustering (e.g. clustering coefficient) and have relatively small shortest path lengths while the directed cycles have very long paths in the network (diameter and characteristic path length) to reach to far nodes when they are unable to utilize a random shortcut in the network.

Considering the information presented in Table 7, several observations are in order. As the average node degree increases, the characteristic path length (average path length) increases in the preferential attachment network while it decreases for the Erdos-Renyi networks, Small-World networks, and the directed cycles. This is primarily because how different network topologies tend to be structured. For instance, for the small world networks and the directed cycles, the increased number of connections increases the immediate neighborhood sizes for all the nodes in the network as well as the number of rewired shortcut edges. Both of these factors contribute to the decreased shortest path length. For preferential attachment networks, the differences in the characteristic path lengths are not too big, as preferential attachment networks tend to have relatively small shortest path lengths. In addition, in all network types studied above, the clustering coefficient increases along with the increasing average node degree.

Table 8 - Network statistics for additional 1000-node directed cycles, varying rewiring probability *p* from 0.2 to 1.00 with a step size of 0.2 (Avg. degree = 6).

p	**Max Btw**	**Avg. Btw**	**Std. Dev. Btw**	**Min Deg**	**Max Deg**	**Std. Dev. Deg**	**Diameter**	**Char. Path Length**	**Clustering Coefficient**
0.2	34020.93	4305.77	3104.97	3	9	1.028	35	9.711	0.154
0.4	15036.88	3183.39	2395.88	3	12	1.423	30	8.017	0.071
0.6	14763.34	2268.86	2463.76	3	11	1.575	31	7.789	0.024
0.8	6779.73	833.58	1161.97	3	12	1.715	22	6.444	0.005
1.0	1026.5	100.79	144.32	3	13	1.765	12	3.862	0.003

Apart from these two sets of synthetic networks, further investigation is performed on the directed cycles as they are well known to have varying topological characteristics, depending on the chosen rewiring probability, p. A sweep of p values covering the range of 0.2 – 1.0 with a step size of 0.2 is performed on 1000-node small world networks and directed cycles, with an average degree of 6.

Table 8 lists the topological features and basic network statistics for 1000-node directed cycles. As the rewiring probability increases, the network becomes more random. As a result, the network diameter and the average path length are reduced along with the clustering coefficient. In addition, the deviation in the distribution of node degrees increases as the nodes' connections become more random, starting to lose the initial lattice structure that exists at their initialization phases.

7.2 REAL-LIFE NETWORKS

Apart from the synthetic networks, the performances of the proposed incremental centrality algorithms are also evaluated with a number of real-life networks that grow over time and are of different scales and topologies.

The real-life networks used in the evaluations are prepared as weighted networks where the cost of an edge is inversely proportional to the strength of the relationship it models. That is, multiple updates for the same pair of nodes are consolidated in a single edge. For instance, if communications/interactions from node x to node y have been recorded twice up to a certain point, then the edge $\{x \rightarrow y\}$ has the cost of 1/2. When a third update from node x to node y is recorded, then the cost of the edge $\{x \rightarrow y\}$ is updated to be 1/3. Next, the real-life datasets that are used in performance evaluations are described in detail.

7.2.1 SocioPatterns Dataset

SocioPatterns dataset was collected during the ACM Hypertext 2009 conference, where the SocioPatterns project deployed the Live Social Semantics application [8]. During the conference, the participants volunteered to wear Radio Frequency Identification (RFID) embedded badges. The RFID badges had coverage of 1 – 1.5 meters, and human bodies acted as RF shields, which allows identification of primarily face-to-face contacts. The power and frequency of the RFID badges were adjusted such that face-to-face communication can be captured with more than 99% reliability. For about 2.5 days, communication between RFID badges is recorded every 20 seconds based on the RFID packet exchanges observed among any two participants' badges.

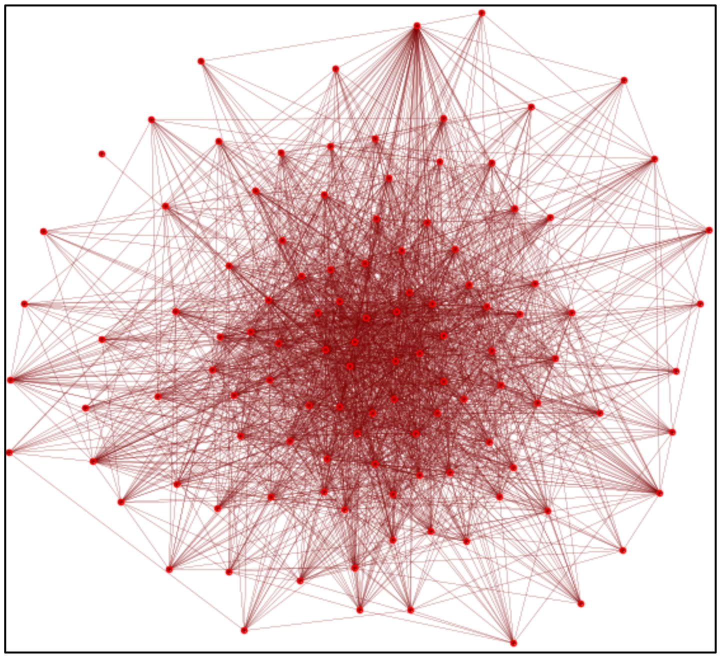
Figure 14 - SocioPatterns network topology.

The resultant social network is a bi-directional, weighted, dynamic contact network of 113 conference attendees with 20,818 dynamic communication updates. This dataset is further explored in [160]. Figure 14 shows the topology of the SocioPatterns social network.

7.2.2 OnlineForum: Facebook-like Social Network

This dataset was collected from an online community for students at University of California, Irvine [14] [13]. This dataset contains 59,835 time stamped online messages sent among 1,899 users. The information on the sender, the receiver, and the length (how many characters) was recorded for each message. After self-loops and the messages with zero length are removed, there were 58,827 messages left in the dataset. The final form of the network is a directed, weighted, dynamic social network and its topology is depicted in Figure 15.

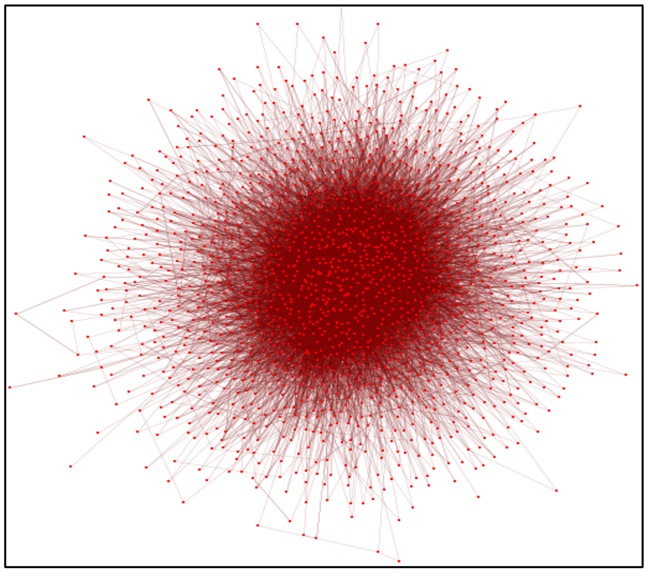

Figure 15 - OnlineForum (Facebook-like Social Network) network topology.

7.2.3 HEP Co-authorship Network

HEP co-authorship dataset is the co-authorship network that has been derived from the High Energy Physics Theory dataset (HEP-TH), which is a publicly available dataset, compiled by arXiv [161] for the KDD Cup'03 competition [162]. The authorship network contains 21,917 edges among 7,508 researchers who are the authors of 29,555 high-energy physics papers that were added to the arXiv online library from 1992 to 2003. A detailed exploration of this dataset has been performed in [15]. Figure 16 shows the final topology of the network, while Figure 17 shows an earlier snapshot of the network when the network was still in growth phase. Out of 7508 nodes, 5786 of them are in the core component of the network. Since transitivity and social networking is very important for the formation of coauthorship relationships, there is a significant level of triangulation observed, which is more visible in the earlier version of the network drawn in Figure 17.

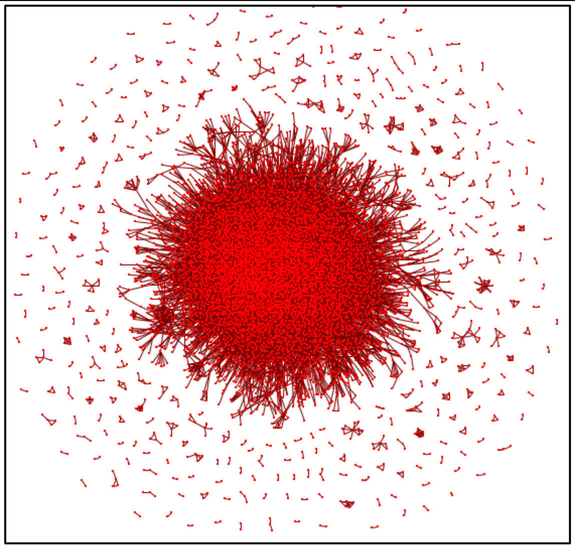
Figure 16 - HEP Coauthorship network topology.

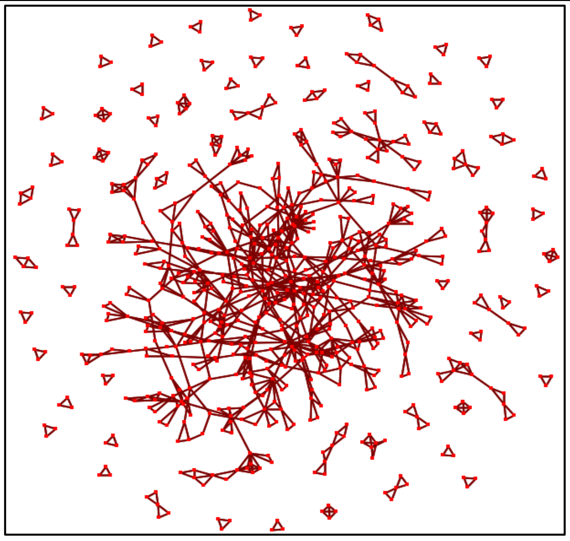
Figure 17 - An earlier version of the HEP Coauthorship network topology.

7.2.4 P2P Communication Network

P2P Communication network is a peer-to-peer file sharing dataset collected on a server running the OpenNap [163] file sharing system. The P2P communication network was collected on two campus networks: the University of Brescia (UNIBS) and the Politecnico di Torino (POLITO). The traces contain the Internet usage data from informed participants who agreed to continue their normal Internet usage during the course of the experiment [17].

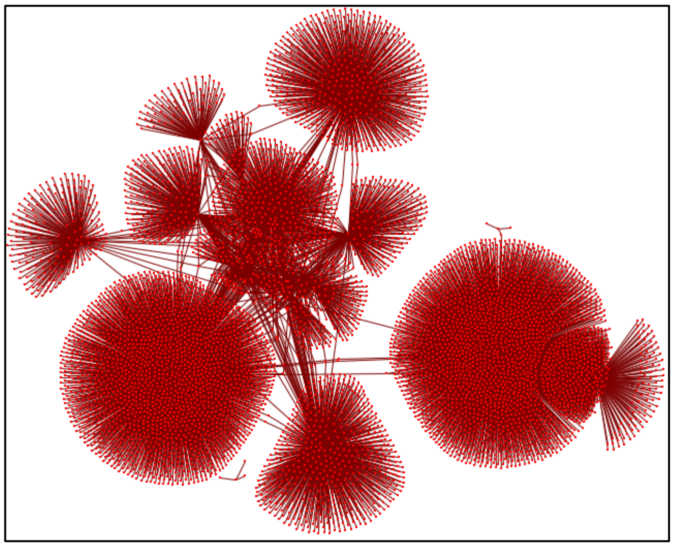

Figure 18 - P2P Communication network topology.

Out of this dataset, I extracted P2P traffic, generated by several different file sharing and communication based P2P applications including amule, bittorrent, edonkey, and Skype. I derive P2P communications and file transmissions and model them as a network where nodes represent the users and links represent the communication/transmission between those two users. The resultant network contains 6843 nodes, and 7572 edges and its topology is presented in Figure 18.

7.2.5 Retweet Network on Iran Sanction News

Iran Twitter dataset is compiled by G.R. Boynton from Iowa State University, and consists of tweets posted online following two different pieces of news on Washington Post [16]:

- WASHINGTON -- Secretary of State Hillary Rodham Clinton says the United States and its partners seeking new sanctions against Iran have come up with a draft proposal for a new round of penalties. (10:50 AM EDT Tuesday, May 18, 2010)
- The U.N. Security Council voted 12 to 2 to impose a fourth round of sanctions on Iran, bringing to a close months of diplomatic efforts by the Obama administration to penalize Tehran for building a covert nuclear facility and accelerating its enrichment of uranium. (11:32 AM EDT Wednesday, June 9, 2010).

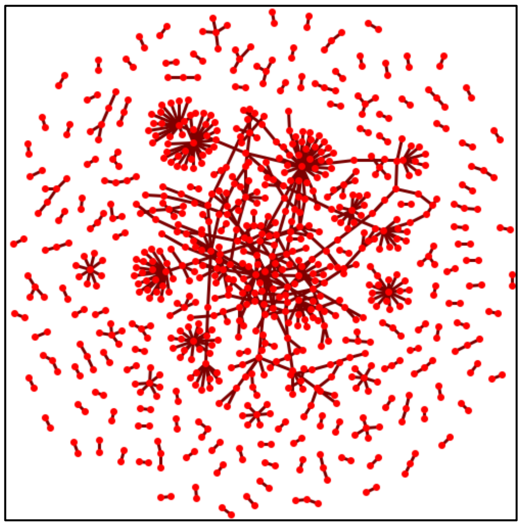

Figure 19 - Iran retweet network topology.

The Twitter stream covers dates between May 18, 2010 and June 15, 2010. There are 4546 tweets posted online on this topic and 707 of them are retweets using the RT feature provided by Twitter. The retweet network extracted from this dataset contains 809 nodes and 665 links, and it is a very sparse network (See Figure 19).

7.2.6 Real-Life Network Datasets

Table 9 shows basic topological features and network statistics of the real-life datasets used in performance evaluations. As shown in Table 9, these networks have substantially different topological features and sizes. For instance, P2P network has almost no clustering and the distribution for the degrees of nodes is highly right skewed (max = 2185, min = 1, average = 2.21). The average number of connections nodes have is so small despite the large maximum value, which means that there is only a handful of nodes that are connected to a good portion of nodes, while the majority of the nodes are have a couple of connections.

Table 9 - Topological features of real-life networks. 'D? U?' column displays information on the directionality of edges in the networks. D represents directed networks while U represents undirected (bidirectional) networks.

Network	D? U?	Nodes	Edges	Avg Deg	Min Deg	Max Deg	Std. Dev. Deg	Diameter	Char. Path Length	Clust. Coeff
SocioPatterns	U	113	4392	38.87	1	98	18.35	3	1.6562	0.534
OnlineForum	D	1897	20290	21.40	3	339	35.61	8	3.1966	0.085
HEP Coauthorship	U	7507	38804	5.16	3	64	6.147	15	5.7426	0.459
P2P	D	6843	7572	2.21	1	2185	38.33	3	1.2481	0
Twitter (Iran)	D	809	665	1.644	1	39	2.349	5	1.404	0.016

On the other hand, although HEP Co-authorship and P2P Communication networks are comparable in terms of the number of nodes they have, HEP Co-authorship network exhibits more of the expected characteristics of social networks. For instance, it has higher local clustering and the phenomenon of six-degrees of separation is also observed. Another dataset whose topology is highly influenced by the social networking concepts is SocioPatterns networks, where there is again high clustering and transitivity in the connections people make. In addition, in terms of network size (number of nodes) this set of networks provide a good sample covering both the small networks a lot of social analysts are interested in and the larger networks that have become popular more recently due to availability of online data collections.

CHAPTER 8 EXPERIMENTS AND RESULTS

This chapter first describes the coding environment used for the implementation of the algorithms proposed in this thesis. Then, the performance benefits of the proposed incremental centrality algorithms obtained on the synthetic and real-life networks are reported.

8.1 IMPLEMENTATION ENVIRONMENT

As the implementation environment, I have extended the GraphStream [164], which is an open source, dynamic graph library written in Java. GraphStream provides a framework to handle the evolution of graphs, and provides support for the addition, removal, and modification of nodes and edges [164] [165]. The performance results are collected on a machine with 3.20Ghz Intel Xeon CPU and 256 GB RAM.

8.2 EXPERIMENT DESIGN

To measure the performance of the incremental centrality algorithms for handling growing network updates, 100 edges are randomly selected from each network and the initial computation of the centrality values is done on the incomplete version of the network that has all but 100 edges. Then, each of these 100 edges is inserted incrementally. The average performance in terms of execution time over the repeated invocations of traditional baseline algorithm (e.g. the Dijkstra's algorithm for the closeness centrality and the Brandes' algorithm for the betweenness centrality) is calculated by averaging the execution time of each incremental update. For instance, if the incremental centrality algorithm takes 5 seconds on average to complete a set of updates and it takes 30 seconds on average for the non-incremental algorithm to complete the same set of updates, we conclude that the incremental algorithm is 6x faster than the corresponding non-incremental baseline algorithm on average.

Similar experiments have been designed for measuring the performance of the proposed incremental centrality algorithms for shrinking network updates as well. In the experiments performed for shrinking network updates, we start with the complete version of the network and incrementally remove the same set of edges used in the experiments for growing network updates. Therefore, we have a way of comparing how different types of network updates affect the performance. The performance results reported in the next section (Chapter 8.3) are obtained on networks with different topologies (e.g. preferential attachment networks [10], Erdos-Renyi networks [166], small-world networks [167], and directed cycles) and different network sizes (e.g. 1000, 3000, and 5000) and average node degrees (e.g. 4, 6, and 8). Herein, the network size is represented in terms of the number of nodes in a network. For small world networks and directed cycles, the rewiring probability is 0.3. However, through a set of additional experiments presented in Chapter 8.3.6, the impacts of the rewiring probability on the network topology and the performance improvements achieved by the incremental centrality algorithms are discussed.

Aside 1 – Adding random edges in a network may result in losing the scale-free properties of a preferential attachment network. While it is unlikely that adding or removing 100 edges will impact the network topology, for the smallest networks the random addition of edges may move the network somewhat away from scale free slightly and this might dilute the impact of the scale free networks. Considering this potential side effect, instead of adding or removing 100 random edges, we start with a scale free network and then remove 100 edges randomly selected out of the entire network topology. Then, these edges are gradually added back in. Thus, the network becomes closer to a pure scale free form.

Next, using the 1000-node preferential attachment network (avg. deg = 6) as an example, we check the distribution of node degrees in the two versions of the network when it is with and without those 100 edges randomly selected out of the entire network topology to observe if they still exhibit the scale free property. Figure 20 - Figure 23 present the distribution of edges before and after the removal of the 100 edges randomly selected out of the network topology. As it can be observed by comparing Figure 20 with Figure 22 and Figure 21 with Figure 23, removal of 100 edges does not disturb the distribution of node degrees.

Mathematically, the distribution of a random variable x obeys a power law if its probability distribution satisfies $p(x) \propto x^{-\alpha}$ where α is the characterizing *scaling exponent* which typically lies in the range of $2 < \alpha < 3$ for power law data. More precisely, when the data is discrete, which is the case in our dataset, $p(x) = \Pr(X = x) = Cx^{-\alpha}$ where C is a constant. Various studies have shown that if an empirical dataset follows a power-law distribution, it usually only does so for values of x, where $x > x_{min}$ [168]. In the cases where x_{min} is not known in advance, accurate estimation of x_{min}

is very important for estimating α accurately. If the value chosen for x_{min} is too low, then we would try to fit power laws to a part of the dataset which does not necessarily follow power laws. Similarly, if the value chosen for x_{min} is too high, then we are effectively reducing the size of the dataset, making it prone to statistical errors [169].

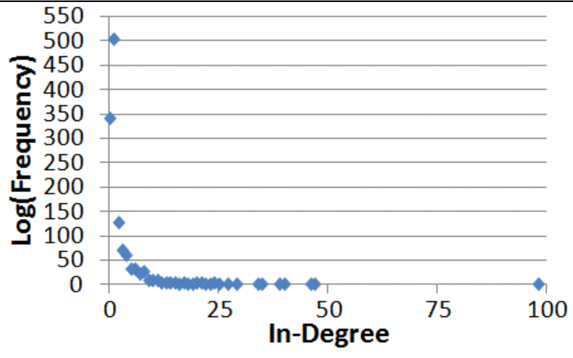

Figure 20 - In-degree distribution of the 1000-node preferential attachment network with 100 of its edges missing.

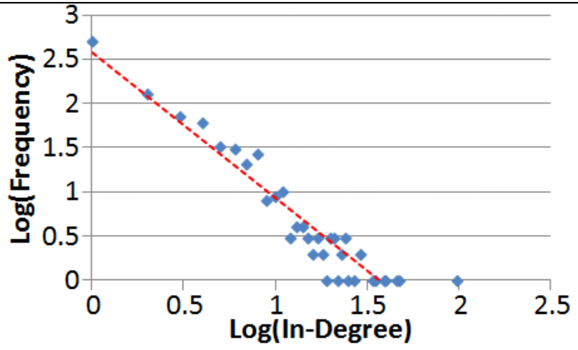

Figure 21 - Log-log scale of in-degree distribution of the 1000-node preferential attachment network with 100 of its edges missing. (Same data as in Figure 20).

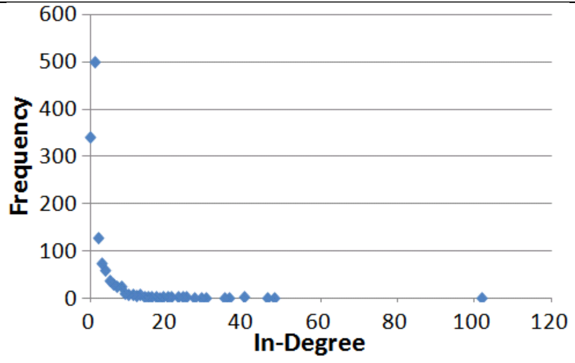

Figure 22 - In-degree distribution of the 1000-node preferential attachment network (Complete topology, avg. degree = 6).

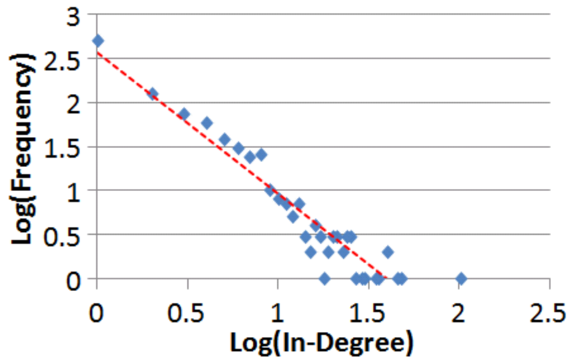

Figure 23 - Log-log scale of in degree distribution of the 1000-node preferential attachment network (Complete topology, average degree = 6). Same data as in Figure 23.

Detection and Characterization of Power Laws

Estimating $\boldsymbol{x_{min}}$**:** To estimate the lower bound of a power law distribution, we use the method proposed by Clauset et al. in [170]. The basic idea is to choose the value of x_{min} such that the maximum absolute distance between the CDF functions of the original data and the pruned data (which contains only $x > x_{min}$) is minimized. The goal is to make the distribution of the original dataset and the best fitted power law as similar as possible.

Estimating $\boldsymbol{\alpha}$**:** For estimating the characterizing scaling exponent, we use the method described in [168], which essentially describes a maximum likelihood estimator that is equivalent to a discrete version of the Hill estimator [171]. Mathematically, the estimated α, $\tilde{\alpha}$ is calculated as $\widetilde{\alpha} = 1 + n\left[\sum_{i=1}^{n} \ln \frac{x_i}{x_{min}}\right]^{-1}$.

Table 10 - Parameters for the power law fit for the degree distribution of the smallest (1000-node) preferential attachment network.

	$\boldsymbol{\alpha}$	$\boldsymbol{x_{min}}$	$\boldsymbol{L}$
Incomplete Topology	2.44	10	1.0406e+04
Complete Topology	2.44	10	1.0468e+04

When Clauset's power-law fitting method is applied both on the complete network topology and the version of the network with 100 missing edges, the following x_{min}, α, and L (log-likelihood of $x > x_{min}$) values are obtained as presented in Table 10. Since α and x_{min} values are unaffected, especially the α value, we conclude that experiments with the removal and insertion of 100 edges randomly selected out of the entire network topology do not cause the preferential attachment networks to lose their topological properties.

Aside 2 – It is possible to run the growing and shrinking network updates in combination, in any order. The reason to show the performance improvements separately in the following tables is to provide opportunities for comparison of the performances of the incremental centrality algorithms for the shrinking and growing network updates.

8.3 PERFORMANCE RESULTS WITH SYNTHETIC NETWORKS

This section reports the performance of the incremental centrality algorithms obtained on the synthetic networks and discusses the performance results in line with the topological features of the synthetic networks. In the rest of this section, the synthetic networks that are described in Chapter 7.1 are used for performance measurements.

8.3.1 Incremental Closeness Centrality Performance Results

The performance values reported in Table 11 describe the speedup obtained by the incremental closeness algorithm over the repeated invocations of the Dijkstra's algorithm used for computing closeness centrality, averaged across 100 incremental updates on the network as described in Chapter 8.2. The standard deviations for the performance improvements are added in parenthesis for each corresponding average performance value. Table 11 also shows the percentage of the total number of nodes that are affected by the growing and the shrinking network update types. The total number of affected nodes is calculated as the size of the set formed as the combination of AffectedSinks and AffectedSources nodes.

Table 11 - Performance improvements of the incremental closeness algorithm over the repeated invocations of computing closeness centrality using the Dijkstra's algorithm obtained on networks with different topologies/sizes described in Chapter 7.1. Average node degree is 6.

Topology	**Growing Network Updates**		**Shrinking Network Updates**	
	Closeness Speedup	**Closeness Affected %**	**Closeness Speedup**	**Closeness Affected %**
Preferential Attachment - 1000	**900 x** (732.53)	3.79	**467.70 x** (905.96)	2.93
Preferential Attachment - 3000	**14714.43 x** (7298.64)	2.13	**3447.89 x** (12233.78)	1.51
Preferential Attachment - 5000	**47738.81 x** (27076.62)	1.33	**10150.29 x** (48386.73)	0.97
Erdos-Renyi - 1000	**123.07 x** (1146.17)	60.86	**58.37 x** (1018.59)	43.71
Erdos-Renyi - 3000	**515.35 x** (485.18)	69.02	**169.83 x** (148.08)	41.31
Erdos-Renyi - 5000	**890.56 x** (1158.4)	70.66	**304.44 x** (389.26)	41.21
Small World - 1000	**233.49 x** (5671.12)	50.10	**118.71 x** (3838.12)	17.51
Small World - 3000	**642.45 x** (80695.62)	52.39	**338.81 x** (27515.92)	18.5
Small World - 5000	**2015.87 x** (145931.31)	50.69	**822.01 x** (91968.11)	17.97
Directed Cycles - 1000	**309.62 x** (1221.32)	26.27	**106.27 x** (1207.63)	20.82
Directed Cycles - 3000	**740.44 x** (15275.67)	26.17	**243.70 x** (12120.51)	21.47
Directed Cycles - 5000	**1822.83 x** (54345.94)	26.54	**495.39 x** (38131.80)	22.17

Consider the results presented in Table 11. First, the performance improvements obtained over the baseline are higher for the growing network updates than for the shrinking network updates. However, in general, the number of affected nodes is lower for the shrinking network updates. The *DELETEUPDATECLOSENESS* algorithm maintains a priority queue while the *INSERTUPDATECLOSENESS* algorithm does not and the *DELETEUPDATECLOSENESS* algorithm is significantly more complicated than the *INSERTUPDATECLOSENESS* algorithm. Therefore, the overall algorithmic complexity and the actual execution time are higher for shrinking network updates.

Second, the incremental closeness algorithm, both for the growing and the shrinking network updates, performs best with the preferential attachment networks. It is also observed that other parameters shown in Table 6 such as network diameter and characteristic path length are inversely related with the performance obtained. For instance, in preferential attachment networks, characteristic path length, diameter, and the clustering coefficient are lower compared to other topologies. When the shortest paths in a directed network are short (i.e. contain a number of hops) and there is not much redundancy, 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).

On the other hand, the average length of the shortest paths in small world networks is not high either. However, the small world networks have undirected (bidirectional) edges. Thus, the number of shortest paths between any two nodes is likely to be higher. Hence, there is a large difference in the portion of the network affected by the growing and the shrinking network updates. Consider the following example. There are three shortest paths from node *a* to node *b*. Then, an edge on one of these shortest paths is removed. Although one of these shortest paths is not a shortest path from node a to b anymore, there are still two more paths that would serve as the shortest paths, and the update terminates relatively quickly. Now, consider the opposite case. There are again three shortest paths from node *a* to node *b*. Then, a direct edge connecting node *a* to node *b* is inserted and all the previously known shortest paths should be invalidated and the rest of the shortest paths in the network that use the shortest path from node a to node b as their subpaths should also be updated. Therefore, it is likely that the portion of a small world network that is affected by an incremental growing network will be higher than the portion of the same network that is affected by its corresponding shrinking network update.

Third, comparing the speedup obtained on different networks, speedup obtained using the incremental closeness algorithm increases with the increased network size. The largest improvements are again observed with preferential attachment networks because the incremental algorithms are able to prune a larger portion of the network from subsequent computations in preferential attachment networks.

When the standard deviations on the performance improvements are examined in detail, additional observations on the performance behavior of the incremental closeness centrality algorithm can be made. First, the highest deviations are observed in the directed cycles and small world networks. This is reasonable given that both network types benefit from the existence of rewired shortcuts. Second, there are some networks on which the standard deviations for the performance of the incremental closeness centrality algorithm is very large compared to the average performance obtained on the same network. This is because in those networks some of the incremental network updates completed very quickly, which causes the deviation to be very large.

Table 12 - Performance improvements of the incremental closeness algorithm over the repeated invocations of the Dijkstra's algorithm for computing closeness centrality obtained on with different network topologies and different average node degree values as described in Chapter 7.1. The number of nodes is fixed at 3000.

Topology	**Avg Deg.**	**Growing Network Updates**		**Shrinking Network Updates**	
		Closeness Speedup	**Closeness Affected %**	**Closeness Speedup**	**Closeness Affected %**
Preferential Attachment	4	**13384.83 x** (30947.95)	0.95	**3344.98 x** (5952.6)	0.79
Preferential Attachment	6	**14714.43 x** (7298.64)	2.13	**3447.89 x** (12233.78)	1.51
Preferential Attachment	8	**15259.79 x** (51827.04)	3.19	**3694.61 x** (10063.46)	1.86
Erdos-Renyi	4	**288.15 x** (114.565)	54.70	**89.74 x** (93.174)	48.33
Erdos-Renyi	6	**515.35 x** (485.18)	69.02	**169.83 x** (148.08)	41.31
Erdos-Renyi	8	**885.19 x** (2480.52)	59.48	**321.66 x** (1174.89)	30.16
Small World	4	**471.70 x** (58927.40)	63.23	**129.20 x** (39859.40)	36.09
Small World	6	**710.11 x** (80695.62)	52.39	**338.81 x** (27515.92)	18.5
Small World	8	**1492.96 x** (1698.92)	45.63	**829.57 x** (1161.88)	11.61
Directed Cycles	4	**393.86 x** (150095.88)	31.18	**118.75 x** (74590.23)	29.84
Directed Cycles	6	**740.44 x** (15275.67)	26.17	**243.70 x** (12120.51)	21.47
Directed Cycles	8	**2391.75 x** (25320.00)	18.50	**928.00 x** (10223.88)	12.69

Although the actual values of standard deviations depend very much on the interaction of the random network update with the rest of the network topology, the general trend is that the deviation on the shrinking network updates is lower than the deviation on the growing network updates as it takes longer to complete the shrinking network updates in general.

Next, Table 12 presents performance results collected in a similar fashion to those presented in Table 11. In Table 12, different from the results presented in Table 11, the average degrees of networks are varied from 4.0 to 8.0 with a step size of 2.0 for each synthetic network topology type while the number of nodes is fixed at 3000 nodes for all. The standard deviations for the performance improvements are added in parenthesis for each corresponding average performance value.

For preferential attachment networks, there is a slight performance improvement with the increasing average degree; however, this trend is not as pronounced as it is with the other network topologies. In Erdos-Renyi networks, the performance improvements of the incremental closeness algorithm increases with the increased average degree, both for growing and shrinking network updates. Yet, this is not necessarily strictly in line with the percentage of affected nodes as the randomness of the connections in Erdos-Renyi networks may result in different proportions of the network getting affected by the issued network update. For small world networks and directed cycles, the performance improvements again increase with the increasing average degree as the percentage of nodes affected by the network updates decrease.

Overall, when the number of immediate connections per node increases, the network is denser but this does not necessarily mean that the shortest paths in the network become longer. As presented earlier in Table 7, in Erdos-Renyi networks, small world networks, and especially in directed cycles, the increased average node degree provides opportunities for including more random connections that connect far apart nodes, reducing the characteristic (average) shortest path lengths and the diameters of the networks. This effect, as a result, reflects as a substantial increase in the performance improvements obtained by the incremental closeness centrality algorithm over computing the closeness centrality with the Dijkstra's algorithm [109].

8.3.2 Incremental Betweenness Centrality Performance Results

Next, the performance of the proposed incremental betweenness centrality algorithm is examined. The design and preparation of the experiments are the same as those used for the incremental closeness centrality algorithm, as discussed earlier in Chapter 8.2. The incremental network updates are the same as those used in the timing experiments done with the incremental closeness centrality algorithm, presented earlier in Chapter 8.3.1. The non-incremental baseline algorithm is the Brandes' betweenness algorithm [28], which is the best performing betweenness algorithm used in standard implementations.

Table 13 and Table 14 list the average speedups obtained by the incremental betweenness centrality algorithm over the Brandes' algorithm both for the growing and the shrinking network updates along with the percentages of the total number of nodes that are affected. The percentages of affected nodes are calculated in terms of the size of the combination of the two sets, AffectedSinks and AffectedSources. The standard deviations for the performance improvements are added in parenthesis for each corresponding average performance value.

In particular, the performance results in Table 13 show how the performance improvements of the incremental betweenness centrality algorithm change along with the changing network size and topology. On the other hand, Table 14 reports the changes in the performance improvements along with the changing average node degree and the network topology.

Similar to the performance results obtained with the incremental closeness algorithm in Chapter 8.3.1, the performance results presented in both tables indicate that the incremental betweenness algorithm provides the highest performance benefits on the preferential attachment networks and the lowest performance improvements on the directed cycles. Directed cycles provide examples of pathological test cases for the incremental centrality algorithms because they tend to have very long diameters and relatively large average (characteristic) shortest path lengths both of which increase the lengths of the shortest paths to be updated.

Comparing the network statistics (Table 6) and the performance results obtained on different networks (Table 13), the speedup obtained using the incremental betweenness update algorithm increases with the increased network size. It is also observed that other parameters such as network diameter, characteristic path length, and average/min/max unscaled betweenness centrality values are inversely correlated with the performance obtained; the longer the shortest paths are in a network, the lower the performance improvements are in that network.

Table 13 - Performance improvements of the incremental betweenness algorithm over repeated invocations of the Brandes' betweenness algorithm obtained on networks with different topologies/sizes described in Chapter 7.1.

	Growing Network Updates		**Shrinking Network Updates**	
Topology	**Betweenness Speedup**	**Betweenness Affected %**	**Betweenness Speedup**	**Betweenness Affected %**
Preferential Attachment - 1000	**178.65 x** (971.51)	3.54	**173.84 x** (967.36)	3.63
Preferential Attachment - 3000	**971.40 x** (9021.75)	1.98	**740.13 x** (8809.35)	1.95
Preferential Attachment - 5000	**3760.48 x** (33818.54)	1.16	**2415.18 x** (32011.24)	1.14
Erdos-Renyi - 1000	**7.99 x** (996.81)	62.89	**6.60 x** (997.09)	62.48
Erdos-Renyi - 3000	**18.97 x** (40.22)	67.77	**15.08 x** (14.89)	68.08
Erdos-Renyi - 5000	**31.18 x** (66.51)	68.31	**24.87 x** (53.91)	68.32
Small World - 1000	**8.77 x** (12.25)	61.03	**7.87 x** (12.98)	61.03
Small World - 3000	**20.94 x** (27.16)	61.63	**17.13 x** (25.95)	61.63
Small World - 5000	**36.32 x** (51.01)	60.73	**29.38 x** (50.51)	60.73
Directed Cycles - 1000	**3.78 x** (803.30)	44.14	**1.79 x** (622.95)	43.98
Directed Cycles - 3000	**4.26 x** (6342.75)	42.07	**2.22 x** (4508.81)	41.65
Directed Cycles - 5000	**6.47 x** (15361.89)	43.87	**2.92 x** (15324.96)	43.12

Considering the network characteristics presented in Table 6, it is observed that in the preferential attachment networks, the characteristic path length and the diameter are substantially lower compared to the other topologies. And, the shortest paths in the directed cycles are substantially longer, yielding performance results that are much lower compared to the other network types as the performance results in Table 13 suggest.

Table 14 - Performance improvements of the incremental betweenness algorithm over repeated invocations of the Brandes' algorithm obtained on networks with different topologies and different average node degrees as described in Chapter 7.1. The number of nodes is fixed at 3000.

Topology	**Avg Deg.**	**Growing Network Updates**		**Shrinking Network Updates**	
		Betweenness Speedup	**Betweenness Affected %**	**Betweenness Speedup**	**Betweenness Affected %**
Preferential Attachment	4	**1856.54 x** (3865.926)	0.90	**1672.78 x** (2979.49)	0.89
Preferential Attachment	6	**971.40 x** (9021.75)	1.98	**740.13 x** (8809.35)	1.95
Preferential Attachment	8	**916.52 x** (6119.21)	2.69	**627.16 x** (3331.99)	2.68
Erdos-Renyi	4	**14.02 x** (66324.74)	56.26	**9.23 x** (53451.25)	55.67
Erdos-Renyi	6	**18.97 x** (40.22)	67.77	**15.08 x** (14.89)	68.08
Erdos-Renyi	8	**29.95 x** (154.4)	58.43	**22.80 x** (164.28)	58.50
Small World	4	**14.23 x** (27.64)	72.89	**11.24 x** (26.17)	72.89
Small World	6	**20.94 x** (27.16)	61.63	**17.13 x** (25.95)	61.63
Small World	8	**28.14 x** (28.41)	54.89	**23.34 x** (27.96)	54.89
Directed Cycles	4	**0.88 x** (29674.85)	38.52	**1.09 x** (17844.20)	36.98
Directed Cycles	6	**4.26 x** (6342.75)	42.07	**2.22 x** (4508.81)	41.65
Directed Cycles	8	**14.60 x** (4273.70)	37.74	**7.19 x** (2743.57)	37.75

Apart from the lengths of the shortest paths, the average betweenness value (unscaled) and the network size are also very important factors, especially the network size. As mentioned earlier, the performance benefits of the incremental betweenness update algorithm increases with the increasing network size. When the average betweenness values are examined (Table 6), it is observed that the difference across different topologies is very large. This is because in preferential attachment networks, compared to other network topologies, there are fewer intermediate nodes that are on the shortest paths. Hence, when there is a network update, there are fewer nodes whose betweenness values should be adjusted. When most of the shortest paths in a network consist of a couple of hops only (i.e., when it takes a social agent only a number of intermediaries to reach out to the majority of the network), then there are fewer nodes that lie on the shortest paths and the overall depth of the shortest path tree is shorter. As a result, there are fewer predecessors to be tracked and maintained when there is need for reconstructing the shortest paths in the network.

Finally, the percentages of nodes that are affected are higher for the incremental betweenness centrality than for the incremental closeness centrality. The closeness centrality is only affected by the changes in the shortest distances in a network. For the incremental closeness centrality, only the nodes that have strictly shorter or strictly longer paths to/from them are inserted into the sets of affected sinks/sources. On the other hand, for the incremental betweenness centrality, not only the changes in the shortest distances but also the changes in the number of shortest paths affect the centrality values. Hence, for the incremental betweenness centrality, there are more reasons to include nodes in the set of affected nodes, resulting in a larger percentage of affected nodes overall. Since the incremental betweenness centrality algorithm needs to maintain more information (e.g. tracking predecessors, maintaining the number of shortest paths), the performance of the incremental closeness centrality algorithm is higher than that of the incremental betweenness centrality algorithm (Compare the performance results in Table 11 with Table 13 and the performance results in Table 12 with Table 14).

8.3.3 How do the Hub Nodes Affect the Performance of Incremental Centrality Algorithms in Preferential Attachment Networks?

'Preferential attachment' is a network generation model that leads to the generation of scale free networks with heavy-tailed, power-law degree distributions. In such networks, there are very few nodes with a large number of connections (e.g. hubs) while the majority of the nodes in the network have very few connections. Figure 24 presents the in-degree distribution of the nodes in a 3000-node preferential attachment network with an average degree of 6. Figure 25 presents the same data in log-log scale.

The preferential attachment network whose degree distribution is presented in Figure 24 and Figure 25 have a number of hub nodes. Next, the hub nodes with the highest number of connections are listed in Table 15.

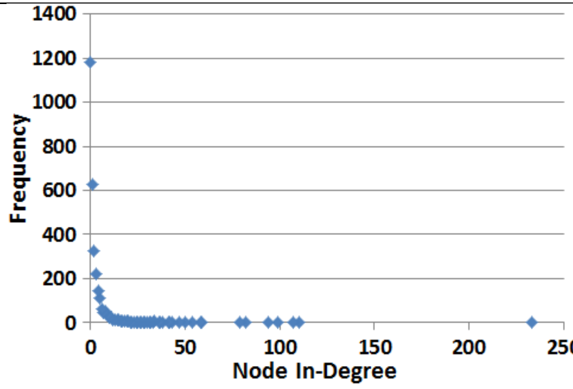

Figure 24 - The distribution of nodes' in-degree values in 3000-node preferential attachment network with average degree 6.

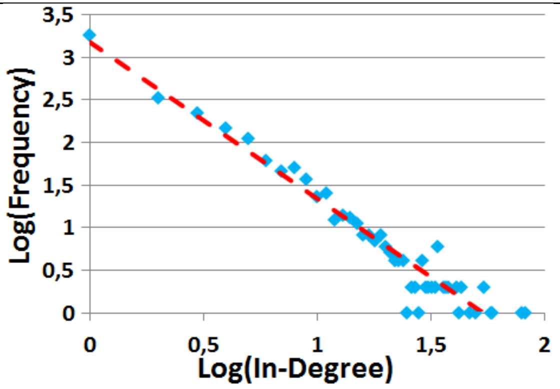

Figure 25 - Log-log scale plot for the in-degree distribution depicted in Figure 24.

Table 15 - List of the hub nodes in 3000-node preferential attachment network with average node degree 6.

Rank	Node ID	#(Connections)	Rank	Node ID	#(Connections)
1	0	233	*11*	9	54
2	6	110	*12*	21	54
3	1	107	*13*	82	50
4	3	99	*14*	31	47
5	11	94	*15*	30	43
6	2	82	*16*	16	42
7	7	79	*17*	24	41
8	15	78	*18*	54	41
9	17	59	*19*	8	38
10	4	58	*20*	25	38

One question that stems from the existence of hub nodes in a network is ‘How do the incremental centrality algorithms perform when a network update that involves one of the hub nodes is issued?’ A hub node x might be the head or the tail of a modified edge $\{x \rightarrow y\}$ or it might be on the shortest paths that are affected by the modification of an edge $\{a \rightarrow b\}$. This section attempts to answer the question posed through detailed examination of the performance results for the incremental betweenness centrality algorithm. Similar observations hold for the performance of the incremental closeness centrality, however, the performance differences across different updates are more pronounced for the incremental betweenness centrality.

How the existence of hub nodes affects the performance is a result of the network structure, the position of the incremental network update with respect to the position of the hub nodes, and the direction of connections into/out of the hub nodes. When the updated edge $\{x \rightarrow y\}$ ends at a hub node y with many incoming connections and no outgoing connections, the update terminates immediately. However, when both updates are made on nodes with high number of connections, then the incremental update takes longer to complete because such nodes are expected lie on many shortest paths. Hence, a change in the network that involves a hub node is expected to trigger a big wave of changes if it has enough many outgoing connections on top the incoming connections it has. In addition, when a hub node is marked as an affected node, then its immediate neighbors should be traversed to see if the incremental network update should propagate further by adding one or more its immediate neighbors in the list of affected nodes as well. Since hub nodes have a lot of connections, it is expected that traversing its immediate neighbors will take longer than traversing the immediate neighbors of a node with only a couple of immediate connections.

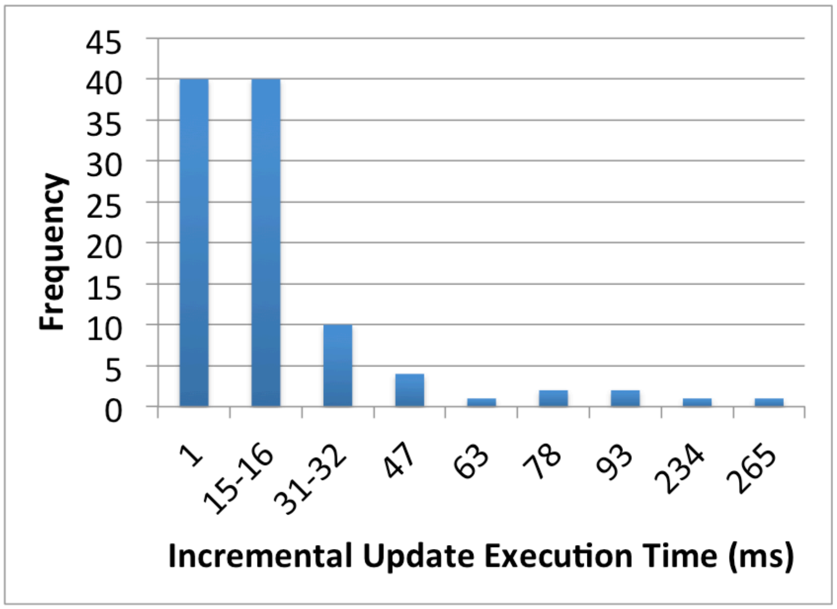

Figure 26 - Distribution of execution times for 100 incremental edge insertions.

Out of the 100 randomly selected network updates, there were only 6 updates that took significantly longer than the rest of the updates. In particular, %80 of the individual updates completed in less than 16 milliseconds, and there are only 6 updates that took more than 75 milliseconds, with only 2 updates taking longer than 200 milliseconds. Figure 26 shows the distribution of execution times for 100 incremental edge insertions.

Insertion of the edge {129 → 37} is the update that took the longest. For instance, node-129 has 24 incoming connections and 2 outgoing connections while node-37 has 18 incoming connections and 3 outgoing connections. Before the insertion of the edge {129 → 37} that would connect node-129 and node-37 directly, these two nodes were connected through 4 different shortest paths of length 3.0 as shown in Figure 27.

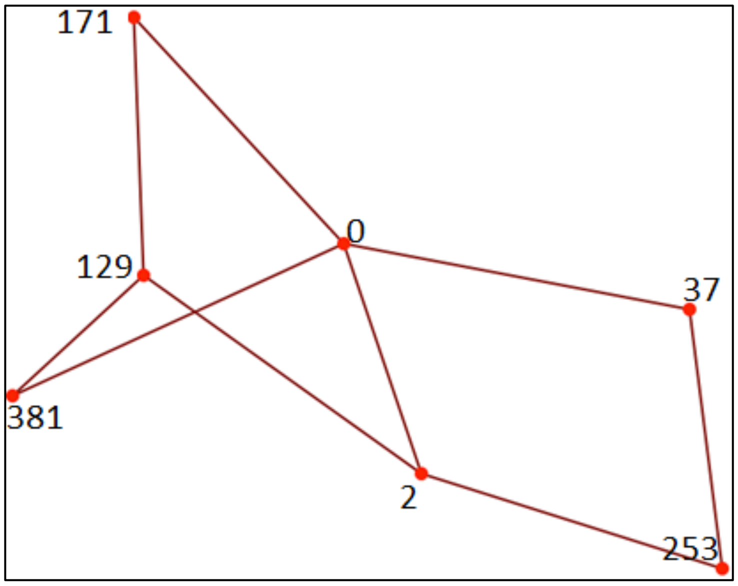

Figure 27 - Shortest paths from node-129 to node-37 before the insertion of the edge {129→37}.

For instance, when the edge {129 → 37} is inserted the old set of shortest paths should be invalidated and all the shortest paths that use these invalidated shortest paths as their subpaths should also be invalidated. Another important point to notice is that although node-129 and node-37 are average nodes with not too many or not too few connections, there are two major hubs on their shortest paths: node-0 (233 connections) and node-2 (82 connections), which cause the slowdown in the completion of this particular update when the shortest paths that pass through them are to be updated. In conclusion, when the hub nodes lie on the affected paths and are inserted into the set of affected nodes, then the network update takes longer than average. The execution time tends to increase with the increased number of hub nodes inserted into the sets of affected nodes.

8.3.4 Incremental *k*-Centrality Performance Results

Next, the performance of the incremental *k*-closeness and *k*-betweenness centrality algorithms are examined. In the performance results presented in Table 16 and Table 17, we use $k = 2$ and $k = 3$, respectively. The baseline algorithms (the Dijkstra's algorithm [109] for the closeness centrality and the Brandes' algorithm [28] for the betweenness centrality) are also bounded by the k hop limit. The experiments are designed as described in Chapter 8.2. Hence, to avoid repetition, they are not described here again. In Table 16 and Table 17, PF stands for preferential attachment networks, ER stands for Erdos-Renyi networks, SW stands for small world networks, and DC stands for directed cycles. Each network topology type is appended by a number: 1, 3, or 5, which denote the network size in terms of the number of nodes and stand for 1000 nodes, 3000 nodes, and 5000 nodes, respectively.

Considering the performance results provided in Table 16 and Table 17, it is observed that the incremental centrality algorithms provide performance benefits that are on the order of thousands. There are three main trends that stand out: (*i*) the incremental *k*-closeness algorithm provides higher performance benefits than the incremental *k*-betweenness algorithm, (*ii*) the incremental *k*-centrality algorithms provide higher performance improvements for the growing network updates than they do for the shrinking network updates, and (*iii*) the performance benefits of the incremental centrality algorithms increase with the increasing network size. All of these observations are in line with the results presented earlier.

However, there is one major difference that can only be observed in the behavior of *k*-centralities. When the shortest paths are not limited by *k* hops, the maximum performance benefits are obtained on the preferential attachment networks (Chapter 8.3.1 and Chapter 8.3.2). However, in *k*-centralities, the incremental centrality algorithms provide the minimum performance benefits in the preferential attachment networks and the maximum performance benefits in the small world networks. In other words, the ordering of the topologies in terms of how much they benefit from the incorporation of the incremental approach is reversed when the shortest paths are limited within *k* hops both for the incremental algorithms and the baseline algorithms.

The performance benefits of the incremental *k*-centralities depend on how much work they can avoid. In preferential attachment networks, the shortest paths are mostly composed of a couple of hops, with relatively smaller average characteristic path length and network diameter compared to the other network topologies. With the introduction of the limiting parameter *k*, the shortest paths become even shorter but the amount of work that can be avoided is not as high as the work that can be avoided in other network topologies.

Table 16 - Performance benefits of the proposed incremental *k*-centrality algorithms and the portion of the network affected by these changes obtained on preferential attachment, Erdos-Renyi, small world networks, and directed cycles. Avg. degree = 6, *k* = 2.

	Growing Network Updates (k = 2)				**Shrinking Network Updates (k = 2)**			
T	**Closeness Speedup**	**Closeness Affected%**	**Betweenness Speedup**	**Betweenness Affected %**	**Closeness Speedup**	**Closeness Affected %**	**Betweenness Speedup**	**Betweenness Affected %**
PF1	**4007.39 x** (2660.30)	1.77	**1169.57 x** (708.83)	0.73	**2318.95 x** (1689.62)	0.67	**561.91 x** (415.49)	1.68
PF3	**67536.43 x** (70166.31)	0.74	**12252.17 x** (8936.66)	0.29	**26128.53 x** (34690.16)	0.27	**4660.43 x** (4350.67)	0.72
PF5	**175671.04 x** (144434.28)	0.36	**33434.23 x** (19134.80)	0.14	**150069.59 x** (95076.37)	0.13	**17114.77 x** (9173.12)	0.35
ER1	**8434.33 x** (8192.99)	2.45	**2214.27 x** (1657.40)	0.77	**9349.37 x** (5936.06)	0.76	**1095.16 x** (1500.38)	2.44
ER3	**93542.53 x** (40662.82)	0.85	**19922.55 x** (6648.24)	0.27	**84290.54 x** (29807.10)	0.26	**7674.98 x** (2773.54)	0.85
ER5	**356374.65 x** (104205.84)	0.53	**64049.13 x** (17893.89)	0.16	**394281.47 x** (117711.62)	0.16	**21370.51 x** (7166.96)	0.53
SW1	**3505.08 x** (3323.48)	3.91	**700.38 x** (313.34)	1.02	**4576.60 x** (3080.00)	0.85	**286.24 x** (211.32)	4.05
SW3	**50380.67 x** (40131.32)	1.40	**5698.70 x** (2223.15)	0.35	**53706.20 x** (36106.66)	0.30	**1944.95 x** (1192.21)	1.48
SW5	**245023.50 x** (199487.08)	0.78	**22649.73 x** (8159.11)	0.20	**246886.51 x** (156819.89)	0.17	**6356.84 x** (3776.83)	0.82
DC1	**8013.70 x** (8339.80)	1.55	**1832.06 x** (1460.83)	0.68	**5939.09 x** (4855.54)	0.57	**1005.99 x** (1720.02)	1.63
DC3	**140584.58 x** (63424.38)	0.53	**22107.68 x** (7744.64)	0.23	**125663.72 x** (40038.24)	0.19	**10258.06 x** (4859.67)	0.56
DC5	**639784.95 x** (246368.46)	0.30	**87351.19 x** (28524.46)	0.13	**630523.27 x** (157903.27)	0.11	**33937.61 x** (14724.15)	0.32

In the incremental *k*-centralities algorithms, the highest performance benefits are obtained on directed cycles. In directed cycles, some nodes are connected by the shortcuts in the network formed by rewiring. However, there are also other node pairs whose shortest paths do not include any shortcut edges and such shortest paths tend to be very long. When the shortest paths are confined to the 2-hop or 3-hop neighborhood of each node, most of the very long shortest paths are eliminated from computations. In addition, how far a network update can propagate across the network is very limited as well (limited to the *k*-hop neighborhood only).

Table 17 - Performance benefits of the proposed incremental k-centrality algorithms and the portions of the network affected by these changes obtained on preferential attachment, Erdos-Renyi, small world networks, and directed cycles. Avg. degree = 6, k = 3.

	Growing Network Updates (k = 3)				**Shrinking Network Updates (k = 3)**			
T	**Closeness Speedup**	**Closeness Affected%**	**Betweenness Speedup**	**Betweenness Affected %**	**Closeness Speedup**	**Closeness Affected %**	**Betweenness Speedup**	**Betweenness Affected %**
PF1	**1933.86 x** (2460.15)	2.79	**467.76 x** (424.54)	1.71	**980.19 x** (957.92)	1.42	**318.11 x** (389.35)	2.60
PF3	**12305.23 x** (56123.57)	1.27	**3880.92 x** (4845.24)	0.73	**12345.63 x** (8129.59)	0.61	**1619.92 x** (2190.31)	1.19
PF5	**70359.54 x** (55104.65)	0.64	**13972.58 x** (7995.69)	0.35	**29268.18 x** (22821.96)	0.31	**6301.81 x** (4698.60)	0.59
ER1	**2648.67 x** (6388.20)	6.99	**665.23 x** (1342.86)	2.45	**8741.28 x** (5309.33)	2.38	**321.24 x** (1114.04)	6.92
ER3	**25453.04 x** (18069.07)	2.55	**5917.13 x** (3475.05)	0.85	**100799.68 x** (27376.85)	0.85	**2058.77 x** (10260.79)	2.56
ER5	**96010.96 x** (45713.23)	1.60	**18529.29 x** (8188.16)	0.52	**377138.79 x** (100462.26)	0.52	**6298.02 x** (2769.37)	1.59
SW1	**1054.90 x** (4155.52)	13.16	**182.17 x** (157.09)	4.15	**955.40 x** (3168)	3.29	**76.80 x** (77.80)	14.61
SW3	**14267.99 x** (41931.26)	5.15	**1294.52 x** (966.03)	1.48	**10661.16 x** (27570.41)	1.27	**431.51 x** (382.85)	5.85
SW5	**53772.04 x** (139668.06)	2.84	**4574.47 x** (3486.16)	0.82	**48955.05 x** (113508.33)	0.69	**1405.23 x** (1291.21)	3.24
DC1	**6760.30 x** (6372.81)	2.94	**1097.46 x** (1506.50)	1.67	**5346.76 x** (4207.69)	1.28	**625.45 x** (1751.89)	3.47
DC3	**80953.65 x** (38993.78)	1.01	**8812.74 x** (5324.65)	0.56	**51067.62 x** (37778.30)	0.45	**4666.67 x** (2620.77)	1.19
DC5	**288379.42 x** (135771.75)	0.58	**34715.31 x** (20091.47)	0.32	**229305.42 x** (158129.62)	0.25	**14197.06 x** (9886.13)	0.67

The abovementioned factors result in a dramatic increase in the speedup that can be achieved by incremental *k*-centrality algorithms as well as a dramatic decrease in the percentage of the nodes that are affected by the incremental network updates. Consider the percentages of the networks affected by the incremental network updates as listed in Table 13, Table 14, Table 16, and Table 17. When the shortest paths are not bounded by any *k*-hop limits, the percentages of the affected nodes are in the range of 42.07% - 68.32% (for Table 13) and 36.98% - 72.89% (for Table 14) if only the Erdos-Renyi, small-world and directed cycles networks are accounted for. These ranges extend to 1.16% - 68.32% (for Table 13) and 0.89% - 72.89% (for Table 14) if the preferential attachment networks are included as well. When the shortest paths are restricted to remain within k hops, these values change as follows. The percentages of the affected nodes remain within 0.11% - 3.91% for all network types (Table 16) for $k = 2$ while for $k = 3$, these affected percentages remain within 0.25% - 14.61% (Table 17).

When the results in Table 16 and Table 17 are compared against one another, it is observed that the performance benefits of the incremental *k*-centralities are higher when $k = 2$ and the percentages of affected nodes are lower. However, in most cases the total percentages of affected nodes are less than 10-15%, mostly staying within less than 5% of the entire network. The underlying reasons are the same as those resulting in high performance benefits. When the shortest paths are confined to *k*-hop neighborhood, how far a network can propagate across the network is also limited.

Next, in Table 18 and Table 19, the performance results of the incremental *k*-betweenness and incremental *k*-closeness algorithms, for networks with varying average node degrees are presented, using the *k*-hop limits $k = 2$ and $k = 3$, respectively. The performance results presented in Table 18 and Table 19 collected on networks with 3000 nodes and varying average node degrees (e.g. 4.0, 6.0, and 8.0). Considering the incremental *k*-betweenness results, the performance improvements provided by the incremental *k*-betweenness algorithm over the Brandes' *k*-betweenness algorithm decreases with the increasing average node degree.

Table 18 - Performance benefits of the proposed incremental *k*-closeness algorithm over *k*-closeness algorithm and portion of the network affected by these changes obtained on preferential attachment, Erdos-Renyi, small world networks, and directed cycles (*k* = 2, number of nodes = 3000). The first column (e.g. *T*) lists the topology names along with their average node degrees.

	Growing Network Updates (*k* = 2)				**Shrinking Network Updates (*k* = 2)**			
T	**Closeness Speedup**	**Closeness Affected%**	**Betweenness Speedup**	**Betweenness Affected %**	**Closeness Speedup**	**Closeness Affected %**	**Betweenness Speedup**	**Betweenness Affected %**
PF4	**42393.01 x** (49332.76)	0.19	**18835.73 x** (10743.23)	0.39	**18338.90 x** (28261.15)	0.19	**6344.60 x** (5411.19)	0.38
PF6	**67536.43 x** (34690.16)	0.74	**12252.17 x** (8936.66)	0.29	**26128.53 x** (70166.31)	0.27	**4660.43 x** (4350.67)	0.72
PF8	**88628.43 x** (69466.50)	0.32	**10587.64 x** (7429.73)	0.95	**28990.65 x** (35282.93)	0.29	**2706.76 x** (3307.44)	0.90
ER4	**112938.19 x** (115455.15)	0.18	**28141.80 x** (21516.05)	0.42	**105115.30 x** (96063.50)	0.19	**15175.37 x** (16955.21)	0.42
ER6	**93542.53 x** (40662.82)	0.85	**19922.55 x** (6648.24)	0.27	**84290.54 x** (29807.10)	0.26	**7674.98 x** (2773.54)	0.85
ER8	**67272.73 x** (57479.85)	0.32	**13440.00 x** (8457.19)	1.35	**79497.84 x** (50806.15)	0.32	**3611.11 x** (3120.86)	1.35
SW4	**80986.09 x** (55057.35)	0.72	**11547.87 x** (4282.55)	0.25	**68229.48 x** (37660.91)	0.23	**4619.78 x** (2522.49)	0.73
SW6	**50380.67 x** (40131.32)	1.40	**5698.70 x** (2223.15)	0.35	**53706.20 x** (36106.66)	0.30	**1944.95 x** (1192.21)	1.48
SW8	**48238.49 x** (33250.30)	2.06	**4235.30 x** (564.37)	0.43	**46067.38 x** (26961.08)	0.33	**1005.81 x** (564.37)	2.26
DC4	**205357.14 x** (130529.90)	0.33	**32911.39 x** (19353.79)	0.18	**179316.73 x** (67508.42)	0.16	**19609.63 x** (13187.93)	0.33
DC6	**140584.58 x** (63424.38)	0.53	**22107.68 x** (7744.64)	0.23	**125663.72 x** (40038.24)	0.19	**10258.06 x** (4859.67)	0.56
DC8	**127924.53 x** (56549.32)	0.68	**19087.64 x** (7900.49)	0.27	**125663.72 x** (31835.46)	0.20	**7260.27 x** (3496.14)	0.75

When *k* = 2, the largest performance improvements are observed in the directed cycles, followed by the Erdos-Renyi networks, preferential attachment networks, and small world networks. This is also in line with the percentage of the nodes affected by the incremental updates. In all three different types of networks, the performance improvements decrease with the increasing average node degree. This is because when there are more immediate neighbors connected to each node, an incremental network update is inevitably felt by more nodes in the network.

Table 19 - Performance benefits of the proposed incremental *k*-betweenness algorithm over *k*-betweenness algorithm implemented as in Brandes and the portion of the network affected by these changes obtained on preferential attachment, Erdos-Renyi, small world networks, and directed cycles ($k = 3$, avg. degree = 6). The first column (e.g. *T*) lists the topology names along with their average node degrees.

	Growing Network Updates (k = 3)				**Shrinking Network Updates (k = 3)**			
T	**Closeness Speedup**	**Closeness Affected%**	**Betweenness Speedup**	**Betweenness Affected %**	**Closeness Speedup**	**Closeness Affected %**	**Betweenness Speedup**	**Betweenness Affected %**
PF4	**20235.23 x** (49632.85)	0.60	**11662.51 x** (7664.97)	0.38	**16813.56 x** (27306.46)	0.36	**4986.82 x** (4115.51)	0.58
PF6	**12305.23 x** (56123.57)	1.27	**3880.92 x** (4845.24)	0.73	**12345.63 x** (8129.59)	0.61	**1619.92 x** (2190.31)	1.19
PF8	**36385.76 x** (73620.80)	1.80	**2996.25 x** (4460.23)	0.91	**9720.92 x** (8793.33)	0.7	**1427.46 x** (2753.62)	1.62
ER4	**57248.21 x** (140441.29)	0.89	**14303.40 x** (23260.63)	0.42	**54836.49 x** (98033.33)	0.42	**6205.19 x** (18471.12)	0.89
ER6	**25453.04 x** (18069.07)	2.55	**5917.13 x** (3475.05)	0.85	**100799.68 x** (27376.85)	0.85	**2058.77 x** (10260.79)	2.56
ER8	**15889.83 x** (18687.31)	5.24	**2917.32 x** (3415.20)	1.35	**12176.84 x** (16390.85)	1.32	**824.32 x** (975.16)	5.21
SW4	**23769.90 x** (47122.22)	1.91	**3262.50 x** (2417.90)	0.73	**19485.04 x** (35948.19)	0.67	**1532.71 x** (1308.55)	2.00
SW6	**14267.99 x** (41931.26)	5.15	**1294.52 x** (966.03)	1.48	**10661.16 x** (27570.41)	1.27	**431.51 x** (382.85)	5.85
SW8	**10072.24 x** (12263.00)	9.01	**886.12 x** (595.87)	2.26	**7431.26 x** (9028.73)	1.76	**220.35 x** (173.73)	10.97
DC4	**101742.36 x** (133504.69)	0.53	**19638.21 x** (21366.49)	0.33	**97027.70 x** (79794.33)	0.29	**10534.19 x** (15765.61)	0.56
DC6	**80953.65 x** (38993.78)	1.01	**8812.74 x** (5324.65)	0.56	**51067.62 x** (37778.30)	0.45	**4666.67 x** (2620.77)	1.19
DC8	**51720.16 x** (32805.24)	1.48	**6220.12 x** (4535.26)	0.76	**39479.64 x** (17969.89)	0.54	**2672.00 x** (2490.92)	1.92

When $k = 3$, similar to the case where $k = 2$, the performance improvements decrease with the increasing average node degree. However, when $k = 2$, most of the longer shortest paths are eliminated while $k = 3$ retains some of the longer shortest paths. The impact of the structure of the shortest paths become more visible and the performance improvements are lower than the performance improvements obtained when the limiting parameter $k = 2$.

For both $k = 2$ and $k = 3$, the percentages of affected nodes are lower for the incremental *k*-closeness centrality than they are for the incremental *k*-betweenness centrality. Similar observations are made for the other results and this effect stems from the difference in the definitions of the closeness and betweenness centrality. The incremental updates for the closeness centrality only propagate when the shortest distances in a network change while the incremental updates for the betweenness centrality have additional reasons for propagation such as the changes in the number of shortest paths, apart from the shortest distance changes.

For the incremental *k*-closeness centrality, since directed cycles have very long shortest paths, limiting the shortest paths into *k* hops provides substantial performance improvements, yielding the highest performance improvements across all types of synthetic networks simulated.

Unlike the behavior of the incremental *k*-betweenness centrality, the incremental *k*-closeness centrality shows different trends of performance improvements for different network topologies. For the incremental *k*-betweenness centrality, the performance improvements consistently decrease with the increasing average node degree in the networks. There are more edges emanating from each node on average. Hence, there are potentially a larger number of shortest paths that are of equivalent length. This would cause the incremental *k*-betweenness algorithm to search for and update a larger number of paths while the same situation might help incremental *k*-closeness for an early termination of an update. Consider the following scenario. From node *x* to node *y*, there are three different shortest paths: p_1, p_2, and p_3. Then, an edge *e*, which was on the path *p1,* is removed. In such a case, the *k*-betweenness centrality values of all the intermediates on the remaining shortest should be updated. However, in the case of incremental *k*-closeness centrality, one of the shortest paths is removed but there are other shortest paths to rely on which would keep the shortest distance from node *x* to node *y* same as before.

While the specific values for the speedups obtained over the baseline algorithms depend very much on the network topology and the shortest paths in the networks, the key takeaway point is that in the computations of the *k*-betweenness and *k*-closeness centralities, the incremental algorithms provide the best performance improvements in directed cycles, followed by the Erdos-Renyi, followed by the preferential attachment networks, and the small world networks. This is the reverse of the trend observed in non-approximate versions of the betweenness and closeness centrality computations, and is related with the amount of work that can be pruned with the incorporation of the incremental computation approach.

Small world networks have a number of properties that cause them to behave slightly differently than the other network types. Small world networks have low average shortest path lengths, the networks are highly clustered compared to the other network types, and their edges are undirected, all of which help increasing the connectivity in the network. Hence, the *k*-hop neighborhood of a node might be more crowded than it is for the other network types that have the same number of nodes and the same average node degree, which reflects as higher portions of networks being affected and a higher number of shortest paths to be updated.

8.3.5 Discussion on the Memory Consumption

This section discusses observations on the memory consumption of the algorithms the experiments were run with. As previously mentioned in Chapter 4.5 and Chapter 5.5, the overall memory consumptions of the incremental centrality algorithms are higher than their corresponding baseline algorithms (the Dijsktra's algorithm [109] for computing closeness centrality and the Brandes' algorithm [28] for computing betweenness centrality).

The maximum memory consumption statistics are collected for each run. In the following set of figures (Figure 28 - Figure 33), the memory consumption of the incremental closeness algorithm is compared against the Dijkstra's algorithm for computing closeness centrality. In the next set of figures (Figure 34 - Figure 39), the memory consumption of the incremental betweenness algorithm is compared against the Brandes' algorithm. The figures on the left hand side show memory consumption results for the varying number of nodes and keeping the average node degree at 6.0. The figures on the right hand side show similar information for the varying average node degrees while keeping the number of nodes at 3000. All these figures show the maximum memory consumption in GBs.

As mentioned earlier, all the experiments were done on a machine with 256 GB main memory. Hence, the memory consumption did not hit the physical memory limit of the machine. The source codes for all algorithms are written under the same coding infrastructure, in GraphStream, a pure Java-based dynamic graph library [164].

In Java, garbage collection is automated. Although it is possible to call for garbage collection explicitly in the source code using System.gc() function call, this does not necessarily enforce the execution of garbage collection [172] [173]. Since the experiments were run on a machine that has 256GB of memory, the garbage collection is almost never executed as the memory consumption does not necessarily get close to the actual limits and this is why very high, climbing memory consumption is observed. Therefore, while the numbers presented in the figures below are not very representative of an implementation with ideal memory management, there are still some conclusions that can be drawn.

First, for the closeness centrality, the network topology type does not change the maximum memory consumption observed; the memory consumption is consistent across preferential attachment, Erdos-Renyi, and directed cycles. This is primarily because the information on the intermediates that lie on the shortest paths is not maintained; only the shortest distances and closeness values are kept. However, it is higher for small world networks. This is because the undirected edges in small world networks are kept as two separate directed edges in opposite directions, which increases the number of edges stored in the memory. Second, both the number of nodes and the sparsity of the networks are important in determining the amount of memory consumed, but, among these two factors, what makes the real difference in how much memory is needed is the number of nodes in a network. For computing the *k*-closeness centrality, bounding the shortest paths by *k* hops does not necessarily affect the amount of memory used up because the amount of memory to keep all the nodes and edges is still needed as well as the shortest distances among nodes and the closeness centrality value of each node.

On the other hand, slightly different trends are observed for the memory consumption of the betweenness computation. Both for the Brandes' betweenness algorithm and the incremental betweenness centrality algorithms, the number of nodes in the network is a more important factor than the average node degree, similar to the results observed with closeness centrality. However, since the intermediates on the shortest paths are also kept in the memory, the topology of the network is also a factor. In addition, for betweenness centrality, bounding the shortest paths within *k* hops helps by reducing the amount of memory further as there are fewer shortest paths with relatively low number of intermediates to be maintained, compared to regular non-approximate version of the betweenness. When the memory consumption results of the *k*-betweenness is compared against that of the regular betweenness centrality, it is still possible to notice some impact of the topology on the amount of memory consumed. Yet, the differences across different network topologies are less pronounced when the shortest paths are bounded to remain within *k* hops.

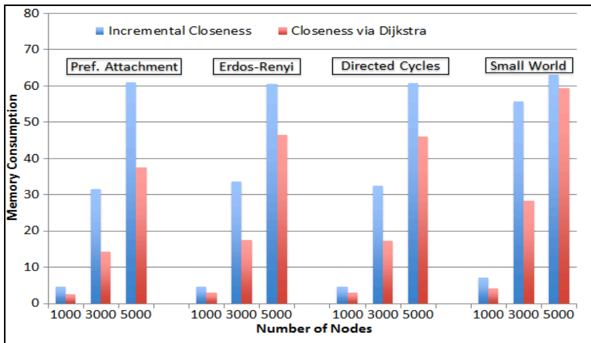

Figure 28 - Memory statistics comparing the incremental closeness algorithm with the Dijkstra's algorithm for closeness.

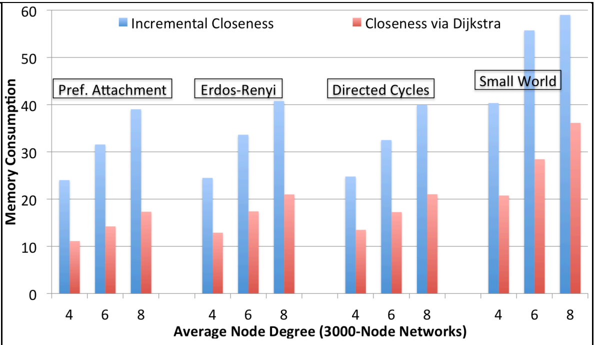

Figure 29 - Memory statistics comparing the incremental closeness algorithm with the Dijkstra's algorithm for closeness.

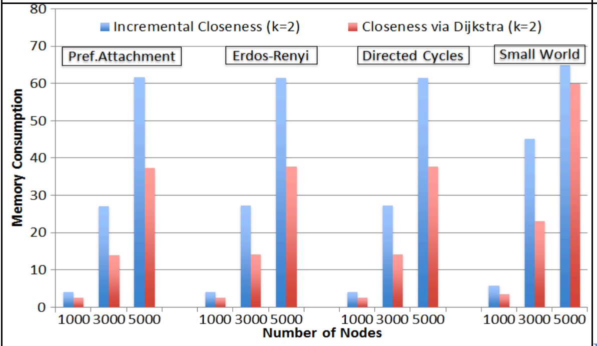

Figure 30 - Memory statistics comparing the incremental closeness algorithm with the Dijkstra's algorithm for closeness. Both are bounded by $k = 2$.

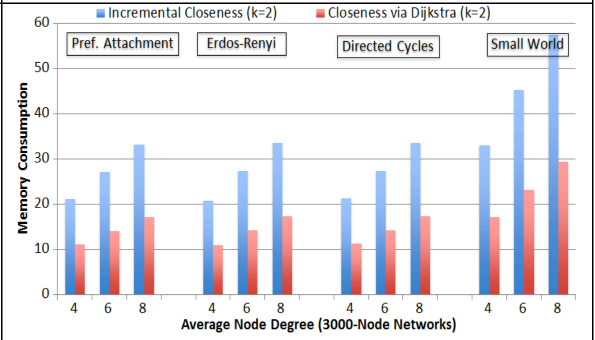

Figure 31 - Memory statistics comparing the incremental closeness algorithm with the Dijkstra's algorithm for closeness. Both are bounded by $k = 2$.

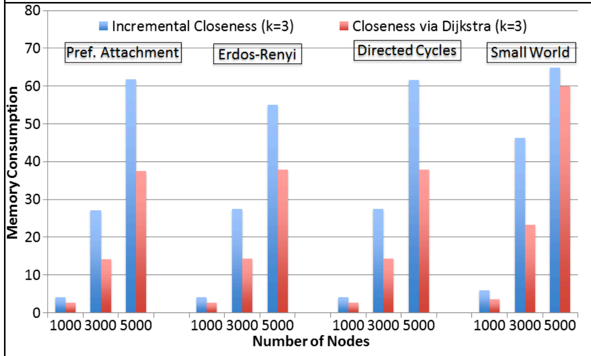

Figure 32 - Memory statistics incremental closeness algorithm with the Dijkstra's algorithm for closeness. Both are bounded by $k = 3$.

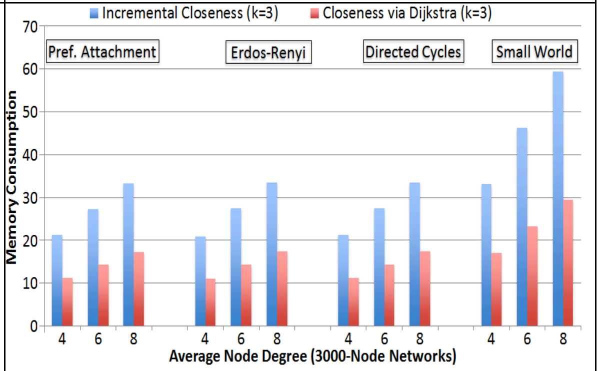

Figure 33 - Memory statistics incremental closeness algorithm with the Dijkstra's algorithm for closeness. Both are bounded by $k = 3$.

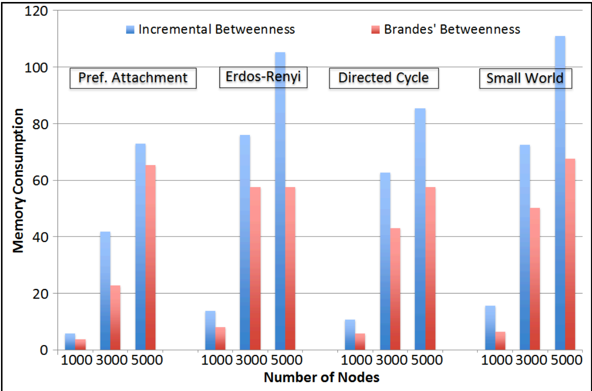

Figure 34 - Memory statistics comparing the incremental betweenness algorithm with the Brandes' algorithm.

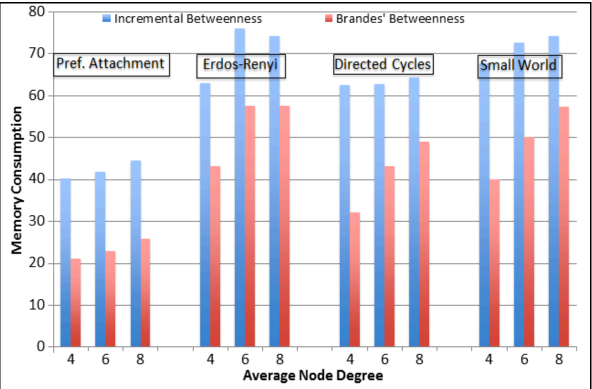

Figure 35 - Memory statistics comparing the incremental betweenness algorithm with the Brandes' algorithm.

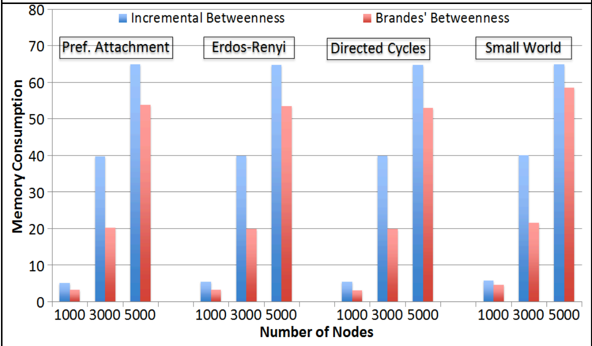

Figure 36 - Memory statistics comparing the incremental betweenness algorithm with the Brandes' algorithm. Both are bounded by $k = 2$.

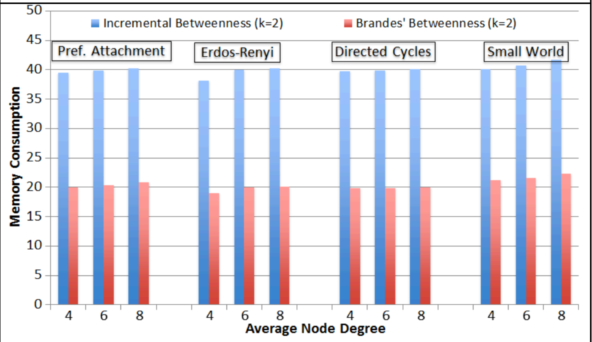

Figure 37 - Memory statistics comparing the incremental betweenness algorithm with the Brandes' algorithm. Both are bounded by $k = 2$.

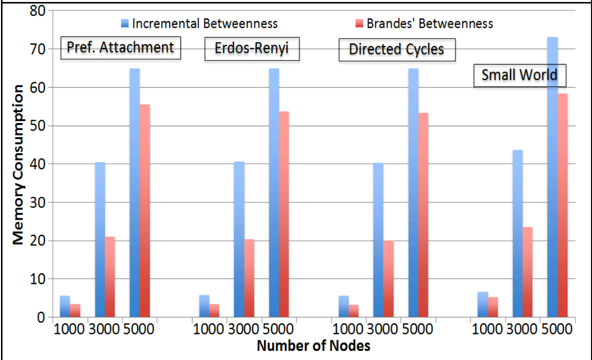

Figure 38 - Memory statistics comparing the incremental betweenness algorithm with the Brandes' algorithm. Both are bounded by $k = 3$.

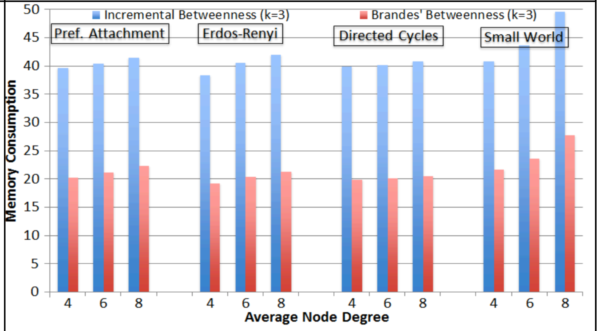

Figure 39 - Memory statistics comparing the incremental betweenness algorithm with the Brandes' algorithm. Both are bounded by k = 3.

The actual numbers of memory usage can be very much affected by the programming language and the computing platform (e.g. CPU, RAM, how much of the main memory is reserved as RamDisk) on which the experiments are run. For instance, for Java, the Java Virtual Machine (VM) arguments also change how the memory allocation is done and how much memory is allocated for identical runs executed on the same physical machine. In our case, the same set of runs was observed to have different memory consumption values in different machines, and in the same machine with different VM arguments. A lot of the memory consumption observed in the figures above are governed by the memory allocation behavior, and would reflect the topology, size, and density differences better if the algorithms were implemented in another language where memory allocation can be managed at fine-grain resolution. Hence, we do not elaborate further on the memory consumption and take the memory consumption analysis done in Chapter 4.5 and Chapter 5.5 as ground.

8.3.6 Additional Performance Results with Directed Cycles

Both in small world networks and directed cycles, the network topology and the speedup values of the incremental centrality algorithms vary substantially depending on the rewiring probability *p* even when the number of nodes and the average node degree are kept the same. In directed cycles, each node has a number of immediate neighbors. Some of the other nodes that are not within the immediate neighborhood of a node can be reached in a small number of steps following the shortcuts formed in the network while for accessing the others long paths in the form of chains that cannot benefit from the existence of the shortcuts should be traversed.

Directed cycles have different topological characteristics and performance values depending on the chosen rewiring probability *p*. As *p* value moves closer to 1.0, the randomness in the network increases, turning the network into more of a random network. As presented earlier in Chapter 7.1.5, the differences observed in the characteristics of the directed cycles are larger than those observed in the small world networks. This enables us to observe how the performances of the proposed incremental centrality algorithms change along with the changing network characteristics easier in directed cycles. Hence, in this section, we examine how the performance of the proposed centrality algorithms change when the rewiring probability *p* changes, which in turn changes both the clustering coefficient and the structure of the shortest paths as reflected by the diameter and the characteristic (average) shortest path lengths.

Table 20 - Network statistics for additional directed cycles based network, sweeping rewiring probability (Size = 1000 nodes, avg. degree = 6. Same as Table 8 in Chapter 7.1).

p	**Max Btw**	**Avg. Btw**	**Std. Dev. Btw**	**Min Deg**	**Max Deg**	**Std. Dev. Deg**	**Diameter**	**Char. Path Length**	**Clustering Coefficient**
0.2	34020.93	4305.77	3104.97	3	9	1.028	35	9.711	0.154
0.4	15036.88	3183.39	2395.88	3	12	1.423	30	8.017	0.071
0.6	14763.34	2268.86	2463.76	3	11	1.575	31	7.789	0.024
0.8	6779.73	833.58	1161.97	3	12	1.715	22	6.444	0.005
1.0	1026.5	100.79	144.32	3	13	1.765	12	3.862	0.003

The following set of tables (Table 21 – Table 24) report performance results for a sweep of the rewiring probability, *p*, in the range of 0.2 to 1.0 with a step size of 0.2, on 1000-node directed cycles networks, with an average degree of 6. Table 21 presents the performance results obtained with the proposed incremental closeness centrality while

Table 22 presents the corresponding performance results of the proposed incremental betweenness algorithm on 1000-node directed cycles.

Table 21 - Performance benefits of the incremental closeness algorithm obtained on directed cycles and the percentages of the networks affected by these changes (1000 nodes, average degree = 6).

	Growing Network Updates		**Shrinking Network Updates**	
p	**Closeness Speedup**	**Closeness Affected %**	**Closeness Speedup**	**Closeness Affected %**
0.2	280.88 x	22.903	121.45 x	18.659
0.4	298.49 x	29.494	126.94 x	22.552
0.6	341.23 x	26.876	122.34 x	21.511
0.8	761.37 x	14.003	234.25 x	11.817
1.0	7246.77 x	2.274	2188.96 x	2.091

Table 22 - Performance benefits of the incremental betweenness algorithm obtained on directed cycles and the percentages of the networks affected by these changes (1000 nodes, average degree = 6).

	Growing Network Updates		**Shrinking Network Updates**	
p	**Betweenness Speedup**	**Betweenness Affected %**	**Betweenness Speedup**	**Betweenness Affected %**
0.2	1.42 x	47.780	0.80 x	47.264
0.4	9.09 x	36.268	7.29 x	36.634
0.6	17.95 x	28.004	15.34 x	27.089
0.8	57.47 x	13.063	43.74 x	12.892
1.0	974.19 x	2.238	688.53 x	2.270

As shown in Table 20, the general trend is that along with the increasing rewiring probability, the clustering coefficient, the diameter, and the characteristic path length reduce. This reflects as an increase in the speedups of the incremental closeness algorithm as presented in Table 21. In addition, the speedup obtained over the repeated invocations of the Dijkstra's algorithm increases with the reducing percentage of affected nodes, in line with the other results presented earlier in this chapter. Similar reasoning holds for the results presented in

Table 22 for the incremental betweenness algorithm. However, the speedups obtained with the incremental algorithms are significantly higher for the closeness centrality than they are for the betweenness centrality as presented in Table 21 and

Table 22.

One interesting point is that when the rewiring probability p is very low, the shortest paths in the network tend to be very long because of the nodes that cannot be connected through the shortcuts in the network. Hence, when there is an update in the network, which results in a change that affects one of those longer, no-shortcuts kind of paths, it might become very costly to update all the predecessors on the path and the related information. Hence, when the rewiring probability p = 0.2, the speedup for the growing networks is very low (only 1.42x) while the non-incremental, Brandes' betweenness algorithm performs slightly better than the incremental algorithm for shrinking network updates (0.80x) due to the superfluous work the incremental betweenness algorithm has to do in this case.

Table 23 - Performance benefits of the incremental k-centrality algorithms and the percentages of the networks affected by these changes obtained on the directed cycles (k = 2, 1000 nodes, avg. degree = 6).

	Growing Network Updates (k = 2)				**Shrinking Network Updates (k = 2)**			
p	**Closeness Speedup**	**Closeness Affected%**	**Betweenness Speedup**	**Betweenness Affected %**	**Closeness Speedup**	**Closeness Affected %**	**Betweenness Speedup**	**Betweenness Affected %**
0.2	10654.97x	1.388	1896.79x	0.656	7284.59x	0.495	792.66x	1.47
0.4	18280.59x	1.676	2656.63x	0.686	10163.28x	0.637	1028.50x	1.70
0.6	18305.36x	1.652	3140.03x	0.674	10318.20x	0.651	1294.06x	1.66
0.8	18357.80x	1.575	3174.32x	0.686	13134.49x	0.68	1336.51x	1.55
1.0	24903.89x	1.052	4172.75x	0.574	14034.04x	0.57	1851.82x	1.05

Table 24 - Performance benefits of the incremental *k*-centrality algorithms and the percentages of the networks affected by these changes obtained on the directed cycles (k = 3, 1000 nodes, avg. degree = 6).

	Growing Network Updates (k = 3)				**Shrinking Network Updates (k = 3)**			
p	**Closeness Speedup**	**Closeness Affected%**	**Betweenness Speedup**	**Betweenness Affected %**	**Closeness Speedup**	**Closeness Affected %**	**Betweenness Speedup**	**Betweenness Affected %**
0.2	4648.45x	2.380	740.56x	1.493	2839.03x	1.045	412.64x	2.90
0.4	5930.71x	3.455	1239.29x	1.719	3474.23x	1.547	619.20x	3.66
0.6	6025.94x	3.400	1274.09x	1.676	4253.95x	1.578	663.20x	3.48
0.8	6746.01x	3.005	1383.60x	1.571	4124.95x	1.524	736.41x	2.92
1.0	12205.53x	1.521	2305.77x	1.044	5975.05x	1.026	1491.10x	1.52

However, this observation does not hold for *k*-centralities. The following tables, Table 23 (k = 2) and Table 24 (k = 3), present the performance results of the incremental algorithms for the *k*-centralities in directed cycles. Since the percentages of affected nodes remain less than 5% of the entire network, in *k*-centralities, we observe speedups that are on the order several hundreds to thousands.

8.3.7 Performance on Larger Networks

One important point about larger networks is that it takes too long for them to compute the centrality measures, especially when one wishes to recompute the centrality measure with every change on networks that are changing over time. And, the performance improvements that were obtained on networks with short average/characteristic shortest path lengths (typical of real-world social networks) are substantial (See performance results obtained on preferential attachment networks presented in Table 11 in Chapter 8.3.1). 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 networks even after all the updates in the networks are final. In other words, incremental closeness algorithms support both on-the-fly computation on dynamic networks and faster computation of the metric on large, static networks.

Table 25 - Performance improvements for the results presented in Figure 40.

#(Nodes)	#(Edges)	Performance
10,000	30,000	14.78 x
20,000	60,000	24.17 x
30,000	90,000	26.28 x
40,000	120,000	29.53 x

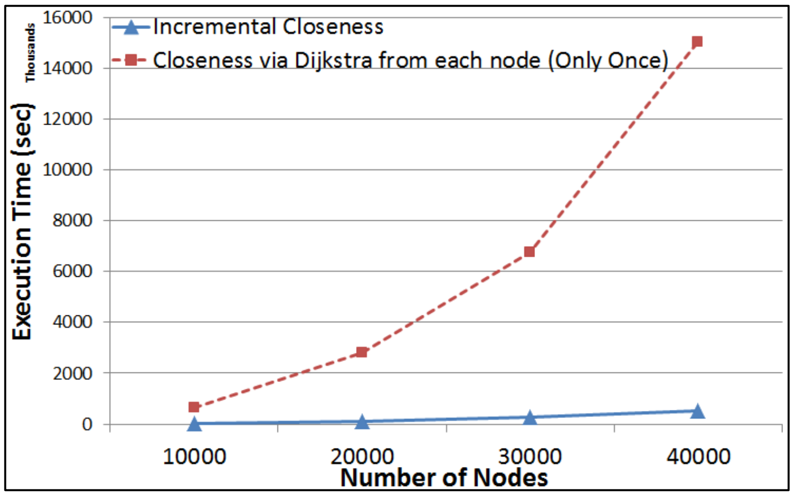

Figure 40 - Performance of incremental closeness computation vs. the computation of closeness centrality via the Dijsktra's algorithm on large preferential attachment networks. The Dijsktra's algorithm is executed only once after all updates completed.

We support this claim by the results presented in Figure 40, in which traditional closeness centrality computation using the Dijkstra's algorithm [109] is executed *only* once instead of modeling the growth of the network. In the results presented in Figure 40 and Table 25, four different preferential attachment networks are used where the number of nodes is varied between 10,000 and 40,000 with a step size of 10,000. The average node degree is 6.

This set of experiments "pretend" to build the network incrementally by invoking the *INSERTCLOSENESS* algorithm once for each edge in the network and compare its performance against computing closeness centrality by running the Dijkstra's algorithm *only* once from each node in the network on the final, static 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, this set of experiments use preferential attachment networks. According to the results presented in Table 25, computing closeness centrality with repeated invocations of the *INSERTCLOSENESS* algorithm performs 15-30x better than using traditional non-incremental algorithms only once (e.g. the Dijkstra's algorithm).

The incremental centrality algorithms work best on mid-scale networks with several thousands of nodes and edges with frequent network updates. In an attempt to see what is possible in a reasonable time using the hardware already described above, we have run two additional exploratory experiments with the incremental closeness centrality algorithm and the incremental betweenness centrality algorithm. These experiments were run on larger networks in the same fashion as the other experiments described earlier in this section: experiments that build a network from scratch by inserting each edge one at a time, incrementally. To be consistent with the other experiments presented earlier in this section, we have again used directed, preferential attachment networks and set the average node degree as 6. In this additional set of experiments, only the incremental algorithms are run, and their static non-incremental counterparts are not run for comparison.

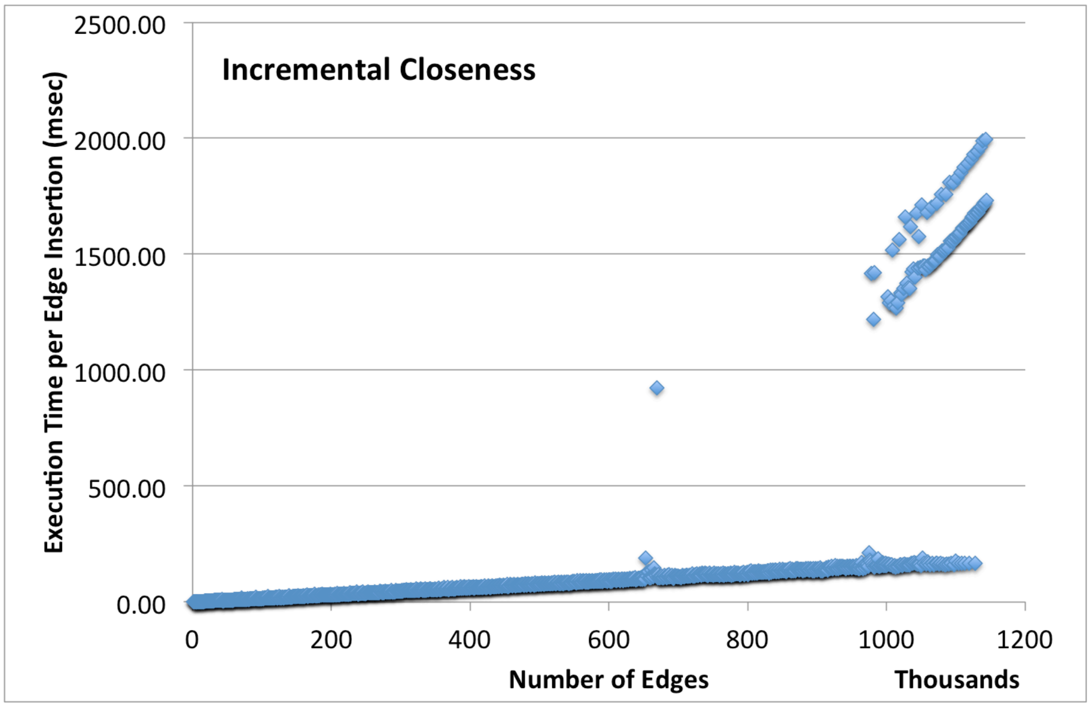

Figure 41 - Incremental closeness centrality algorithm execution time per edge insertion in milliseconds versus the number of edges inserted. The numbers of edges are represented in thousands, where ‘1200’ on the *x*-axis represents 1,200,000 edges. The execution time per edge insertion is given in milliseconds, 2500 represents the 2.5 seconds.

The results of this exploratory experiment suggest that the incremental closeness centrality algorithm run on the available hardware (in particular a machine with 256GB of RAM) can easily handle a network with 100.000 nodes and 600,000 edges. Then, we tried another run on a larger network with 250,000 nodes and 1,500,000 edges. Figure 41 and Figure 42 present the execution time of the incremental closeness centrality algorithm per edge insertion in this larger experiment versus the number of edges and the number of nodes, respectively. In these figures, the numbers of edges and nodes are presented in thousands. The ‘1200’ on the *x*-axis of Figure 41 represents 1,200,000 edges while ‘200’ on the *x*-axis of Figure 42 represents 200,000 nodes.

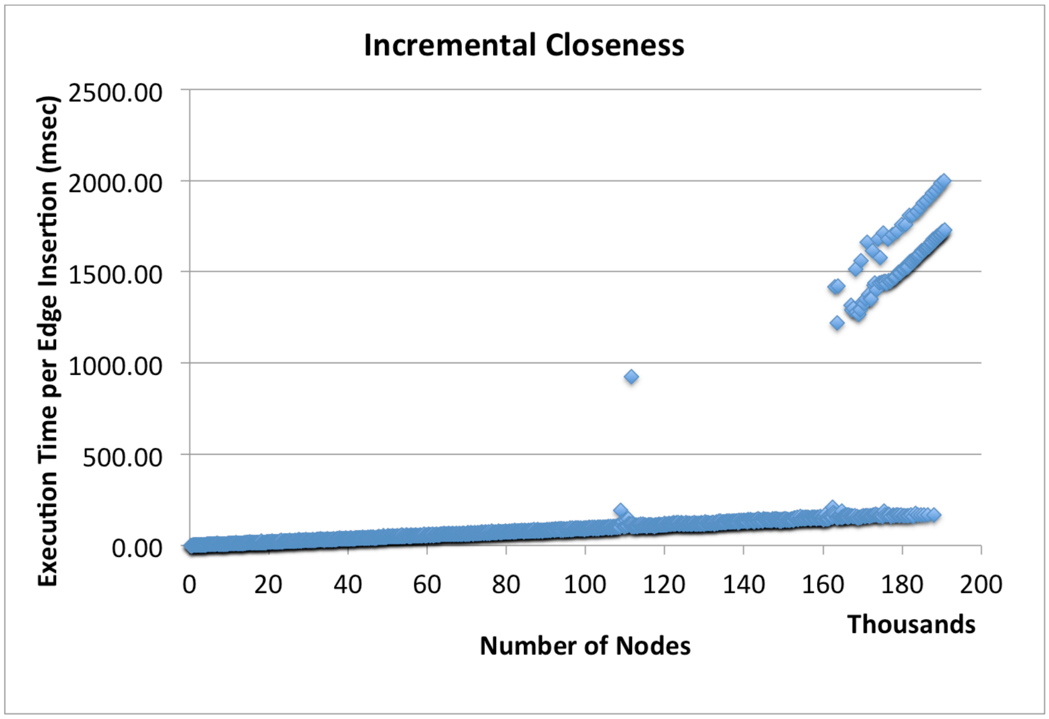

Figure 42 - Incremental closeness centrality algorithm execution time per edge insertion in milliseconds versus the number of nodes inserted. The numbers of nodes are represented in thousands, where ‘200’ on the *x*-axis represents 200,000 nodes. The execution time per edge insertion is given in milliseconds, 2500 represents the 2.5 seconds.

Before being able to insert all the edges into the 250,000-node network, the system started swapping at around 165,000 nodes and 1,000,000 edges. This is why the execution times oscillate between low and high values after the number of edges in the network reach 1,000,000 and the time required for each edge insertion increased dramatically on average. The higher execution time values represent the execution times for the network updates that are affected by swapping and the lower values represent the execution times of the updates that were executed when the memory was freed up. This data point is not exhaustive, but it suggests that on a machine with 256GB of physical memory can process incremental closeness centrality on preferential attachment based social networks with degree of 6 and somewhere between 150,000 to 200,000 nodes.

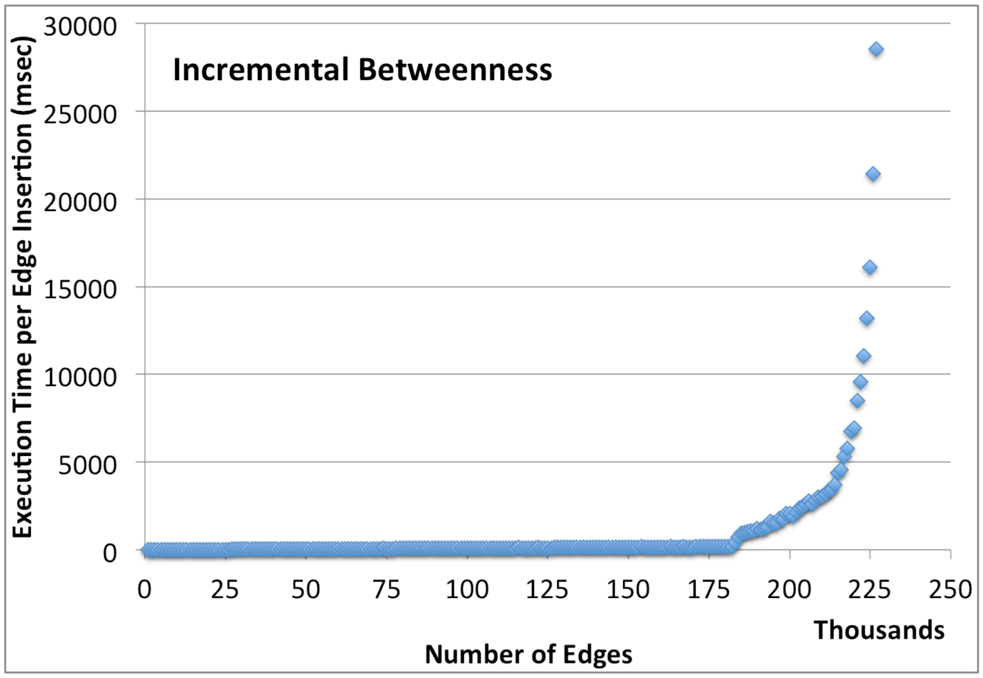

Figure 43 - Incremental betweenness centrality algorithm execution time per edge insertion in milliseconds versus the number of edges inserted. The numbers of edges are represented in thousands, where ‘200’ on the *x*-axis represents 200,000 edges. The execution time per edge insertion is given in milliseconds, 30,000 represents the 30 seconds.

Similar results were also collected for the behavior of the incremental betweenness centrality algorithm, as presented in Figure 43 and Figure 44. Figure 43 and Figure 44 present the execution time of the incremental closeness centrality algorithm per edge insertion versus the number of edges and the number of nodes, respectively. In these figures, the numbers of edges and nodes are given in thousands. The ‘200’ on the *x*-axis of Figure 43 represents 200,000 edges while ‘40’ on *x*-axis of Figure 44 represents 40,000 nodes.

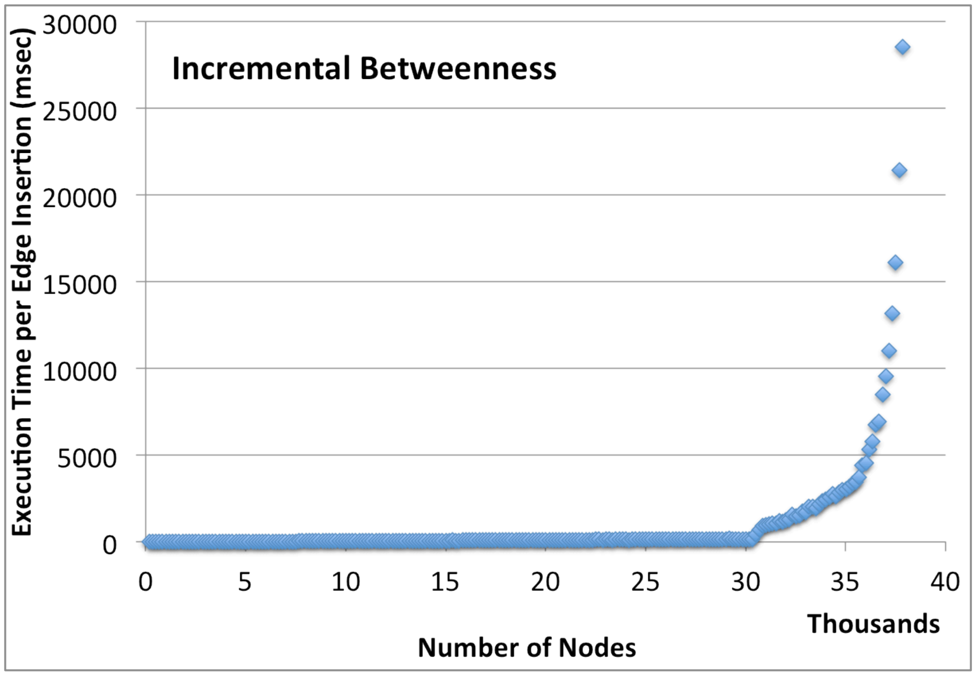

Figure 44 - Incremental betweenness centrality algorithm execution time per node insertion in milliseconds versus the number of nodes inserted. The numbers of nodes are represented in thousands, where '40' on the *x*-axis represents 40,000 nodes. The execution time per edge insertion is given in milliseconds, 30,000 represents the 30 seconds.

For the case of betweenness centrality, the fact that additional information must be stored for the incremental computations, the corresponding network sizes are much smaller for the betweenness centrality. For betweenness centrality, the execution time per edge insertion starts increasing considerably when the network reaches 30,000 nodes and 180,000 edges. Once the network becomes larger than 35,000 nodes and 225,000 edges, the execution time of running the incremental betweenness algorithm on a new edge is observed to increase dramatically. However, the cause of this slowness is not the physical memory limits; the system does not yet swap due to lack of memory. Rather, it is slow because as the network gets larger, the algorithm needs to update several pieces of information including the number of shortest paths, the shortest distances, and the predecessors on the shortest paths. In other words, the amount of information the algorithm needs to update has increased substantially with the increased node/edge size, which results in slower update progression.

As one final note, it is also observed that, in terms of memory requirements, the stack size is more important for the incremental betweenness centrality than it is for the incremental closeness centrality for such larger networks. This is because there are more sub-algorithms for the incremental betweenness centrality than for the incremental closeness centrality. Hence, the number of sub-algorithms invoked during the execution of incremental betweenness centrality tends to be much larger than it is for the incremental closeness centrality, which might call for a larger stack size to be allocated for the execution of the incremental betweenness centrality.

8.4 Results with Real-Life Networks

This section evaluates the performance of the proposed incremental centrality algorithms on the real life networks that are described in Chapter 7.2 and interprets the obtained performance results in line with the topological features of the networks.

In a similar fashion to the experiments with synthetic networks, for growing network updates, an earlier version of the network is formed which has all the information except 100 edges. Then, those edges are inserted incrementally one by one. Similarly, to obtain performance results with shrinking network updates, we start with the complete version of the network and remove the same set of 100 edges that were used in the timing runs with the growing network updates. The performance of the incremental closeness algorithm is compared against the Dijkstra's algorithm while the performance of the incremental betweenness centrality is compared against the Brandes' algorithm as baseline algorithms. When measuring the performance of the *k*-centralities, the baseline algorithms are also bounded by *k* hop limits.

8.4.1 Incremental Closeness Centrality Performance Results

First, the performance of the incremental closeness centrality algorithm is discussed. Table 26 and Table 27 present the performance results of the incremental closeness algorithm collected on real life networks that are described in Chapter 7.2. In the performance results presented in Table 26, the networks were modeled as weighted networks while Table 27 presents the performance results collected on the unweighted (binary) versions of the same real-life networks. Both Table 26 and Table 27 provide the percentages of the total number of nodes that are affected by the network updates.

Table 26 - Performance improvements of the incremental closeness algorithm over computing closeness centrality using repeated invocations of the Dijkstra's algorithm, collected on real-life networks described in Chapter 7.2. Information on the affected portion of the network is also provided. The results presented in this table are obtained on the weighted versions of the networks.

	Growing Network Updates		**Shrinking Network Updates**	
Network	**Closeness Speedup**	**Closeness Affected%**	**Closeness Speedup**	**Closeness Affected%**
SocioPatterns	452.31 x	2.51	49.00 x	4.19
Twitter (Iran)	21769.19 x	0.33	11650.32 x	0.33
Email	1917.84 x	5.41	585.01 x	4.58
HEP Coauthorship	8328.39 x	6.84	3732.56 x	6.16
P2P	356527.35 x	0.04	23494.72 x	0.04

Considering the performance results presented in Table 26, in general, similar to the results obtained on the synthetic networks, the performance benefits of the proposed incremental closeness algorithm increase with the increasing network size and growing network updates yield higher performance benefits than the corresponding shrinking network updates. However, how much performance improvement can be obtained is also a factor of the structure of the shortest paths in the network and the portion of the network that is affected.

For instance, in the 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, slightly more than a single hop, which reflects as an enormous speedup that is obtained over the non-incremental algorithm. Thus, the majority of the edges are on the shortest paths for the nodes they are connecting and 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 26. This is 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 the node x to see if it is possible to find another path to y and which one is the shortest if any. Given that the network has hub-like nodes that have 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.

Similar behavior is observed in the Twitter dataset collected on Iran sanctions. In the Iran Twitter dataset, the average shortest path is 1.4, and it is a very sparse network as well. This is why it breaks the pattern of increased performance benefits with the increasing network size and provides substantial speedup with the use of incremental centrality algorithms.

Another interesting observation is that the affected portion of the network is higher when there is higher impact of personal acquaintances or in-person communication due to the way the networks are structured. In such networks, there is usually higher transitivity and clustering. 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 a conference-like environment. For instance, the clustering coefficient of the SocioPatterns network is 0.534 while the clustering coefficient for the Iran retweet network is 0.016.

Table 27 - Performance improvements of the incremental closeness algorithm over computing closeness centrality using repeated invocations of the Dijkstra's algorithm, collected on real-life networks described in Chapter 7.2. Information on the affected portion of the network is also provided. The results presented in this table are obtained on the unweighted (binary) versions of the networks.

	Growing Network Updates		**Shrinking Network Updates**	
Network	**Closeness Speedup**	**Closeness Affected%**	**Closeness Speedup**	**Closeness Affected%**
SocioPatterns	339.51 x	64.21	115.79 x	1.90
Twitter (Iran)	25411.20 x	0.33	4177.58 x	0.33
Email	6728.97 x	20.01	1680.93 x	4.11
HEP Coauthorship	4395.16 x	23.55	679.55 x	18.89
P2P	301301.33 x	0.04	20662.69 x	0.04

Next, Table 27 presents the performance results similar to the results presented in Table 26. In Table 27, the presented results are collected on the unweighted versions of the real-life networks. Although they are relatively close, the performance results presented in Table 27 are different from the performance results presented in Table 26, as well as the percentages of the affected nodes in each network.

When the networks are unweighted, all edges have the same edge cost. Assigning costs to the edges causes the structure of the shortest paths to be different. Consider the abstract example depicted in Figure 45. The cost of each edge is written on top of the edge.

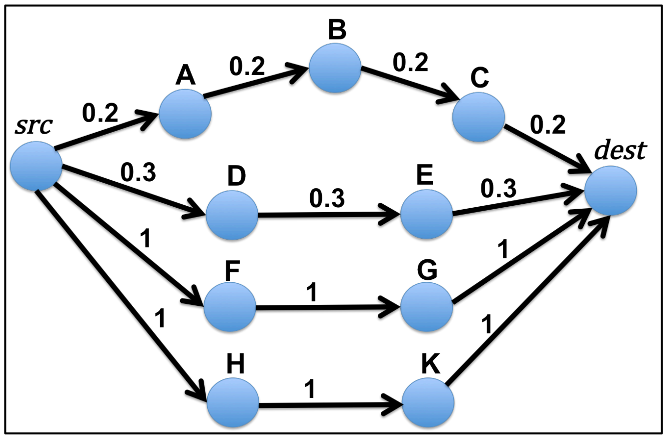

Figure 45 - Example network to demonstrate how the use of edge costs might change the structure of the shortest paths in a network.

In the network drawn in Figure 45, there are four possible paths from node *src* to node *dest*. Out of these four possible paths, only the path $\{src \rightarrow A \rightarrow B \rightarrow C \rightarrow dest\}$ is the shortest path with a total path cost of 0.8 when the edge costs written on the edges are taken into account. However, when the same network is modeled as an unweighted network where the edge costs are ignored, the structure of the shortest paths change dramatically. In the unweighted version of the network depicted in Figure 45, there are three distinct shortest paths, each with a total path cost of 3: $\{src \rightarrow D \rightarrow E \rightarrow dest\}$, $\{src \rightarrow F \rightarrow G \rightarrow dest\}$, and $\{src \rightarrow H \rightarrow K \rightarrow dest\}$. Another effect of ignoring the edge costs is that when the edge costs are less than or equal to 1 as is the case with the real-life networks used in these experiments, the shortest paths in the unweighted, binary version of the network might have paths with lower number of hops as demonstrated in the example network depicted in Figure 45.

In addition to potentially changing the structure of shortest paths and the shortest distances, ignoring the edge costs might lead to a higher number of equivalent shortest paths in the network, resulting in a higher percentage of affected nodes when an incremental growing network update is issued on the network. While the chances are lower to get multiple paths with exactly the same cost when the edge costs come from a large range of values, it is relatively easier to get multiple paths of the same cost when the path costs are determined in terms of number of hops. As demonstrated in Table 27, the most dramatic changes in the percentages of the affected nodes are observed in the SocioPatterns and HEP Co-authorship networks. This is because, the percentages of affected nodes are especially higher for the growing network updates on the better-connected networks with higher clustering coefficients and undirected edges where there are more redundant paths and a higher number of nodes are reachable from each node.

The redundancy in the shortest paths reflects as a lower percentage of the affected nodes in the SocioPatterns, Twitter, and Email networks because removal of an edge from one of the shortest paths does not necessarily change the shortest distances between nodes, as there might be other shortest paths to fall back.

In the HEP Co-authorship network, the percentage of the affected nodes is also higher for the shrinking network updates, which can be interpreted as a certain feature (e.g. funneling) of the shortest path structure in the coauthorship networks. *Funneling* is a concept that has been discussed in the context of co-authorship networks. Newman found that most of the shortest paths to a particular scientist pass through a disproportionately small number of his or her collaborators [174]; this phenomenon was later termed as "funneling" [175]. When funneling is observed frequently, the shortest paths mostly converge at some point (i.e. at a funneling node), which results in the change of most of the shortest paths in the case of decremental updates. This can also be observed from the higher percentage of affected nodes for the shrinking network updates in HEP Co-authorship network where funneling is an important concept.

8.4.2 Incremental Betweenness Centrality Performance Results

Next, the performance of the incremental betweenness centrality algorithm is discussed. Table 28 and Table 29 present the performance results of the incremental betweenness algorithm collected on real life networks. In the performance results presented in Table 28 the networks were modeled as weighted networks while Table 29 presents performance results collected on the unweighted (binary) versions of the same networks. Both Table 28 and Table 29 provide the percentages of the total number of nodes that are affected by the network updates.

Table 28 - Performance improvements of the incremental betweenness algorithm over repeated invocations of the Brandes' algorithm, collected on real-life networks described in Chapter 7.2. Information on the affected portion of the network is also provided. The results presented in this table are obtained on the weighted versions of the networks.

	Growing Network Updates		**Shrinking Network Updates**	
Network	**Betweenness Speedup**	**Betweenness Affected %**	**Betweenness Speedup**	**Betweenness Affected %**
SocioPatterns	113.67 x	2.64	24.66 x	12.63
Twitter (Iran)	8429.88 x	0.32	6681.51 x	0.33
Email	217.79 x	13.54	163.70 x	14.41
HEP Coauthorship	233.27 x	33.55	160.71 x	27.39
P2P	53722.49 x	0.04	29753.69 x	0.04

Similar to the incremental closeness performance results, the highest performance benefits are obtained on P2P communication network and the Iran retweet network while the SocioPatterns dataset provides the lowest speedup. However, the general trend is that the performance benefits increase with the increasing network size for networks of similar structures.

In addition, the impact of real social community structure and how this affects the affected percentages of the networks is more visible for the incremental betweenness centrality (Table 28) than it is for the incremental closeness centrality (Table 26). Comparing the affected percentages of the networks provided in both tables, it is observed that the percentage of a network that is affected by the incremental betweenness centrality updates is usually higher than the percentages affected by the incremental closeness centrality updates. As mentioned earlier, this is because incremental betweenness centrality handles changes both in the length and the number of the shortest paths in a network while the incremental closeness centrality algorithm focuses only on the changes in the lengths of the shortest paths.

When the performances of both incremental centrality algorithms are compared, the performance benefits of the incremental closeness centrality algorithm is higher than that of incremental betweenness centrality, which is in line with the results obtained on the synthetic networks. Similar observations hold for the performance results presented in Table 29. The differences in the interpretation and scale of the performance results presented in Table 29 against the performance results presented in Table 28 stem from the structural differences in the shortest paths as discussed in the interpretation of performance differences for incremental closeness centrality algorithm in Chapter 8.4.1.

Table 29 - Performance improvements of the incremental betweenness algorithm over the repeated invocations of the Brandes' algorithm, collected on real-life networks. Information on the affected portion of the network is also provided. The results presented in this table are obtained on the unweighted (binarized) versions of the networks.

	Growing Network Updates		**Shrinking Network Updates**	
Network	**Betweenness Speedup**	**Betweenness Affected %**	**Betweenness Speedup**	**Betweenness Affected %**
SocioPatterns	8.73 x	44.14	8.12 x	44.17
Twitter (Iran)	8442.96 x	0.32	6734.63 x	0.33
Email	14.05 x	42.55	12.57 x	42.58
HEP Coauthorship	64.66 x	44.66	48.16 x	44.64
P2P	54536.88 x	0.04	29428.90 x	0.04

The performance values and the percentage of affected nodes are different in unweighted versions of the networks, when compared to the results collected on their weighted versions. Due to increased redundancy in the number of shortest paths in SocioPatterns, Email, and HEP Coauthorship networks, the performance improvements of the incremental betweenness algorithm over the Brandes' algorithm has dropped down while the percentages of the affected nodes have increased substantially. On the other hand, the performance results remain similar in weighted and unweighted version of the Twitter and P2P networks where the redundancy of shortest paths is not a significant factor.

When comparing the performance results obtained for the incremental closeness (Table 27) and incremental betweenness (Table 29) algorithms on the unweighted versions of the networks, it is observed that the percentages of affected nodes for shrinking network updates demonstrate different behaviors. This is observed due to differences in the definitions of closeness and betweenness centrality metrics. In betweenness, the number of equivalent shortest paths from one node to another is important while closeness only need information on the shortest distances. Assume that an edge e which lies on a shortest path from a node x to another y is removed. When one of the shortest paths from a node x to node y changes, the betweenness values of all the intermediate nodes on the shortest paths from node x to node y change. However, in closeness centrality, when one of the shortest paths from node x to y changes, if there are other shortest paths which would keep the shortest distance from node x to y as is, then the incremental network update concludes without adding more nodes into the set of affected nodes. Hence, the sets of affected nodes for the incremental betweenness algorithm handling shrinking network updates are substantially larger than those for the incremental closeness algorithm handling shrinking network updates. In addition, due to the reasons discussed above, unlike incremental closeness centrality algorithm, the percentages of the affected nodes for the incremental betweenness algorithm handling shrinking network updates are comparable to those for the incremental betweenness algorithm handling growing network updates.

8.4.2.1 Additional Notes on the Performance of the Incremental Betweenness Centrality Algorithm on Co-authorship Dataset

An accepted way of evaluating the performance of the incremental algorithms in the literature is to issue a number of random incremental updates on the network and to report the average performance across all the random updates [56]. When a batch of random updates is issued on a network, there is an implicit assumption that all random updates are equally likely to be observed. However, this is usually not the case for the real-life networks whose properties are studied in depth in prior literature. Certain types of updates are more likely to occur than the others in real life networks.

For instance, authorship networks have well-understood properties that are previously documented in the literature [169] [15]:

- Links in co-authorship networks are reciprocal (*i.e.*, symmetric).
- The link weights between two authors in co-authorship networks can increase over time if two authors have further collaboration.
- Vertices and edges added to the citation/co-authorship networks are permanent and cannot be removed at a later time.
- The already formed part of the network is mostly static; in most cases, only the leading edge of the network changes over time.

A lot of real life networks, including authorship networks, have time information for every piece of action taken. For instance, an authorship network have such information available from the date/year the authors publish a paper. When the entire lifetime of a network is considered, there are usually different social agents that are active during different periods of time. For instance, when an authorship network for a certain research field is considered, there are usually new people that join the network, who are likely to be M.S. or Ph.D. students that start their graduate school career. In co-authorship networks, a lot of the authors publish only a couple of papers during their graduate school career and then their publications cease. On the other hand, there are usually a core set of academicians that publish a substantial number of papers, and form the shortest paths at the core of the network. Such core-authors usually tend to be the authors that have published a lot of papers together and established the research field. The core-authors also stop publishing after a certain number of years. However, since the relationships between them are very strong compared to the authors in the rest of the network, the paths that pass through the core set of authors tend to remain as the shortest paths in the network for a long time.

Depending on the phase an authorship network is in, these observations might have different implications on the shortest paths in the network and how an incremental update propagates in a co-authorship network. For instance, if an authorship network is in early growth stages, the shortest paths are likely to be modified every time a new edge/node is inserted. However, when the research field is more established, it is likely that a relationship between two newer authors will not affect all the shortest paths in the network; which causes the incremental network update not to propagate too far in the network.

Another likely-to-occur update type is a new node joining a network, which represents, for instance, a graduate student. Usually, such new nodes join the network through a funneling node (which is likely to be the student's advisor) through which all the shortest paths to/from this new node pass [174]. When such a new node joins the network, the shortest paths to/from this new node should be discovered from scratch, which usually affects the entire component the funneling node (potentially the node representing the advisor) is a part of. This situation potentially increases the percentage of the network affected by this incremental network update.

Instead of inserting 100 edges that are selected randomly out of the entire network topology, when we issue last 100 updates on the same authorship network in the order the updates arrive at the network in real life, the performance of the incremental betweenness centrality algorithm over the Brandes' betweenness algorithm is different. For instance, when the last 100 updates are issued on the HEP authorship network considering the timestamps of the network updates, the incremental betweenness algorithm runs 357.96 times faster than the Brandes' algorithm and 42.08% of the network is affected on average. However, when the 100 updates are randomly selected out of all network updates, the incremental betweenness algorithm performs 233.27 times faster than the Brandes' betweenness algorithm and the 33.55% of the network is affected on average as presented in Table 28.

As one last point, since the published papers are archived permanently, in authorship networks, only growing network updates are observed: insertion of new edges and nodes, and edge cost decrease. No shrinking network updates are observed for the authorship networks in real life.

8.4.2.2 Performance Comparison against the QuBE Algorithm

Next, the performance of the proposed incremental betweenness algorithm is compared against the performance of a recent betweenness algorithm called QuBE [56]. The idea of the QuBE algorithm [56] depends on estimating the nodes whose betweenness values might change due to an update in a network while avoiding computation of all-pairs shortest paths. In contrast, the incremental betweenness algorithm proposed in this dissertation depends on the dynamic maintenance of all-pairs shortest paths and the related auxiliary data, which might be useful in other problems beyond betweenness as discussed earlier in Chapter 5.6.

The QuBE algorithm covers edge insertions/deletions, leaving out node insertions and deletion and edge cost modifications for weighted network types. In contrast, our algorithm supports edge cost modifications for the weighted networks as well as support for node insertion and removal.

Providing support for weighted networks makes the algorithm design harder as it introduces several additional cases to be handled. Consider the following simple example. Assume that there is a path from node x to node y. Then, an incremental network update inserts a direct edge from node x to node y. In binary networks, it is obvious that no path from node x to node y can be smaller than a direct edge from x to y, and several changes on the shortest paths can be kept up to date only by accounting for the number of hops. However, in weighted networks, when an edge $\{x \rightarrow y\}$ from node x to node y is inserted, it is not necessarily given that this newly inserted edge will be on the shortest paths. Depending on the cost of the edge $\{x \rightarrow y\}$ and the previously known shortest path length from node x to node y, it may or may not be. Hence, it will still be necessary to check the paths of equivalent length before ruling out all previously known shortest paths between x and y due to the insertion of a direct edge $\{x \rightarrow y\}$, which increases the complexity of the algorithm design.

Next, the performance of the incremental betweenness centrality algorithm proposed in this dissertation is compared against the QuBE algorithm using the same datasets Lee et al. used in their QuBE paper [56]. We select two of their datasets: the dataset on which QuBE performs the best (Eva), and the dataset on which QuBE performs the worst (CAGrQc).

Table 30 - Performance comparison of QuBE and our proposed algorithm.

Network	**Type**	**#(Node)**	**#(Edge)**	**QuBE**	**Incremental Betweenness**
Eva [176]	Ownership	4457	4562	2418.17	25425.87
CAGrQc [177]	Collaboration	4158	13422	2.06	67.86

Table 30 reports the average performance results for 100 random updates on the networks. Both the QuBE algorithm and the incremental betweenness centrality algorithm proposed in this dissertation are compared against the Brandes' algorithm as baseline. Our algorithm performs 10-30 times better than the QuBE algorithm while providing substantial improvements over the Brandes' algorithm.

8.4.3 Incremental *k*-Centrality Performance Results

Next, the performances of the incremental *k*-closeness and *k*-betweenness centrality algorithms on real life networks are discussed. For the performance evaluations with the incremental *k*-centrality algorithms, the unweighted (binary) versions of the real life networks are used. In the performance results presented in Table 31 and Table 32, the bounding parameter k is set as $k = 2$ and $k = 3$, respectively. The baseline algorithms (the Dijkstra's algorithm [109] for closeness centrality and the Brandes' algorithm [28] for betweenness centrality) are also bounded by k hops. The experiments are designed in a similar fashion to those in Chapter 8.4.1 and Chapter 8.4.2 (i.e. 100 random edge insertions and their removals). Hence, to avoid repetition, they are not described here again.

Table 31 - Performance benefits of the incremental *k*-centrality algorithms obtained on real-life networks and the portions of the networks affected by these changes ($k = 2$).

	Growing Network Updates ($k = 2$)				**Shrinking Network Updates ($k = 2$)**			
Topology	**Closeness Speedup**	**Closeness Affected%**	**Betweennes s Speedup**	**Betweenness Affected %**	**Closeness Speedup**	**Closeness Affected %**	**Betweenness Speedup**	**Betweenness Affected %**
Socio Patterns	616.86 x	64.21	10.17 x	43.85	471.55 x	1.90	12.66 x	43.90
Twitter (Iran)	7985.74 x	0.31	9808.72 x	0.29	4259.78 x	0.29	7876.06 x	0.31
Email	6114.62 x	17.51	327.61 x	3.72	5691.02 x	1.45	118.77 x	27.85
HEP Coauthor	233476.79 x	1.15	38011.02 x	0.22	224353.97 x	0.15	6958.46 x	1.54
P2P	312118.62 x	0.04	53980.70 x	0.03	21023.51 x	0.03	26442.56 x	0.04

Both Table 31 and Table 32 provide performance results and percentages of the affected nodes both for the incremental *k*-closeness and the incremental *k*-betweenness algorithms. To understand how the incremental *k*-centrality algorithms improve over the incremental centrality algorithms, the performance results presented in this section (Table 31 and Table 32) should be compared against the performance results presented earlier in Table 27 (incremental closeness centrality performance results collected over the *unweighted* versions of the real-life networks) and Table 29 (incremental betweenness centrality performance results collected over the *unweighted* versions of the real-life networks).

Table 32 - Performance benefits of the incremental *k*-centrality algorithms obtained on real-life networks and the portion of the networks affected by these changes (*k* = 3).

	Growing Network Updates (*k* = 3)				**Shrinking Network Updates (*k* = 3)**			
Topology	**Closeness Speedup**	**Closeness Affected%**	**Betweenness Speedup**	**Betweenness Affected %**	**Closeness Speedup**	**Closeness Affected %**	**Betweenness Speedup**	**Betweenness Affected %**
Socio Patterns	603.89 x	64.21	8.23 x	44.14	466.58 x	1.90	9.66	44.17
Twitter (Iran)	5902.69 x	0.32	9764.90 x	0.31	3807.37 x	0.31	6949.98	0.33
Email	4558.35 x	19.97	34.04 x	30.26	2224.55 x	3.52	30.69	39.74
HEP Coauthor	35248.73 x	4.63	3640.68 x	1.55	23075.61 x	0.78	845.69	7.35
P2P	290828.22x	0.04	52633.31 x	0.04	20027.29 x	0.04	29980.52	0.04

First, the performance of the incremental *k*-closeness centrality algorithm is discussed. Considering performance results presented in Table 31 (*k* = 2) and Table 32 (*k* = 3), it is observed that performance improvements of the incremental *k*-centrality algorithms are higher when the bounding parameter *k* is lower in general. However, the specific performance values are affected by the amount of work that can be avoided by setting *k* to a lower value. For instance, in the Email and HEP Coauthorship networks, the performance difference is dramatic when k = 3 or k = 2. The average shortest path length for the Email network is 3.2 while the HEP coauthorship network has an average path length of 5.76. Since the Email and HEP Coauthorship networks tend to have longer shortest paths, this as a result, causes a large fraction of nodes to be affected. Hence, bounding the shortest paths within k hops helps improving the performance substantially, and the performance difference is also visible when k is updated from k = 3 to k = 2. However, the other real-life networks (especially the P2P network) have relatively low average shortest path lengths, less than 2.0. The average path length is 1.65 for the SocioPatterns network, 1.4 for the Iran Twitter network, and 1.24 for the P2P communication network. Hence, although there are slight improvements with the limiting k value decreasing, the performance is not all that different when k is updated from k = 3 to k = 2.

The percentage of affected nodes remain the same as the incremental closeness centrality (no limiting k is set) when k = 3, and have small reductions when k = 2 for the SocioPatterns, Twitter (Iran), and P2P networks. Considering the average shortest path lengths are less than 2.0, such an outcome is reasonable. However, we do observe substantial reductions in the size of the affected nodes set for the Email and HEP Coauthorship network. For instance, for the part of the incremental closeness algorithm that handles the growing network updates, the percentage of affected nodes is 23.55% of the HEP Coauthorship network. This percentage reduces to 4.63% when k = 3, and to 1.15% when k = 2. Hence, the performance improvements of the incremental closeness algorithm over the corresponding versions of the Dijsktra's algorithm increase from

4395.16x (over the Dijkstra's algorithm) to 35248.73x (over the Dijkstra's algorithm with $k = 3$), and to 233476.79x (over the Dijkstra's algorithm with $k = 2$), respectively.

Similar observations hold for the performance results for the incremental k-betweenness centrality. The performance and the percentage of affected nodes remain similar to the incremental betweenness centrality with no limiting k in SocioPatterns, Twitter (Iran), and P2P networks while the performance improvements again become larger with decreasing k. The most dramatic performance improvements for the incremental k-betweenness centrality are observed for the HEP coauthorship network, which is larger than the other networks, and benefit more from k-hop limit as its shortest paths tend to be longer on average. Comparing the performance results of the incremental k-closeness and incremental k-betweenness algorithms, the incremental k-closeness algorithm provides higher performance improvements over its baseline because it has fewer pieces of information to maintain and the percentage of affected nodes are lower.

8.5 SUMMARY: PERFORMANCE TRENDS

This section discusses how the performances of the incremental centrality algorithms scale with varying network sizes and different network features. First, the performance scaling on synthetic networks is discussed in Chapter 8.5.1 and Chapter 8.5.2. Then, the performance scaling on real-life networks is discussed in Chapter 8.5.3 and Chapter 8.5.4. Finally, Chapter 8.5.5 discusses the performance trends observed in real-life and synthetic networks in comparison to one another.

Chapter 8.5.1 and Chapter 8.5.3 discuss performance scaling of the incremental centrality algorithms with respect to network size while Chapter 8.5.2 and Chapter 8.5.4 discuss performance scaling considering other network features. Performance scaling with respect to the network size is presented and discussed separately because the network size is the main driving factor for the performance scaling and the memory consumption.

8.5.1 Network Size and Performance Trends in Synthetic Networks

This subsection analyzes the performance trends of the incremental centrality algorithms with respect to the network size on synthetic networks.

Earlier in Chapter 8.3.2, the performance values of the incremental betweenness centrality algorithm were presented in Table 13 for the growing network updates for preferential attachment, Erdos-Renyi, small world networks, and directed cycles. Figure 46 visualizes the same information and presents the performance values in logarithmic scale. Similarly, Figure 47 presents the performance of the incremental closeness centrality algorithm based on the data in Table 11 using logarithmic scale for the performance values.

Since preferential attachment performance improvements are at a different scale, when they are displayed in the same figure with the other network types, the trends for the other network types are not clearly visible. Hence, the performance values are drawn in logarithmic scale. In these figures (Figure 46 and Figure 47), the *x*-axis shows the number of nodes and *y*-axis shows the log-10 of the speedup obtained by the incremental centrality algorithm over its non-incremental, traditional baseline algorithm.

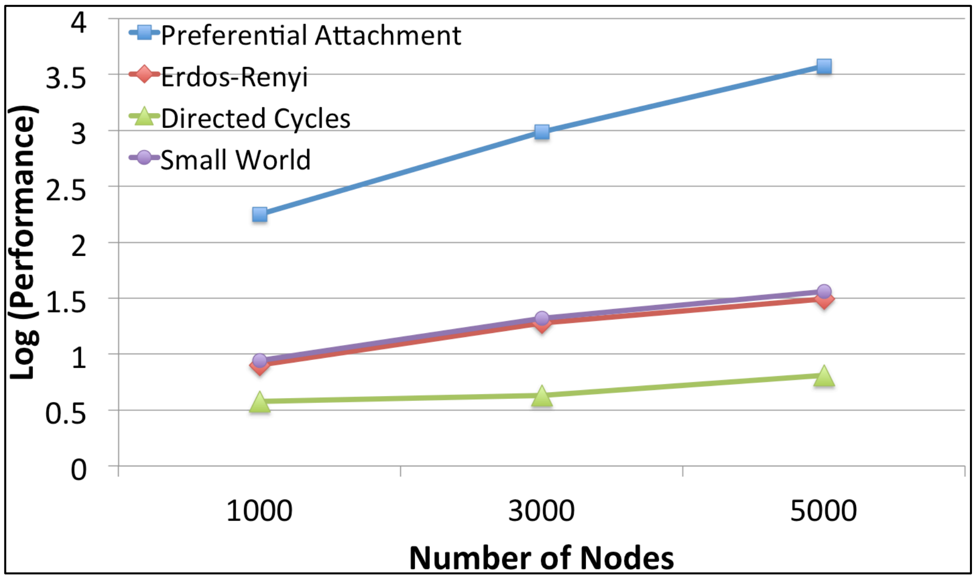

Figure 46 - Performance of the incremental betweenness centrality algorithm for the data in Table 13. Growing network updates on the preferential attachment, small world, Erdos-Renyi networks, and directed cycles are analyzed.

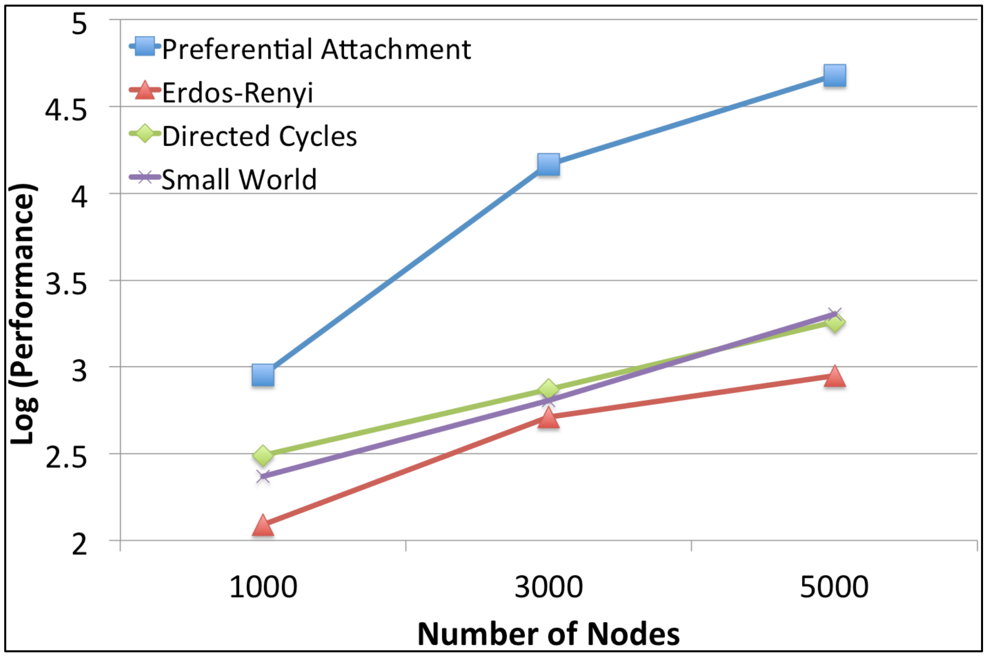

Figure 47 - Performance of the incremental closeness centrality algorithm for the data in Table 11. Growing network updates on the preferential attachment, small world, Erdos-Renyi networks, and directed cycles are analyzed.

Below is a list of the equations for the curves fitted to the data presented in Figure 46 and Figure 47. For clarity, the regression equations are listed below. These equations are obtained by plugging the network size (in terms of the number of nodes) as the *x*-values (e.g. 1000, 3000, and 5000) and the actual performance improvement values (not the logarithmic) as the *y*-values:

Polynomial curves fitted on the performance of the incremental closeness centrality algorithm:

- $y = 0.0024x^2 - 2.698x + 1196.8$ (Preferential Attachment)
- $y = 0.0001x^2 - 0.2775x + 390$ (Small World)
- $y = 8\text{e-}05x^2 - 0.11x + 337.63$ (Directed Cycles)
- $y = -2\text{e-}06x^2 + 0.2045 - 79.375$ (Erdos-Renyi)

Polynomial curves fitted on the performance of the incremental betweenness centrality algorithm:

- $y = 0.0002x^2 - 0.6015x + 530$ (Preferential Attachment)
- $y = 4\text{e-}07x^2 + 0.0045x + 3.88$ (Small World)
- $y = 2\text{e-}07x^2 - 0.0006x + 4.1888$ (Directed Cycles)
- $y = 0.0058x + 2.0833$ (Erdos-Renyi)

The regression equations obtained for the performance of the incremental closeness and betweenness centrality algorithms resemble one another. However, the constants for the curves fitted to the incremental closeness centrality algorithm are larger than those in the curves fitted to the performance of the incremental betweenness centrality. All equations are polynomial and are selected to fit all the data points in the curves.

Finally, for the network size data points analyzed in this dissertation, the general tendency is that, as the network sizes increase, the densities of the networks tend to get smaller and the performance benefits of the incremental centrality algorithms increase. However, if the network to be analyzed becomes very large, then the hardware capacity and the requirements for the memory consumption would govern the limits of this trend.

8.5.2 Network Features and Performance on Synthetic Networks

Next, in Chapter 8.5.2, we discuss how the performances of the incremental centrality algorithms scale with the other network features. To be able to analyze these aspects, we primarily benefit from the information provided in the network statistics tables in Chapter 7.1.5 and the performance results presented earlier in this section on synthetic networks.

In particular, we select some of the network features that are expected to have an impact on the performance of the incremental centrality algorithms such as the average shortest path length, diameter, and the clustering coefficient and draw their scatter plots against the performance values. More precisely, we generate multiple scatter plots using the average shortest path length vs. the performance of the incremental betweenness centrality algorithm or the diameter vs. the performance of the incremental closeness centrality algorithm, and so on. Once we have the scatter plots ready, we perform regression analysis and try fitting curves on them to be able to comment on the general trends observed.

Table 33 - Network statistics for varying network sizes accompanied by the performance improvements of the incremental closeness centrality algorithm. The average degree is 6.

Topology	Size	Density	Avg. Btw	Min Deg	Max Deg	Diameter	Char. Path Length	Clustering Coefficient	Closeness Speedup
Pref. Attach.	1000	0.003	94.37	3	89	10	3.45	0.014	900
Pref. Attach.	3000	0.001	197.59	3	233	14	4.126	0.007	14714.43
Pref. Attach.	5000	0.0006	292.48	3	212	16	4.442	0.005	47738.81
Erdos-Renyi	1000	0.003	4777.28	1	14	15	6.305	0.003	123.07
Erdos-Renyi	3000	0.001	18136.74	2	13	14	7.086	0.001	515.35
Erdos-Renyi	5000	0.0006	32073.39	2	11	14	7.492	0.001	890.56
Small World	1000	0.003	1809.59	3	11	8	4.623	0.212	233.49
Small World	3000	0.001	6617.851	3	11	9	5.413	0.208	642.45
Small World	5000	0.0006	12079.138	3	12	9	5.833	0.216	2015.87
Directed Cycles	1000	0.003	3544.12	3	11	38	8.38	0.106	309.62
Directed Cycles	3000	0.001	14986.28	3	11	57	11.37	0.104	740.44
Directed Cycles	5000	0.0006	28762.20	3	12	77	12.98	0.108	1822.83

Table 34 - Network statistics for varying average node degrees accompanied by the performance improvements of the incremental betweenness centrality algorithm. The average degree is 6.

Topology	Size	Density	Avg. Btw	Min Deg	Max Deg	Diameter	Char. Path Length	Clustering Coefficient	Betweenness Speedup
Pref. Attach.	1000	0.003	94.37	3	89	10	3.45	0.014	178.65
Pref. Attach.	3000	0.001	197.59	3	233	14	4.126	0.007	971.40
Pref. Attach.	5000	0.0006	292.48	3	212	16	4.442	0.005	3760.48
Erdos-Renyi	1000	0.003	4777.28	1	14	15	6.305	0.003	7.99
Erdos-Renyi	3000	0.001	18136.74	2	13	14	7.086	0.001	18.97
Erdos-Renyi	5000	0.0006	32073.39	2	11	14	7.492	0.001	31.18
Small World	1000	0.003	1809.59	3	11	8	4.623	0.212	8.77
Small World	3000	0.001	6617.851	3	11	9	5.413	0.208	20.94
Small World	5000	0.0006	12079.138	3	12	9	5.833	0.216	36.32
Directed Cycles	1000	0.003	3544.12	3	11	38	8.38	0.106	3.78
Directed Cycles	3000	0.001	14986.28	3	11	57	11.37	0.104	4.26
Directed Cycles	5000	0.0006	28762.20	3	12	77	12.98	0.108	6.47

Combining the information provided on Table 6 (network statistics for synthetic networks –varied by size–) and Table 11 (performance results for the incremental closeness centrality algorithm), Table 33 is obtained. Similarly, Table 34 combines information from Table 6 and Table 13 (performance results for the incremental betweenness centrality algorithm).

Using the information provided in Table 33 and Table 34, I have generated multiple scatter-plot figures to show how the performance improvements change with the changing network parameters. The scatterplots (Figure 48 - Figure 71) are accompanied by non-linear regression equations (trend lines) that would provide the best possible match, as close as possible to all data points in the graph. Although the actual performance values are different, the general performance trends for the incremental closeness and betweenness centrality algorithms are similar. Thus, I first provide figures comparing the performance values of the incremental closeness centrality algorithm (Figure 48 - Figure 59 for the information presented in Table 33) and then the figures for the performance of the incremental betweenness centrality algorithm (Figure 60 – Figure 71 for the information presented in Table 34) against the network features. To be consistent with the results presented earlier in Chapter 8.5.1 and to make the figures more visible, the *y*-axes are also drawn in logarithmic scale. For comparison purposes, the scatter plots with linear *y*-axes are also included. In particular, Figure 48 - Figure 53 show log-10 of the performance of the incremental closeness centrality algorithm versus various network features and Figure 54 - Figure 59 show their counterparts drawn in linear *y*-axes. Similarly, Figure 60 - Figure 65 show log-10 of the performance of the incremental betweenness centrality algorithm versus various network features and Figure 66 - Figure 71 show their counterparts drawn in linear *y*-axes.

For the data presented in Figure 48 - Figure 71, regression equations are fit in an attempt to better explain the trends. In the regression equations, *x*-value represents the network feature under investigation such as the clustering coefficient, the average shortest path length, or the network diameter while the *y*-value represents either the performance improvement or the log-10 of the performance improvement of the incremental centrality algorithms depending on the figure.

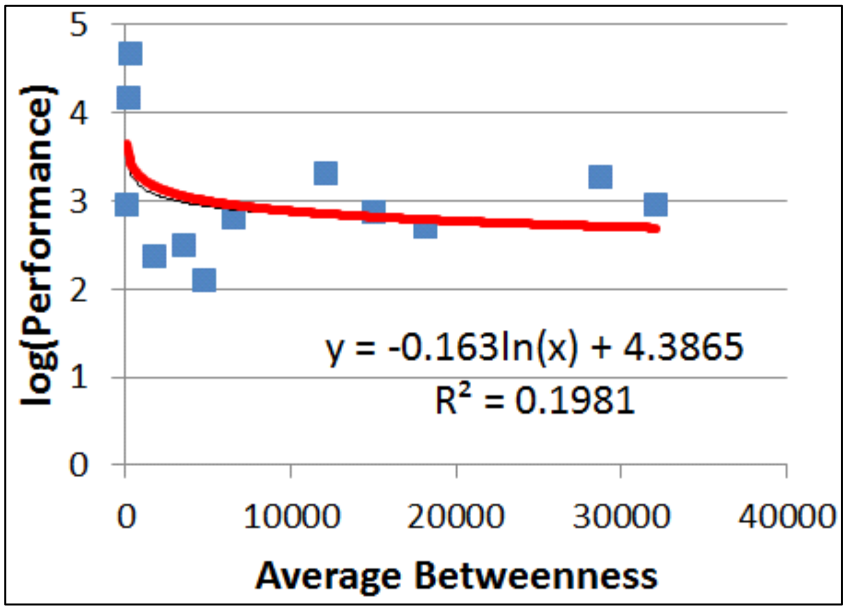

Figure 48 - Log-10 of the performance improvement of the incremental closeness centrality algorithm vs. Average betweenness centrality.

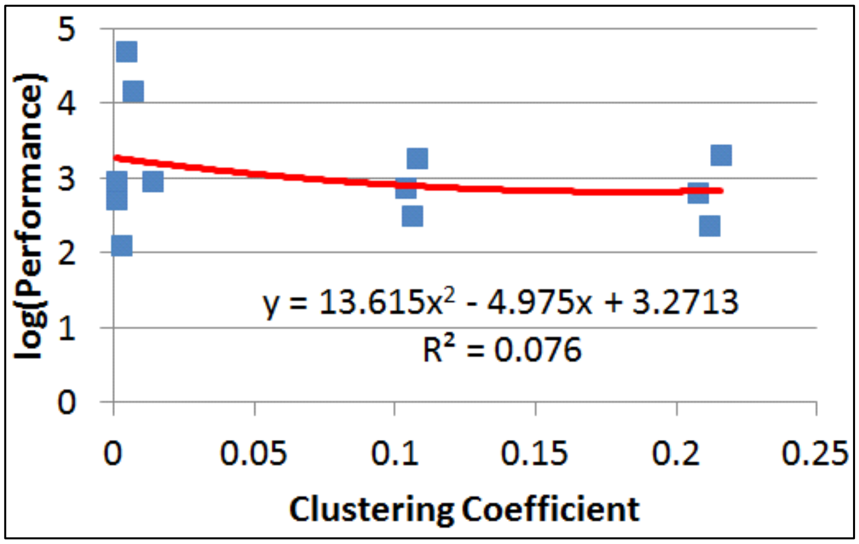

Figure 49 - Log-10 of the performance improvement of the incremental closeness centrality algorithm vs. Global clustering coefficient.

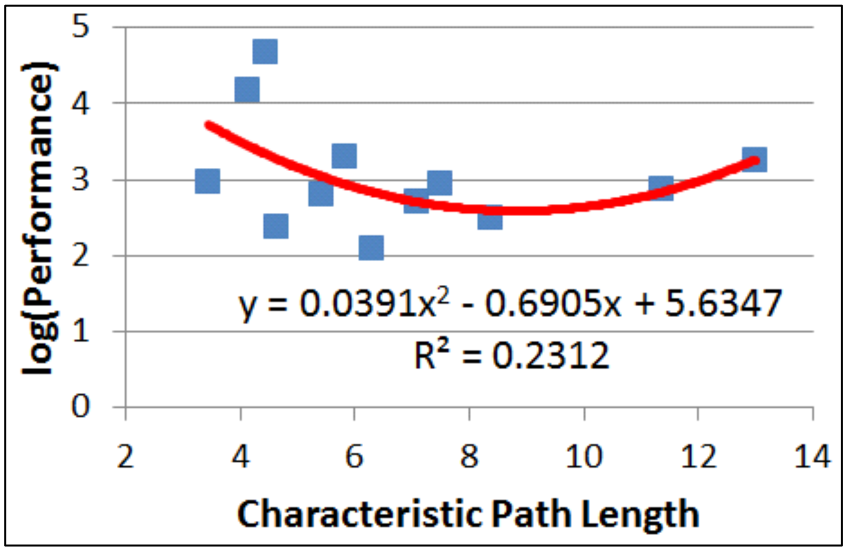

Figure 50 - Log-10 of the performance improvement of the incremental closeness centrality algorithm vs. Characteristic path length.

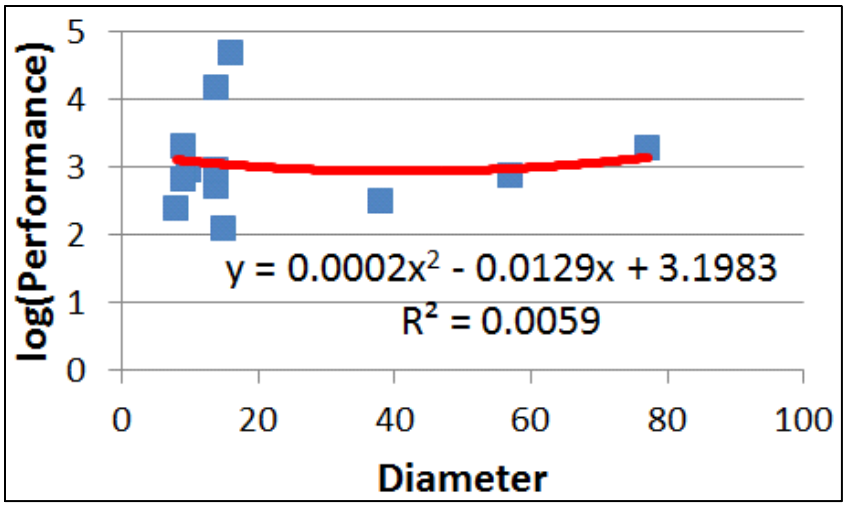

Figure 51 - Log-10 of the performance improvement of the incremental closeness centrality algorithm vs. Network diameter.

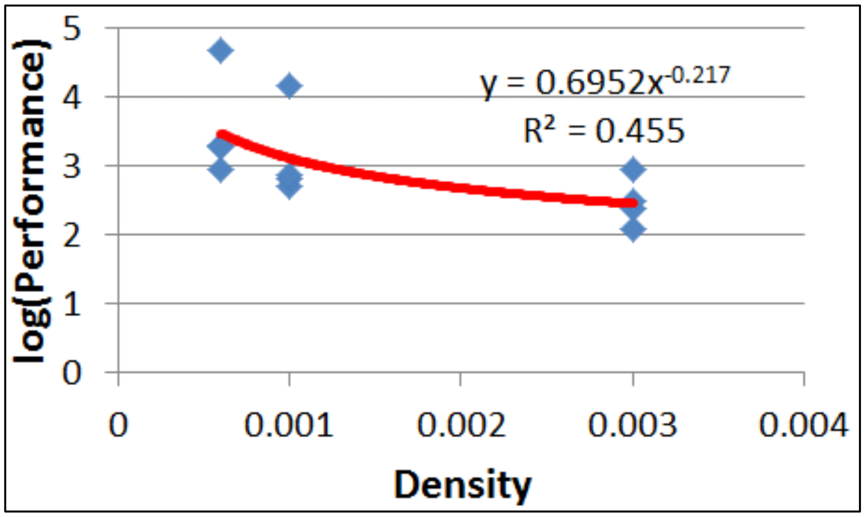

Figure 52 - Log-10 of the performance improvement of the incremental closeness centrality algorithm vs. Network density.

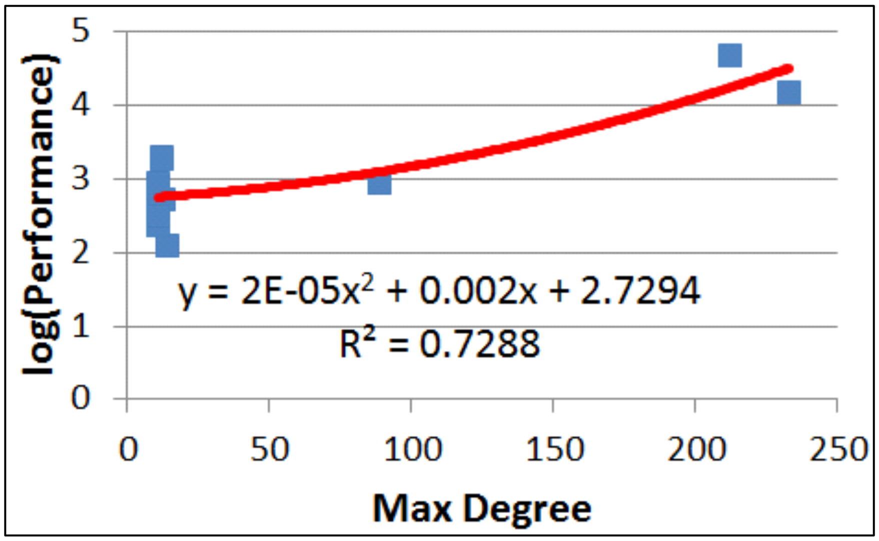

Figure 53 - Log-10 of the performance improvement of the incremental closeness centrality algorithm vs. Maximum node degree.

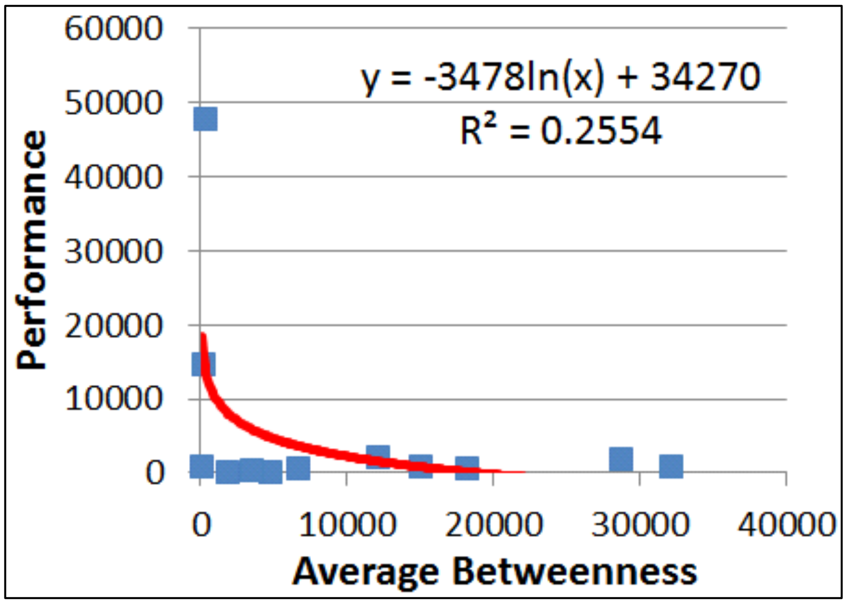

Figure 54 - Performance improvement of the incremental closeness centrality algorithm vs. Average betweenness centrality.

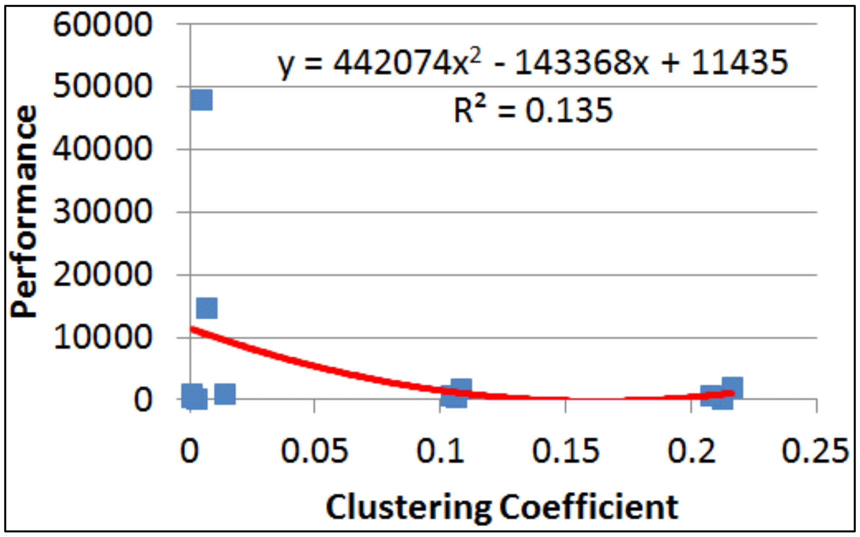

Figure 55 - Performance improvement of the incremental closeness centrality algorithm vs. Global clustering coefficient.

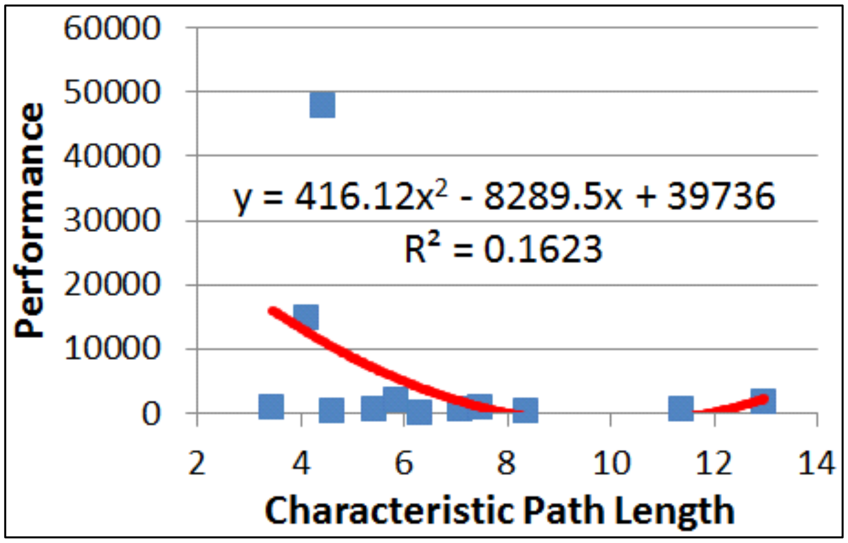

Figure 56 - Performance improvement of the incremental closeness centrality algorithm vs. Characteristic path length.

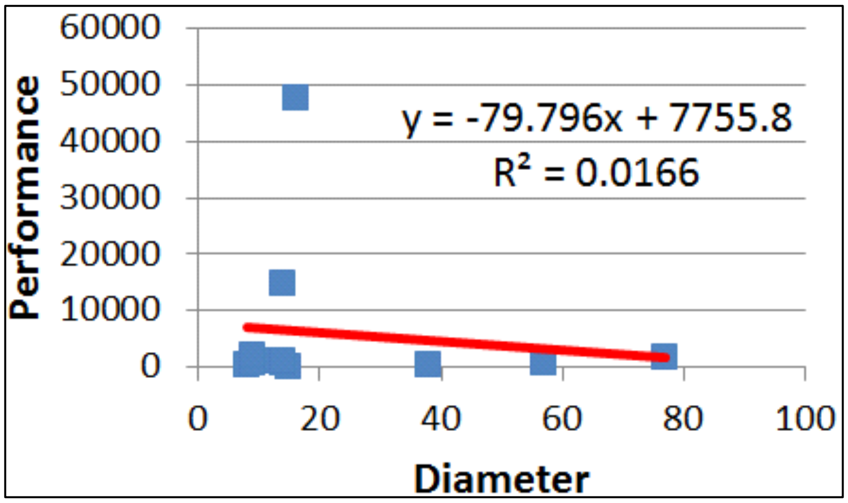

Figure 57 - Performance improvement of the incremental closeness centrality algorithm vs. Network diameter.

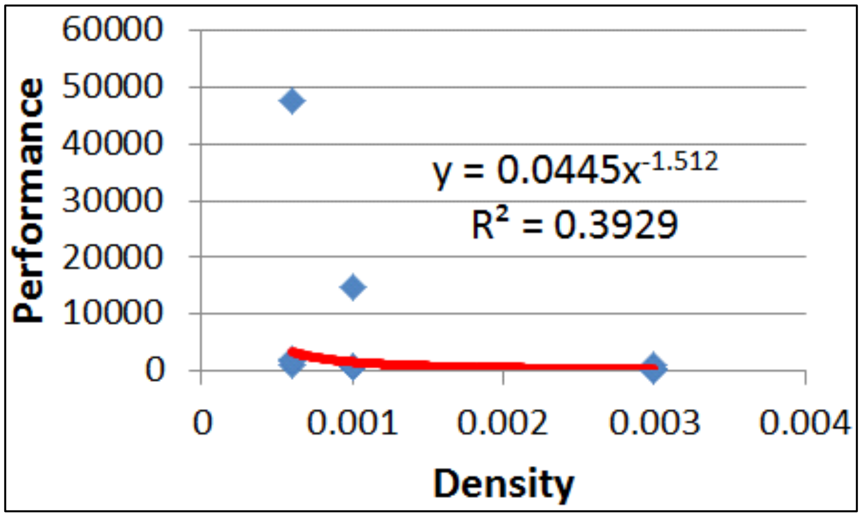

Figure 58 - Performance improvement of the incremental closeness centrality algorithm vs. Network density.

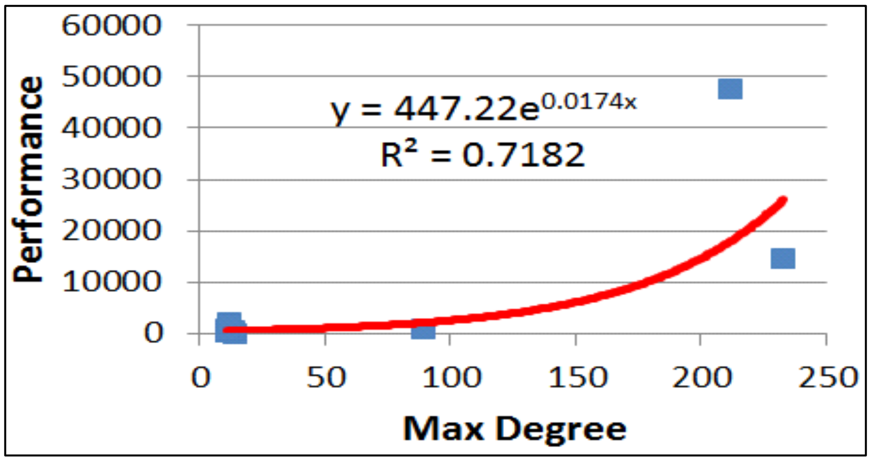

Figure 59 - Performance improvement of the incremental closeness centrality algorithm vs. Maximum node degree.

The next set of figures present the performance scaling of the incremental betweenness algorithm. First, the set of figures with logarithmic scale on the performance axes are presented (Figure 60 - Figure 65). Then, their counterparts with linear scale are presented for comparison purposes (Figure 66 - Figure 71).

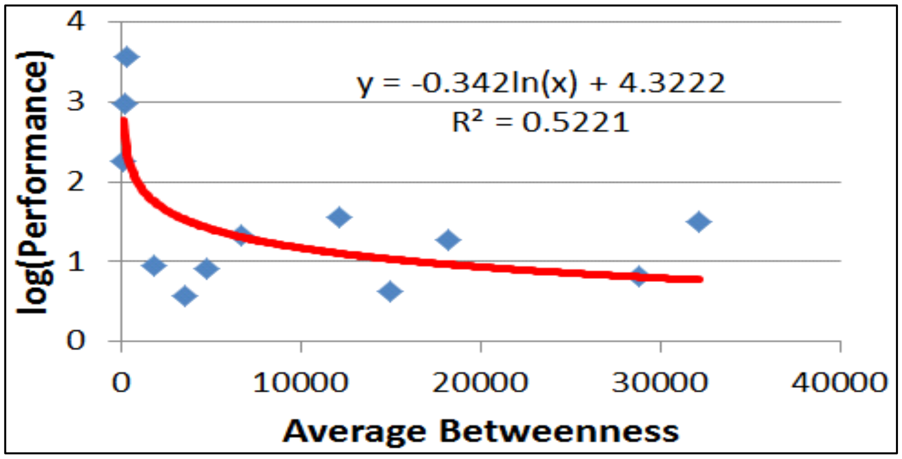

Figure 60 - Log-10 of the performance improvement of the incremental betweenness centrality algorithm vs. Average betweenness centrality.

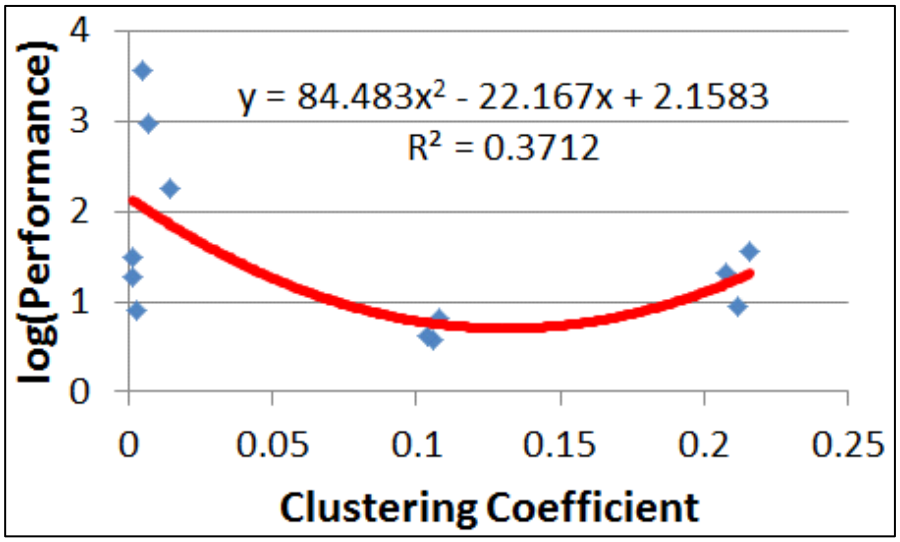

Figure 61 - Log-10 of the performance improvement of the incremental betweenness centrality algorithm vs. Global clustering coefficient.

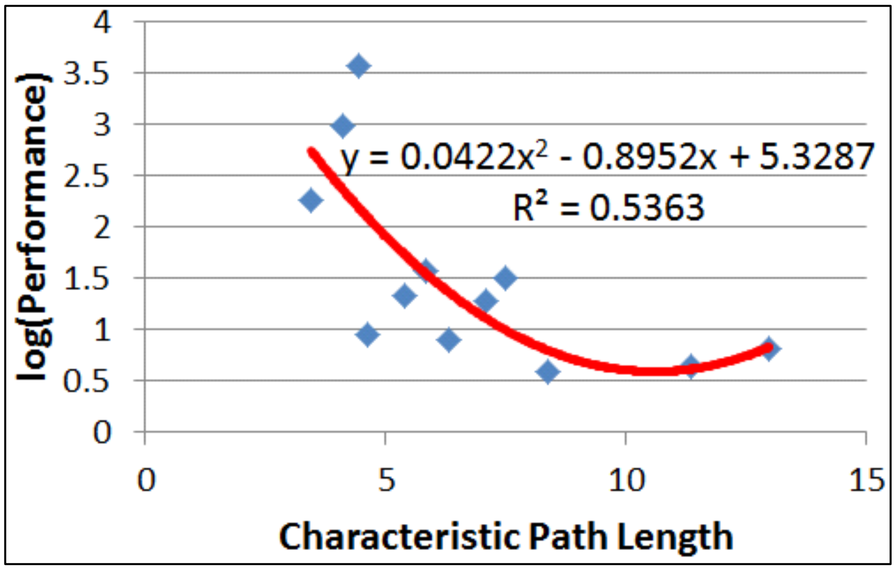

Figure 62 - Log-10 of the performance improvement of the incremental betweenness centrality algorithm vs. Characteristic path length.

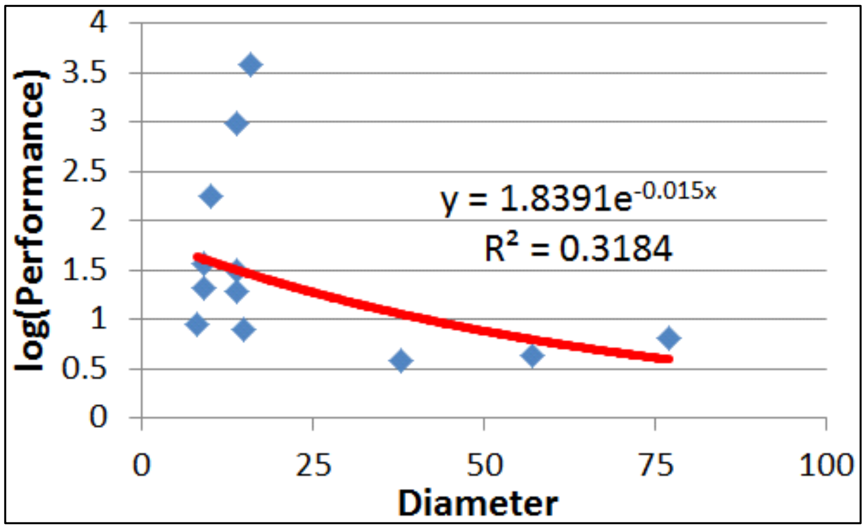

Figure 63 - Log-10 of the performance improvement of the incremental betweenness centrality algorithm vs. Network diameter.

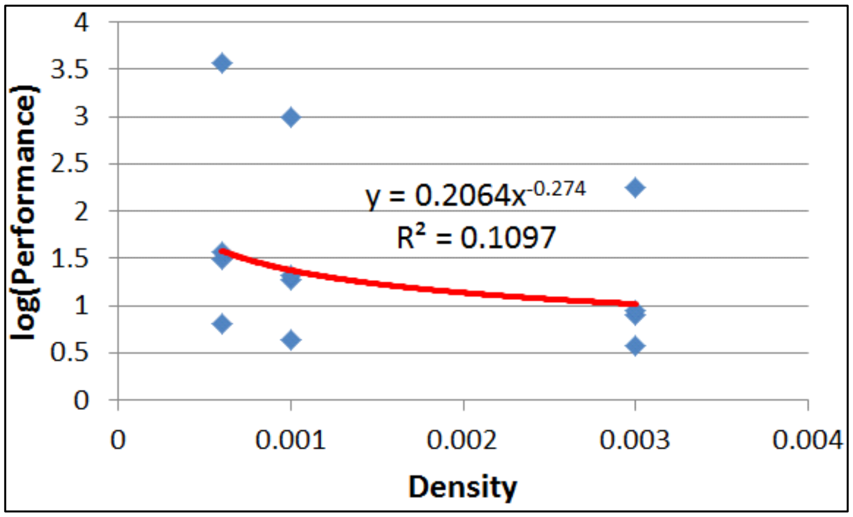

Figure 64 - Log-10 of the performance improvement of the incremental betweenness centrality algorithm vs. Network density.

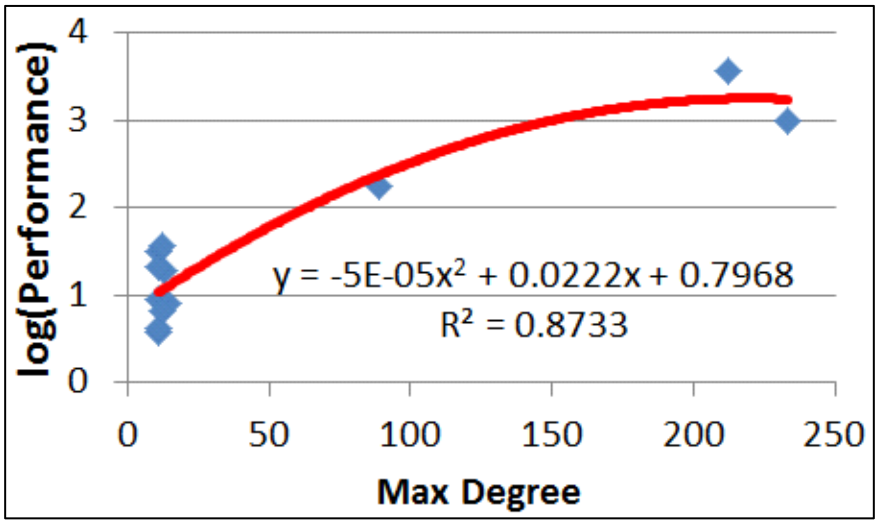

Figure 65 - Log-10 of the performance improvement of the incremental betweenness centrality algorithm vs. Maximum node degree.

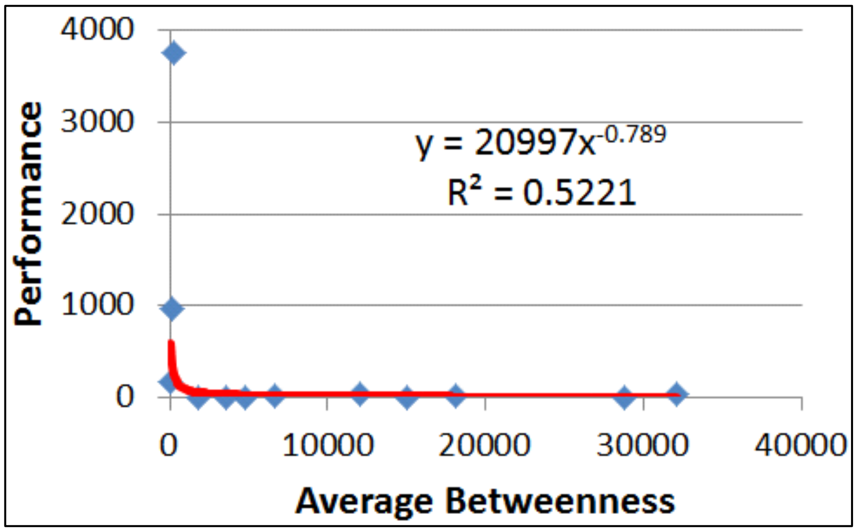

Figure 66 - Performance improvement of the incremental betweenness centrality algorithm vs. Average betweenness centrality.

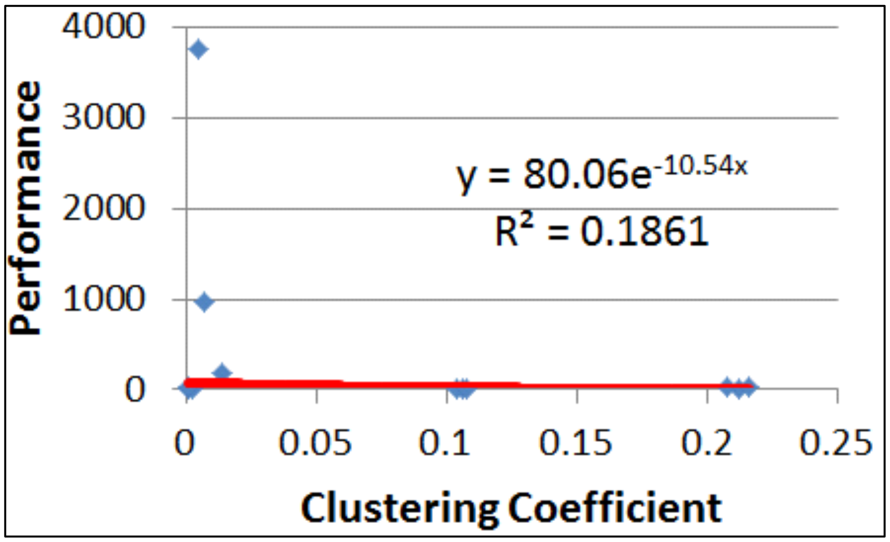

Figure 67 - Performance improvement of the incremental betweenness centrality algorithm vs. Global clustering coefficient.

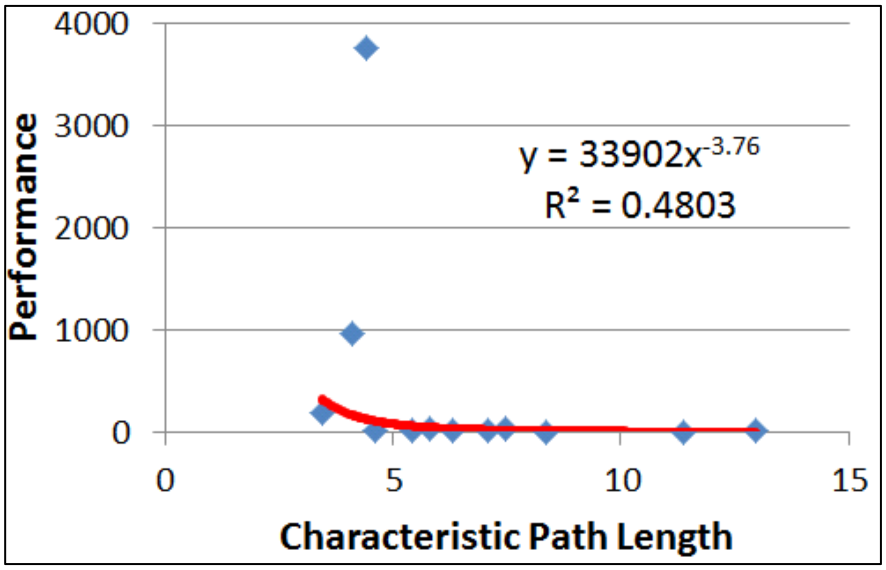

Figure 68 - Performance improvement of the incremental betweenness centrality algorithm vs. Characteristic path length.

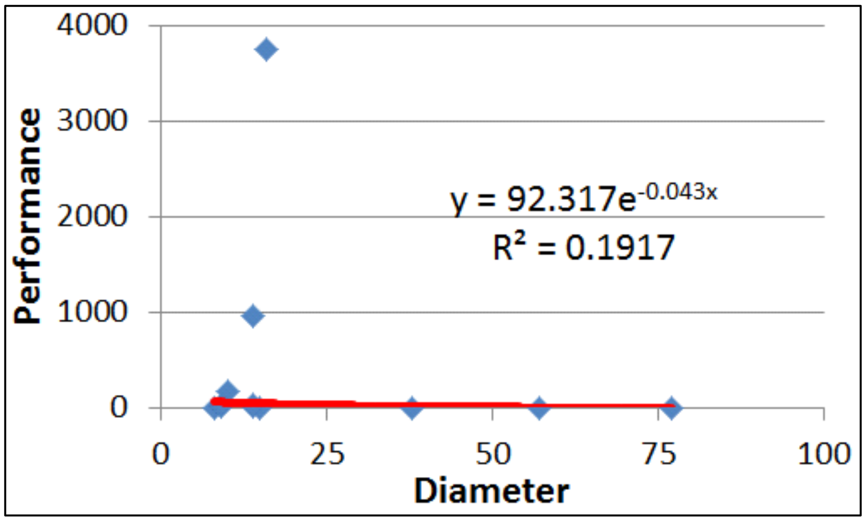

Figure 69 - Performance improvement of the incremental betweenness centrality algorithm vs. Network diameter.

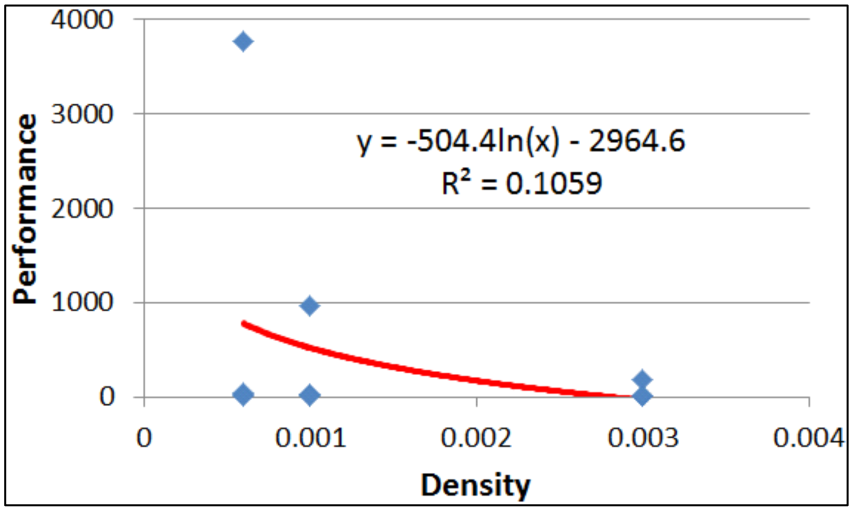

Figure 70 - Performance improvement of the incremental betweenness centrality algorithm vs. Network density.

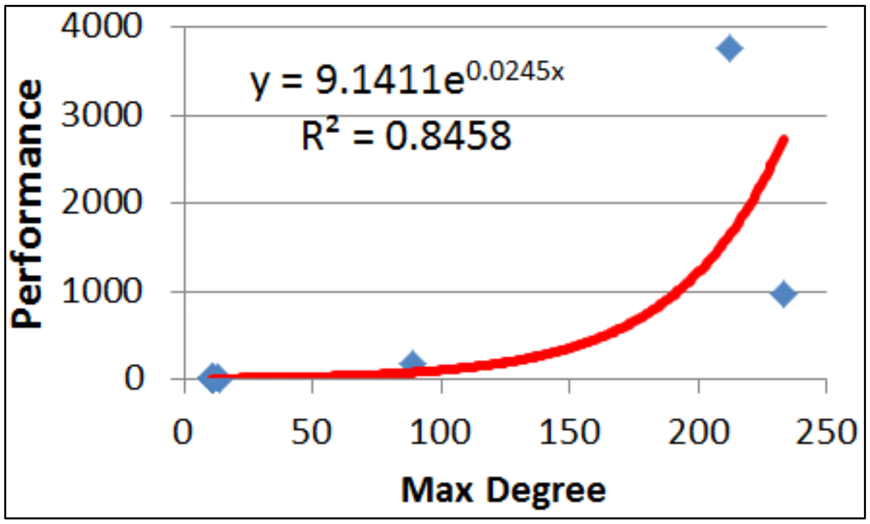

Figure 71 - Performance improvement of the incremental betweenness centrality algorithm vs. Maximum node degree.

The trends observed for the incremental betweenness and incremental closeness centrality algorithms are similar. For instance, Figure 50 and Figure 62 suggest that the performance improvements of the incremental centrality algorithms increase with the decreasing average shortest path length. This is reasonable as there are fewer nodes on the shortest paths to be updated on average. This would also cause the memory consumption to be lower as there are fewer nodes to be tracked as predecessors when constructing shortest paths trees with lower depth on average.

One surprising results is the relationship of the maximum node degree and the performance of the incremental centrality algorithms (Figure 53 and Figure 65). For the same average node degree, increased maximum degree implies a network structure that is moving towards having hub nodes with many connections as in preferential attachment and scale-free networks. Although all the previous performance results suggested higher performance benefits for the preferential attachment networks, it is striking to see the impact through the maximum node degree.

Finally, Table 35 summarizes our findings about how different network properties affect the performance of the incremental centrality algorithms on synthetic networks. A sample row in Table 35 is read as follows. While the diameter of a network decreases, the performance benefits of the incremental centrality algorithms are expected to increase and the memory consumption is expected to decrease.

For most of the network features, the memory consumption and performance improvements go hand in hand: the memory consumption decreases while the performance improvements increase. For instance, when the shortest paths in a network are not very long, the average shortest path length is relatively small. Lower average shortest path lengths result in fewer affected nodes, which causes the performance improvements of the incremental algorithms to be higher. In addition, lower average shortest path length also causes the shortest path trees that are stored in the background to contain fewer nodes and to have low depth.

Table 35 - Table summarizing how different network features affect the performance of the incremental centrality algorithms on synthetic networks.

Network Feature	Performance	Memory
Clustering Coefficient: *Decrease*	Increase	Governed by others
Max. Degree: *Increase*	Increase	Governed by others
Avg. Shortest Path Length: *Decrease*	Increase	Decrease
Diameter: *Decrease*	Increase	Decrease
Density: *Decrease*	Increase	Decrease
Size: *Increase*	Increase	Increase
Density: *Decrease*	Increase	Increase

8.5.3 Network Size and Performance Trends on Real-Life Networks

This section discusses the regression analysis performed on how the performances of the incremental centrality algorithms scale with the network size on the real life datasets. The discussion presented in this section is the counterpart of the regression analysis performed on synthetic networks in Chapter 8.5.1.

Figure 72 presents the performance improvements of the incremental betweenness centrality algorithm collected on real life networks using growing network updates. Figure 73 presents similar results for the closeness centrality algorithm. The performance improvements range from 113x to 53722x for the incremental betweenness centrality algorithm and range from 452x to 356527x for the incremental closeness centrality algorithm. Thus, the performance values are drawn on logarithmic scale to ensure that all data points are clearly visible. Figure 72 and Figure 73 (real-life networks) are the counterparts of Figure 46 and Figure 47 (synthetic networks).

As suggested by the results presented both in Figure 72 and Figure 73, out of these five networks, two networks (e.g. *Group-I*: Iran Retweet network and P2P file communication network) above the general trend line behave differently than the remaining three networks (e.g. *Group-II*: SocioPatterns network, Email network, and HEP Co-authorship network) that remain below it.

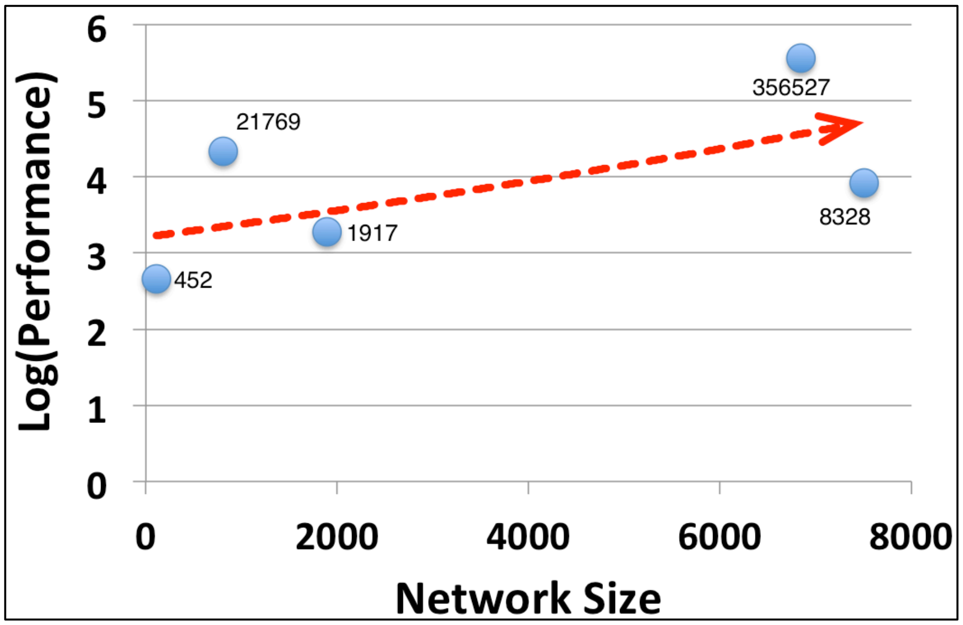

Figure 72 - Performance of the incremental betweenness centrality algorithm in real life networks for the growing network updates vs. Network size.

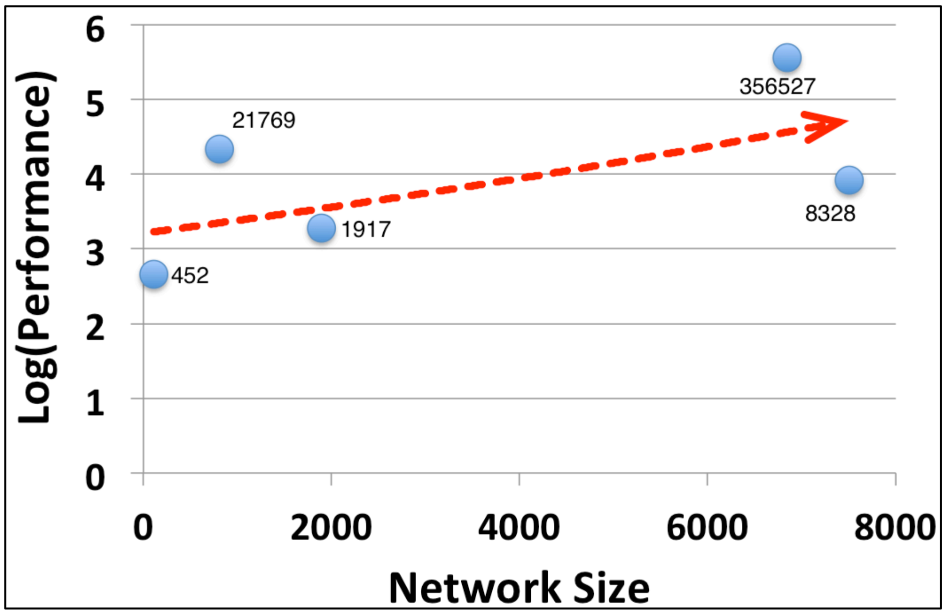

Figure 73 - Performance of the incremental closeness centrality algorithm in real life networks for the growing network updates vs. Network size.

There is no regression curves found that would fit all data points and present a clear trend. Hence, separate curves are fitted for the *Group-I* and *Group-II* networks as follows. In the equations presented below, the y values denote the performance improvements of the incremental algorithm over its non-incremental static counterpart. The x values represent the network size in terms of number of nodes.

Incremental Betweenness Centrality:

$y = 0.1625x - 303.3$ (*Group-I*: Iran Retweet, P2P Communication)

$y = 29.826\, ln(x) - 23.091$ (*Group-II*: SocioPatterns, Email, HEP Coauthorship)

Incremental Closeness Centrality:

$y = 0.0599x + 398.83$ (*Group-I*: Iran Retweet, P2P Communication)

$y = 4\text{e-}05x^2 + 0.7338x + 368.53$ (*Group-II*: SocioPatterns, Email, HEP Coauthorship)

These above formulated equations have a number of implications on how the performances of the incremental centrality algorithms scale with network size while accounting for different properties of networks. First, considering the constants in the above given equations, the incremental closeness centrality is more likely to provide higher benefits as the network size (in terms of the number of nodes) increases. Second, the performance of the incremental betweenness centrality algorithm scales with a logarithmic equation (e.g. the *ln* function) for the *Group-II* networks, which implies that the performance benefits become more incremental with the increasing network size. However, it is the opposite for the *Group-I* networks, which is supported by a linear scaling trend. This is because the overall structures of the *Group-I* networks in are different from those in *Group-II* networks. *Group-I* networks benefit from sparsity of the networks and the social structure that causes most of the relationship to be in one direction. However, in two out of three *Group-II* networks are undirected and Email network has a lot of connections that go in both directions. Altogether, these properties cause the percentages of affected nodes to be higher in *Group-II* network than in *Group-I* networks, which is also reflected in the scaling of performance improvements as the regression analysis suggested.

8.5.4 Network Features and Performance on Real-life Networks

Next, we discuss how the performances of the incremental centrality algorithms scale with the other network features on real life networks. This section is the counterpart of Chapter 8.5.2 that analyzes and discusses how the performances of the incremental centrality algorithms scale with varying network features for synthetic networks. To be able to analyze these aspects, we primarily benefit from the information provided in network statistics in Chapter 7.2.6 and the real-life networks' performance results that are presented earlier in this chapter.

Table 36 - Network statistics for real-life networks accompanied by the performance improvements of the incremental closeness centrality algorithm.

Network	**D? U?**	**Nodes**	**Edges**	**Den sity**	**Avg. Btw.**	**Max Deg**	**Diam eter**	**Char. Path Len**	**Clust. Coef**	**Closeness Speedup**
SocioPatterns	U	113	4392	0.35	36.752	98	3	1.66	0.53	452.31
OnlineForum	D	1897	20290	0.0056	7159.92	339	8	3.2	0.08	1917.84
HEP Coauthorship	U	7507	38804	0.00068	15620.65	64	15	5.74	0.46	8328.39
P2P	D	6843	7572	0.00016	0.33	2185	3	1.25	0	356527.35
Twitter (Iran)	D	809	665	0.00102	0.464	39	5	1.4	0.02	21769.19

Combining the information provided on Table 9 (network statistics for real life networks) and Table 26 (performance results for the incremental closeness centrality algorithm), we obtain Table 36. Similarly, Table 37 combines information from Table 9 and Table 28 (performance results for the incremental betweenness centrality algorithm).

Table 37 - Network statistics for real-life networks accompanied by the performance improvements of the incremental betweenness centrality algorithm.

Network	D? U?	Nodes	Edges	Density	Avg. Btw.	Max Deg	Diameter	Char. Path Len	Clust. Coef	Betweenness Speedup
SocioPatterns	U	113	4392	0.35	36.752	98	3	1.66	0.53	113.67
OnlineForum	D	1897	20290	0.0056	7159.92	339	8	3.2	0.08	217.79
HEP Coauthorship	U	7507	38804	0.00068	15620.65	64	15	5.74	0.46	233.27
P2P	D	6843	7572	0.00016	0.33	2185	3	1.25	0	53722.49
Twitter (Iran)	D	809	665	0.00102	0.464	39	5	1.4	0.02	8429.88

Using the information provided Table 36 and Table 37, I have again generated multiple scatter-plot figures to show how the performance improvements change with the changing network parameters. The scatterplots (Figure 74 - Figure 97) are accompanied by non-linear regression equations (trend lines) that would provide the best possible match, as close as possible to all data points in the graph. Although the actual performance values are different, the general performance trends for the incremental closeness and betweenness centrality algorithms are similar. Thus, I first provide figures comparing the performance values of the incremental closeness centrality algorithm (Figure 74 - Figure 85 for the information presented in Table 36) and then the figures for the performance of the incremental betweenness centrality algorithm (Figure 86 - Figure 97 for the information presented in Table 37) against the network features. To be consistent with the results presented earlier in Chapter 8.5.1, Chapter 8.5.2, and Chapter 8.5.3 and to make the figures more visible, the *y*-axes of the figures used in main discussion are again drawn in logarithmic scale, accompanied by their equivalents in linear scale for comparison purposes. In particular, Figure 74 - Figure 79 show log-10 of the performance of the incremental closeness centrality algorithm versus various network features and Figure 80 - Figure 85 show their counterparts drawn in linear *y*-axes. Similarly, Figure 86 - Figure 91 show log-10 of the performance of the incremental betweenness centrality algorithm versus various network features and Figure 92 - Figure 97 show their counterparts drawn in linear *y*-axes.

For the data presented in Figure 74 - Figure 97, regression equations are fit in an attempt to better explain the trends. In the regression equations, *x*-value represents the network feature under investigation such as the clustering coefficient, the average shortest path length, or the network diameter. The *y*-value represents either the logarithm of the performance improvement or the performance improvement of the incremental centrality algorithms.

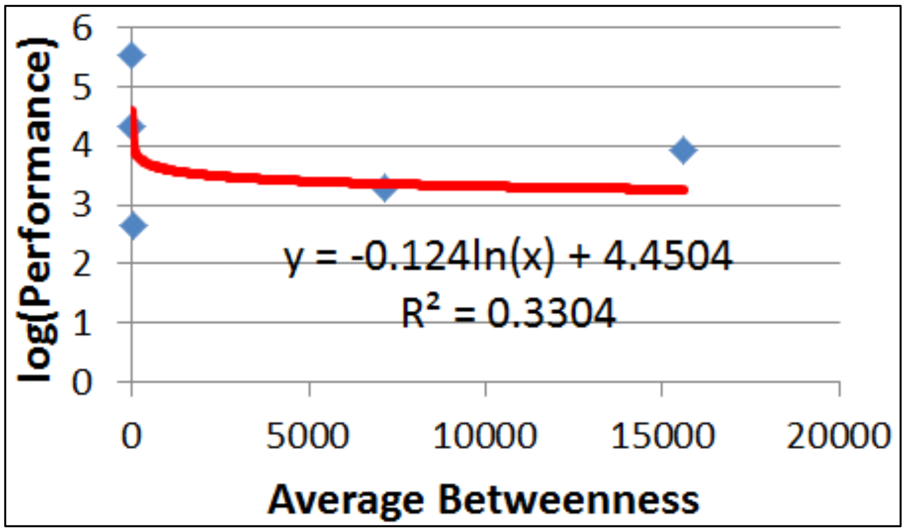

Figure 74 - Log-10 of the performance improvement of the incremental closeness centrality algorithm vs. Average betweenness centrality.

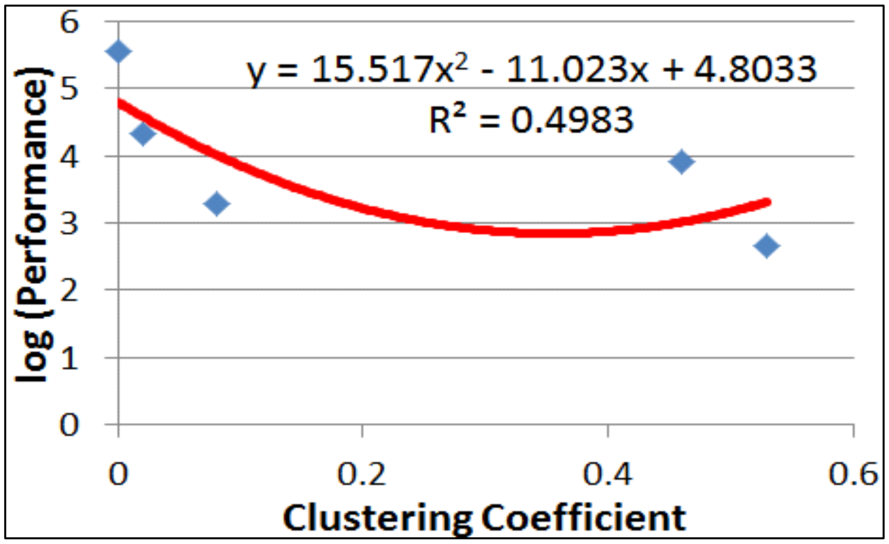

Figure 75 - Log-10 of the performance improvement of the incremental closeness centrality algorithm vs. Global clustering coefficient.

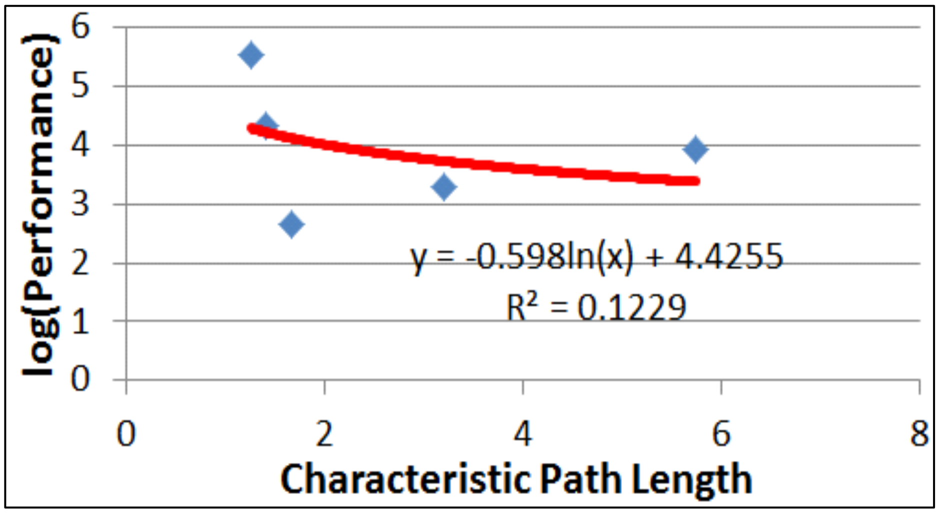

Figure 76 - Log-10 of the performance improvement of the incremental closeness centrality algorithm vs. Characteristic path length.

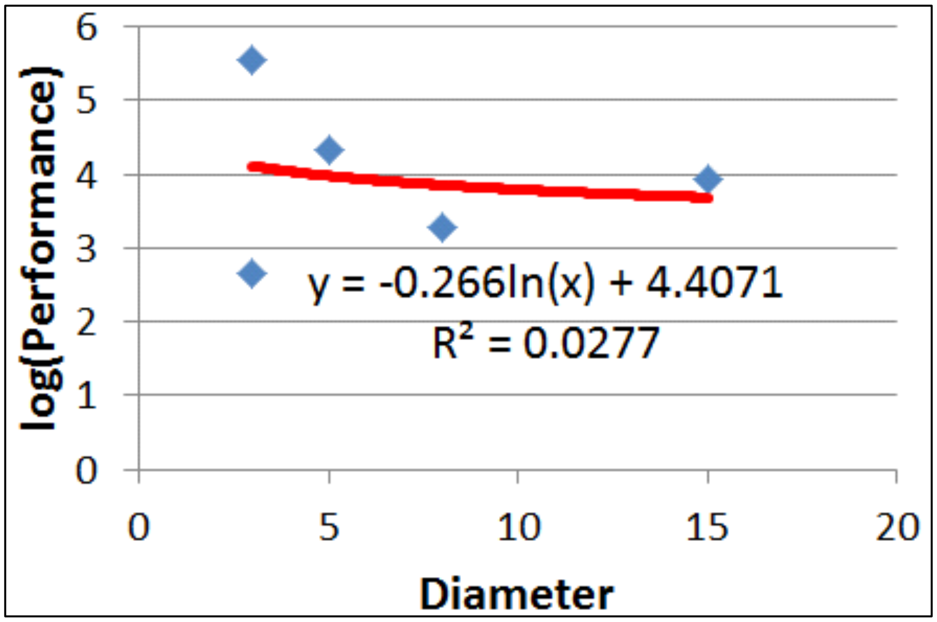

Figure 77 - Log-10 of the performance improvement of the incremental closeness centrality algorithm vs. Network diameter.

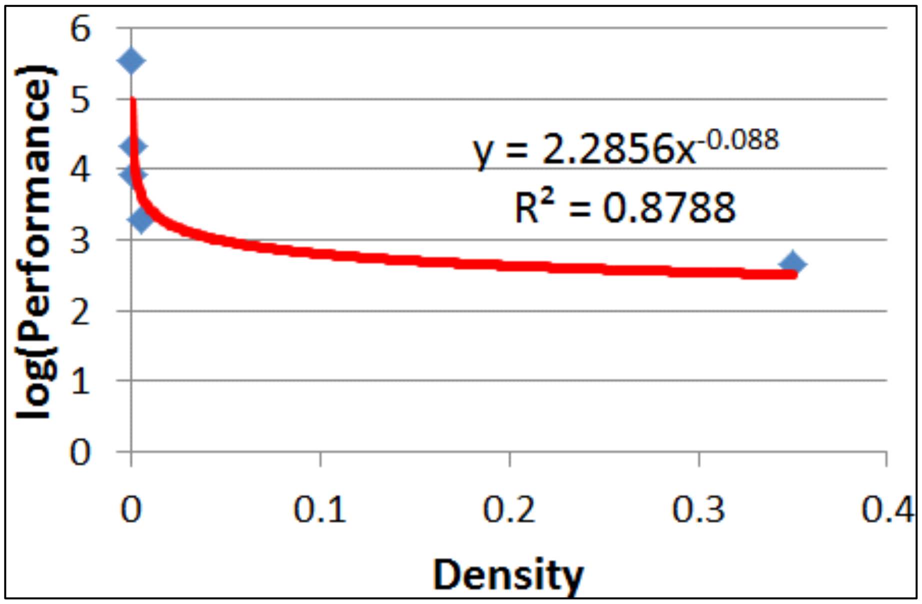

Figure 78 - Log-10 of the performance improvement of the incremental closeness centrality algorithm vs. Network density.

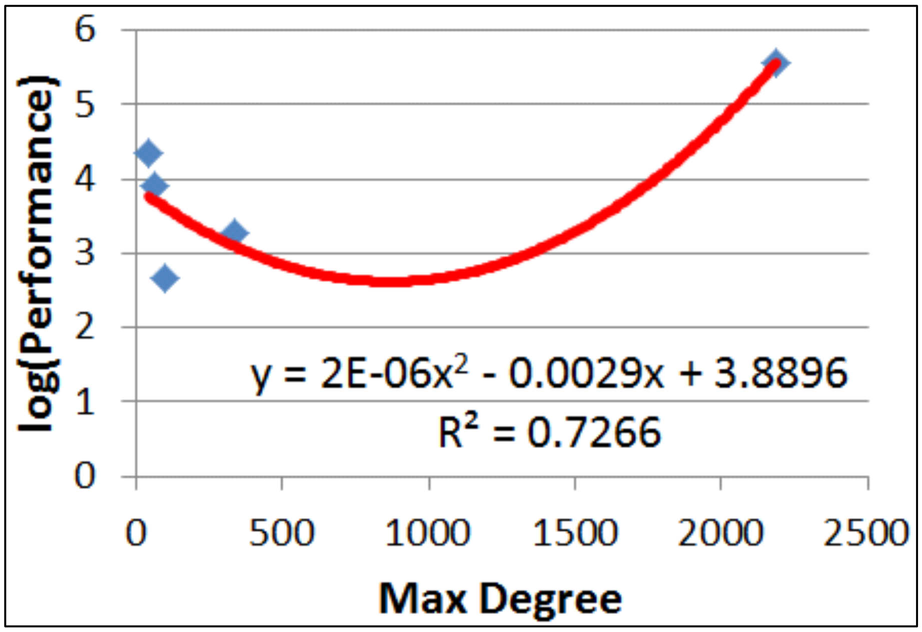

Figure 79 - Log-10 of the performance improvement of the incremental closeness centrality algorithm vs. Maximum node degree.

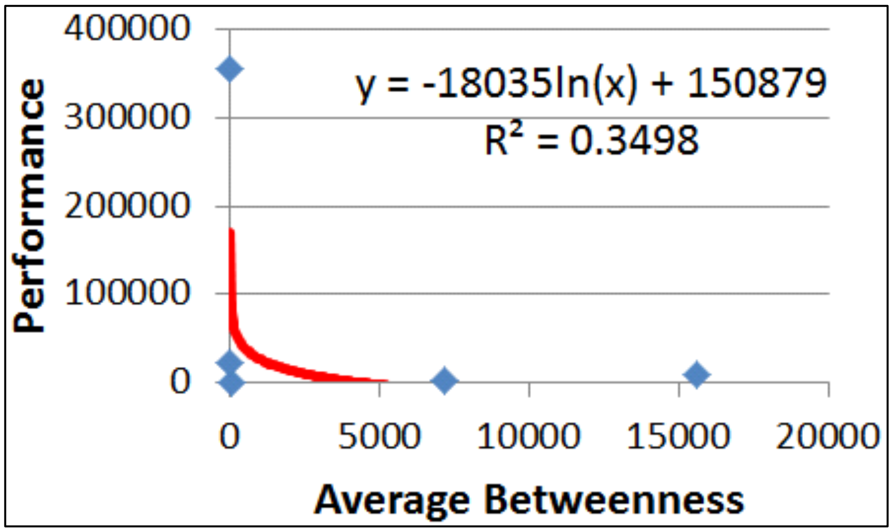

Figure 80 - Performance improvement of the incremental closeness centrality algorithm vs. Average betweenness centrality.

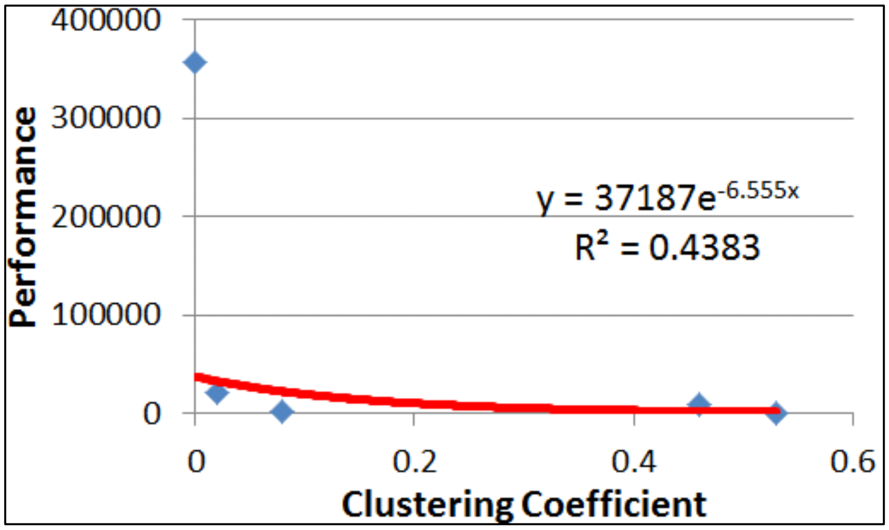

Figure 81 - Performance improvement of the incremental closeness centrality algorithm vs. Global clustering coefficient.

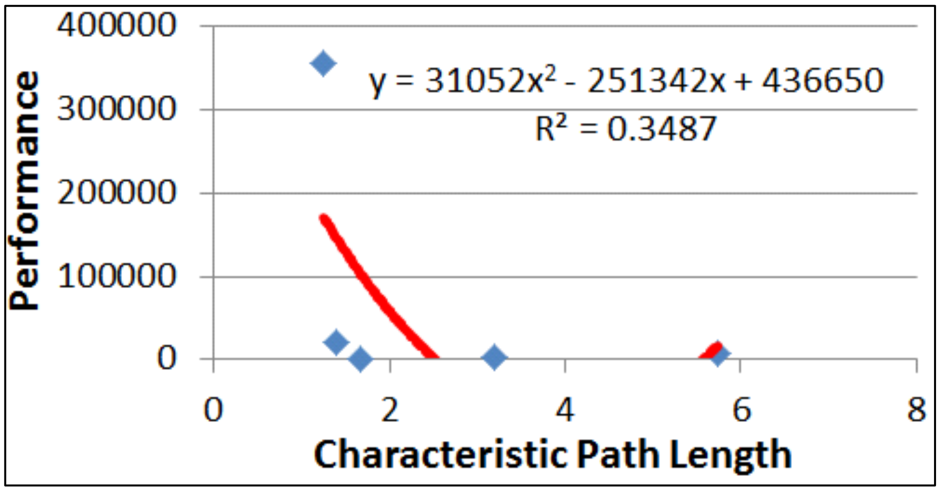

Figure 82 - Performance improvement of the incremental closeness centrality algorithm vs. Characteristic path length.

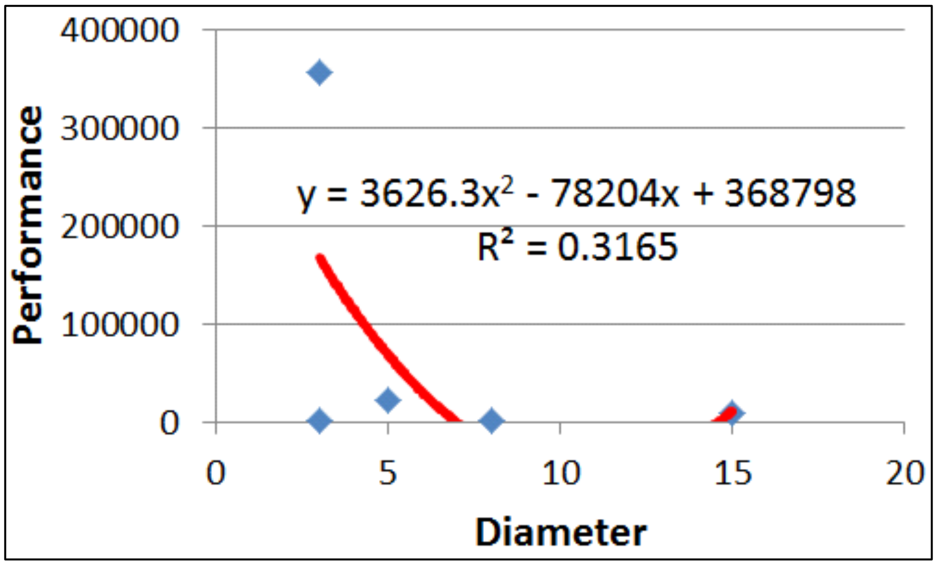

Figure 83- Performance improvement of the incremental closeness centrality algorithm vs. Network diameter.

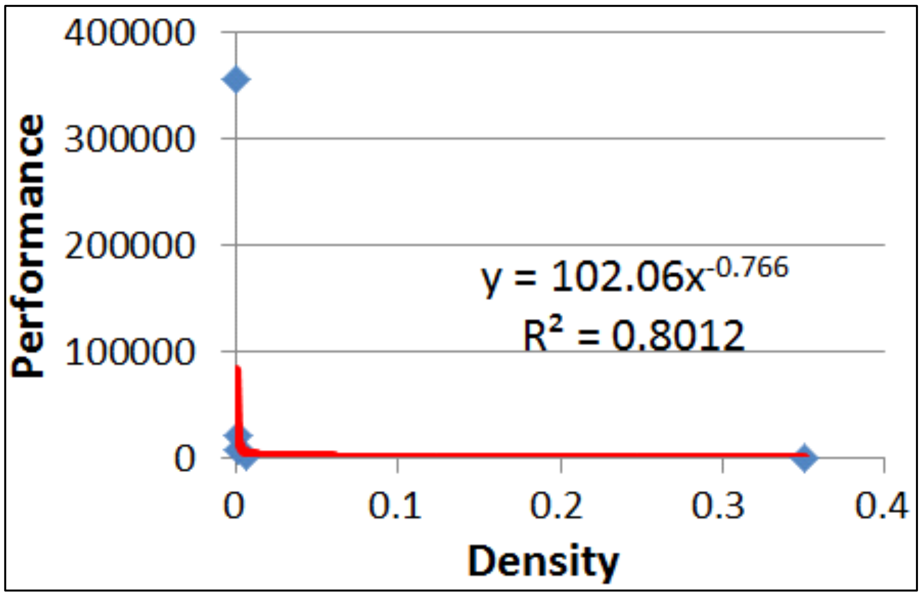

Figure 84 - Performance improvement of the incremental closeness centrality algorithm vs. Network density.

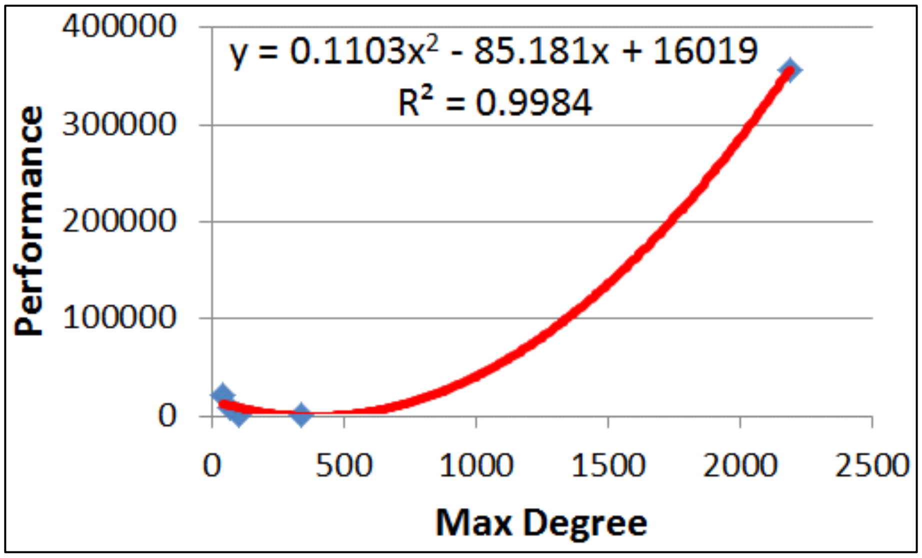

Figure 85 - Performance improvement of the incremental closeness centrality algorithm vs. Maximum node degree.

The next set of figures present the performance scaling of the incremental betweenness algorithm. The set of figures with logarithmic *y*-axes (Figure 86 - Figure 91) are presented first, followed by the figures with linear performance axes (Figure 92 - Figure 97) for comparison purposes.

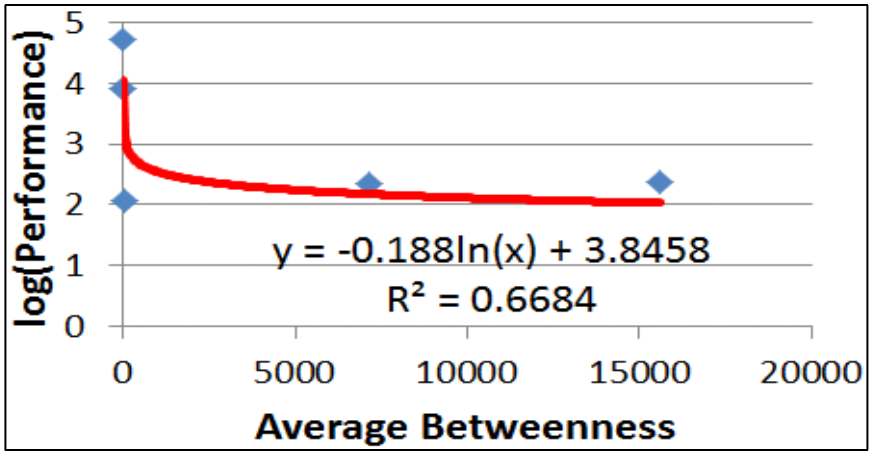

Figure 86 - Log-10 of the performance improvement of the incremental betweenness centrality algorithm vs. Average betweenness centrality.

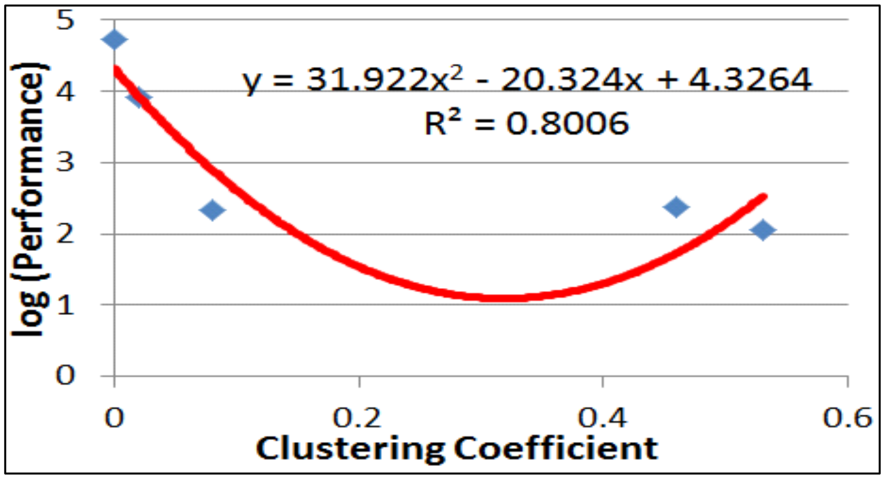

Figure 87 - Log-10 of the performance improvement of the incremental betweenness centrality algorithm vs. Global clustering coefficient.

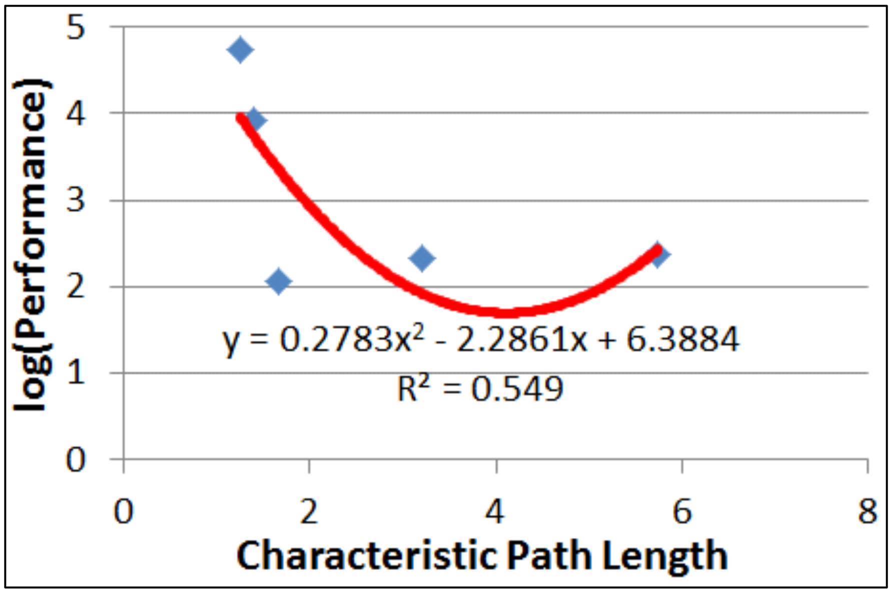

Figure 88 - Log-10 of the performance improvement of the incremental betweenness centrality algorithm vs. Characteristic path length.

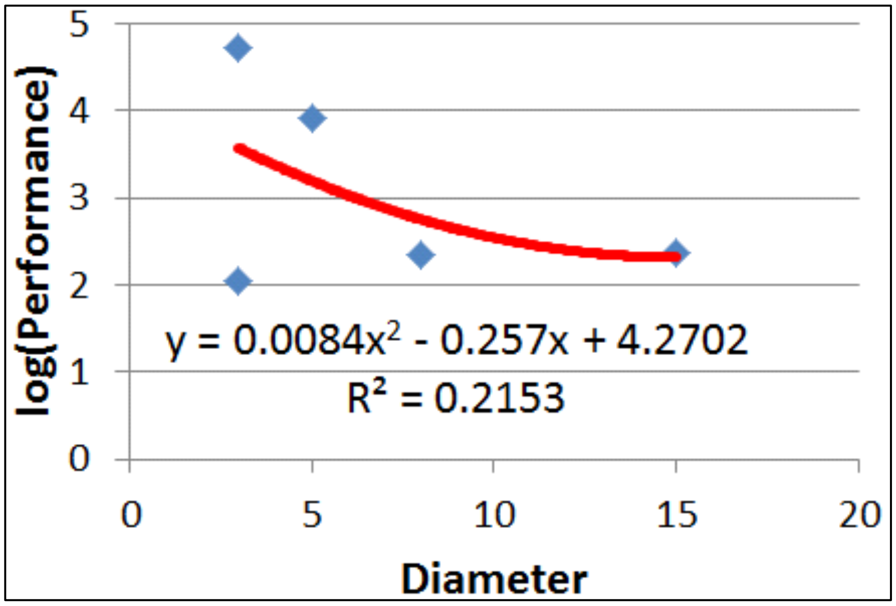

Figure 89 - Log-10 of the performance improvement of the incremental betweenness centrality algorithm vs. Network diameter.

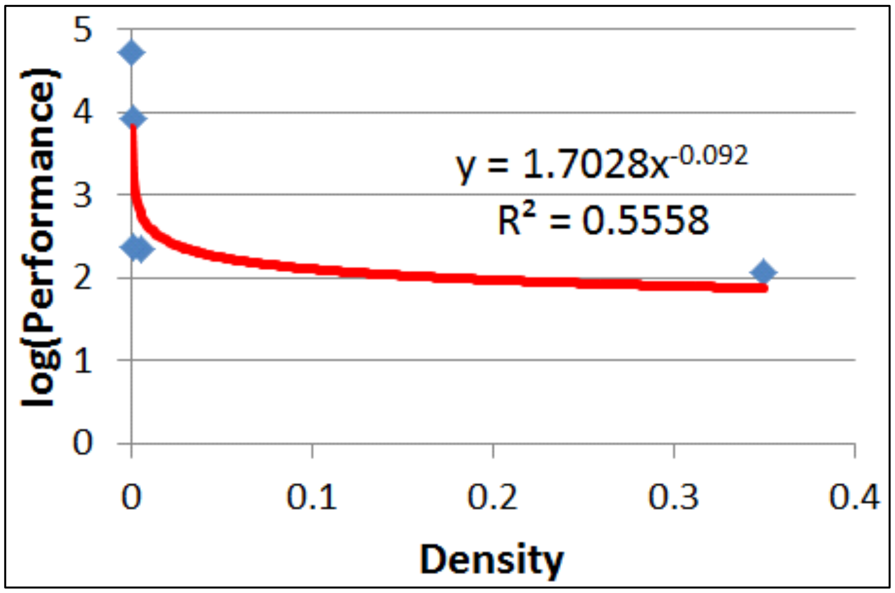

Figure 90 - Log-10 of the performance improvement of the incremental betweenness centrality algorithm vs. Network density.

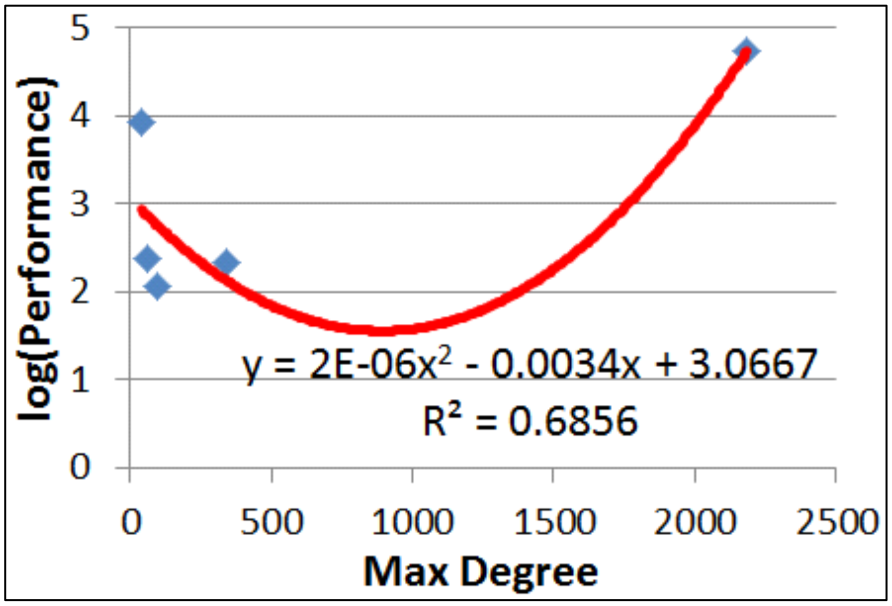

Figure 91 - Log-10 of the performance improvement of the incremental betweenness centrality algorithm vs. Maximum node degree.

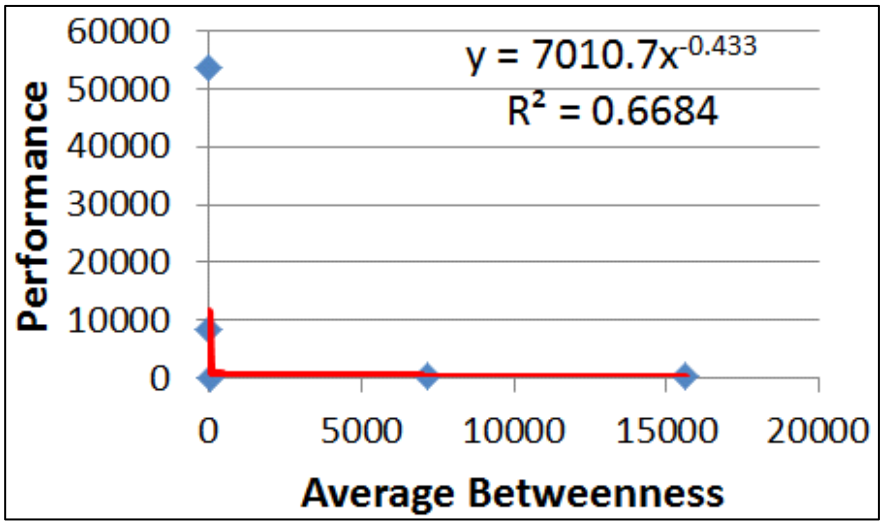

Figure 92 - Performance improvement of the incremental betweenness centrality algorithm vs. Average betweenness centrality.

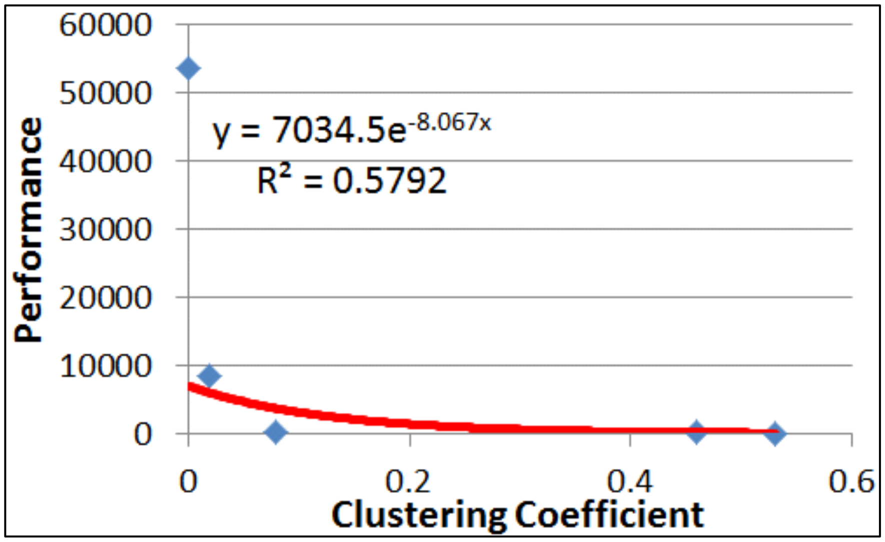

Figure 93 - Performance improvement of the incremental betweenness centrality algorithm vs. Global clustering coefficient.

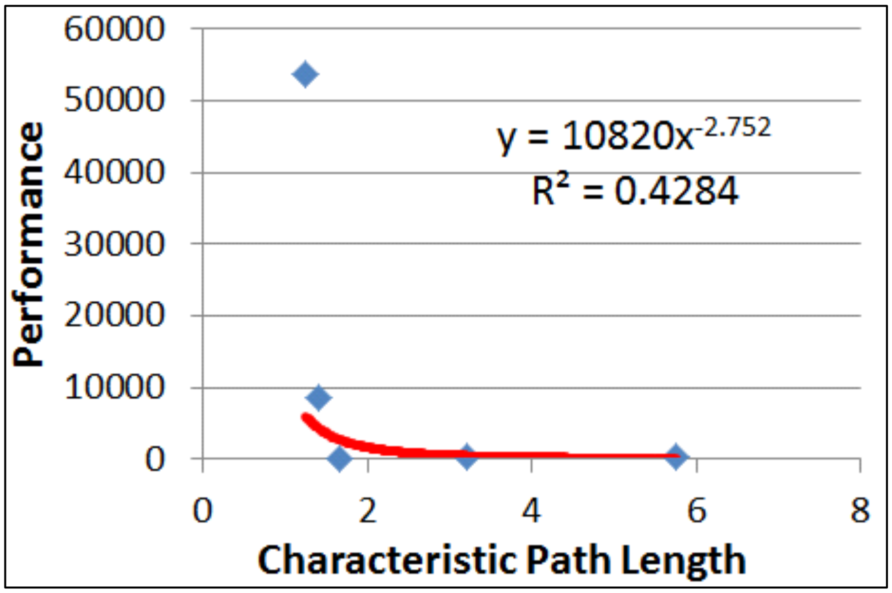

Figure 94 - Performance improvement of the incremental betweenness centrality algorithm vs. Characteristic path length.

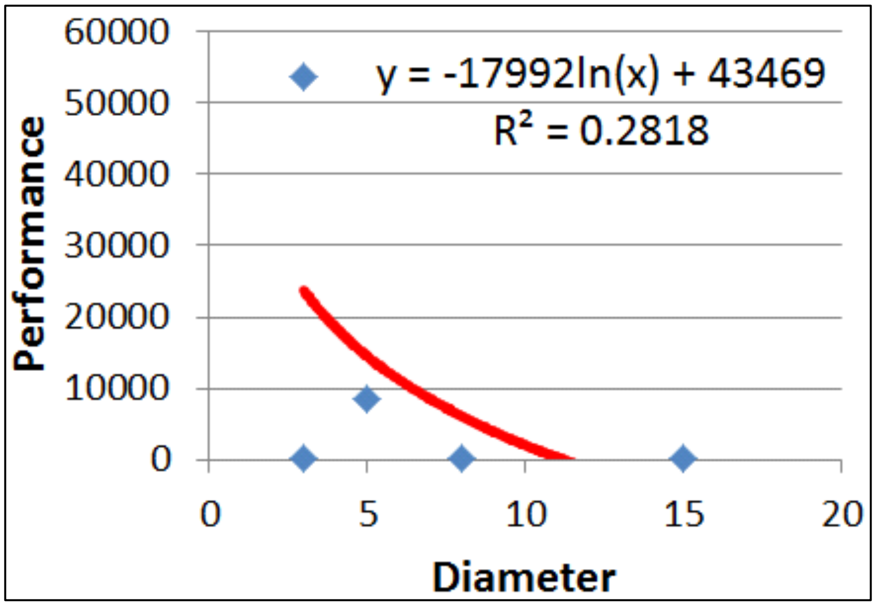

Figure 95 - Performance improvement of the incremental betweenness centrality algorithm vs. Network diameter.

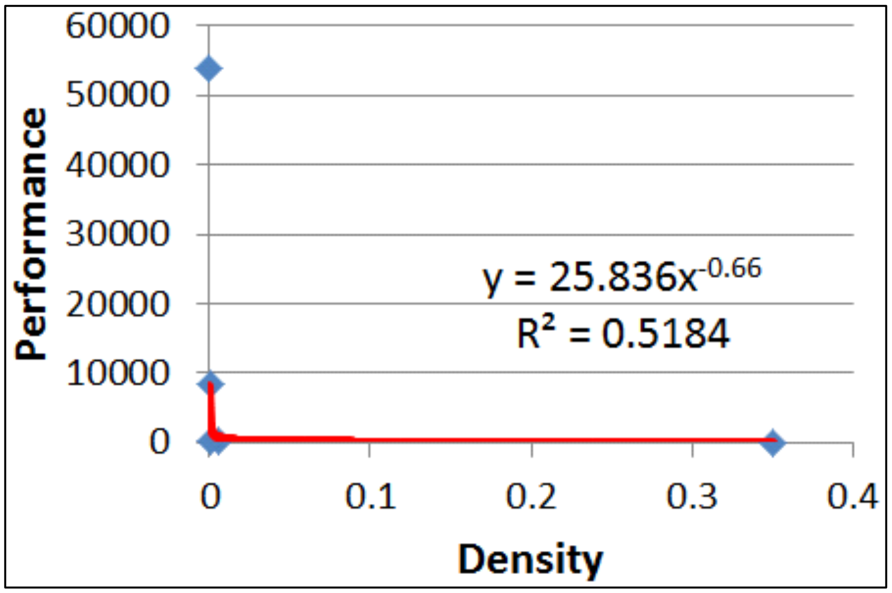

Figure 96 - Performance improvement of the incremental betweenness centrality algorithm vs. Network density.

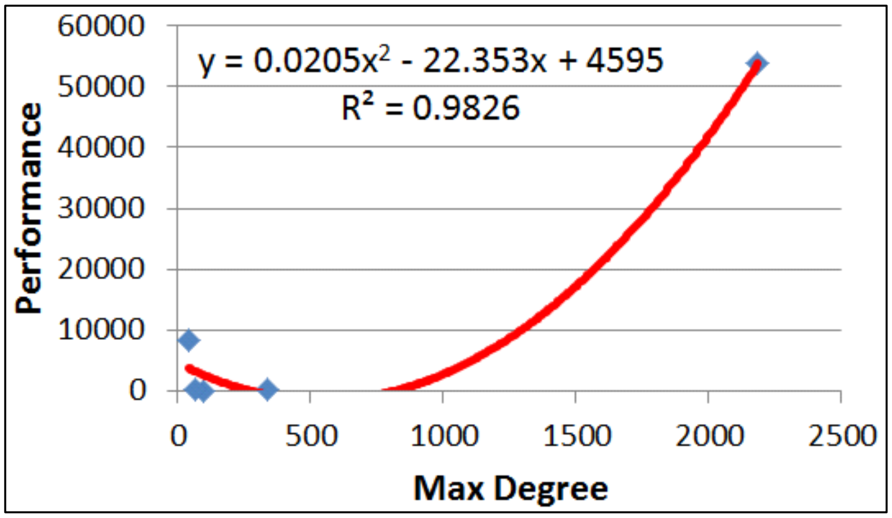

Figure 97 - Performance improvement of the incremental betweenness centrality algorithm vs. Maximum node degree.

Considering the trend lines fit on the performance results presented in the figures above, the maximum degree and the network density have trend lines that closely match the data presented. Although the other network features have some visible trends, most of the time, these network features interact with one another and cause the actual performance values to deviate from the expected values. Table 38 summarizes the general trends on how specific network features would steer the performances of the incremental centrality algorithms. In general, the trends observed on the real life and synthetic networks are in line with one another, which will be discussed next in Chapter 8.5.5 in detail.

Table 38 - Table summarizing how different network features affect the performance of the incremental centrality algorithms on real life networks.

Network Feature	Performance	Memory
Max. Degree: *Increase*	Increase	Governed by others
Density: *Decrease*	Increase	Decrease
Size: *Increase*	Increase	Increase
Clustering Coefficient: *Decrease*	Increase	Governed by others
Avg. Shortest Path Length: *Decrease*	Increase	Decrease
Diameter: *Decrease*	Increase	Decrease
Fragmentation: *Increase*	Increase	Decrease

8.5.5 Comparative Discussion on Performance Trends in Synthetic and Real-Life Networks

This section discusses the performance trends observed in synthetic and real-life networks together in an attempt to compare between the synthetic network data and the real-life network data and comment on how well the performance trends on the synthetic data reflect the performance trends of the real-life networks.

It is observed that the trends observed in real life and synthetic networks are comparable as outlined in Table 35 and Table 38. However, there are minor differences. For instance, the clustering coefficients in real life networks have the potential to be higher than those in the synthetic networks. Another difference is that in real life networks, the networks' diameters are short, i.e., less than 15 hops. However, in synthetic networks, especially for directed cycles, the diameters can become artificially long, exceeding 100 hops.

In real-life networks is that a substantial portion of relationships among social agents is mutual and modeled as undirected (bidirectional) networks. The impact of converting a network from directed to undirected is different in different networks. In most real life networks, the average shortest path lengths and diameters are low as discussed earlier. When the network is converted to an undirected network from a directed network, this usually results in higher number of pairs that can reach out to one another and higher clustering coefficient, without having too much impact on the lengths of the shortest paths. Converting the relationships from directed to undirected also results in an increased number of nodes affected when an incremental update such as an edge insertion is issued on the network. This usually reflects as decreased performance benefits when the network is undirected compared to directed networks. However, on an artificially created network, the impact might be different. For instance, the small world networks are undirected versions of the directed cycles. The small world networks' shortest path lengths are very low compared to those in the directed cycles, reducing the network diameter from 102 (3000-node directed cycles with avg. node degree 4) to 13 (3000-node small world network with avg. node degree 4). In contrast to what we observe for most real life networks, we observe that small world networks have higher performance benefits than directed cycles due to decreased shortest path lengths on average.

In addition, real-life networks might have fragmentation and disconnected components that would help increasing the performance obtained with incremental algorithms on such networks. Among the datasets we have used, only two networks have considerable, non-zero fragmentation: Iran retweet network has 0.86 fragmentation and the co-authorship network has 0.239 fragmentation. The synthetic networks are modeled to reflect the claimed network sizes as the entire node space; hence, they have zero fragmentation. Since we have only two data points for fragmentation, we don't have enough data points and they are not included in the scatter plots. Yet, the impact of the fragmentation can be understood from the very high performance improvements obtained on the Iran retweet network despite the fact that it is a small network with less than 1000 nodes.

Apart from the differences discussed so far, there are also similarities that are well covered, which causes similar trends to be observed in both real life and synthetic networks when regression analysis is performed on the scatter plots of different network features. As a specific example, the performances of the incremental centrality algorithms on the P2P file transfer network resemble the performance trends of the directed preferential attachment networks. This is because both the preferential attachment networks and the P2P file transfer network are modeled as directed networks. In addition, they both have central, hub nodes with very large number of connections and very few nodes with high betweenness centrality values, resulting in low average betweenness centrality across the entire network.

8.6 Incremental Betweenness Clustering Performance Results

In this section, the performance of the clustering algorithm that is discussed in Chapter 6.5.1 is evaluated. The performance results presented in Table 39 and Table 40 are collected as follows. For each network used in the evaluations, using the corresponding *k*-hop limit, the edge betweenness centrality values are calculated. Then, 1000 edges are removed either by recomputing the edge betweenness values using the incremental edge *k*-betweenness algorithm or by using the Brandes' edge *k*-betweenness algorithm. The performance results compare the total (cumulative) execution time that includes (*i*) the initialization of the edge *k*-betweenness centrality values, (*ii*) searching for the edge with the maximum edge betweenness to remove in the next iteration, and (*iii*) re-computation of the edge *k*-betweenness values after the removal of the edge with the largest edge *k*-betweenness centrality value.

As explained earlier in Chapter 6.5.1.2, using the incremental edge betweenness algorithm within the edge betweenness based clustering might trigger worst-case network updates. This is because the edge betweenness based clustering algorithms (e.g. the Girvan-Newman clustering algorithm and its variants) remove the edge with the highest edge betweenness centrality value, which is the edge that tends to lie on many shortest paths. The removal of such an edge may result in a big change in the network, which, in turn, might trigger an update for all the shortest paths in the network. To ensure that the possibility of the worst case updates which would minimize the potential performance benefits of the incremental centrality is ruled out, using the *k*-hop limited approximation of the edge betweenness centrality is recommended in this thesis. Therefore, the results presented in this section focus on comparing the performance of the edge betweenness based clustering algorithm using the incremental and the non-incremental (Brandes') edge betweenness algorithms with *k* hop limits. In the results presented in Table 39 and Table 40 the performance improvements of the proposed clustering algorithm discussed are around 100x – 200x for 1000 edge removals. However, since there are other parts of the algorithm accounted for such as initialization and the search for the maximum edge betweenness centrality value, the performance improvement values do not smoothly increase with the increasing network size or the average node degree. Therefore, as a part of the results presented in Table 39 and Table 40, the actual execution times are included as well. The actual execution times were collected in terms of nanoseconds, and converted to seconds for presentation in Table 39 and Table 40.

In Table 39, the evaluations for the preferential attachment, Erdos-Renyi, small-world, and directed cycles network topologies for increasing network sizes (e.g. 1000, 3000, and 5000) are presented. All networks have an average node degree of 6. For instance, PF1 stands for preferential attachment network with 1000 nodes while SW5 stands for small world network with 5000 nodes.

Table 39 - Performance results for the clustering algorithm based on incremental *k*-betweenness compared against the clustering algorithm based on *k*-betweenness, collected on synthetic networks.

	(*k* = 2)			**(*k* = 3)**		
T	**Brandes *k*-btw (sec)**	**Incremental *k*-btw (sec)**	**Performance Improvement**	**Brandes *k*-btw (sec**	**Incremental *k*-btw (sec)**	**Performance Improvement**
PF-1	925.65	6.61	139.98x	1002.34	8.58	116.81x
PF-3	9411.91	57.33	164.16x	9696.87	65.34	148.41x
PF-5	39940.17	249.94	159.79x	43935.9	415.37	105.77x
ER-1	952.29	5.91	161.09x	998.63	7.51	132.93x
ER-3	10180.22	83.9	121.33x	13434.94	109.03	123.23x
ER-5	46442.98	263.49	176.26x	49698.67	308.65	161.02x
SW-1	1004.88	8.04	124.98x	1203.82	15.77	76.33x
SW-3	9793.34	59.85	163.63x	12436.9	84.77	146.71x
SW-5	37075.02	262.23	141.38 x	58208.12	293.24	198.49x
DC-1	978.56	5.69	171.93x	998.29	6.81	146.64x
DC-3	8374.41	56.98	146.96x	9119.83	58.87	154.92x
DC-5	42406.91	255.89	165.72x	43632.24	277.58	157.18x

In Table 40, the evaluations for the preferential attachment, Erdos-Renyi, small-world, and directed cycles network topologies for varying average node degrees (e.g. 4, 6, and 8) are presented. All networks have 3000 nodes. For instance, PF4 stands for preferential attachment network with 3000 nodes and average degree of 4 while SW8 stands for small world network with 3000 nodes and average degree of 8.

Finally, in Table 41, the performance results collected on the real life networks are presented. In these results, 100 edges are removed, and similar to the synthetic networks' performance results presented in Table 39 and Table 40, the total execution times are compared which includes the initial computation of the edge betweenness values, selecting the edge with the maximum edge betweenness value.

Table 40 - Performance results for the clustering algorithm based on incremental *k*-betweenness compared against the clustering algorithm based on *k*-betweenness, collected on synthetic networks.

	(*k* = 2)			**(*k* = 3)**		
T	**Brandes *k*-btw (sec)**	**Incremental *k*-btw (sec)**	**Performance Improvement**	**Brandes *k*-btw (sec)**	**Incremental *k*-btw (sec)**	**Performance Improvement**
PF-4	8222.77	53.76	152.94x	9258.58	53.85	171.93x
PF-6	9411.91	57.33	164.16x	9696.88	65.33	148.41x
PF-8	8907.49	60.36	147.56x	11119.39	97.87	113.61x
ER-4	8505.08	53.62	158.63x	9045.75	57.73	156.69x
ER-6	10180.22	83.9	121.33x	13434.94	109.02	123.23x
ER-8	9883.49	56.83	173.89x	12624.59	73.91	170.79x
SW-4	8780.47	58.78	149.37x	9273.4	61.69	150.32x
SW-6	9793.34	59.85	163.63x	12436.9	84.77	146.71x
SW-8	12890.4	63.56	202.81x	18012.4	135.17	133.25x
DC-4	9227.45	60.94	151.40x	11150.38	53.86	207.01x
DC-6	9119.83	56.98	160.04x	8374.41	58.87	142.26x
DC-8	11488.83	89.83	127.90x	13457.54	101.65	132.39x

The results presented in Table 41 present the performance improvements obtained over the clustering performed using the regular Brandes' edge betweenness algorithm. Imagine a missing column for the clustering algorithm that uses the Brandes' edge betweenness algorithm with all the values set as 1x. Hence, it is possible to see how much of the performance improvement comes from the use of *k*-betweenness algorithm, and how much of the performance improvement comes from adding the benefits of the incremental algorithm design. The performance results presented in Table 41 show that the majority of the performance improvements come from the incorporation of the incremental algorithm in the edge betweenness based clustering. The larger performance improvements are observed in cases where the introduction of the *k*-hop limits can prune some of the extra work that would have been performed otherwise.

Table 41 - Performance results for the clustering algorithms based on the incremental *k*-betweenness and the Brandes' *k*-betweenness. Collected on real-life networks.

	(*k* = 2)		**(*k* = 3)**	
Topology	**Brandes *k*-betweenness**	**Incremental *k*-betweenness**	**Brandes *k*-betweenness**	**Incremental *k*-betweenness**
Socio Patterns	1.438 x	11.676 x	1.395 x	5.397 x
Twitter (Iran)	1.18 x	28.291 x	1.04 x	30.368 x
Email	2.95 x	79.656 x	1.05 x	1.05 x
HEP Coauthor	5.698 x	30.867 x	2.58 x	24.280 x
P2P	1.002 x	18.037 x	1.002 x	17.753 x

As one last point, the performance improvements presented for the real life networks are lower than those presented for the synthetic networks because the number of edges removed in the experiments done with real networks is lower than the number of edges removed in synthetic network experiments. Different numbers of edges are selected for removal from synthetic and real life networks so that it would also be possible to comment on how the numbers of recomputations affect the absolute values of performance improvements. The results presented in Table 39, Table 40, and Table 41 imply that the performance improvements obtained by the incorporation of the incremental *k*-betweenness computation increase with the increasing number of removed edges.

8.7 Comments on Verification

During the initial phases of algorithm development, I have used several small networks of 5 to 10 nodes that model different network structures such as cliques, chains, star topology, cycles, and various combinations of those to cover various basic network structures and potential corner cases. One I started running experiments with the larger networks that were used in the results presented in this dissertation, I initially recorded the betweenness and the closeness centrality values from the incremental centrality algorithms and from the non-incremental algorithms (the Brandes' algorithm for betweenness and the Dijkstra's algorithm for closeness) to compare the results generated at each step after every incremental network update. Since writing all the centrality values at each network update to the disk takes a long time, after getting satisfactory results with the accuracy results, I have started using the following verification method: I have recorded the betweenness/closeness centrality values only once after all 100 network updates complete for both the incremental centrality algorithms and their non-incremental counterparts, and compared both result files against one another. Since the incremental centrality values build on the results from prior runs, if there is a discrepancy in the algorithm, the centrality values at the end of 100 updates tend to be way off from the value suggested by the non-incremental, baseline algorithm. This is a good way for validating the results as well as identifying and debugging the potential bugs in the algorithm during the design and development stage.

A small sample from the betweenness centrality values generated on a 1000-node directed cycle network with $p = 0.2$ is presented in Table 42. The betweenness centrality values presented in Table 42 are recorded after 100 edge insertion updates are issued on the network.

Table 42 - Comparison of output from the incremental and non-incremental betweenness centrality algorithms.

Incremental Betweenness Algorithm		Brandes' Betweenness Algorithm	
Node ID	**Betweenness Centrality**	**Node ID**	**Betweenness Centrality**
0	0.0	0	0.0
1	2.5	1	2.5
986	3.5	986	3.5
989	6.0	989	6.0
3	8.333333333333332	3	8.333333333333332
971	2.8333333333333335	971	2.8333333333333335
974	6.0	974	6.0
4	5.833333333333333	4	5.833333333333333
2	0.0	2	0.0
5	2.833333333333333	5	2.833333333333333
870	7.833333333333332	870	7.833333333333333
6	3.8333333333333335	6	3.8333333333333335
817	5.5	817	5.5
7	4.333333333333333	7	4.333333333333333
8	5.333333333333334	8	5.333333333333333
9	9.5	9	9.5

The centrality values do match across the incremental and non-incremental algorithm, with very minor differences in how the floating-point values are handled (See the centrality values for node-870 and node-8). However, we consider the betweenness centrality values of such nodes equal. Similar results were obtained for the shrinking/growing network updates for closeness centrality, betweenness centrality, and their *k*-hop limited approximations; however, they are not included here for brevity.

CHAPTER 9 CONCLUSIONS AND FUTURE WORK

This chapter summarizes the key findings of this dissertation and discusses several additional ideas that are particularly promising as future research directions.

9.1 SUMMARY OF FINDINGS

This dissertation proposes incremental centrality algorithms that perform dynamic maintenance of closeness and betweenness centrality values while providing support for new edge/node insertions and deletions and edge cost modifications. The goal is to avoid re-computations involved in the analysis of dynamic social networks and reflect changes triggered by an incremental network update as efficiently as possible. The performance results collected on different synthetic and real-life networks indicate that incremental computations of social centrality metrics is a high-performance method, providing substantial performance improvements in revealing temporal patterns of closeness centrality and betweenness centrality.

Closeness centrality was selected as the first metric of interest because its computation is only dependent on the shortest distances across the nodes, which is the core information required by all other shortest path based social centrality metrics. The other metric that is selected as the metric of interest is the betweenness centrality due to its wide application across several fields.

In general, the incremental centrality algorithms proposed in this dissertation increases the speed with which the centrality metrics can be calculated for all nodes in a network, exploiting the classical tradeoff between memory and speed.

Considering the performance results presented in Chapter 8, several observations are in order. First, the performance gains increase with the increasing network size. In larger networks, the opportunity for early pruning is larger, which in turn results in increased performance benefits.

Another point of observation is that the network topology is an important factor in the performance improvements the incremental centrality algorithms provide. The performance results obtained on real-life networks suggest that the structure of the shortest paths in a network might differ considerably from the binary, unweighted version of the same network depending on the weights/costs of the edges. The difference between the binary, unweighted version and the weighted version of the same network is especially higher if the weights of the edges come from a large range of values. Hence, the performance improvements of the incremental centrality algorithms obtained on the binary and the weighted versions of the same network tend to be different from one another.

The impact of topology on the performance benefits of the incremental centrality algorithms were also visible in experiments performed with the synthetic networks. The incremental centrality algorithms obtain the best performance improvements on the preferential attachment networks. The performance improvements were inline with the average shortest path length and the diameter of the network; increasing with the decreased shortest path length in general. Such behavior also impacts how much performance improvement can be obtained by the incremental *k*-centralities. When the shortest paths in a network are relatively short (i.e. composed of a small number of hops), there is not too much room for opportunity for early pruning due to limiting the shortest paths to remain within the first *k* hops. In the case of incremental *k*-centralities, the best performance benefits are obtained on small world networks while the performance benefits obtained on the preferential attachment networks are lower.

Considering the behaviors of the incremental centrality algorithms both in real-life and synthetic networks, it is observed that the performance trends of the incremental centrality algorithms are similar for the real world and synthetic networks. In general, the performance improvements increase with increasing network size, increasing node maximum degree, and decreasing network density. Moreover, the performance improvements decrease with the increasing average betweenness centrality, increasing average shortest path length, increasing network diameter, and increasing global clustering coefficient.

It is also observed that most real life networks have low diameters and average shortest path lengths. Hence, networks with very long diameters such as the directed cycles are less likely to be observed when real-life relationships are modeled. In addition, real life networks benefit more from fragmentation and the existence of disconnected components, which helps to increasing the performance improvements obtained by the incremental algorithms.

Another trend in real-life networks is that a substantial portion of relationships among social agents is mutual and modeled as undirected (bidirectional) networks. The impact of converting a network from directed to undirected is different in different networks. In most real life networks, the average shortest path lengths and diameters are low as discussed earlier. When the network is converted to an undirected network from a directed network, this usually results in higher number of pairs that can reach out to one another and higher clustering coefficient, without having too much impact on the lengths of the shortest paths. Converting the relationships from directed to undirected also results in an increased number of nodes affected when an incremental update such as an edge insertion is issued on the network. This usually reflects as decreased performance benefits when the network is undirected compared to directed networks. However, on an artificially created network, the impact might be different. For instance, the small world networks are undirected versions of the directed cycles. The small world networks' shortest path lengths are very low compared to those in directed cycles, reducing the network diameter from 102 (3000-node directed cycles with avg. node degree 4) to 13 (3000-node small world network with avg. node degree 4). In contrast to what we observe for most real life networks, we observe that small world networks have higher performance benefits than directed cycles due to decreased shortest path lengths on average.

As discussed in Chapter 8.6, the incremental *k*-centrality algorithms are quite effective at improving the performance of the clustering algorithms that use edge betweenness to partition the network into clusters (e.g. the modified Girvan-Newman clustering algorithm). However, the performance improvements obtained with the clustering algorithm that uses the incremental edge betweenness algorithm instead of the edge betweenness algorithm based on the Brandes' betweenness algorithm do not justify the additional algorithmic complexity and memory used for its computation. This is because modifying the edge with the maximum edge betweenness can trigger updates where almost the entire network is affected. Such a problem is avoided when the shortest paths are restricted to stay within first *k* hops. Hence, we recommend the use of incremental k-centrality algorithms in clustering.

As one final note, an additional use case for the incremental closeness algorithm is to use it on a large static network. By "pretending" that the network is being built incrementally, one can apply the incremental closeness algorithm and calculate closeness centrality more quickly than one can do using the non-incremental algorithm only once.

In an attempt to see what is possible in a reasonable time, we have run two additional exploratory experiments with the incremental closeness centrality algorithm and the incremental betweenness centrality algorithm.

The results of these exploratory experiments suggest that the incremental closeness centrality algorithm run on the available hardware (a quad-core PC with 256GB of RAM) can easily handle a network with 100,000 nodes and 600,000 edges. Then, we tried another run on a network with 250,000 nodes and 1,500,000 edges. Before being able to complete the entire run, the system started swapping at around 165,000 nodes and 1,000,000 edges. Once the number of edges in the network reached 1,000,000, the time required for each edge insertion increased dramatically on average due to this swapping.

This data point is not exhaustive, but it suggests that a PC with 256GB of physical memory can process incremental closeness centrality on preferential attachment based social networks with degree of 6 and somewhere between 150,000 to 200,000 nodes.

The corresponding network sizes are much smaller for the betweenness centrality. For betweenness centrality, the execution time per edge insertion starts increasing considerably when the network reaches 30,000 nodes and 180,000 edges. Once the network becomes larger than 35,000 nodes and 225,000 edges, the execution time of running the incremental betweenness algorithm on a new edge is observed increase dramatically. Note, this increase in execution time per insertion is necessarily a memory limit in this case but a result of the structure of the problem.

Finally, the ability for fast incremental computations of how power, influence, and centralities change in a dynamic network enables the use of incremental centrality metrics to rapidly identify over-time change and to set up alerts. 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 traditionally static centrality metric computational methods.

9.2 FUTURE WORK

Inline with the behavior of most incremental algorithms, the algorithms presented in this dissertation are costly in terms of memory requirements. Hence, they are primarily geared towards mid-scale networks that are composed of several thousands to tens of thousands of nodes. To make them work with large-scale networks whose numbers of nodes are mostly on the order of millions, parallelization and approximation are among the possible directions.

Parallelization: With the advents in the multi-core technology, support for multithreaded execution and utilization of computing resources is important, and as previously discussed in Chapter 2.1.3, there are a number of research studies that already focus on parallelizing betweenness algorithms for large-scale networks. Therefore, one fruitful future research direction for the work presented in this dissertation is the design of parallel, incremental centrality algorithms based on the incremental centrality algorithms proposed in this dissertation.

The incremental centrality algorithms proposed in this dissertation are based on the idea of identifying the set of nodes that are affected by an incremental network update and the subgraph defined by these nodes, which is usually a subset that is much smaller than the entire network. For instance, assume that there are two incremental network updates issued one after another and they propagate on totally disparate parts of the network. In such a case, these two incremental network updates can be issued in totally parallel with no problem because the subsets those two incremental network updates work on would have no overlap. However, for the cases when there is some overlap between the affected parts of the network for two different updates, the updates should be handled with care. For such a case, one reasonable approach is to use CREW (Concurrent Read Exclusive Write) based parallelization techniques that would allow reading different variables concurrently but would allow only one process to write to an individual cell. If the write-protection is to be achieved with atomic locks instead, then metadata common across all network updates such as the shortest path distances, the centrality values, the number of shortest paths, and the predecessors on the shortest paths need to be protected.

Approximation: This dissertation discusses the approximations of the incremental centrality algorithms through the *k*-centrality extensions in Chapter 6, where the number of hops the shortest paths are composed of should be less than or equal to *k*. However, as mentioned earlier in Chapter 2.1.3, there are other methods for approximating the centrality metrics such as the methods discussed in [57] [58], which are based on sampling a number of source (seed) nodes to calculate the shortest paths from. Towards this end, an interesting direction is to attempt combining seed based approaches with incremental algorithm design, which would selectively propagate the incremental updates in a network depending on the relationship of the source of an update with the set of sampled seed nodes. The evaluation of the accuracy (validation) for such an approximation method is also necessarily, posing another interesting but challenging problem. Another direction in this context is to combine the incremental updates with the adaptive sampling of seed nodes where the set of sampled seed nodes evolve based on the frequency and topological position of the incremental updates issued dynamically in the network.

Extensions to other centrality metrics: Apart from parallelization and approximation techniques, there are also other directions the work presented in this dissertation can be extended to. As previously mentioned in Chapter 5.6, one other potential future work direction is to design incremental centrality algorithms for other shortest path based centrality metrics such as the stress centrality [20] or the information centrality [178].

Stress centrality is defined as the number of shortest paths that pass through a node across all pairs of nodes in a network. Betweenness centrality is defined as the fraction of shortest paths that pass through a node across all pairs of nodes in a network. The definition and computation of stress centrality are very similar to that of betweenness centrality. Hence, the incremental betweenness centrality algorithm can be converted to compute the stress centrality only by changing the lines where the betweenness centrality of a node is modified.

The information centrality of a node [178] is defined as the harmonic mean of the 'bandwidth' for all the shortest paths originating from the node whose centrality is being calculated. The bandwidth of a path is calculated as the inverse of its path length. This metric is always mentioned in relation to the closeness centrality, as it also needs information on the inverse of the shortest paths lengths. The algorithm for the incremental computation of the information centrality can be obtained by modifying the lines inserted into the Ramalingam and Reps dynamic all-pairs shortest paths algorithm to support the computation of closeness centrality.

BIBLIOGRAPHY

[1] S. Wasserman and K. Faust, *Social Network Analysis*. Cambridge, UK: Cambridge University Press, 1994.

[2] S. A. Levin, "Ecosystems and the Biosphere as Complex Adaptive Systems," *Ecosystems*, vol. 1, no. 5, pp. 431--436, 1998.

[3] M. Kas et al., "What if Wireless Routers were Social? Approaching Wireless Mesh Networks from a Social Networks Perspective," *IEEE Wireless Communications*, vol. 19, no. 6, pp. 36-43, December 2012.

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

[5] E. Otte and R. Rousseau, "Social network analysis: a powerful strategy, also for the information sciences," *Journal of information Science*, vol. 28, no. 6, pp. 441-453, 2002.

[6] M. Batty, "The Size, Scale and Shape of Cities," *Science*, vol. 319, no. 5864, pp. 769--771, 2008.

[7] M. Kas et al., "Analyzing scientific networks for nuclear capabilities assessment," *Journal of the American Society for Information Science and Technology (JASIST)*, vol. 63, no. 7, pp. 1294-1312, April 2012.

[8] SocioPattern Project. (2009) Hypertext 2009 Dynamic Contact Network Dataset. [Online]. http://www.sociopatterns.org/datasets/hypertext-2009-dynamic-contact-network/

[9] K. I. Goh, E. Oh, H. Jeong, B. Kahng, and D. Kim, "Classification of Scale-free Networks," *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, vol. 99, no. 20, pp. 12583–12588, October 2002.

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

[11] P. Erdos and A. Renyi, "On the Evolution of Random Graphs," *Magyar Tud. Akad. Mat. Kutató Int. Közl*, vol. 5, pp. 17-61, 1960.

[12] D. Watts and S. Strogatz, "Collective Dynamics of 'Small-World' Networks," *Nature*, vol. 393, pp. 440-442, 1998.

[13] T. Opsahl, "Structure and Evolution of Weighted Networks," School of Business and Management, Queen Mary College University of London, London, Ph.D. Thesis 2009.

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

[15] 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.

[16] G. R. Boynton. (2010, May) G. R. Boynton's New Media and Politics. [Online]. http://ir.uiowa.edu/polisci_nmp/108/

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

[18] J. Pfeffer and K. M. Carley, "k-Centralities: Local Approximations of Global Measures Based on Shortest Paths," in *WWW*, 2012, pp. 1043-1050.

[19] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, p. 7821, 2002.

[20] A. Shimbel, "Structural parameters of communication networks," *Bulletin of Mathematical Biology*, vol. 15, no. 4, pp. 501--507, 1953.

[21] N. E. Friedkin, "Theoretical foundations for centrality measures," *The American Journal of Sociology*, vol. 96, no. 6, pp. 1478--1504, 1991.

[22] U. Brandes and D Fleischer, "Centrality measures based on current flow," in *Symposium on Theoretical Aspects of Computer Science (STACS)*, Stuttgart, Germany, 2005, pp. 533--544.

[23] S. P. Borgatti and M. G. Everett, "A graph-theoretic perspective on centrality," *Social Networks*, vol. 28, no. 4, pp. 466--484, 2006.

[24] D. Shah and T Zaman, "Rumors in a Network: Who's the Culprit?," *IEEE Transactions on Information Theory*, vol. 57, no. 8, pp. 5163--5181, 2011.

[25] P. Bonacich, "Power and Centrality: A Family of Measures," *The American Journal of Sociology*, vol. 92, no. 5, pp. 1170-1182, March 1987.

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

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

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

[29] T. Opsahl, F. Agneessens, and J. Skvoretz, "Node Centrality in Weighted Networks: Generalizing Degree and Shortest Paths," *Social Networks*, vol. 32, no. 1, pp. 245--251, 2010.

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

[31] I. McCulloh and K. M. Carley, "Longitudunal Dynamic Network Analysis: Using the Over Time Viewer Feature in ORA," Institute for Software Research, Carnegie Mellon University, Pittsburgh, Technical Report CMU-ISR-09-118, 2009.

[32] T. M. Newcomb, *The acquaintance process*. New York, NY, USA: Holt, Rinehart & Winston, 1961.

[33] S. Sampson, *Crisis in a Cloister (Doctoral Dissertation)*. Ithaca, NY, USA: Cornell University, 1969.

[34] J. Tang, C. Mascolo, M. Musolesi, and Vito Latora, "Exploiting Temporal Complex Network Metrics in Mobile Malware Containment," in *Proceedings of the 12th IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, Lucca, Italy, 2011, pp. 1--9.

[35] 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.

[36] R. Yang and L. Zhuhadar, "Extensions of closeness centrality?," in *Proceedings of the 49th Annual Southeast Regional Conference*, Kennesaw, GA, 2011, pp. 304--305.

[37] 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, pp. 186--195.

[38] 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, pp. 228--229.

[39] S. Borgatti, K. M. Carley, and D. Krackhardt, "Robustness of Centrality Measures under Conditions of Imperfect Data," *Social Networks*, vol. 28, no. 2, pp. 124--136, 2006.

[40] A.E. Sariyuce, K. Kaya, E. Saule, and U.V. Catalyurek, "Incremental Algorithms for Network Management and Analysis based on Closeness Centrality," Technical Report Uploaded on arXiv 2013.

[41] L. Leydesdorff, "Betweenness Centrality as an Indicator of the Interdisciplinarity of Scientific Journals," *Journal of the American Society for Information Science and Technology*, vol. 58, no. 9, pp. 1303--1319, 2007.

[42] P. Holme, B. J. Kim, C. N. Yoon, and S. K. Han, "Attack Vulnerability of Complex Networks," *Physical Review E*, vol. 65, no. 5, p. 056109, May 2002.

[43] M. Kitsak et al., "Betweenness Centrality of Fractal and Nonfractal Scale-free Model Networks and Tests on Real Networks," *Physical Review E*, vol. 75, no. 5, p. 056115, May 2007.

[44] T. L. Frantz, M. Cataldo, and K. M. Carley, "Robustness of centrality measures under uncertainty: Examining the role of network topology," *Computational and Mathematical Organization Theory*, vol. 15, no. 4, pp. 303-328, December 2009.

[45] L. Gulyas, G. Horvath, T. Cseri, Z. Szakolczy, and G. Kampis, "Betweenness Centrality Dynamics in Networks of Changing Density," in *19th International Symposium on Mathematical Theory of Networks and Systems (MTNS)*, Budapest,Hungary, 2010.

[46] M. Everett and S.P. Borgatti, "Ego Network Betweenness," *Social Networks*, vol.

37, no. 1, pp. 31--38, January 2005.

[47] U. Brandes, "On Variants of Shortest-Path Betweenness Centrality and Their Generic Computation," *Social Networks*, vol. 30, no. 2, pp. 136--145, May 2008.

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

[49] K. Lerman, R. Ghosh, and J. H. Kang, "Centrality Metric for Dynamic Networks," in *Proceedings of the Eighth Workshop on Mining and Learning with Graphs (MLG)*, Washington, DC, 2010, pp. 70-77.

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

[51] J. Tang, M. Musolesi, C. Mascolo, and V. Latora, "Temporal distance metrics for social network analysis," in *Proceedings of the 2nd ACM Workshop on Online Social Networks (WOSN)*, Barcelona,Spain, 2009, pp. 31-36.

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

[53] A. Berger, M. Muller-Hannemann, S. Rechner, and A. Zock, "Efficient computation of time dependent centralities in air transportation networks," in *Proceedings of the 5th international conference on WALCOM: algorithms and computation*, 2011, pp. 77-88. [Online]. http://dl.acm.org/citation.cfm?id=1966169.1966183

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

[55] O. Green, R. McColl, and D. A. Bader, "A fast algorithm for streaming betweenness centrality," in *International Conference on Privacy, Security, Risk and Trust (PASSAT) and International Conference on Social Computing (SocialCom)*, Amsterdam, Netherlands, 2012, pp. 11-20.

[56] M. J. Lee, J. Lee, J. Y. Park, R.H. Choi, and C. W. Chung, "QUBE: a Quick algorithm for Updating BEtweenness Centrality," in *Proceedings of the 21st international conference on World Wide Web (WWW)*, Lyon,France, 2012, pp. 351--360.

[57] D. Bader, S. Kintali, K. Madduri, and M. Mihail, "Approximating Betweenness Centrality," *Algorithms and Models for the Web-Graph*, vol. 4863, pp. 124--137, 2007.

[58] R. Geisberger, P. Sanders, and D. Schultes, "Better Approximation of Betweenness Centrality," in *Workshop on Algorithm Engineering and Experiments (ALENEX)*,

2008.

[59] T. Alahakoon, R. Tripathi, N. Kourtellis, R. Simha, and A. Iamnitchi, "K-path centrality: A new centrality measure in social networks," in *4th Workshop on Social Network Systems*, Salzburg, Austria, 2011, p. Article 1.

[60] D.A Bader and K. Madduri, "Parallel algorithms for evaluating centrality indices in real-world networks," in *International Conference on Parallel Processing (ICPP)*, 2006, pp. 539-550.

[61] K. Madduri, D. Ediger, K. Jiang, D.A. Bader, and D. Chavarria-Miranda, "A Faster Parallel Algorithm and Efficient Multithreaded Implementations for Evaluating Betweenness Centrality on Massive Datasets," in *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)* , Rome, 2009, pp. 1-8.

[62] G. Tan, D. Tu, and N. Sun, "A parallel algorithm for computing betweenness centrality," in *Internation Conference on Parallel Processing (ICCP)*, 2009, pp. 340-347.

[63] N. Edmonds, T. Hoefler, and A. Lumsdaine, "A Space-Efficient Parallel Algorithm for Computing Betweenness Centrality in Distributed Memory," in *International Conference on High Performance Computing (HiPC)*, Dona Paula, 2010, pp. 1--10.

[64] G. Tan, V. C. Sreedhar, and G. R. Gao, "Analysis and performance results of computing betweenness centrality on IBM Cyclops64," *The Journal of Supercomputing*, vol. 56, no. 1, pp. 1-24, 2011.

[65] P. R. Pande and D. A. Bader, "Computing betweenness centrality for small world networks on a GPU," in *Poster at 15th High Performance Embedded Computing Conference*, Lexington,MA, 2011.

[66] S. Jin et al., "A Novel Application of Parallel Betweenness Centrality to Power Grid Contingency Analysis," in *International Symposium on Parallel & Distributed Processing (IPDPS)*, Atlanta, GA, 2010, pp. 1-7.

[67] O. Coudert, J. Cong, S. Malik, and M. Sarrafzadeh, "Incremental CAD," in *Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design*, 2000, pp. 236--244.

[68] J. Cong and M. Sarrafzadeh, "Incremental Physical Design," in *Proceedings of the 2000 International Symposium on Physical Design (ISPD)*, 2000, pp. 84-92.

[69] G. S. Taylor and J. K. Ousterhout, "Magic's incremental design-rule checker," in *Proceedings of the 21st Design Automation Conference (DAC)*, Washington, DC, 1984, pp. 160-165.

[70] B. Alpern, R. Hoover, B. K, Rosen, P. F. Sweeney, and F. K. Zadeck, "Incremental evaluation of computational circuits," in *Proceedings of the First Annual SIAM Symposium on Discrete Algorithms*, Philedelphia, PA, 1990, pp. 32-42.

[71] J. Doyle, "A truth maintenance system," *Artificial Intelligence*, vol. 2, no. 3, pp. 231–272, November 1979.

[72] J. de Kleer, "An assumption based TMS," *Artificial Intelligence*, vol. 28, no. 2, pp.

127-224, March 1986.

[73] A. Howard, M. J. Mataric, and G. S. Sukhatme, "An Incremental Self-Deployment Algorithm for Mobile Sensor Networks," *Autonomous Robots, Special Issue on Intelligent Embedded Systems*, vol. 13, no. 2, pp. 113--126, September 2002.

[74] Z. Liang and Y. Li, "Incremental support vector machine learning in the primal and applications," *Neurocomputing*, vol. 72, no. 10, pp. 2249–2258, June 2009.

[75] A. Gijsberts, "Incremental Learning for Robotics with Constant Update Complexity," Department of Communication, Computer, and System Sciences, University of Genoa, Ph.D. Thesis 2011.

[76] W. Pugh and T. Teitelbaum, "Incremental computation via function caching," in *16th ACM Symposium on Principles of Programming Languages (POPL)*, Austin, TX, 1989, pp. 315-328.

[77] L. A. Lombardi and B. Raphael, "LISP as the language for an incremental computer," in *The programming Language LISP: Its Operation and Applications*. Cambridge, MA, USA: M.I.T. Press, 1964, pp. 204-219.

[78] M. Burke and L. Torczon, "Interprocedural optimization: eliminating unnecessary recompilation," *Transactions on Programming Languages and Systems*, vol. 15, no. 3, pp. 367-399, 1993.

[79] G. Ramalingam and R. Thomas, "A categorized bibliography on incremental computation," in *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, New York, NY, 1993, pp. 502-510.

[80] U. A. Acar, G. E. Blelloch, M. Blume, and K. Tangwongsan, "An experimental analysis ofr self-adjusting computation," in *Programming Language Design and Implementation (PLDI)*, Ottawa, Canada, 2006, pp. 96-107.

[81] M. Carlsson, "Monads for incremental computing," in *Proceedings of the 7th SIGPLAN International Conference on Functional Programming*, 2002, pp. 26-35.

[82] U. A. Acar, G. E. Blelloch, and R. Harper, "Adaptive functional programming," in *Proceedings of the 29th Annual Symposium on Principles of Programming Languages*, 2002, pp. 247-259.

[83] A. Gupta and I. S. Mumick, "Maintenance of Materialized Views: Problems, Techniques, and Applications," *IEEE Quareterly Bulletin on Daqta Engineering*, vol. 18, no. 2, 1995, Special Issue on Materialized Views and Data Warehousing.

[84] L. Baekgaard and L. Mark, "Incremental computation of nested relational query expressions," *Transactions on Database Systems*, vol. 20, no. 2, pp. 111 - 148 , June 1995.

[85] A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, and K. A. Ross, "Implementing incremental view maintenance in nested data models," in *1997*, Proceedings of the International Workshop on Database Programming Languages, pp. 202-221.

[86] J. Liu, M. Vincent, and M. Mohania, "Incremental evaluation of nest and unnest

operators in nested relations," in *roceedings of the Second International Symposium on Cooperative Database Systems for Advanced Applications (CODAS)*, Wollongong, Australia, 1999, pp. 264-275.

[87] D. Gluche, T. Grust, C. Mainberger, and M.H. Scholl, "Incremental updates for materialized views user-defined functions," in *Proceedings of the 5th International Conference on Deductive and Object Oriented Databases*, 1997.

[88] M. A. Ali, A. A. A. Fernandes, and N. W. Paton, "Incremental maintenance of materialized OQL views," in *International Workshop on Data Warehousing and OLAP (DOLAP)*, Washington, D.C, USA, 2000.

[89] H. Nakamura, "Incremental Computation of Complex Object Queries," in *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Tampa, FL, USA, 2001, pp. 156-165.

[90] B. Fortz and M. Thorup, "Internet traffic engineering by optimizing OSPF weights," in *Proceedings of 19th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, vol. 2, 2000, pp. 519--528.

[91] P. Bhatotia, M. Dischinger, R. Rodrigues, and U. A. Acar, "Slider: Incremental Sliding-Window Computations for Large-Scale Data Analysis," CITI, Universidade Nova de Lisboa, Lisbon, Technical Report 2012.

[92] M, Dischinger et al., "Glasnost: Enabling End Users to Detect Traffic Differentiation," in *Proceedings of the 7th Conference on Networked Systems Design and Implementation (NSDI)*, 2010.

[93] G. A. Cheston, "Incremental algorithms in graph theory," Department of Computer Science, University of Toronto, Toronto, Ph.D. Dissertation 1976.

[94] W. Fan et al., "Incremental Graph Pattern Matching," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, vol. 11, Athens, Greece, 2011, pp. 925-936.

[95] D. Conte, P. Foggia, C. Sansone, and M. Vento, "Thirty years of graph matching in pattern recognition," *International Journal of Pattern Matching and Artificial Intelligence*, vol. 18, no. 3, pp. 265-298, May 2004.

[96] C. M. Hoffmann and M. J. O'Donnell, "Pattern Matching in Trees," *Journal of ACM*, vol. 29, no. 1, pp. 68--95, January 1982.

[97] G. Frederickson, "Data structures for online updating of minimum spanning trees," *SIAM Journal of Computing*, vol. 14, no. 4, pp. 781-798, November 1985.

[98] S. Liu, D. Epley, and W. Decker, "An Incremental Distributed Minimum Directed Spanning Tree Algorithm," in *Parallel and Distributed Computing and Systems (PDCS)*, Phoenix, AZ, USA, 2005.

[99] G.F. Italiano, "Finding paths and deleting edges in directed acyclic graphs," *Information Processing Letters*, vol. 28, no. 1, pp. 5-11, May 1988.

[100] J. A. La Poutre, "Maintenance of triconnected components of a graph,"

Proceedings of the 19th International Colloqium on Automata, Languages, and Programming, pp. 354-365, July 1992.

[101] V. King and G. Sagert, "A fully dynamic algorithm for maintaining the transitive closure," in *Proceedings of the 31st Annual Symposium on Theory of Computing (STOC)*, Atlanta, GA, USA, 1999a, pp. 492-498.

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

[103] 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*, 1999b, pp. 81--89.

[104] G. Ramalingam and T. Reps, "On the Computational Complexity of Incremental Algorithms," Computer Science, University of Wisconsin at Madison, Madison, Technical Report 1991.

[105] G. Ramalingam and T. Reps, "On the Computational Complexity of Dynamic Graph Problems," *Theoretical Computer Science*, vol. 58, no. 1-2, pp. 233-277, 1996.

[106] J. Moy. (1998, April) OSPF Version 2. Online. [Online]. http://tools.ietf.org/html/rfc2328

[107] T. Reps. (2011, December) Thomas W. Reps -- Past Research Accomplishments. [Online]. http://pages.cs.wisc.edu/~reps/past-research.html

[108] 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.

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

[110] M. R. Anderberg, *Cluster analysis for applications*. New York, NY, USA: Academic Press Inc., 1973.

[111] A. K. Jain and R. C. Dubes, *Algorithms for Clustering Data*. Upper Saddle River, NJ, USA: Prentice Hall Inc., 1988.

[112] A. K. Jain, "Data clustering: a review," *ACM Computing Surveys (CSUR)*, vol. 31, no. 3, pp. 264-323, 1999.

[113] M. Ester, H. P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *In Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining*, 1996, pp. 226-231.

[114] C. Fraley and A. E. Raftery, "How many clusters? Which clustering method? Answers via model-based cluster analysis," *The computer journal*, vol. 41, no. 8, pp. 578-588, 1998.

[115] G. Fung, "A comprehensive overview of basic clustering algorithms," June 2001.

[116] R. Sibson, "SLINK: an optimally efficient algorithm for the single-link cluster method," *The Computer Journal*, vol. 16, no. 1, pp. 30-34, 1973.

[117] T. Sorensen, "A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on Danish commons," *Biol. Skr.*, vol. 5, pp. 1-34, 1948.

[118] D. Defays, "An efficient algorithm for a complete link method," *The Computer Journal*, vol. 20, no. 4, pp. 364-366, 1977.

[119] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, no. 2, p. 026113, 2004.

[120] M. Chavent, Y. Lechevallier, and O. Briant, "DIVCLUS-T: a monothetic divisive hierarchical clustering method," *Computational Statistics and Data Analysis*, vol. 52, no. 2, pp. 687-701, 2007.

[121] N. Yuruk, M. Mete, X. Xu, and T. A. J. Schweiger, "A Divisive Hierarchical Structural Clustering Algorithm for Networks," in *IEEE ICDM Workshop on Mining Graphs and Complex Structures, In conjunction with the Seventh IEEE Int. Conf. of Data Mining*, Omaha, NE, 2007.

[122] P. N. Tan, M. Steinbach, and V. Kumar, "Cluster Analysis: Basic Concepts and Algorithms," in *Introduction to Data Mining.*: Pearson Education India, 2006, pp. 487-568.

[123] R. Xu and D. Wunsch II, "Survey of Clustering Algorithms," *IEEE Transactions on Neural Networks*, vol. 16, no. 3, pp. 645-678, May 2005.

[124] L. Zadeh, "Fuzzy Sets," *Information and Control*, vol. 8, pp. 338--353, 1965.

[125] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3, pp. 75-174, 2010.

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

[127] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. Cambridge, MA, USA: MIT Press, 2001.

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

[129] A. M. Berman, M. C. Paull, and B. G. Ryder, "Proving relative lower bounds for incremental algorithms," *Acta Informatica*, vol. 27, no. 7, pp. 665-683, 1990.

[130] D.R. White and D. Smith, "Flow Centralities: Do they Predict the Economic Rise and Fall of States?," in *INSNA Sunbelt Conference*, San Diego, CA, 1988.

[131] K. Jiang, D. Ediger, and D. A. Bader, "Generalizing k-Betweenness Centrality Using Short Paths and a Parallel Multithreaded Implementation," in *International Conference on Parallel Processing (ICPP)*, Vienna, Austria, 2009, pp. 542 - 549.

[132] D. Ediger et al., "Massive Social Network Analysis: Mining Twitter for Social Good," in *The 39th International Conference on Parallel Processing (ICPP)*, San

Diego, CA, USA, 2010, pp. 583-593.

[133] D. Gkorou, J. Pouwelse, and D. Epema, "Efficient Approximate Computation of Betweenness Centrality," in *Advanced School for Computing and Imaging (ASCI)*, Veldhoven, Netherlands, 2010.

[134] D. Gkorou, J. Pouwelse, and D. Epema, "Betweenness Centrality Approximations for an Internet Deployed P2P Reputation System," in *International Parallel and Distributed Processing Symposium Workshops (HotP2P)*, 2011, pp. 1627 - 1634.

[135] S. Gregory, "Local Betweenness for Finding Communities in Networks," Computer Science, University of Bristol, Bristol, UK, Technical Report 2008.

[136] F. Salvetti and S. Srinivasan, "Local Flow Betweenness Centrality for Clustering Community Graphs," in *1st Workshop on Internet and Network Economics (WINE)*, vol. 3828, Hong Kong, 2005, pp. 531-544.

[137] F. Radicchi, C. Castellano, F. Cecconi, V. Loreto, and D. Parisi, "Defining and identifying communities in networks," *PNAS*, vol. 101, no. 9, pp. 2658–2663, March 2004.

[138] S. Gregory, "An Algorithm to Find Overlapping Community Structure in Networks," in *Knowledge Discovery in Databases: PKDD*, vol. 4702, 2007, pp. 91-102.

[139] S. Gregory, "A Fast Algorithm to Find Overlapping Communities in Networks," *Machine Learning and Knowledge Discovery in Databases*, vol. 5211, pp. 408-423, 2008.

[140] J. O'Madadhain, D. Fisher, S. White, and Y. Boey. (2003) The JUNG (Java Universal Network/Graph) Framework. [Online]. http://jung.sourceforge.net/doc/

[141] S. Nanda and D. Kotz, "Localized Bridging Centrality for Distributed Network Analysis," in *ICCCN*, 2008.

[142] M. Kas, I. Korpeoglu, and E. Karasan, "OLSR-aware channel access scheduling in wireless mesh networks," *Journal of Parallel and Distributed Computing*, vol. 71, no. 9, pp. 1225-1235, September 2011, Special Issue on Advancement of Research in Wireless Access and Mobile Systems.

[143] M. Kas, I. Korpeoglu, and E. Karasan, "OLSR-aware distributed channel access scheduling for wireless mesh networks," in *Wireless Communications and Networking Conference (WCNC)*, Budapest, Hungary, 2009, pp. 1-6, DOI:10.1109/WCNC.2009.4917841.

[144] IEEE Standards, "Draft of Wireless LAN Medium Access Control (MAC) and Phyical Layer Specifications," Institute of Electrical and Electronics Engineers, 802.11, 1997.

[145] M. Kas, I. Korpeoglu, and E. Karasan, "Utilization-based dynamic scheduling algorithm for wireless mesh networks," *EURASIP Journal on Wireless Communications and Networking*, vol. 2010, p. 312828, October 2010, Open Access: http://jwcn.eurasipjournals.com/content/2010/1/312828.

[146] IEEE Standards, "IEEE Standard for Local and Metropolitan Area Networks Part 16: Air Interface for Fixed Broadband Wireless Access Systems," Institute of Electrical and Electronics Engineers, 2004.

[147] IEEE 802.11s Task Group, "Draft Amendment to Standard for Information Technology - Telecommunications and Information Exchange Between Systems - LAN/MAN Specific Requirements- Part 11: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications:Amendment: ESS Mesh," Institute of Electrical and Electronics Engineers, 2006.

[148] R. Albert and A. L. Barabasi, "Statistical mechanics of complex networks," *REVIEWS OF MODERN PHYSICS*, vol. 74, no. 1, pp. 47-97, January 2002.

[149] R. K. Merton, "The Matthew Effect in Science," *Science*, vol. 159, no. 3810, pp. 56-63, 1968.

[150] E. N. Gilbert, "Random graphs," *Ann. Math. Statist.*, vol. 30, pp. 1141-1144, 1959.

[151] M.D. Humphries and K. Gurney, "Network 'Small-World-Ness': A Quantitative Method for Determining Canonical Network Equivalence," *PLoS One*, vol. 3, no. 4, 2008.

[152] J.W. Bohland and A.A. Minai, "Efficient associative memory using small-world architecture," *Neurocomputing*, pp. 489-496, 2001.

[153] F. Jansson, "Pitfalls in Spatial Modelling of Ethnocentrism: A Simulation Analysis of the Model of Hammond and Axelrod," *Journal of Artificial Societies and Social Simulation*, vol. 16, no. 3, p. 2, June 2013.

[154] D.H. Zanette, "Critical behavior of propagation on small-world networks," February 2008.

[155] H. Zhao, J. Zhou, A. Zhang, G. Su, and Y. Zhang, "Self-organizing Ising model of artificial financial markets with small-world network topology," *Europhysics Letter (EPL)*, vol. 101, no. 1, December 2012.

[156] A Ramezanpour and V. Karimipour, "Simple models of small world networks with directed links," Department of Physics, Sharif University of Technology, Tehran, Iran, Tech Report 2008.

[157] J. Xu, "Markov Chain Small World Model with Asymmetric Transition Probabilities," *Electronic Journal of Linear Algebra*, vol. 17, pp. 616-636, November 2008.

[158] W. Liao et al., "Small-world directed networks in the human brain: multivariate Granger causality analysis of resting-state fMRI," *Neuroimage*, vol. 54, no. 4, pp. 2683-94, February 2011.

[159] C.P. Zhu, S.J. Xiong, Y.J. Tian, N. Li, and K.S. Jiang, "Scaling of Directed Dynamical Small-World Networks with Random Responses," *Physical Review Letters*, vol. 92, no. 24, pp. 218702:1-4, May 2004.

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

[161] Cornell University. (2011, January) Cornell University Library (arXiv). [Online]. http://arxiv.org/

[162] (2003, August) SIGKDD CUP. [Online]. http://www.sigkdd.org/kdd2003/kddcup.html

[163] OpenNap. (2011, November) OpenNap: Open Source Napster Server. [Online]. http://opennap.sourceforge.net/

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

[165] A. Dutot, F. Guinand, D. Olivier, and Y. Pigné, "Graphstream: A tool for bridging the gap between complex systems and dynamic graphs," in *Satellite Conference within the 4th European Conference on Complex Systems*, Dresden, Germany, 2007, pp. 1-10.

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

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

[168] Cosma Rohilla Shalizi, M.E.J. Newman Aaron Clauset, "Power-Law Distributions in Empirical Data," 2009.

[169] M. Kas, "Structures and Statistics of Citation Networks," Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, MS Thesis 2011.

[170] C. Moore and M. E. J. Newman A. Clauset, "On the Frequency of Severe Terrorist Attacks," vol. 51, no. 58, 2007.

[171] B. M. Hill, "A Simple General Approach to Inference about the Tail of a Distribution," vol. 3, no. 5, 1975.

[172] Oracle. (2011) JAVA API. [Online]. http://docs.oracle.com/javase/6/docs/api/java/lang/System.html

[173] StackExchange. (2012, July) System.gc() in Java. [Online]. http://stackoverflow.com/questions/66540/system-gc-in-java

[174] M. E. J. Newman, "Scientific collaboration networks. II. Shortest paths, weighted networks, and centrality.," *Physical Review E*, vol. 64, no. 1, p. 016132, 2001.

[175] I. Gorodezky and Y. Tang, "The funneling phenomenon in a coauthorship network," Center for Applied Mathematics (CAM), Cornell University, Ithaca, NY, Project Report 2008.

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

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

[178] K. Stephenson and M. Zelen, "Rethinking centrality: Methods and examples," *Social Networks*, vol. 11, no. 1, pp. 1--37, 1989.