GRAPH ANALYSIS OF STREAMING RELATIONAL DATA

A Dissertation
Presented to
The Academic Faculty

By

Anita N. Zakrzewska

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computational Science and Engineering

Georgia Institute of Technology

May 2018

Copyright © Anita N. Zakrzewska 2018

GRAPH ANALYSIS OF STREAMING RELATIONAL DATA

Approved by:

Dr. David A. Bader, Advisor
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Ümit Çatalyürek
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Bistra Dilkina
School of Engineering
University of Southern California

Dr. Constantine Dovrolis
School of Computer Science
Georgia Institute of Technology

Dr. Srinivas Aluru
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. E. Jason Riedy
School of Computational Science and Engineering
Georgia Institute of Technology

Date Approved: March 16, 2018

To my parents, Elżbieta and Radosław, and to my sister, Aleksandra.

ACKNOWLEDGEMENTS

I would like to thank my advisor, David Bader, for all his expertise and mentorship throughout these years. Working in his lab has been a fulfilling and engaging experience. A special thanks goes to Jason Riedy for his valuable advice and collaboration. Thank you to the rest of my committee, Ümit Çatalyürek, Bistra Dilkina, Constantine Dovrolis, and Srinivas Aluru. I am grateful for their feedback and time. I would also like to thank all my colleagues at Georgia Tech, especially my friends and co-authors Eisha Nathan and James Fairbanks. My family and friends have provided great support throughout the years. A heartfelt thanks goes to Lucas Giordano for his support and making me laugh when it was most needed. Finally, I would like to express my deepest gratitude to my parents and sister for their continual encouragement and help during my Ph.D. days and throughout my life. I could not be where I am today without them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Graphs are used to represent data from a variety of sources. As the volume of information available has exploded, there has been growing interest in the application of graph analysis to real-time data. For example, graph analysis techniques can be applied to activity from social networks, online communication, financial transactions, and sensor data. Studying such datasets poses many challenges, including dealing with the high volume of data and the speed with which it is generated and then transforming the relational information into dynamic graphs. This dissertation addresses challenges that occur throughout the graph analysis process.

Because many datasets are large and growing, it may be infeasible to collect and build a graph from all the data that has been generated. This dissertation addresses the challenges created by large volumes of streaming data through new sampling techniques. The algorithms presented can sample a subgraph in a single pass over an edge stream and are therefore appropriate for dynamic data. This dissertation also presents a sampling algorithm that can produce a temporally biased subgraph. Because this method emphasizes edges with recent timestamps, it is useful for applications in which newer data is more relevant.

Before graph analysis techniques can be applied, a graph must first be created from the data collected. This is especially challenging when creating dynamic graphs, which represent relationships that change over time. In particular, it is not obvious how to remove old data, especially when edges are derived from interactions, which are not explicitly reversed, but rather may decrease in relevance. This dissertation evaluates several methods of aging old data to create dynamic graphs. In addition to methods used in the literature, a new approach is proposed, based on the concept of preserving edges in active portions of the graph. The method of aging data is important and should be carefully chosen because it will affect the structure of the dynamic graph and therefore the result of algorithmic analysis.

This dissertation also contributes new techniques for dynamic community detection and analysis. In order to be useful for streaming data, community detection algorithms should handle rapid changes to the graph. This work presents a new algorithm for local community detection, which incrementally updates when the graph changes. By incrementally updating, it can produce new results faster compared to re-computing from scratch with a static algorithm. The creation of dynamic graphs allows us to study community changes over time. This work addresses the topic of community analysis with a new vertex-level measure of community change.

Together, these contributions advance the study of streaming relational data through graph analysis. The dissertation addresses challenges arising in multiple stages of the process, improving our ability to learn from the vast and ever-growing amounts of data at our disposal.

CHAPTER 1
INTRODUCTION

Every day, people log into social networks to consume content, make posts, and keep in touch with contacts. All this activity generates an enormous amount of data. According to the website Internet Live Statistics, which tracks internet use, there are currently over 300 million active users on Twitter, generating around 500 million tweets a day. Facebook has over 2 billion active users, who create data by posting, reacting to others' posts, sending instant messages, and managing a list of contacts. Every second, hundreds of photos are uploaded to Instagram, thousands of calls are placed with Skype, and millions of emails are sent [1]. Another large source of data generation is personal tracking. It is now common to track activities such as exercise, location, and sleep. One example is Strava, a platform on which users can upload their rides and runs, react to friends' activity, and see which other users they came into contact with while exercising. This platform was recently in the news when it was discovered that data collected by users revealed locations of military bases and even road networks inside them.

Much of this data stores relationships between entities. This relational data can therefore be represented as a graph of vertices and edges. An edge between two people may represent interaction between them, such as one user re-tweeting another's post. On Facebook, edges can be formed from interaction such as reacting to a post or tagging a contact in a photo. Communication graphs can be created from email or instant messaging contact between people. Edges can also represent explicit relationships, such as friend or contact lists.

Graph analysis can provide insight into the structure of relational data and the underlying processes and behaviors that generate it. Here we use the term relational to refer to data that expresses relationships between entities. Applying graph analysis to data from sources such as online activity, however, is not straightforward. Challenges include the large volume of data, the speed with which it is generated, and the transformation of the data into graphs. First, data must be collected. For applications in which it is continually being generated, the collection process must take into account this streaming nature. For example, the data may be available only as a stream of elements of unknown order. Very large datasets may be too large to fully store or to run computationally intensive analytics. If so, it may be necessary to sample only a portion of the dataset. Once data is collected, a graph must be created from it and the method of doing so is not always straightforward. If dynamic graph analysis is performed, then it is necessary to decide how a changing graph will be built from temporal data. Finally, once a graph is created, structural properties can be evaluated with graph metrics. The graph algorithms must scale to the size of the data and be able to keep up with the pace of change.

1.1 Contributions

This dissertation contributes to the field of dynamic graph analysis by addressing several of the challenges described above. Because many real-life datasets are large and continually growing, it is often infeasible to collect, store, and build a graph from the entire dataset. In these cases, a representative portion of the data may be sampled. When sampling a graph, the goal is typically to obtain a sampled subgraph that is structurally similar to the graph that would be obtained from the full dataset. If the dataset is dynamic, with new relational edges continually generated, then it may only be available as a stream of edges. For these cases, sampling algorithms will need to process a stream of edges in a single pass. Most graph sampling algorithms from the literature, however, sample a static graph and assume random access. Chapter 3 presents new algorithms for sampling a graph from a stream of edges in a single pass. The algorithms Weighted Edge Sampling and Randomly Induced Edge Sampling produce a non-temporally biased subgraph with a pre-determined number of edges. Experiments show that these new algorithms perform better than previous methods from the literature. Chapter 3 also describes Temporal Weighted Edge Sampling, a new algorithm that samples an edge stream to produce a temporally biased subgraph. Because this method includes more edges with recent timestamps, it is useful in applications where newer data is considered more relevant.

Dynamic graphs represent relationships that change over time. Therefore, they can be created from a stream of edge data. For example, new edges may be added to reflect new relationships and edges may be removed when the underlying relationships lose relevance. In graphs with edge weights, weight changes will represent an increase or decrease in the strength of a relationship. There exists a large body of work presenting algorithms for dynamic graph analysis. However, before any such analysis can be performed, a dynamic graph must first be created. It is necessary to decide under which conditions edges will be added, modified, and removed. The way in which this is done will affect the structure of the graph and therefore analysis results. While it is typical for new edges to always be added to a graph, the question of when to remove or down-weight edges is less straightforward. Chapter 4 evaluates methods of removing old data, or aging, that have been used in the literature. It also presents a new approach based on the concept of preserving edges in active portions of the graph. The motivation for the new method is to preserve information about relationships between important entities in the network. Because the method of aging data may affect results of algorithmic analysis, it should be carefully chosen.

Once a dynamic graph has been created, graph metrics may be computed and analysis performed. Chapters 5 and 6 contribute new algorithms for community detection and analysis on dynamic graphs. Chapter 5 presents Ordered Dynamic Seed Expansion, a new algorithm for finding local communities for seed vertices of interest. The algorithm finds a community relevant to a given seed or set of seeds and incrementally updates the result when the graph changes. By incrementally updating, Ordered Dynamic Seed Expansion can produce new results faster than re-computing from scratch with a static algorithm. This makes it more able to keep up with a fast pace of new incoming data. Once communities are computed, it is possible to analyze how they change over time. For example, one can track their evolution, flagging events such as new communities appearing or communities merging together and splitting apart. Vertex level events can also be found. Chapter 6 presents a new measure to detect when vertices switch communities. This local measure uses changes in a vertex's neighborhood to flag a community change. Because it is local, it works well when global communities are unstable over time. The measure can be helpful in flagging entities with unusual behavior for further analysis.

CHAPTER 2
BACKGROUND

2.1 Graph Analysis

Graphs are used to represent relationships from a variety of data sources, such as online communication, biological networks, and financial transactions. A graph $G = (V, E)$ is composed of a set of vertices V and edges E connecting the vertices. In the most basic case, an edge is a tuple of two vertices (u, v). However, edges may also have a weight, timestamp, and additional attributes. In this dissertation, edges are undirected and may contain a weight attribute $(u, v).weight$ and a time attribute $(u, v).time$. Edge weights may represent the relative importance of particular edges in graph analysis; for example, they may be related to edge aging, as explained later herein. Edge timestamps may correspond to times when edges are created based on an input data stream, or to other events underlying the input data. In the case when a graph has no time attributes, the timestamp may be omitted. For unweighted graphs, the weight of each edge is simply 1 and may also be omitted. Each vertex has a set of neighbors $N(v)$, which are the vertices it is connected to through an edge. The weighted degree of each vertex is then the sum of all its edge weights.

2.1.1 Static Analysis

Graph analysis is used to interpret relational data and provide insights into the underlying real life processes that generate the relationships. Here we use the term relational to refer to data that expresses relationships between entities. Such data can be represented as a set of vertices and edges. The structure of the data can be characterized with various graph measures. A basic measure is the distribution of vertex degrees. Networks representing social connections, for example, typically have a skewed degree distribution, with a small number of vertices with high degree and a large number with few neighbors. Road networks, on the other hand, do not, as a single intersection can connect only a few streets. Another basic metric is the diameter, or the maximum distance between any pair of vertices. A path of length h between vertices u and w is a sequence of h edges $((u, v_1), (v_1, v_2), \ldots, (v_{h-1}, w))$ in G. The distance between two vertices is then the length of the shortest path between them. Social networks tend to be "small world" graphs with low diameter in which each entity is only a few hops away from any other entity. Along with the diameter, other statistics of the distribution of shortest path lengths between vertices are used to understand graph structure. They may be useful, for example, in understanding how information or diseases spread.

Triangles are a heavily studied graph feature. A triple is a set of three vertices for which at least one is connected to the other two. A triangle, or closed triple, is a set of three vertices, each of which is connected to the other two. Graphs with many triangles are heavily clustered, meaning that two connections of an entity are also likely to be connected. This concept is measured with the clustering coefficient. The local clustering coefficient of a vertex gives the ratio of the number of triangles it participates in over the number of triples it participates in. The global clustering coefficient is the ratio of the number of triangles to the number of triples within the entire graph.

Another feature used to analyze the structure of graphs is vertex centrality. Several different types of centrality have been proposed in the literature, each measuring vertex importance in some way. The simplest, degree centrality, refers to the number of neighbors of a given vertex. Closeness centrality measures the distance of a vertex to other vertices. Betweenness centrality measures the number of shortest paths between all vertex pairs that pass through a given vertex. Centrality measures can be used to identify important vertices which are key to the spread of information or diseases. For example, betweenness centrality was used to find important Twitter accounts that disseminated information during the H1N1 influenza outbreak and a flood in Atlanta, Georgia [2].

The measures discussed above all apply to static graphs, which do not change and have a constant set of vertices and edges. Such graphs represent permanent relationships, whose existence may be possible because the underlying data is fully available and does not change. Alternatively, data spanning a certain period of time may be collected and then used to create a static graph, thus ignoring any data created earlier or later. Analysis of static graphs ignores any possible temporal relationships that might be present in the source data and thus provides only limited insights into relationships and processes that gave rise to that data.

2.1.2 Dynamic Analysis

A dynamic graph represents time-changing relational data. In this dissertation, a dynamic graph is a sequence of graph snapshots over time $\{G_0, \ldots, G_t\}$. Each $G_t = (V_t, E_t)$ is a static graph representing the state of the dynamic graph at time t. In this work, time t is a discretized variable and not continuous time. Dynamic graph analysis is used when relationships may change and entities may enter or leave the network. This is represented by the addition, deletion, or modification of edges from one snapshot to the next.

The static metrics discussed above may be applied to dynamic graphs by simply computing and updating them for each new graph snapshot in time. For example, the clustering coefficient may be maintained, either by computing it from scratch every time the graph changes and an updated result is needed or by incrementally updating the value based on the changes that occurred.

Dynamic graphs also allow for dynamic analysis, which measures how the graph changes and detects important or unusual events. For example, Leskovec *et al.* study graph densification over time and find that the diameter of graphs decreases as more edges are added [3]. By measuring and updating the centrality scores of vertices, it is possible to track the behavior of important vertices and detect changes in their behavior. For example, a vertex with low importance suddenly becoming very central may be a significant event. Another example, dynamic community analysis, is discussed here in Section 2.2.4

2.2 Community Detection

A commonly studied feature of graphs is community structure. A graph community, or cluster, may be broadly defined as a set of vertices that is densely connected. While there is no single definition of communities, they should have high internal edge density and few inter-community edges. For example, in graphs representing online social networks, multiplayer games, or project management tools, graph communities can show groups of friends or family, game teammates, or officemates who work together on the same project, respectively.

Community detection is related to the task of graph partitioning because both attempt to find sets of vertices that are strongly connected to each other and weakly connected to the rest of the graph. However, the two tasks have different goals. Graph partitioning divides a graph into a pre-determined number of groups, typically of similar size, while minimizing the number of edges between them. A typical graph partitioning application is dividing computational tasks between processors for parallel computation so that inter-processor communication is minimized. This differs from community detection in several ways. First, the number of partitions is determined by the number of processors and the application and is therefore known at the outset. Second, the partitions should be of approximately equal size to load balance the problem. Third, the best partitions must be returned regardless of whether the graph naturally divides well. Community detection, on the other hand, seeks to determine what intrinsic structure exists in the graph. Therefore, the number of communities to return is not pre-determined, but depends on the graph. The sizes of communities may differ both in the number of vertices and edges they contain. Finally, some community detection algorithms place each vertex in exactly one community so that the communities form a partition of the graph. Others methods return overlapping, or fuzzy, clusters by placing each vertex in one or more communities.

A set of r communities in a graph G is represented by $\{C_1, \ldots, C_r\}$. When referring to communities in a dynamic graph, the r_t communities in snapshot G_t are $\{C_{1,t}, \ldots, C_{r_t,t}\}$. In those cases where there is only a single community discussed, it is simply denoted C. Let k_{in}^C be the sum of all edge weights interior to community C and k_{out}^C be the sum of all edge weights on the border of C.

$$k_{in}^C = \sum_{(u,v) \in E | u \in C \wedge v \in C} (u,v).weight \tag{2.1}$$

$$k_{out}^C = \sum_{(u,v) \in E | u \in C \wedge v \notin C} (u,v).weight \tag{2.2}$$

2.2.1 Quality Measures

While there is no single definition of what constitutes a community in a graph or how to measure its quality, many edges inside the community and few edges connecting it to the rest of the graph are desired for the cluster to be considered well formed. One approach defines and evaluates a community individually, focusing on the vertices and edges within the community or adjacent to it and ignoring the rest of the graph. Such definitions often require each member vertex to be sufficiently connected to a number of other members. Several defintions are discussed in [4]. For example, an *n*-clan is a maximal subgraph for which the distance between any pair of vertices, using edges within the subgraph, is not greater than n. A *k*-core is a maximal subgraph for which each member is a neighbor of at least k other members. Other definitions also impose limits on the number of edges between the community and the rest of the graph. One example is an LS-set, a subgraph in which each member vertex has more neighbors within the community than outside of it. A relaxed version requires only the total number of edges within the community to be greater than those connecting it to the rest of the graph. Such definitions, however, may be too restrictive for many applications.

Instead, sets of vertices can be evaluated as communities using quality measures, also known as fitness measures. The higher the value of a quality measure, the better the community structure of the set. The community detection problem then becomes the task of selecting those sets of vertices whose community fitness is sufficiently high. The intra-cluster density $f_{int}(C)$ measures the ratio of the number of intra-community edges to the maximum number that would be possible if all vertices were connected to each other.

$$f_{int}(C) = \frac{k_{in}^{C}}{|C|\,(|C|-1)/2} \tag{2.3}$$

Another fitness measure is the relative density, which gives the ratio of the number of intra-community edges to the total community degree. Lancichinetti *et al.* use $f_{LFM}(C)$, a modification of relative density with a resolution parameter α, which controls the relative importance of having many intra-community edges compared to having few border edges.

$$f_{LFM}(C) = \frac{2k_{in}^{C}}{(2k_{in}^{C} + k_{out}^{C})^{\alpha}} \tag{2.4}$$

Another fitness measure is $f_{MONC}(C)$, which further modifies the quality measure to allow for singleton communities [5].

$$f_{MONC}(C) = \frac{2k_{in}^{C} + 1}{(2k_{in}^{C} + k_{out}^{C})^{\alpha}} \tag{2.5}$$

Instead of evaluating individual communities locally, communities forming a partition of a graph can also be evaluated globally with a quality measure. A popular measure is modularity, shown in Equation 2.6, which compares the number of intra-community edges to the expected number under a random null model [6]. The motivation behind modularity is that a random graph will not have community structure. A given partition with more intra-community edges, and thus more community structure, than expected, will have a higher modularity. The null model maintains the vertex degrees found in the graph, but with randomly rewired edges.

$$Q = \frac{1}{|E|} \sum_{C_i} k_{in}^{C_i} - \frac{(2k_{in}^{C_i} + k_{out}^{C_i})^2}{4\,|E|} \tag{2.6}$$

A good community assignment also minimizes the number of inter-community edges, or the edge cut. However, considering only the number of inter-community edges alone can lead to individual vertices forming singleton clusters. Therefore, quality measures commonly used also take into account the size of communities. Two such measures are the ratio cut [7], which divides the number of inter-community edges by the number of vertices in each community, and the normalized cut [8, 9], which uses the total number of inter- and intra-community edges as the normalizing factor.

$$RatioCut(C_1, \ldots, C_r) = \frac{1}{2} \sum_{C_i} \frac{k_{out}^{C_i}}{|C_i|} \tag{2.7}$$

$$NCut(C_1, \ldots, C_r) = \frac{1}{2} \sum_{C_i} \frac{k_{out}^{C_i}}{2k_{in}^{C_i} + k_{out}^{C_i}} \tag{2.8}$$

2.2.2 Detection Algorithms for Static Graphs

The most common type of community detection algorithm finds global, non-overlapping communities that form a partition of the vertices. Such a global approach is useful when it is necessary to know the structure of the entire graph. The results are simple to interpret and to input to other algorithms because each vertex belongs to exactly one community and thus receives a single label. However, many graphs are composed of overlapping communities and in these cases non-overlapping algorithms may output low quality results.

An early algorithm by Girvan and Newman divides a graph into clusters by repeatedly removing the edge with highest betweenness centrality [10]. This removal eventually disconnects the vertices and returns the resulting connected components as clusters. The intuition behind this approach is that highly central edges are likely to form bridges between less connected portions of the graph. However, because the centrality measure must be recomputed after each edge removal, the worst case complexity of $\mathcal{O}(m^2n)$ for n vertices and m edges is high, and the algorithm can only be run on smaller datasets.

Many algorithms used are greedy, agglomerative methods. The Clauset Newman Moore algorithm greedily maximizes modularity by initially placing each vertex in its own singleton cluster and then repeatedly contracting edges [11]. At each iteration the change in modularity that would result from joining two neighboring clusters into one is computed. The pair with the highest change is chosen and the two communities are joined into one. Multiple works have suggested improvements to this algorithm to achieve better modularity optima or to speed up computation [12, 13, 14]. Riedy *et al.* present a parallel version that uses a maximal edge matching to contract multiple edges at once [15]. The Louvain algorithm is a popular and fast modularity maximizing method [16]. Each vertex also starts as a singleton cluster. Next, for each vertex, the algorithm computes the change in modularity that would result from moving it to the cluster of a neighbor and moves it to the best one. At the end of this stage, all clusters are contracted into super-vertices and the same process is repeated on the super-vertices. The process is repeated until no improvements to modularity can be achieved. In general, the Louvain algorithm tends to output communities with higher modularity than the Clauset Newman Moore algorithm.

Spectral methods provide an entirely different approach to community detection. When a graph is represented by an adjacency matrix, the eigenvectors of the adjacency matrix, or of other matrices derived from it, can be used to partition the graph into clusters [17, 18]. In spectral clustering, the eigenvectors of the Laplacian of the adjacency matrix are computed. Each vertex can then be represented by an r dimensional vector from r eigenvectors. Clustering these vectors with a technique such as *k*-means clustering assigns the vertices to communities [4]. Newman computes a modularity matrix and uses its leading eigenvector to repeatedly bisection the vertices [19]. At each stage vertices are divided based on their sign in the leading eigenvector, which increases the modularity of the partition.

Different community detection algorithms may produce differing clusters and it can be difficult to know which to use. Instead of relying on the results of a single algorithm, ensemble methods combine the output of multiple detection algorithms or of multiple runs of a stochastic algorithm [20, 21, 22, 23].

While most methods partition the graph into mutually disjoint groups, there is a growing body of work in detecting overlapping communities [24], where each vertex may belong to multiple groups. These methods are valuable because in many real datasets, communities of vertices do not form clean partitions. Rather, clusters may overlap each other or be hierarchically nested. For example, in a social network, a single person may belong to multiple clusters corresponding to work, family, and friends. Because they can account for the cluster overlap that occurs in many real datasets, these algorithms are likely to output better results. However, compared to non-overlapping algorithms, they are often more computationally intensive and may require more parameter setting.

A well known algorithm for overlapping cluster detection is clique percolation, which is based on the idea that communities are dense and likely to contain subgraphs in which each vertex has an edge to every other vertex, called cliques [25]. A community returned by the algorithm is a subgraph composed of overlapping cliques. Link partitioning takes an entirely different approach by applying agglomerative clustering to edges instead of vertices [26]. While a vertex may belong to multiple communities, each of its edges belongs to only one. This intuition makes sense in the case of a social network. A person may belong to multiple groups, but typically has one primary relationship type with any particular contact.

Label propagation is a technique that has been used to find both non-overlapping communities [27, 28] and overlapping ones [29]. In this approach, each vertex is initialized with its own unique community label. Next, a sweep of all vertices is performed in which each is assigned one or more labels that occur most frequently among its neighbors. Successive sweeps are repeated until some convergence criteria are met, after which each vertex belongs to one or more communities based on its stored labels. These techniques are near linear in time complexity and thus can be scaled to large graphs.

A fourth type of method to detect overlapping communities is through multiple local expansions. In this approach, a local community is grown around a seed vertex or set of seed vertices. By repeating such expansions multiple times, a graph is covered with local communities, some of which may overlap. Lanchichinetti *et al.* repeatedly use a random vertex as a seed and greedily expand to create a local community [30]. Overlapping communities are produced by sequentially running expansions from a node not yet in a community. A similar approach is used in [5], where the expansion technique differs, and in [31], which uses maximal cliques as starting seeds.

Local community detection is the task of finding a relevant cluster for a set of vertices of interest, called seed vertices. This problem is sometimes called seed set expansion and we will use those two terms interchangeably. Because many graphs may now have millions or billions of vertices, a full graph may be too large for certain tasks such as visualization, human inspection of anomalies, and running computationally intensive algorithms. Local community detection can be used in such cases to extract a smaller, but still relevant, subgraph. Additionally, as mentioned above, seed set expansion is used as a module in some global, overlapping community detection algorithms [5].

Clauset presents a greedy method for seed expansion that starts from a single vertex and then iteratively adds neighboring vertices to maximize the local modularity score [32]. The complexity of this approach for general graphs is $\mathcal{O}(c^2 d)$, where d is the average degree and c the final community size, although this will depend on the graph structure. Riedy *et al.* assume the the set of seed vertices given as input may belong to different communities. Each vertex starts out in its own cluster. Merges may then occur between a seed's community and either another seed's community or a singleton vertex in order to maximize global modularity [33]. Bagrow and Bollt use a different approach in the L-shell method [34], in which vertices are added from successive shells. A shell is a set of vertices at a fixed distance from the seed. Unlike in [32], multiple vertices are added to the cluster at once. This will likely improve running time, but may lower quality of resulting communities.

Local community detection has also been achieved through spectral methods. Andersen *et al.* [35] use the Spectral PageRank-Nibble method. Their final community is formed by adding vertices in order of decreasing PageRank values. In the random walk approach of Andersen and Lang [36], some vertices in the seed set may not be placed in the final community.

2.2.3 Detection Algorithms for Dynamic Graphs

While the community detection algorithms discussed thus far are applied to static graphs, different methods can be applied to dynamic graphs. A survey of work involving communities in dynamic graphs can be found in [37] and [38]. When the community structure must be computed for each snapshot G_t of a dynamic graph, any of the static algorithms discussed above could be simply applied to each snapshot. Dynamic algorithms are used instead for two reasons. First, a dynamic algorithm may seek to output a higher quality sequence of communities over time by using multiple snapshots of the graph. To do so, it may account for temporal relationships between graph snapshots and prevent formation of drastically differing communities in consecutive time instances. Second, the algorithm may reduce computation by reusing the previously found communities and incrementally updating.

In the former case, the goal is typically to return a sequence of communities that are both high in quality at each point in time and change smoothly over time. Multiple graph snapshots $\{G_1, \ldots, G_t\}$ may be used to compute the communities for a single snapshot. If a graph is no longer changing and the entire history of temporal graph data is available during computation, it may be used to obtain a sequence of communities [39, 40, 41].

Because all snapshots of the graph are used as input, the output communities are likely to both have good quality at each point in time and change smoothly over time. However, such an approach can be computationally expensive and requires knowledge of all data. In many real life applications, online results are needed at regular time intervals when the data changes. This type of approach would need to be re-run whenever new data appears and may produce different community histories for each run.

Instead, dynamic communities may be computed in an online fashion, with new clusters found for each new snapshot of the dynamic graph as it appears. The online approach can be achieved by both maximizing the quality of clusters in the new snapshot and minimizing the transition cost from the previous community decomposition. Examples include evolutionary clustering by Chakrabarti *et al.* [42] and FaceNet by Lin *et al.* [43].

The second body of work in dynamic community detection aims to reduce the computational cost of finding communities in quickly changing dynamic graphs. This is done by using the previously found result as a starting point and incrementally updating the communities to reflect changes in the graph. The workflow for many of these algorithms is as follows. A current graph exists with previously detected communities. When a set of updates occur, the previous community structure is modified based on the updates. In some cases, a static community detection algorithm is then applied to this modified community state. Some of these approaches are discussed below.

In [44], the authors present incremental spectral clustering by updating eigenvalues. In [45] the authors use an incremental version of the Louvain algorithm [16]. When the graph changes, instead of restarting from scratch, the previous community assignment is used as a starting point. Each node can then move to a different community to increase modularity. Shang *et al.* [46] also present an incremental algorithm to update the Louvain method. After each update, communities either remain the same, are merged due to inter-community edge addition, or new vertices are placed in an existing or new community. Static Louvain clustering is then restarted. The MIEN algorithm [47] is an incremental version of greedy agglomerative community detection, such as CNM. After edges and vertices are added and removed at a time step, all directly affected vertices, (endpoints of an inserted or removed edge or a vertex that was added or removed) are moved into their own singleton communities. Next, the chosen static community detection method is applied to current community structure to obtain any further merges. Aktunc *et al.* [48] present an incremental version of the SLM algorithm [49], which uses the clustering from the previous time step as a starting point, with new vertices in their own singleton communities. Riedy and Bader move vertices of inserted or deleted edges from their communities into singleton clusters before restarting their static, parallel, agglomerative algorithm [50].

Takaffoli *et al.* [51] present an incremental version of the local community detection algorithm from [52]. The static algorithm greedily adds the best neighboring vertex to the community, based on a fitness function, after which all vertices are checked for removal. The incremental version uses the connected components of communities found at the previous time step as starting points before continuing expansion with the static algorithm.

Many of these incremental approaches have a similar principle. Vertices directly affected by graph updates are removed from their previous community and either moved to a different one or placed as a singleton. A static algorithm then uses this modified community state as a starting point.

2.2.4 Analysis of Communities

On static graphs, communities reveal group structure. On dynamic graphs, changes to this structure can be discovered and unusual or important events flagged. Over time communities may grow, shrink, split apart, merge together, disappear, and reappear [53]. Detecting these events requires both finding correct communities for each graph snapshot and matching them across time. For example, communities C_{i,t_1}, C_{j,t_2}, C_{h,t_3} may be matched if they share enough vertices. In [54], such a set of matched communities is labeled as a single entity evolving over time. In addition to tracking community level operations, vertex level changes can be detected. Vertices can alter their behavior by moving from one community to another or by changing the strength of their membership. Asur *et al.* define and detect a variety of community and vertex level events using overlap of communities in consecutive graph snapshots [55]. One of the challenges in matching communities across graph snapshots in this manner is cluster instability. Because the output of many algorithms is sensitive to small variations in input, the communities assigned to consecutive snapshots may differ even when the underlying structure has not changed much. Hopcroft *et al.* [56] address this problem by using multiple runs to detect stable clusters, or natural communities, which are then tracked. Palla *et al.* use overlap to match communities over time in a research paper co-authorship graph and a phone call graph to track how long they persist [57]. Nevertheless, matching clusters from different graph snapshots is a challenge. The way in which new edges are added to and old edges removed from dynamic graphs will affect the stability of communities, and thus should be carefully chosen.

2.3 Graphs from Real Data

2.3.1 Edge Creation and Removal

Before computing any graph metrics and performing analysis, a graph must first be created from relational data. Researchers will often use datasets in which vertices and edges have already been defined, such as from the SNAP repository [58] or the Koblenz Network Collection [59]. In such cases, a graph can be easily formed by reading a file and including all edges listed in it. However, determining how to form edges is not always straightforward and will affect results of analysis.

Edges may be created by inferring relationships between entities using interaction between them. It is then necessary to determine how much and what type of interaction between two entities is sufficient to connect them with an edge. De Choudhury *et al.* [60] examine how to transform raw email communication data into a single, static relationship graph. Structurally different graphs will be formed based on what, if any, communication frequency or volume threshold is set for edge creation. For example, adding an edge between every pair of people who send an email to each other at least once a day will create a very different graph compared to adding an edge for those that exchange emails at least once a year. The authors set a threshold on the geometric mean of the annual rate of messages exchanged and study how the graph structure changes as the threshold varies. Social network ties can also be determined using multiple different forms of interaction or communication [61, 62, 63].

Physical proximity of people has also been used to infer relationships [64, 65, 66, 67, 68, 69, 70, 71]. Factors such as the duration of proximity, location, time of day, number of people nearby, and number of encounters can be used to determine whether a relationship between two people should be established. Such inferred relationships can then be represented by an edge in a graph.

While the issue of determining what type of relationship constitutes an edge is a challenge of creating a static graph, creating dynamic graphs poses additional difficulties. Given a graph snapshot G_t, we must decide how to create G_{t+1} based on the passage of time and new relational data. New edges that appear between times t and $t+1$ will typically be added to create snapshot G_{t+1}. For example, if interactions occur between times t and $t+1$ to create new edges, these edges will be added to create snapshot G_{t+1}. However, as time passes, the importance of old interactions may decrease and they may no longer indicate a relationship. Therefore, it is also necessary to determine whether edges representing older interactions will be removed or de-emphasized and if so, how.

Clauset and Eagle [72] create a dynamic graph of the Reality Mining dataset using the sliding window approach and examine how the size of the sliding window affects periodicity. They also investigate a “natural” window size based on the power spectra of metrics. The window size in the sliding window approach is also studied by Kossinets and Watts [73], who find that vertex-level properties are less stable than global graph ones.

Saganowski *et al.* [74] study the effect on community evolution of using a sliding time window to build a dynamic graph compared to aggregating all past data. They examine how the length and amount of overlap of the window size affects how many communities continue, split, merge, dissolve, and appear between consecutive snapshots. Oliveira *et al.* [75] also compare the sliding window approach to aggregating all data. Both works find that removing old data with a sliding window approach causes more change in community structure compared to aggregating all data.

2.3.2 Sampling

Another challenge in creating graphs is their collection and storage. Many graphs are too large to store or analyze in their entirety. Moreover, they are often constantly growing as new data is generated. Sampling techniques can be used to obtain a smaller, representative subgraph that can then be processed and analyzed. In order for a sample to be representative, it should be structurally similar to the full dataset. Therefore, the quality of sampling methods is typically evaluated by computing graph metrics, such as those discussed in Section 2.1.1, on both the subgraph and the full graph and comparing their values. A sampling method produces a subgraph $G_S = (V_S, E_S)$ where $V_S \subseteq V$ and $E_S \subseteq E$. Typically, $V_S \subset V$ and $E_S \subset E$. Sampling algorithms will either target a reduced number of vertices $\mid V_S \mid$ or limit the number of edges $\mid E_S \mid$.

Static Graph Sampling

Most previous work on graph sampling applies only to static graphs. These methods assume that the entire dataset is available beforehand and can be accessed as needed. Some approaches assume random access ability, which may be infeasible on very large datasets, while others are more suitable to reading large, disk bound graphs. However, they all assume both that the entire dataset already exists and is not growing and that the whole graph can be stored and accessed. Static graph sampling has been studied in [76, 77, 78].

A number of basic methods are described in detail in [76]. These include Random Edge sampling, which selects edges uniformly at random from G to form the sampled graph G_S. This strategy can produce sparsely connected graphs with a larger diameter, but can easily be extended to the fully streaming context, as discussed later. Another is Random Node sampling, in which nodes are chosen uniformly at random from G. The sampled graph G_S is then formed from all edges induced by the selected vertices (edges with both endpoint vertices selected). Unlike in Random Edge sampling, the method can only target a specific number of vertices in the sample, $\mid V_S \mid$, but the resulting number of edges $\mid E_S \mid$ is not fixed and will not be known ahead of time. Thus, the final size of memory needed to store the graph cannot be specified. Variations of Random Node choose the vertices not uniformly, but with some bias. These include Random Degree Node and Random PageRank Node, where the vertices are chosen with probability proportional to their PageRank score and degree, respectively. Because these methods require information about the structure of the graph prior to sampling, they may not be useful for scenarios in which the full dataset is too large to process.

Traversal based methods, which start with some initial vertices and expand by accessing neighbors, can also be used to obtain a subgraph [76]. In Random Walk sampling, random walks are performed from a starting node and G_S is then formed from vertices encountered and edges traversed. Snowball sampling performs a bread first search like traversal from an initial seed vertex, but only a fixed number of neighbors is added for each vertex. In Forest Fire sampling, the traversal is executed by starting with a seed vertex, visiting or "burning" a geometrically distributed random number of its neighbors, and repeating this recursively [76, 79]. Each vertex can only be "burned" or visited once.

Lindner *et al.* [80] present Local Degree Sparsification, in which for each node v, only a fraction of neighbors with the highest degree are kept. This approach aims to select edges that lead to high degree hubs in the graph.

Streaming Graph Sampling

Streaming graph sampling techniques create and maintain a subgraph G_S by processing a stream of edges S. Each element of the stream is an edge (u, v), possibly along with some attributes such as a weight or timestamp. While the entire stream of edges together forms the full graph G, the stream may be never ending as new edge activity is generated. Imagine, for example, tapping into a stream of Twitter activity such as re-tweets or users "following" each other. The full dataset is extremely large and new data is rapidly being generated. In such a scenario, static methods cannot be applied because new data is constantly produced, while a static approach assumes access to the entire dataset. To address this case, streaming graph sampling must be able to process a stream of edges in a single pass.

Note that some static sampling algorithms work by performing several passes over the edges of a graph. Although these methods are often called streaming, they are different from what is here referred to as streaming sampling. In this work, streaming sampling algorithms are those that can process a real time stream of continuously generated data. The full dataset may never be stored so sampling must be performed in a single pass over the edge stream.

While Reservoir sampling is not a graph-specific method, some streaming graph sampling algorithms in the literature use the concept of Reservoir sampling [81]. This method returns a fixed size sample from a stream so that each element has an equal chance of being returned in the sample, without needing to know the total number of elements in the stream. To obtain a sample of size k, the first k elements are added to the reservoir. Each subsequent element is added with decreasing probability. The i^{th} element is added with probability k/i for $i > k$ and if added, replaces a random element of the reservoir.

The simplest form of streaming graph sampling is Random Edge sampling (RE), which uses Reservoir sampling to select random edges from the stream. This approach is used for structural compression of a stream in [82]. Tabassum and Gama [83] apply both Random

Edge and a streaming edge sampling technique called Biased Random sampling on a phone call graph. The Biased Random sampling method inserts each new edge into the sampled graph, replacing an old edge if the sample has reached its size limit.

In addition to RE, which returns a subgraph with a predetermined number of edges, two streaming sampling methods that limit the number of vertices have been proposed in the literature. The first is Streaming Time Node sampling by Ahmed *et. al* [84], in which the stream of edges is divided into time intervals, such as a day or hour of activity. Each interval is selected with a certain probability and all vertices that appear within a chosen interval are added to the sample. An edge from the stream is then inserted if both its endpoint vertices are in the sampled subgraph. This is similar to the static Random Node sampling with induced edges, except that in the streaming case edges are only included if they appear after both vertices are selected. Edges occurring earlier in the stream are not included. By choosing vertices that appear together within a given time interval, the authors state that the vertices are more likely to be connected. The Streaming Time Node sampling method cannot be used to sample from a true, real-time stream of edges because it requires knowledge of the entire graph stream. In order to select a fraction ϕ of vertices in the sample, the vertices present in each unit of time are selected with probability m/T, where T is the total number of timestamps and m is the average number of timestamps needed to achieve the sampling fraction ϕ. This information is likely not present in a real streaming sampling scenario, but rather calculated once the data is collected.

The second streaming sampling method that limits the number of vertices chosen is Partially Induced Edge Sampling (PIES) [85, 86]. The PIES method is similar in principle to Streaming Time Node Sampling, but eliminates the need to know beforehand information about the entire graph stream, making it more suitable to real sampling use cases. This approach uses Reservoir sampling to select a fixed number of vertices from the stream and builds a subgraph from all edges whose endpoint vertices are in the reservoir. Since edges are only included after their endpoint vertices are selected, the edges are partially, not fully, induced. Although the authors also consider an alternative in which vertices are independently chosen, the PIES method samples on edges and either rejects the edge or adds both endpoint vertices of the edge to the sample. This approach biases the sample towards higher degree vertices because they will appear on more edges and thus have a higher probability of being selected.

In addition to methods that limit the number of edges or the number of vertices in the sample, some streaming graph sampling approaches do not limit the final size of the graph. One such approach is used for the DOULION algorithm for streaming triangle counting [87]. In order to reduce the amount of required computation, DOULION first samples a stream of edges before performing triangle counting. The sampling approach used selects each edge with a uniform probability. In a streaming environment in which new data is being generated, the total number of edges in the stream will not be known and this approach will therefore not restrict the final number of edges in the sample. Another example of a sampling method that does not limit the number of vertices or edges in the sample is the one used in the Graph Sample and Hold Framework [88]. There, each edge is selected with varying probability depending on how it connects to the current sample. Because edges are never removed, the size of the sample increases continually.

CHAPTER 3
STREAMING GRAPH SAMPLING

3.1 Introduction

Graphs are widely used to represent relational datasets from a variety of domains, such as online social networks, financial transactions, and biological data. In recent years, there has been increasing interest in the analysis of large, real-world networks, especially from online activity. However, many such online datasets are not only large, but constantly growing as new data are rapidly generated. Both the size and streaming nature of these graphs present challenges.

Datasets that are very large may not be able to be stored in their entirety. If they can be stored, computation time may suffer as algorithms take longer to run. The issue of size can be addressed through sampling. Several graph sampling algorithms have been presented in the literature, but most sample static graphs. Sampling streaming data poses an additional difficulty because the entire dataset cannot be accessed in any order desired. Many graph sampling techniques rely on full access to the graph. For example, given a vertex, they may require access to all of its edges and neighboring vertices. This type of access may not be feasible in a real-life application, either because new edges are continually being generated or because such random access is not available.

Therefore, this work focuses on streaming graph sampling: methods that can sample a subgraph with a single pass over a stream of edges. The single pass requirement is important in order to be able to tap into a stream of new data as it is produced and, in real time, make a decision about what to include and what to forget. The goal of sampling techniques presented in this chapter is to obtain a smaller subgraph because the full dataset is too large to store or too large to run computationally intensive algorithms. Therefore, the sampling algorithms must limit the size of the sampled subgraph. This means that sampling algorithms must limit not only the number of vertices, but also the number of edges in the subgraph. This chapter addresses this need by introducing two new sampling algorithms: Weighted Edge Sampling and Randomly Induced Edge Sampling. This work was published in [89].

In addition to obtaining a structurally similar subgraph, applications may have other requirements. In dynamic graph analysis and general stream processing, newer data may be more relevant and therefore preferred. Aggarwal presents a method for general, non-graph, stream sampling that returns a sample biased towards newer data [90]. Dynamic graphs may be updated to include newest data and often old data is removed when it loses relevance. This chapter addresses the need for temporally biased graph sampling with the new Temporal Weighted Edge Sampling algorithm.

3.1.1 Contributions

This chapter introduces new algorithms for streaming graph sampling. Section 3.4 presents two new algorithms that sample graph streams without temporally biasing: Weighted Edge Sampling (WES) and Randomly Induced Edge Sampling (RIES). Each of these methods can (1) sample a graph stream in a single pass, thus allowing for true sampling of real-time data streams, and (2) limit the size of subgraph by restricting the number of edges. To the best of our knowledge, the only previously proposed sampling algorithm that satisfies both of these requirements is streaming Random Edge (RE). Other approaches discussed in the literature either require more than a single pass over the data (or require random access to the graph) or they only restrict the number of vertices in the sampled subgraph and not the number of edges, which means that the final size of the subgraph is not known. Experiments on several relational social network datasets compare the performance of RIES and WES against the previously known RE. They show that the new algorithms produce subgraphs that are more structurally similar to the original graph compared to RE. The main advantage of WES over RIES is that its parameters affect the sample in a more predictable manner and are thus easier to set.

Section 3.5 presents a new algorithm for temporally biased graph sampling: Temporal Weighted Edge Sampling (TWES). Like WES, RIES, and RE, TWES samples a subgraph from a stream in a single pass while restricting the number of edges sampled. In addition, it can return a sample that is temporally biased towards newer edges. Experiments show that TWES samples subgraphs that are temporally biased and display similar quality to non-temporally biased subgraphs produced by WES.

3.2 Background

3.2.1 Sampling Goals

Graph sampling may be used to achieve four categories of goals [86]. Firstly, it may be performed to estimate specific graph parameters, such as the average degree or clustering coefficient. For example, triangle counting has received much attention in the literature [91, 92]. Another goal may be to obtain representative vertices in order to measure their attributes. This may be useful, for example, in the social science setting where vertices represent people in a network with features that need to be estimated. In this case, it is most important to obtain an unbiased sample of vertices. Third, it may be performed to obtain a representative set of edges in order to analyze their attributes. For example, in a network where vertices have attributes, sampling edges can be used to measure the homophily of such attributes: do vertices tend to connect to similar vertices or not? Finally, sampling a graph can be used to obtain a smaller, but structurally similar subgraph. In this case, simply obtaining representative vertices or representative edges is not sufficient. The goal is to obtain a representative subgraph. This is useful when the entire dataset in question is too large to either store or to analyze. Analysis may then be performed on the smaller subgraph instead. Many graphs available and used in the literature for testing are in fact samples themselves. This chapter addresses the fourth aforementioned goal of sampling:

to obtain a smaller, but structurally similar subgraph that lends itself to further processing.

Streaming graph sampling creates and maintains a subgraph by processing a stream of edges S, which may be never ending as new edge activity is generated. Imagine, for example, tapping into a stream of Twitter activity such as re-tweets or users following others. The full dataset is both extremely large and rapidly growing, with new data being continually added to it. In such a scenario, static methods cannot be applied because new data is constantly produced, while a static approach assumes we already have the entire dataset. To address this case, streaming graph sampling must be able to process a stream of edges in a single pass.

Note that some static sampling algorithms work by performing several passes over the edges of a graph. Although these methods are often called streaming, they are different from what we refer to as streaming sampling. In this work, streaming sampling algorithms are those that can process a real time stream of continuously generated data without going back in time to re-process previously stored data. The full dataset may never be stored so sampling must be performed in a single pass over the edge stream.

3.2.2 Notation and Problem Statement

Given a full graph $G = (V, E)$, a sampling algorithm can be used to obtain a sampled subgraph $G_S = (V_S, E_S)$ where $V_S \subseteq V$ and $E_S \subseteq E$. Since a sample should be smaller than the original graph, typically both $V_S \subset V$ and $E_S \subset E$. In static sampling, G already exists and can be accessed directly. Often, random access is assumed so that all edges of any vertex can be examined. In contrast, in the streaming sampling problem, the full dataset can be accessed only one element at a time as a stream. Each element of the stream is a tuple that represents an edge event. It must contain two vertices and may include other features such as a timestamp and weight. In this work, the elements of a stream S are tuples (u, v, t), where u and v are endpoint vertices and t is a timestamp. We assume there is no weight parameter in the source data stream S, i.e. all input edges have the same initial importance. The timestamps represent discrete time in whatever units are used in the application. In Section 3.5, the timestamps are used by the algorithms to introduce temporal biasing, while algorithms in Section 3.4 do not take timestamps into consideration. The full graph $G = (V, E)$ is created by including all edges from the stream in E and all endpoint vertices in V. If the stream is never-ending, then the full graph G may constantly grow. Note that the word "edge" is used in this work to refer both to the edges of a graph G and to the elements of a stream S. In other words, the graph G is constructed from edges of the stream S as they arrive. However, as explained later, edges of G may include additional weight parameters, even if edges in S are weightless.

The sampling algorithms presented in this chapter are flexible and can be applied to multiple types of edge streams. The algorithms return a sample of the edges of S, denoted S_S, and the elements of S_S can then be turned into a subgraph G_S. We now describe how this is done.

There are two main cases to consider. In the first case, the elements of S may correspond one-to-one to edges of G. Since only one edge exists between any given pair of vertices u and v in G, only one edge in S can then contain u and v. That is, the first two components of the tuple, corresponding to the vertices, uniquely identify the tuple in S. In this case, each edge in S_S is simply added to E_S and the endpoint vertices are added to V_S. If G_S needs to have timestamps, then the timestamp of the tuple may be used to set a time $(u, v).time$ for each $(u, v) \in E_S$. This simpler case is of less practical importance, as it requires that only one edge event may ever occur between any two vertices.

In the second, more general, case, multiple elements of S may contain the same two vertices. For example, each element may correspond to a relational event such as an email sent between two people. Since any two people may communicate repeatedly over time, multiple edges in S may contain the same two vertices. They may even contain the same timestamp if the events took place within the same interval of time. In this case, the subgraph G_S will be created from S_S as follows. For each element $(u, v, t) \in S_S$, if the undirected edge (u, v) does not already exist in E_S, it is added. Otherwise, the weight of $(u, v) \in E_S$ can be incremented and if $t > (u, v).time$, its timestamp updated to reflect the most recent time.

The algorithms presented in this chapter will take as input a stream S and return a sample of the observed elements S_S. The subgraph $G_S = (V_S, E_S)$ can then be created as discussed above. In the experiments presented later in this chapter, edges in S_S may repeat. However, the sampling algorithms are flexible and could be adapted to applications in which graphs must contain additional information. For example, an edge $(u, v) \in G_S$ may need to have as a feature a list of all timestamps at which a relational event between u and v occurred.

3.2.3 Related Work

Section 2.3.2 provides a broader overview of the existing literature on graph sampling techniques. This section discusses work in streaming sampling that is directly relevant to this chapter.

Reservoir Sampling

Reservoir sampling is a general stream sampling technique that has been utilized in graph sampling [81]. This method processes a stream one element at a time to return a random sample of predetermined size. At any point in time, each element that has been processed has an equal chance of being in the sample. A uniformly random sample is simple to achieve when the total number of elements to be processed is known. The techniques's value lies in the fact that a uniformly random sample is achieved without needing to know the total number of elements in the stream. To obtain a sample of size k, the first k elements are first added to the reservoir. Then, each subsequent element is added with decreasing probability. For $i > k$, the i^{th} element is added with probability k/i and if added, replaces a random element of the reservoir. This simple technique assures that at any given time there are always k elements in the sample, and all i elements processed so far each may be found in the sample with the same probability.

Efraimidis and Spirakis presented Weighted Reservoir sampling, in which each element of the stream is included in the returned sample with probability proportional to its weight [93]. Each element with weight w is assigned a random value $r \in Uniform(0, 1)$ in order to generate a key $r^{\frac{1}{w}}$. At the end of the stream, or whenever a sample should be returned, the k elements with largest keys are returned. The Weighted Reservoir sampling technique is utilized in the WES and TWES algorithms presented in this chapter. In this work, edge weights do not come from the input stream edges but are appended to the data based on edge timestamps or on how much they connect to the current sample.

Graph Sampling

Perhaps the simplest form of graph sampling involves creating a subgraph from a uniformly random sample of edges. This approach is called Random Edge sampling (RE). RE can be performed in a single pass over a stream of edges using reservoir sampling, as shown in Algorithm 1. This is the only graph sampling technique from the literature that directly limits the number of edges in the sampled subgraph and can be performed in a single pass over a stream of edges. Therefore, it is used as a baseline for the new algorithms presented in this chapter.

In addition to RE, other streaming algorithms have been discussed in the literature that either limit only the number of vertices in the sample or impose no limit on the size of the sample at all. The Streaming Time Node Sampling [84] and Partially Induced Edge Sampling (PIES) [85, 86] algorithms sample a subgraph with a specified number number of vertices. Since PIES is an improvement on Streaming Time Node Sampling, we use it as our main reference when discussing our proposed algorithms. The PIES algorithm uses Reservoir sampling to produce a subgraph formed from a predetermined number of vertices from the stream. The Reservoir sampling is performed on edges so that if selected, both

```
Data: stream $S$ of edges, sample size $k$
Result: $G_S = (V_S, E_S)$
initialize empty sample $S_S$;
$i = 0$;
while $S$ has edge $e_i$ do
    $(u_i, v_i) = e_i$;
    if $i < k$ then
        add $(u_i, v_i)$ to $S_S$;
    else
        $p = \frac{k}{i}$;
        draw $r$ from $Uniform(0, 1)$;
        if $r < p$ then
            remove random element from $S_S$;
            add $(u_i, v_i)$ to $S_S$;
        end
    end
    $i = i + 1$;
end
for edge $(u, v) \in S_S$ do
    $V_S = V_S \cup \{u, v\}$;
end
create $E_S$ from edges in $S_S$;
return $G_S = (V_S, E_S)$;
```

Algorithm 1: `Streaming Random Edge (RE) Sampling`

endpoint vertices are added to the reservoir, replacing other vertices if full. A subgraph is built by including all edges whose endpoint vertices are in the reservoir when the edge is processed. If a vertex is removed from the reservoir, its edges are dropped as well. Consequently, the number of edges in PIES cannot be predetermined even if the number of vertices remains constant. Section 3.3 discusses why it is important to keep the number of edges, instead of vertices, constant.

While RE keeps the number of edges in the sample at a predefined level and PIES does the same with vertices, some streaming graph sampling approaches do not limit the final size of the graph. In order to reduce required computation, the DOULION algorithm samples a stream of edges before performing triangle counting [87]. Each edge is selected with a predetermined uniform probability. In a streaming environment in which new data is being generated, the total number of elements in the stream cannot be known. Therefore, with any predetermined probability chosen, the final number of edges selected cannot be known. Another such example is the method used in [88], in which each edge is selected with varying probability depending on how it connects to the current sample. Edges are never removed and thus the size of the sample increases as more stream elements are processed.

3.3 Targeting a Vertex Versus Edge Size

Sampling methods typically create a subgraph G_S with either a specific number of vertices $\mid V_S \mid$ or a specific number of edges $\mid E_S \mid$. For example, in static node based sampling, described in Section 2.3.2, k vertices will be randomly chosen and all edges induced by these k vertices are included in E_S. The resulting number of edges $\mid E_S \mid$ in the sample cannot be predicted and can be as low as $\frac{k}{2}$ or as high as $\frac{k*(k-1)}{2}$. The same phenomenon applies to the PIES method, where k vertices are sampled from a stream and edges are included if both endpoint vertices belong to the stream. Given the fact that most social network type graphs are sparse, a bound of $\mathcal{O}(k^2)$ edges for k vertices is not relevant

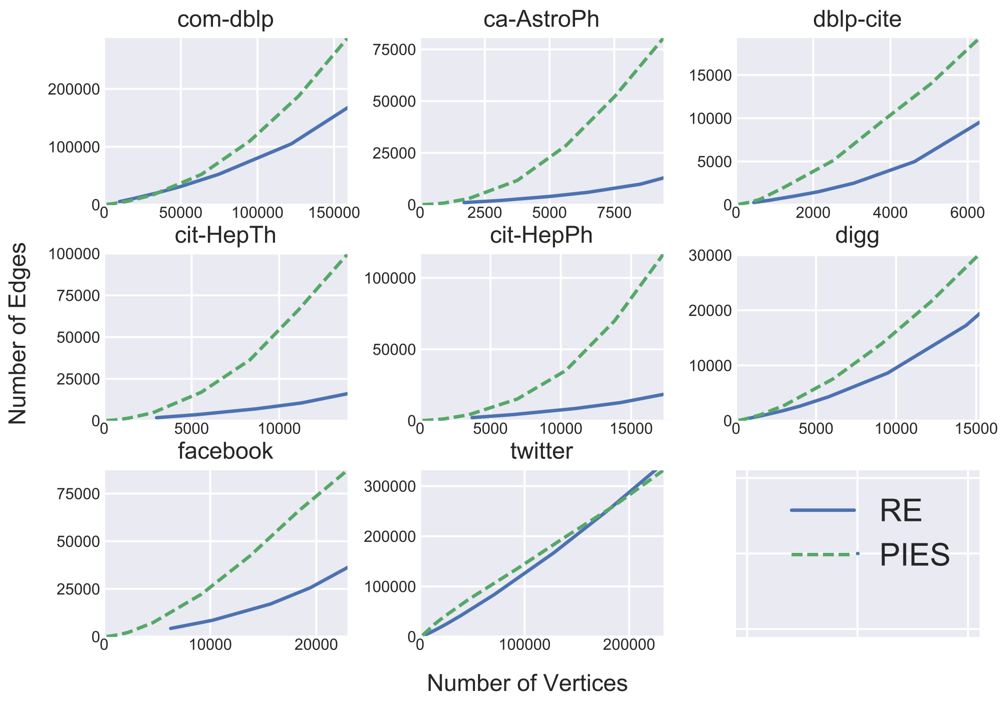


Figure 3.1: The number of vertices and edges for subgraphs sampled with RE and PIES is shown. For samples with the same number of vertices, the PIES method results in many more edges compared to the RE method.

in practice. The resulting number of edges is especially hard to determine in streaming sampling because the dataset is not available ahead of time and average degree estimation cannot be performed.

This is a problem when the purpose of graph sampling is to obtain a smaller subgraph because the full dataset is too large to be stored. The size of $G_S = (V_S, E_S)$ depends on the number of edges $\mid E_S \mid$. Note that $\mid V_S \mid \leq 2 \mid E_S \mid$ so that bounding the number of edges does bound the entire size of the graph. In cases when sampling is performed because computationally expensive analytics cannot be run on the full dataset, it is also important to limit $\mid E_S \mid$ because the running time of most algorithms will depend in part on the number of edges.

To further motivate the need to consider the full size of the sampled graph, Figure 3.1 compares the number of vertices and edges in subgraphs created with PIES and RE on test datasets described in Section 3.4.3. Because in PIES it is not possible to target a specific number of edges, we repeatedly sample subgraphs with increasing numbers of vertices and plot the resulting number of edges against the number of vertices. Similarly, in streaming RE, it is not possible to target a specific number of vertices, so we repeatedly sample subgraphs with increasing numbers of edges and plot the number of edges against the resulting number of vertices. Figure 3.1 shows two important points. First, for subgraphs with a given number of vertices, those subgraphs created by PIES tend to have many more edges compared to subgraphs created by RE. This is as expected because PIES includes all induced edges going forward in time. Second, for any target number of vertices, the number of resulting edges will vary depending on the graph. Thus, it is impossible to predict $\mid E_S \mid$ based on a selected $\mid V_S \mid$ without additional knowledge about the dataset.

3.4 Non-Temporal Graph Sampling

This section presents two new algorithms for streaming graph sampling, WES and RIES. Each processes an edge stream in a single pass and outputs a sample of a specified number of edges. In this section, the timestamps of edges are not considered. The goal of the sampling algorithms is to create a subgraph that is structurally similar to the full graph. With this goal in mind, the quality of samples produced by WES and RIES is compared to the baseline RE algorithm in Section 3.4.3.

3.4.1 Weighted Edge Sampling (WES) Algorithm

The first new streaming graph algorithm presented is Weighted Edge Sampling (WES). The method samples k_e edges from a graph stream S. The source stream S may have any number of elements or be never ending if edges are constantly generated. WES uses the concept of Weighted Reservoir sampling to give more bias to edges that connect to the current sample S_S at the the time when the edge is processed. By biasing towards such edges, WES creates a subgraph that is more connected compared to RE. Specifically, WES has three parameters, w_0, w_1 and w_2. When a new edge is processed in the stream, it is given a weight that indicates how likely it is to be included in the final sample. Edges that have no endpoint vertices in the sample when they are processed are assigned $weight = w_0$, those that have one endpoint vertex in the sample are assigned $weight = w_1$, and those for which both vertices are already included in the sample are assigned $weight = w_2$. Since only the relative weights are important, w_0 is always set to 1 in experiments shown in this section.

For each edge, after its weight is set, a random number r is drawn from $Uniform(0, 1)$. The key for the edge is then set to be $r^{1/weight}$. At any point in the stream, the k_e edges with largest keys seen so far form the sample. The probability that an edge is in the sample is proportional to its weight [93]. By setting $w_0 \leq w_1 \leq w_2$, WES biases towards edges that will connect to the current sampled subgraph. Full details of WES are given in Algorithm 2. In streams with repeated edges, each instance is sampled and stored separately. Top keys can be stored and retrieved using a min-heap, which allows the element with lowest value to be accessed in $\mathcal{O}(1)$ time.

```
Data: stream $S$ of edges, edge limit $k_e$, parameters $w_0, w_1, w_2$
Result: $G_S = (V_S, E_S)$
initialize empty min-heap $M$;
$V_S = \emptyset, i = 0$;
while $S$ has edge $e_i$ do
    $(u_i, v_i) = e_i$;
    if $u_i \in V_S$ and $v_i \in V_S$ then
        $weight = w_2$;
    else if $u_i \in V_S$ or $v_i \in V_S$ then
        $weight = w_1$;
    else
        $weight = w_0$;
    end
    draw $r$ from $Uniform(0, 1)$;
    $key_i = r^{1/weight}$;
    if $\mid M \mid < k_e$ then
        add $(u_i, v_i)$ to $M$ with key $k_i$;
        $V_S = V_S \cup \{u_i, v_i\}$;
    else if $key_i >$ smallest key in $M$ then
        Remove element $(u_j, v_j)$ with smallest key from $M$;
        if $u_j$ has no more edges in $M$ then
            $V_S = V_S \setminus u_j$;
        end
        if $v_j$ has no more edges in $M$ then
            $V_S = V_S \setminus v_j$;
        end
        add $(u_i, v_i)$ to $M$ with key $key_i$;
        $V_S = V_S \cup \{u_i, v_i\}$;
    end
    $i = i + 1$;
end
create $E_S$ from edges in $M$;
return $G_S = (V_S, E_S)$;
```

Algorithm 2: `Weighted Edge Sampling (WES)`

3.4.2 Randomly Induced Edge Sampling (RIES) Algorithm

Randomly Induced Edge Sampling (RIES) is a streaming graph sampling algorithm which creates a subgraph with no more than k_v vertices and no more than k_e edges. RIES is a modification of the streaming algorithm PIES described in Section 3.2.3. Recall that PIES randomly selects k vertices in pairs by sampling on edges and either accepting an edge and adding both its endpoint vertices to the sample or rejecting the edge. In addition, PIES adds all edges in the stream whose two endpoint vertices are in the sample V_S at the time the edge arrives. RIES, on the other hand, includes two levels of stream sampling. On the first level, k_v vertices are sampled from the stream in pairs as in PIES. On the second level, Reservoir sampling is performed on all edges in the stream whose two endpoint vertices are in the sample at the time that the edge arrives. Thus, instead of including all future induced edges, only k_e such induced edges are sampled among all those encountered in the stream. This process is shown in Algorithm 3.

Using RIES requires choosing both a maximum number of vertices k_v and maximum number of edges k_e. The number of vertices k_v can be set by selecting an assumed average degree $d = \frac{k_e}{k_v}$ of the subgraph. This choice of d can be difficult to make because in the streaming context, the full dataset is never available and information about the degree distribution cannot be obtained. Choosing a low value of d will cause the exclusion of many of the induced edges that would have been included with PIES. On the other hand, choosing too high a value of d will mean that the number of vertices k_v is too small to collect k_e edges and therefore the sampled graph will be smaller than it could have been. For experiments presented in Section 3.4.3, d is chosen based on values of average degree typically seen when running PIES.

```
Data: stream $S$ of edges, vertex limit $k_v$, edge limit $k_e$
Result: $G_S = (V_S, E_S)$
initialize empty sample $S_S$, $V_S = \emptyset, i = 0, m = 0, j = 0$;
while $S$ has edge $e_i$ do
    $(u_i, v_i) = e_i$;
    if $| V_S |< k_v$ then
        $V_S = V_S \cup \{u_i, v_i\}$;
        $m = m + 1$;
    else
        $p = \frac{m}{i}$;
        draw $r$ from $Uniform(0, 1)$;
        if $r < p$ then
            if $u_i \notin V_S$ then
                choose random vertex $q \in V_S$;
                $V_S = V_S \setminus q$;
                remove all edges from $S_S$ with endpoint vertex $q$;
                $V_S = V_S \cup u_i$;
            end
            if $v_i \notin V_S$ then
                choose random vertex $q \in V_S$;
                $V_S = V_S \setminus q$;
                remove all edges from $S_S$ with endpoint vertex $q$;
                $V_S = V_S \cup v_i$;
            end
        end
    end
    if $u_i \in V_S$ and $v_i \in V_S$ then
        $j = j + 1$;
        if $| S_S |< k_e$ then
            add $(u_i, v_i)$ to $S_S$;
        else
            $p = \frac{k_e}{j}$;
            draw $r$ from $Uniform(0, 1)$;
            if $r < p$ then
                remove random element from $S_S$;
                add $(u_i, v_i)$ to $S_S$;
            end
        end
    end
    $i = i + 1$;
end
$V_S = \emptyset$ for edge $(u, v) \in S_S$ do
    $V_S = V_S \cup \{u, v\}$;
end
create $E_S$ from edges in $S_S$;
return $G_S = (V_S, E_S)$;
```

Algorithm 3: `Randomly Induced Edge Sampling (RIES)`

Table 3.1: Datasets used

Dataset	$\mid V \mid$	$\mid E \mid$
com-dblp	317,080	1,049,866
ca-AstroPh	18,771	198,050
dblp-cite	12,591	49,743
cit-HepTh	27,770	352,807
cit-HepPh	34,546	421,578
digg-reply	30,398	87,627
facebook	46,952	876,993
twitter	465,017	834,797

3.4.3 Quality Results

Experimental Setup

As discussed earlier, the goal of streaming sampling is to obtain a subgraph $G_S = (V_S, E_S)$ that is structurally similar to the full graph $G = (V, E)$. The sampling algorithm returns a sample of the stream S_S and G_S is created by including all edges and vertices from S_S as described in Section 3.2.2. The full, baseline graph G is similarly constructed by adding all edges from the entire stream S. Of course, in a real-life streaming applications new data may be continually generated so that S has no end and the full graph G grows. In the experiments presented here, however, we use existing test datasets and simulate streaming. Because the streams have a definite beginning and end, the full graph G can be built. The datasets used are publicly available social networks from the Koblenz Network Collection [59] and are listed in Table 3.1. Each dataset contains a list of edges, which are here randomly permuted to form S.

The quality of subgraphs created by RIES and WES is compared to those created by the baseline RE from the literature. We evaluate to what extent the subgraphs created by each method are structurally similar to the full graph. This is done by comparing the following structural properties: vertex degrees, lengths of shortest paths, clustering coefficients, and weakly connected components. In order to compare vertex degrees, the degree distribution found in a sampled subgraph is compared to the distribution from the full graph using the two sample Kolmogorov-Smirnov (K-S) statistic. The statistic is defined as

$$D = max_x\{\mid F_1(x) - F_2(x) \mid\} \tag{3.1}$$

where F_1 and F_2 are two empirical cumulative distribution functions (CDFs). In this case, F_1 would be the degree distribution in G_S and F_2 the degree distribution in G. The lower the value, the more similar the distributions. The K-S statistic is a standard way of comparing how structurally similar a sampled graph is to the full graph and has been used in previous work on static sampling [76] and streaming sampling [85] [86]. Path length similarity is also evaluated in this manner. The shortest paths between random pairs of vertices are computed for G_S and G and the two resulting distributions are compared using the K-S statistic. Because the total number of pairs is very large, the distribution is estimated by selecting random pairs. Clustering similarity in G_S and G is evaluated by comparing the two distributions of local clustering coefficients. Local clustering coefficients are the ratios of the number of triangles a vertex participates in over the number of triples it participates in.

In contrast to other metrics, the connected components of G_S and G are not compared using distributions. The reason for this is that many of the test graphs are close to fully connected and have few components. If only a single component exists, a proper non-trivial distribution of sizes cannot be formed. Distributions cannot not be compared meaningfully unless the number of connected components is large. Therefore, the similarity of connected components is evaluated by computing, for both G_S and G, the percentage of vertices in the largest connected component. In the results, this measure is referred to as global connectivity. A sampling algorithm has good quality for this graph feature if G_S and G have a similar global connectivity value.

For results shown in Figures 3.2-3.6, the sampling size k_e is varied between 0.5% and 20% of the original graph size. While results are shown in terms of the percent of edges sampled, it is important to note that WES and RIES take as input not a percentage, but a fixed number of edges k_e to include in the sample. It is not necessary to know ahead of time how many edges will be processed in the stream. The sampling size k_e is shown as a percentage of total edges so that results on differently sized test datasets can be easily shown together and compared.

For RIES, an average degree $d = 4$ is used and k_v is then set to $k_v = \frac{k_e}{d}$ accordingly. For WES, we use a $w_0 = 1$, $w_1 = 1$, and $w_2 = 100$. We found that biasing heavily towards edges with two endpoint vertices already in the sample created the best results in terms of similarity to the full graph. The particular chosen value of w_2 yielded good results across datasets and sampling percentages. In Section 3.4.3 we discuss the effect of different parameter settings of both WES and RIES. For each value of k_e and each sample size, 10 runs are performed and mean results plotted. For each run, the edges from the dataset are randomly permuted to vary the order of the edge stream.

Results

Figures 3.2, 3.3, and 3.4 show the mean K-S statistic for shortest path length distributions, degree distributions, and clustering coefficient distributions plotted against sample size. The mean is taken over 10 runs, each on a randomly permuted edge stream. Values near zero indicate that the sampling method produces distributions that are very similar to that of the full graph, while values near one indicate dissimilar structure. For each sampling percentage shown on the y-axis, k_e is set to that percent of the total number of edges in the full graph. K-S values are plotted against edge percentages instead of the actual number of edges k_e in order to make comparisons between datasets easier and to show what proportion of edges is needed for good results. However, WES, RIES, and RE all take as input a fixed number of edges to include in the sample, not a percentage.

Figure 3.2 shows that both WES and RIES produce much better results in terms of

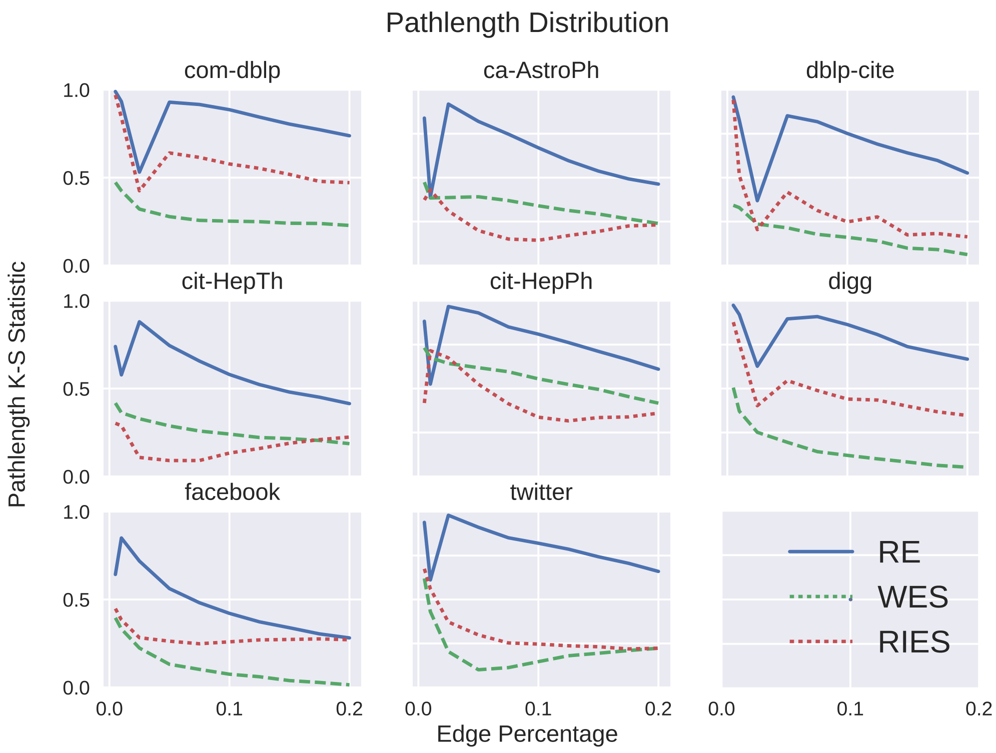


Figure 3.2: The similarity of shortest path lengths between the sampled subgraph G_S and the full graph G is shown for WES, RIES, and RE. Lower K-S values indicate better results.

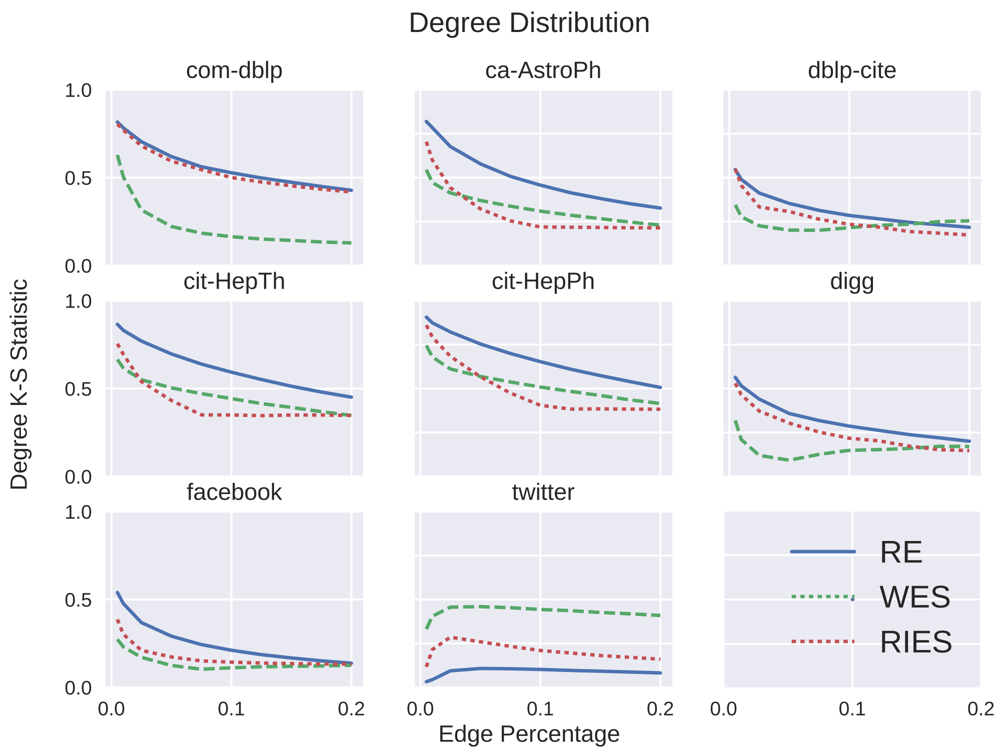


Figure 3.3: The similarity of vertex degrees between the sampled subgraph G_S and the full graph G is shown for WES, RIES, and RE. Lower K-S values indicate better results.

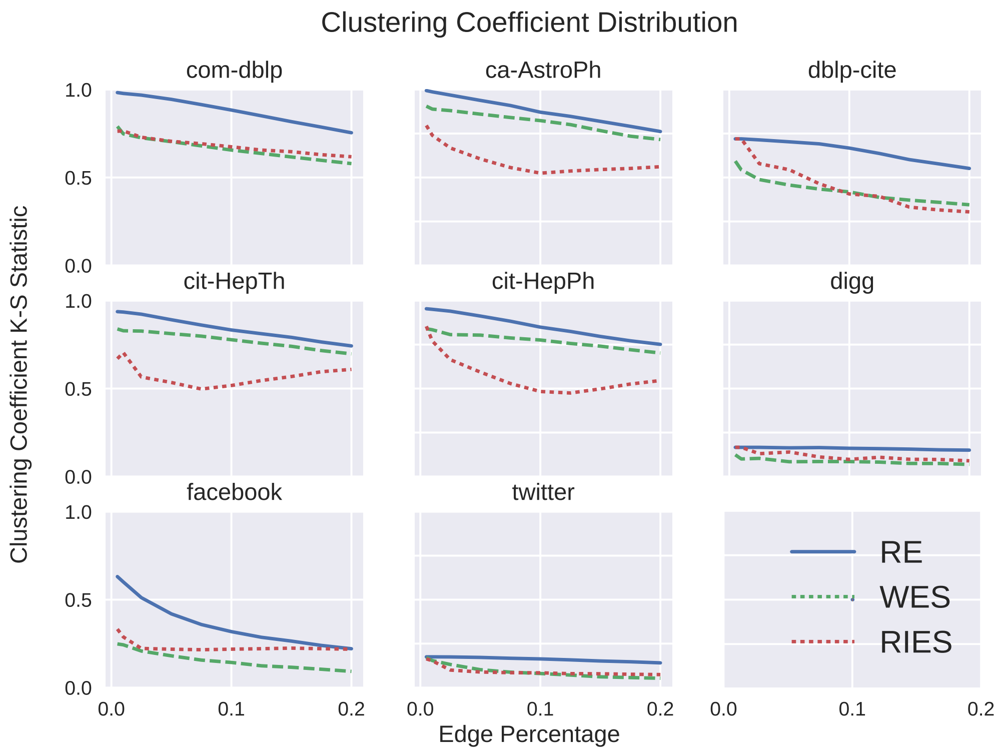


Figure 3.4: The similarity of local clustering coefficients between the sampled subgraph G_S and the full graph G is shown for WES, RIES, and RE. Lower K-S values indicate better results.

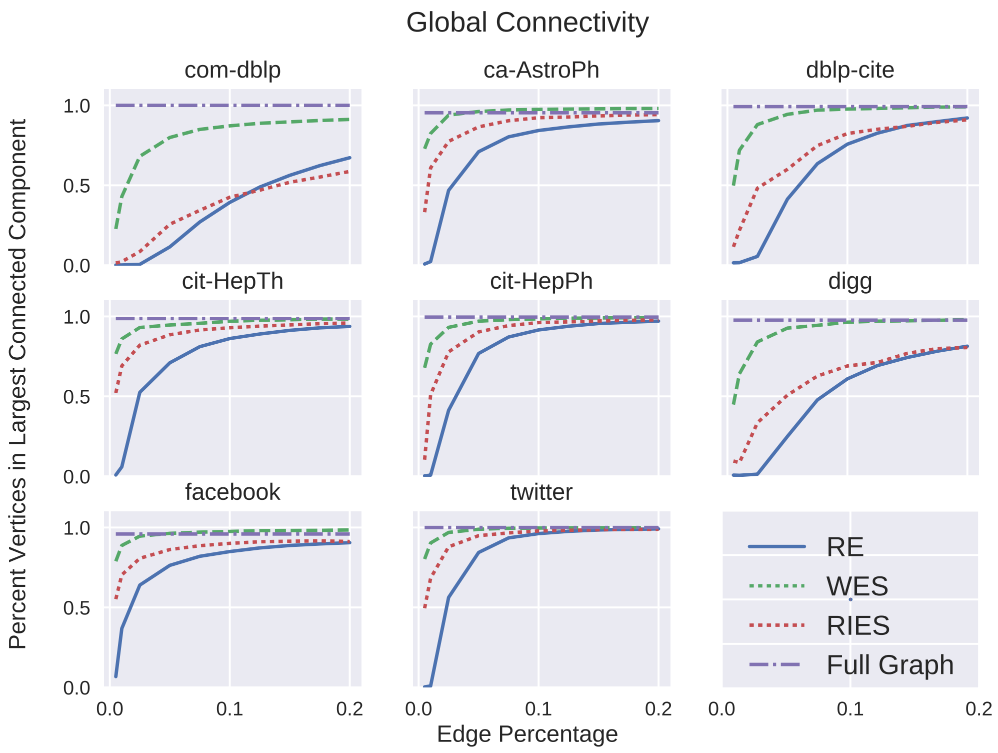


Figure 3.5: The percentage of vertices in the largest connected component is plotted for both the full graph and subgraphs sampled by WES, RIES, and RE. Values similar to the full graph (higher) are better.

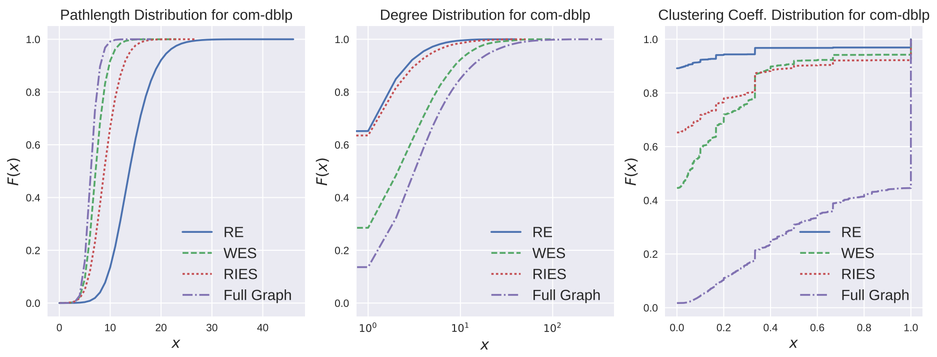


Figure 3.6: This plot shows the CDFs of shortest path lengths between pairs of vertices, vertex degrees, and local clustering coefficients for the *com-dblp* graph. CDFs are shown for the full graph, and samples of 10% of edges using the RE, WES, and RIES methods.

shortest path lengths compared to RE. Overall WES and RIES perform similarly well, with WES performing better on some datasets and RIES on others. To understand the cause of these K-S values, we can examine the left panel of Figure 3.6, which shows the distribution of shortest path lengths for the full graph and for subgraphs created by RE, WES, and RIES. This example is for the *com-dblp* graph and sampling percentage of 10%. In this case, RE performs poorly because it creates a subgraph with shortest paths that are longer than those found in G. That is, pairs of vertices tend to be at a greater distance compared to the full graph. WES and RIES, on the other hand, are able to create a subgraph with path lengths closer to those of the original graph. This is because RE samples edges from the input stream without considering the previously sampled edges, while both WES and RIES take into account how stream elements connect to the already existing sample.

For most datasets, there is a dip in the path length K-S values for RE (and sometimes RIES) at a sampling percentage of 1% or 2.5%. The *com-dblp* dataset can be used as an example to explain this behavior. At sampling percentages of 0.5% and 1%, the subgraphs created by RE are very disconnected. There may be many small connected components containing only single edges or a small number of edges. If a connected component has only n edges, the maximum path length between any of its vertices is n. Therefore, for these low sampling sizes of RE, path lengths are much *lower* than in the full graph, resulting in high K-S values. As the sampling size increases, larger connected components form, allowing for longer path path lengths. At 5%, however, the path lengths are typically *higher* in the subgraphs created by RE than in the full graph. This also results in high K-S values. As the sampling rate increases and the path lengths change from being too short to being too long, the crossover point occurs around 2.5%. The dip in K-S values corresponds to this crossover point. Then, for all sampling rates between 5% and 20%, the subgraphs of *com-dblp* created by RE have shortest path lengths that are longer than those in the full graph.

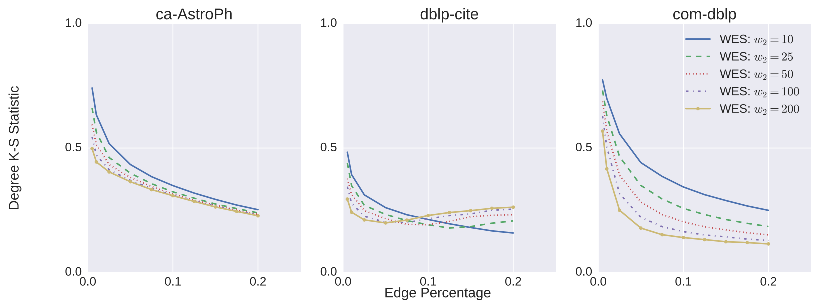
Figure 3.3 and the middle panel of Figure 3.6 show results for vertex degree distributions. Again, WES and RIES tend to sample subgraphs with more similar structure to that of the full graph. This is due to the fact that both methods obtain higher vertex degrees compared to RE. WES achieves this by biasing to edges that already have endpoint vertices in the sample (thus targeting vertices with multiple edges), while RIES does so by including only edges that connect the relatively small set of sampled vertices.

Clustering coefficient similarity is shown in Figure 3.4 and the right panel of Figure 3.6. For most graphs, especially those with many triangles, all methods underestimate the number of triangles and therefore the local clustering coefficients. However, both RIES and WES perform better than RE, with RIES producing the best clustering results.

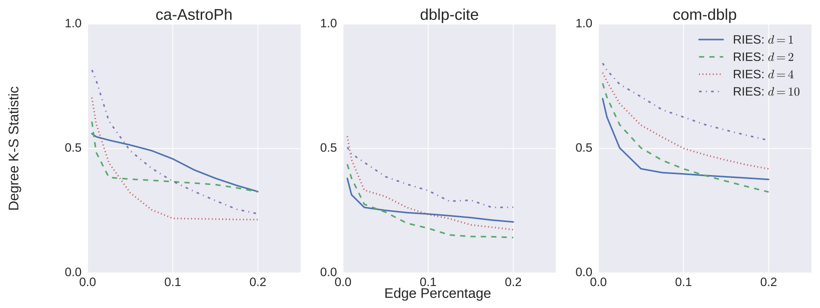
Finally, connected component results are plotted in Figure 3.5, which shows the percentage of vertices that are in the largest connected component both for the full graph and for each sampling method. Note that unlike in Figures 3.2, 3.3, and 3.4, the y-axis in Figure 3.5 does not plot either a distance or similarity metric. Good results are instead indicated by values that are close to those of the full graph. Because the test graphs we used are highly connected, the majority of vertices in the full graph are in the largest connected component. Therefore, for these particular experiments, higher values indicate better results. WES creates the most connected samples and thus performs best, while RE creates the most disconnected graphs. The reason why WES creates more connected subgraphs than RIES is because the method automatically biases towards including edges that connect the rest of the sample. In RIES, the connectivity will depend on whether the randomly chosen vertices will tend to connect or not, so this may vary quite a bit.

Effect of Method Parameters

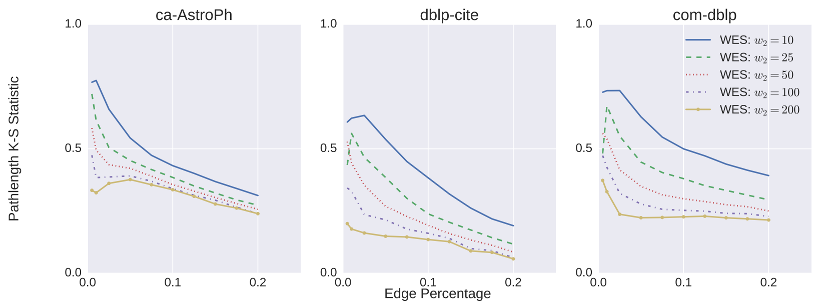
The previously discussed results for WES and RIES were obtained using a single parameter setting for each. In Figure 3.7, the effect of varying method parameters is shown. Degree and shortest path length results are plotted for different parameter settings for both WES and RIES. Figures 3.7a and 3.7c show K-S statistic values on three graphs sampled with



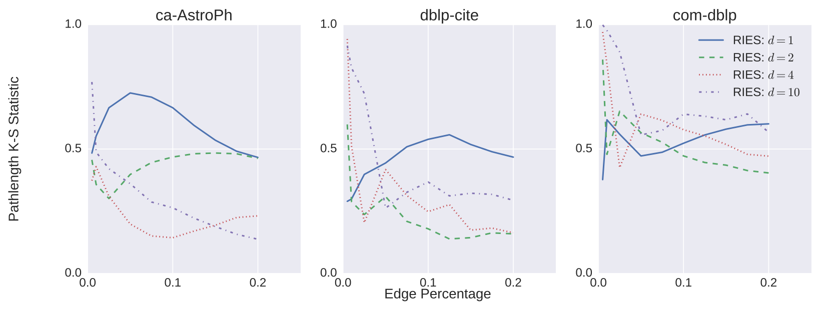
(a) Degree results when varying parameter settings of WES



(b) Degree results when varying parameter settings of RIES



(c) Shortest path length results when varying parameter settings of WES



(d) Shortest path length results when varying parameter settings of RIES

Figure 3.7: The effect of varying method parameters of WES and RIES is shown.

WES. The parameter settings used are $w_2 = 10, 25, 50, 100, 200$, with $w_0 = 1$ and $w_1 = 1$ held constant. Increasing the value of w_2 creates consistent results: both the distribution of shortest path lengths and the degree distribution become more similar to that of the full graph and the K-S statistic decreases. This occurs because a higher value of w_2 compared to w_0 and w_1 creates a more connected graph with higher average degrees, higher clustering coefficients, and shorter distances between pairs of vertices. Because the social network datasets used here are highly connected with small diameter, skewed degree distribution, and many triangles, a higher w_2 parameter usually results in samples more similar to the full graph. This does not mean that increasing the w_2 parameter always produces better results. For example, if a graph has very low clustering coefficients or is very disconnected, a high w_2 might create a subgraph with too many triangles and too few connected components. However, the effect of increasing w_2 on the structure of the subgraph is predictable. Conversely, increasing w_0 relative to w_1 and w_2 has the opposite result. The fact that the effect of increasing or decreasing the parameters of WES is predictable is important for the method to be useful in practice. Section 3.5.3 further shows how the parameters of WES can be adjusted to achieve specific sample features.

Figures 3.7b and 3.7d show results for RIES with $d = 1, 2, 4, 10$. As the edge sample size k_e increases, k_v is set so that $k_v = \frac{k_e}{d}$. Unlike with WES, the effect of the parameter d is not predictable as it varies both across datasets and different edge sample sizes. Figure 3.7b shows that lower values of the assumed average degree d produce better results for smaller edge sample sizes. As the edge sample size increases, the relative performance for higher values of d improves. If the value of d is set too high, then the number of vertices sampled k_v will be too low to obtain k_e edges and the final sample will be smaller than desired. In the streaming context, information about the degree distribution will likely not be available, making it more difficult to choose a value for d. While both RIES and WES perform better than RE, the main advantage of WES over RIES is that the parameters can be set more easily and their effect is more predictable.

3.5 Temporal Graph Sampling

For many applications, a sample may become less relevant over time and newer data may be needed. For these cases, sampling algorithms should be able to produce a temporally biased sample. That is, newer data points will be more likely to be included in the sample. This section addresses the topic of streaming graph sampling with a temporal bias. A new algorithm, Temporal Weighted Edge Sampling (TWES), is presented. Just like WES and RIES, TWES processes an edge stream of unknown length in a single pass to produce a sample with a fixed number of edges. In addition, TWES can bias towards edges with newer timestamps and the degree of bias is parametrically controlled.

3.5.1 Temporal Weighted Edge Sampling (TWES) Algorithm

TWES extends the previously described WES algorithm by using two separate bias factors. In addition to biasing towards edges that connect to the current sample, TWES also biases towards edges with newer timestamps. Recall that when an edge is processed using WES, it is assigned a weight based on how connected it is to the current sample. Let this connectivity weight component be denoted W_C. In TWES, the weight assigned to an edge is a product of W_C and a temporal weight component W_T.

The first weight component, W_C, is obtained the same way as for WES. One of three values is chosen, depending on how many endpoint vertices of the edge are already in the sample at the time the edge is processed. If both endpoint vertices are in the sample, $W_C = w_2$, if only one is, $W_C = w_1$, and if neither is, $W_C = w_0$.

The second weight component, W_T, is derived from the timestamp of the edge. In this work, timestamps represent discrete units of time. For example, if time is measured in the granularity of days, a timestamp may refer to the number of days that have passed since a baseline time. The degree of temporal bias should be tunable, so that anything from strong temporal bias to no temporal bias is possible. When temporal bias is used, more recent timestamps should yield higher weights W_T. This is achieved using exponential decay to obtain weights from timestamps. Under exponential decay, the temporal weight component of an edge with timestamp t is at time t_{curr} given by $e^{-\alpha(t_{curr}-t)}$. The weight depends on the edge's age $t_{curr} - t$, where the lower the timestamp of the edge, the greater its age is and the lower W_T is. Exponential decay has been used in many applications, including in graph analysis for down weighting old data [94, 95, 96, 97].

The problem with using standard exponential decay is that the weight depends on the age of the edge, which changes whenever the current time increases. This poses challenges for stream sampling. Recall that an edge's weight is set to be $W_C W_T$. After drawing a random number $r \in (0, 1)$, its key is set to $key = r^{1/(W_C W_T)}$. Therefore, whenever an edge's age increases, W_T decreases and its key must be recomputed. While this approach is feasible, it requires all keys of the sample to be modified whenever the current time moves forward.

To minimize computation, TWES instead utilizes the method described in [98]. In [98] this is called forward exponential decay, but it may more intuitively be described as exponential growth. Instead of using an element's age, which constantly changes, the forward decay model uses the amount of time between its timestamp t and a fixed point in time L, called the landmark time. Elements that appear more recently have higher values $t - L$ and these values do not change as the current time t_{curr} progresses. For forward exponential decay, the temporal weight W_T of an edge with timestamp t is at time t_{curr} given by

$$\frac{e^{\alpha(t-L)}}{e^{\alpha(t_{curr}-L)}} \tag{3.2}$$

This coincides with standard, or backward, exponential decay.

$$\frac{e^{\alpha(t-L)}}{e^{\alpha(t_{curr}-L)}} = e^{\alpha t-\alpha L-\alpha t_{curr}+\alpha L} = e^{-\alpha(t_{curr}-t)} \tag{3.3}$$

In Weighted Reservoir sampling, elements with the highest keys are chosen and the

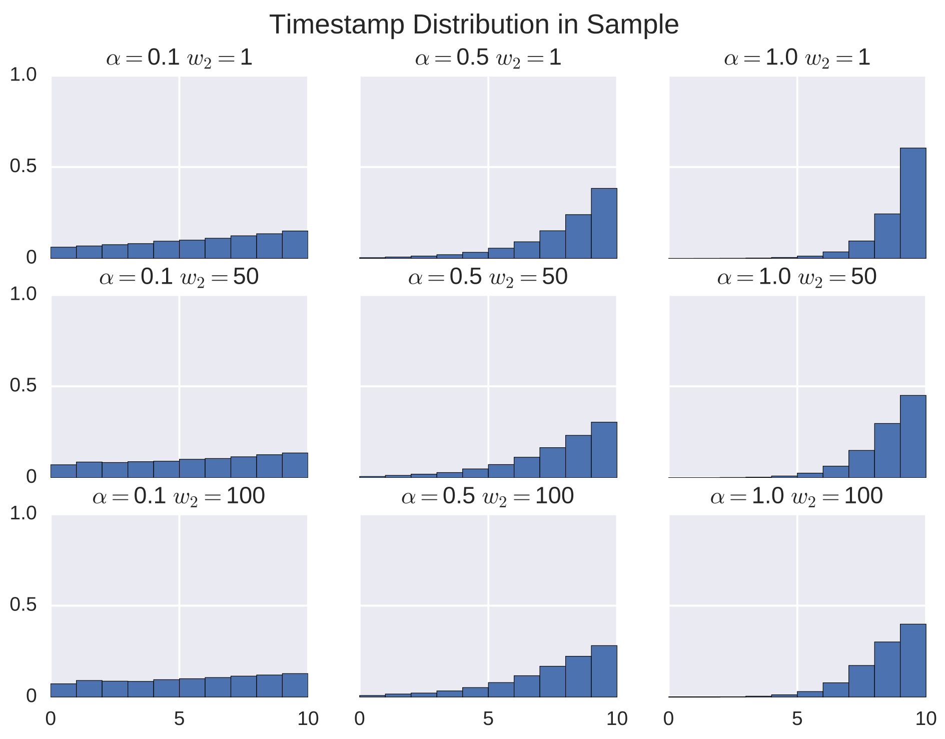


Figure 3.8: The distribution of edge timestamps in samples created by TWES is shown for the *com-dblp* dataset. In each column, the connectivity weight parameter w_2 changes. In each row, the temporal weight parameter α changes.

absolute values of the keys are not relevant. Therefore, only relative weights need to be computed and the denominator of forward exponential decay, which is the same for all elements at any given t_{curr}, can be dropped. The temporal weight component of each edge is then given by $W_T = e^{\alpha(t-L)}$. As this definition depends only on the edge's timestamp t, which does not change, W_T and the resulting key do not need to be recomputed when time moves forward. Thus, the use of forward exponential decay reduces computation compared to normal, backward decay.

Full details of TWES are given in Algorithm 4. In practice, the precision of keys generated with Weighted reservoir sampling are restricted by floating point precision. Implementation solutions related to floating point precision are discussed in Section 3.5.4.

```
**Data**: stream $S$ of edges, edge limit $k_e$, parameters $w_0, w_1, w_2, \alpha$
**Result**: $G_S = (V_S, E_S)$
initialize empty minheap $M$;
$V_S = \emptyset, i = 0$;
**while** $S$ *has edge* $g_i$ **do**
    $(u_i, v_i, t_i) = g_i$;
    $W_T = e^{\alpha(t_i - L)}$;
    **if** $u_i \in V_S$ *and* $v_i \in V_S$ **then**
        $W_C = w_2$;
    **else if** $u_i \in V_S$ *or* $v_i \in V_S$ **then**
        $W_C = w_1$;
    **else**
        $W_C = w_0$;
    **end**
    $weight = W_C W_T$;
    draw $r$ from $Uniform(0, 1)$;
    $key_i = r^{1/weight}$;
    **if** $\mid M \mid < k_e$ **then**
        add $(u_i, v_i, t_i)$ to $M$ with key $key_i$;
        $V_S = V_S \cup \{u_i, v_i\}$;
    **else if** $key_i >$ *smallest key in* $M$ **then**
        Remove element $(u_j, v_j, t_j)$ with smallest key from $M$;
        **if** $u_j$ *has no more edges in* $M$ **then**
            $V_S = V_S \setminus u_j$;
        **end**
        **if** $v_j$ *has no more edges in* $M$ **then**
            $V_S = V_S \setminus v_j$;
        **end**
        add $(u_i, v_i, t_i)$ to $M$ with key $key_i$;
        $V_S = V_S \cup \{u_i, v_i\}$;
    **end**
    $i = i + 1$;
**end**
create $E_S$ from edges in $M$;
return $G_S = (V_S, E_S)$;
```

Algorithm 4: `Temporal Weighted Edge Sampling (TWES)`

3.5.2 Quality Results

This section evaluates the quality of TWES using the same social network datasets shown in Table 3.1 and used in Section 3.4.3. As before, the edges of each dataset form the stream S. Timestamps between 1 and 10 are assigned to each edge in increasing order and the landmark time is $L = 0$. Because TWES is an extension of WES that allows for temporal bias, results for TWES will be the same as for WES when $\alpha = 0$ and no temporal bias is used. Therefore, this section evaluates quality with temporal bias. To achieve good results, TWES should exhibit two properties. First, the temporal bias should allow for higher representation of newer edges over older ones. Second, the sampled subgraph quality should remain high even when temporal bias is used.

To satisfy the first goal the edges of a sample produced by TWES with $\alpha > 0$ should be biased towards higher timestamps. Higher values of α should produce more biased results as long as there are enough edges with high enough timestamps to fill the sample. Figure 3.8 shows that TWES does produce temporally biased samples. Each plot shows the histogram of timestamps in a sample produced by TWES with different parameter settings. Both the connectivity bias (w_2) and temporal bias (α) are varied. In each row, the temporal bias parameter α increases from left to right. As α increases, the timestamp histograms become more skewed, showing that temporal bias is achieved as desired. The top row shows results when $w_0 = 1$, $w_1 = 1$, and $w_2 = 1$, which means that there is no bias towards edges that connect to the sample. In this case the only bias is temporal. In the second and third rows, however, w_2 increases and TWES biases towards edges that connect to the sample. There is both temporal and connectivity bias and these two biases may compete. Even with high values of w_2, however, the samples produced by TWES are temporally biased and as α increases, the bias increases as well.

In addition to producing temporally biased samples, TWES should continue to produce subgraphs that are structurally similar to the full graph. As before in Section 3.4.3, four graph features are used to measure how structurally similar a sampled subgraph G_S

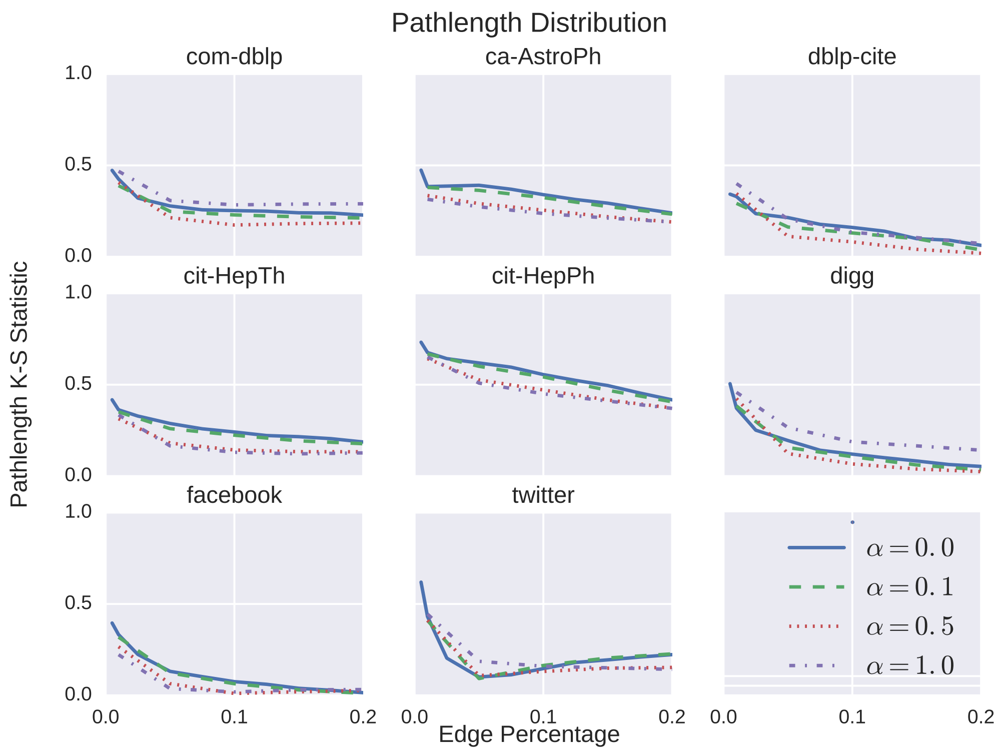


Figure 3.9: The similarity of shortest path lengths between the sampled subgraph G_S and the full graph G is shown for TWES with different degrees of temporal bias. Lower K-S values indicate better results.

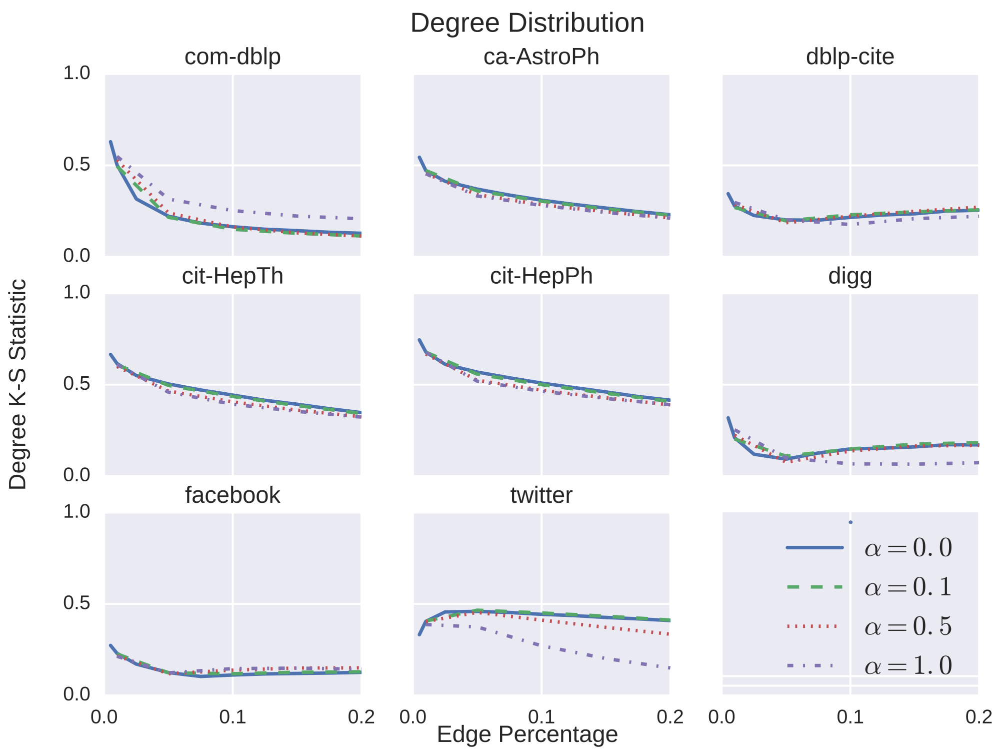


Figure 3.10: The similarity of vertex degrees between the sampled subgraph G_S and the full graph G is shown for TWES with different degrees of temporal bias. Lower K-S values indicate better results.

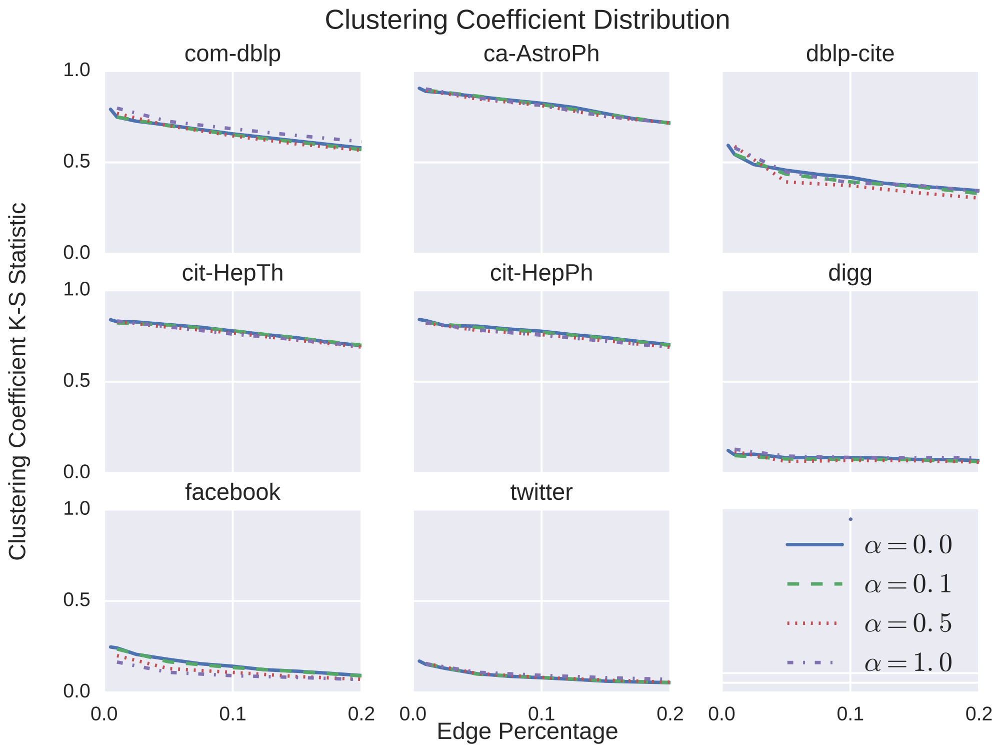


Figure 3.11: The similarity of local clustering coefficients between the sampled subgraph G_S and the full graph G is shown for TWES with different degrees of temporal bias. Lower K-S values indicate better results.

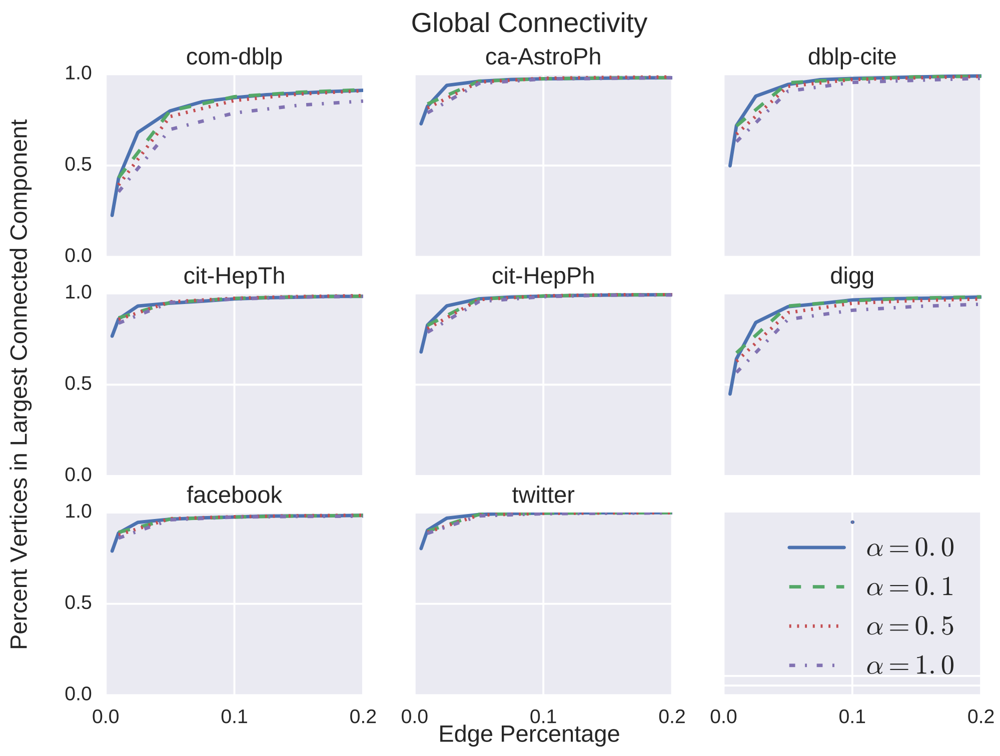


Figure 3.12: The percentage of vertices in the largest connected component is plotted for TWES with different degrees of temporal bias.

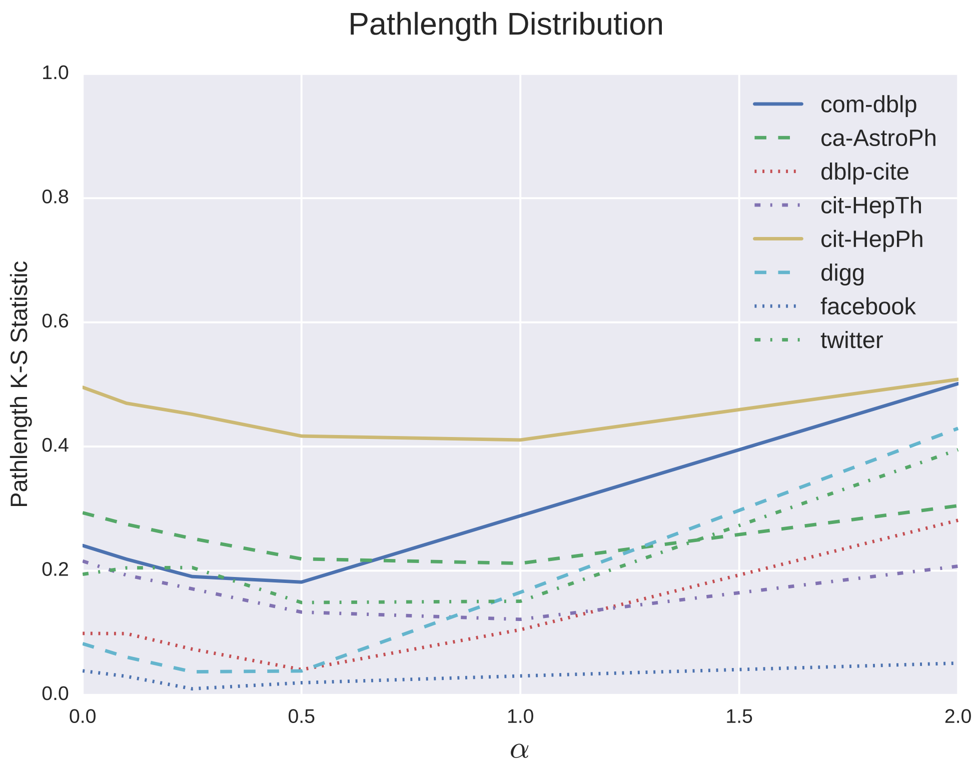


Figure 3.13: The effect of varying temporal bias is shown. The quality (average shortest path length K-S statistic at 15% sampling) of samples produced by TWES is plotted against the temporal bias parameter α.

is to the full graph G. These are the degree distribution, shortest path length distribution, clustering coefficient distribution, and the percentage of vertices in the largest connected component. As in Section 3.4.3, the similarity of distributions in G_S and G is measured using the two sample Kolmogorov-Smirnov (K-S) statistic. The K-S statistic measures the distance between two CDFs, so lower values indicate more similar graph structure. The similarity of connected components in G_S and G is not computed using distributions because in some graphs tested, only a few connected components exist, making distribution comparison infeasible. Instead, the percentage of vertices in the largest connected component is computed.

Figures 3.9 – 3.12 show mean results for the four graph properties discussed. The mean is taken over 10 runs, each on a randomly permuted edge stream. For each permutation, timestamps between 1 and 10 are assigned to each edge in increasing order. Results are shown for TWES where $w_0 = 1$, $w_1 = 1$, $w_2 = 100$, and the temporal bias parameter α is varied. When $\alpha = 0.0$, no temporal bias exists and results are the same as for WES. Overall, Figures 3.9 – 3.12 show that TWES produces similar quality results when $\alpha = 0.0$, $\alpha = 0.1$, $\alpha = 0.5$, and $\alpha = 1.0$. This means that even with temporal bias, TWES produces similar quality subgraphs to those produced by WES. TWES can produce temporal bias while still producing subgraphs that are structurally similar to the full graph, compared to RE from the literature.

On the datasets used, it is possible to both bias towards edges with newer timestamps and to bias towards edges that connect to the sample. For some data streams, however, there may be a trade-off. This may occur if the vertices that appear in a stream change quickly because then new edges would not tend to share vertices with the sample. Futhermore, if temporal bias is high enough, it may dominate over connectivity bias. In the extreme case, the most recently processed edges will form the entire sample.

As Figures 3.9 – 3.12 show, a higher temporal bias parameter α can produce either better or worse structural similarity results, depending on the nature of the dataset in question.

The effect of varying temporal bias through α is explored next. Figure 3.13 shows the K-S statistic for path length distribution as α varies from 0.0 to 2.0. Results are shown for a single sampling percentage of 15%. For all graphs, a similar trend exists. Increasing α from 0.0 initially produces better, more structurally similar, subgraphs. As the temporal bias continues to increase, however, the subgraph quality worsens. The decreased quality for high values of α is intuitive. Because TWES biases both structually and temporally, a trade-off may exist with enough temporal bias. At a high value of α, temporal bias may be strong enough that most or all edges with highest timestamps are included in the sample, regardless of how much they connect to the sample. Then, results may become similar to those seen in RE.

The initial improvement in quality when α rises above 0.0 is less intuitive. It occurs when TWES strongly biases towards edges for which both endpoint vertices are in the sample ($w_2 \gg w_1$ and $w_2 \gg w_0$). In this case, most edges that are added to the sample do not add any new vertices to the subgraph. Let us refer to these edges as fully connected. When a fully connected edge is added, it replaces an edge that was previously in the sample. The new edge will not add any new vertices to V_S. However, if the old edge (u, v) it replaces was the only edge in the sample with endpoint vertex u or v, then $\mid V_S \mid$ decreases. Therefore, every time a new fully connected edge is added to the sample, replacing a previously chosen edge, $\mid V_S \mid$ is more likely to decrease than increase. The number of edges in the sample will stay the same. It follows that the more such replacements occur, the higher the average degree of the sampled subgraph will be. The subgraphs will also be more connected, have shorter path lengths, and have higher clustering coefficients. Values of α greater than 0.0 initially yield better results because they produce temporal bias. With temporal bias, newer edges will have higher weights and will replace previous elements from the sample more frequently. As explained above, this increase in edge replacements can yield more connected, denser subgraphs that are more similar to the full graph.

3.5.3 Parameter Settings

This section discusses how the parameters of WES and TWES can be tuned to produce sampled subgraphs with desired structure. Both algorithms share the same parameters w_0, w_1, and w_2. They assign relative weights to edges that share zero, one, and two vertices, respectively, with the sample when the edge is processed. These parameters can each be increased and decreased during the sampling process to change the structure of the sampled subgraph. Relatively high values of w_2 compared to w_0 and w_1 create denser, more connected subgraphs. These tend to have higher average degrees, fewer connected components, more triangle formation, and shorter distances between pairs of vertices. Therefore, if a denser subgraph is desired, w_2 should be increased or kept at a high value. On the other hand, a higher value of w_1 relative to w_0 and w_2 will create a more disconnected subgraph with fewer triangles. If these qualities are desired w_1 can be increased. A high value of w_0 relative to w_1 and w_2 will produce the most disconnected, sparse subgraph.

Using this information, it is possible to target specific properties by modifying parameters during sampling. The simplest to target is the average graph degree, but others may be possible as well. Figure 3.14 shows results for the standard sampling procedure, in which parameters are constant and for the targeted version, in which parameters are flexible. For the constant parameter version the settings are $\alpha = 0.5$, $w_0 = 1$, $w_1 = 1$, and $w_2 = 100$. For the flexible parameter version, $\alpha = 0.5$ and $w_0 = 1$, but w_1 and w_2 change as the stream is processed, depending on the current characteristics of the sampled subgraph. In the experiment, the goal of the flexible method was to produce a subgraph with a target degree, shown in the plot in solid blue. If the average degree of G_S was below the target value, then $w_1 = 1$ and $w_2 = 100$ to make G_S more dense. If it was above the target value, then $w_1 = 100$ and $w_2 = 1$ to make G_S less dense.

In Figure 3.14, both sampling versions start out with subgraphs of average degree below the target. Once the target is reached, however, the flexible parameter version is able to maintain the desired average degree target value. In fact, by targeting a stable average

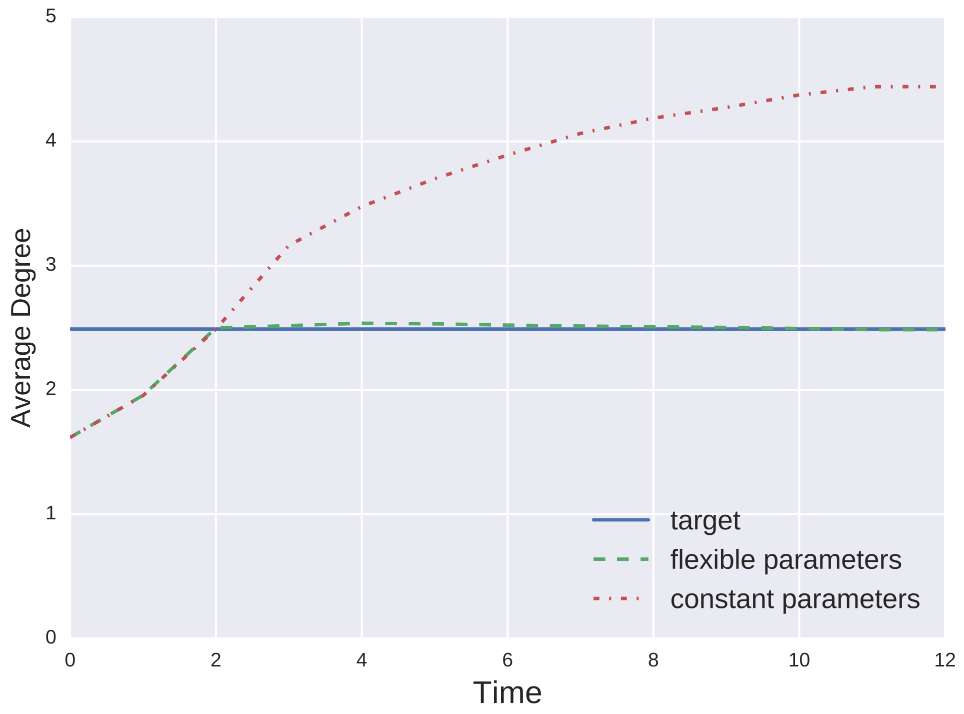


Figure 3.14: The average degree of subgraphs sampled by TWES is shown both for constant parameter settings and when parameters are tuned to achieve a target average degree.

degree value, the average clustering coefficient and the average shortest path length are also stabilized. This experiment suggests that WES and TWES can be used to sample subgraphs with specific structural properties.

3.5.4 Implementation

In the TWES algorithm, the weight assigned to each edge of the stream is $W_C W_T$, where W_C represents the connectivity component and will be set equal to w_0, w_1, or w_2, depending on how many of its endpoint vertices are in the sample when the edge is processed. The temporal component biases towards more recent edges and is $W_T = e^{\alpha(t-L)}$ for timestamp t. In order for TWES to bias towards edges that have higher timestamps and that are more connected to the sample, higher keys should be assigned to edges with higher weights $W_C W_T$.

However, if timestamps reach a high enough value, the possibility of distinguishing between keys of edges will diminish due to floating point precision. Recall that for each edge, a random number $r \in Uniform(0,1)$ is drawn and then $key = r^{1/(W_C W_T)}$. Therefore, as timestamps increase, the temporal weight component W_T will also increase and values of keys will rise and be closer to 1. For example, for $L = 0$, $\alpha = 0.5$, $r = 0.5$, $W_C = 1$, and $t = 1$, the key is approximately 0.657. If the timestamp rises to $t = 5$, the key rises to approximately 0.945. When the keys shift closer to 1, they also shift closer together. Very high keys may be rounded up to 1. Thus, the ability to distinguish between edges with different weights diminishes due to a limitation of floating point precision. Once timestamps are high enough to meet cutoff criteria, a timestamp shift must be performed. First the cutoff condition is described and later, the timestamp shift is explained.

The timestamp cutoff occurs when $t - L$ is large enough that, for a pre-determined high value of $r \in (0,1)$ ($r = 0.999$ in the implementation),

$$r^{1/(w_2 exp(\alpha(t-L)))} = r^{1/(w_2 exp(\alpha(t+1-L)))} \tag{3.4}$$

If the above condition is satisfied, then for the same random r drawn, a higher timestamp does not yield a higher key and the temporal bias component of TWES will be weakened. Given two edges e_i and e_j, if e_i has timestamp t_i, e_j has a higher timestamp $t_j = t_i + 1$, both have the same connectivity weight $W_C = w_2$, and both edges draw the same $r \in (0,1)$, then the the key of e_j should be greater than the key of e_i. If not, then the temporal bias will not be correct. The above condition will be met for higher values of r at lower timestamps. Therefore, in order to be conservative, our implementation of TWES plugs in a high value of $r = 0.999$ and checks for the lowest timestamp for which the condition is satisfied. 99.9% of edges draw a lower value of r and temporal bias will work well overall.

When the condition above is met, a timestamp shift occurs. The landmark time L is increased so that the relative time $t - L$ is lowered. Then, for all edges stored in the sample, their weight and key must be adjusted. If L increases to $L + D$, the temporal component W_T of each edge is multiplied by $e^{-\alpha D}$ and the key is recomputed with the same values of r and W_C as before. This adjustment produces the correct new value of W_T because $e^{\alpha(t-L)}e^{-\alpha D} = e^{\alpha(t-(L+D))}$.

Claim: Performing this temporal shift will preserve the relative order of all keys in the reservoir. If $key_i > key_j$ before the shift with landmark L, then $key_i \geq key_j$ after the shift with landmark $L + D$.

Proof: For edge e_i, let the connectivity weight component be denoted W_{Ci}. Before the shift, its key is then $r_i^{1/(W_{Ci}exp(\alpha(t_i-L)))}$ and after the shift it is $r_i^{1/(W_{Ci}exp(\alpha(t_i-L-D)))}$.

$$\begin{aligned}
&r_i^{1/(W_{Ci}exp(\alpha(t_i-L)))} > r_j^{1/(W_{Cj}exp(\alpha(t_j-L)))} \\
\Longrightarrow\ &\big(r_i^{1/(W_{Ci}exp(\alpha(t_i-L)))}\big)^{exp(\alpha D)} > \big(r_j^{1/(W_{Cj}exp(\alpha(t_j-L)))}\big)^{exp(\alpha D)} \\
\Longrightarrow\ &r_i^{1/(W_{Ci}exp(\alpha(t_i-L))exp(-\alpha D))} > r_j^{1/(W_{Cj}exp(\alpha(t_j-L))exp(-\alpha D))} \\
\Longrightarrow\ &r_i^{1/(W_{Ci}exp(\alpha(t_i-L-D)))} > r_j^{1/(W_{Cj}exp(\alpha(t_j-L-D)))}
\end{aligned} \tag{3.5}$$

The equality condition will only hold if the new keys both are rounded to the same value due to floating point precision. □

It is important to note that the time shift preserves the ability of TWES to distinguish between keys of edges with newer, higher timestamps. The trade-off for this is that it becomes more difficult to distinguish between the keys of edges with lowest timestamps. When $t - L$ is already low, shifting it lower will eventually result in negative values $t - L$, which moves keys towards 0 and leads to precision issues again. However, this trade-off has very limited effect on TWES. Temporal bias indicates that newer edges are most important and few edges with low timestamps will be in the sample.

3.6 Conclusion

This chapter presented new algorithms for streaming graph sampling, WES and RIES. These methods sample a subgraph from an edge stream in a single pass without assuming any order of the edges and without requiring any information about the full graph or size of the stream. Because our goal in sampling is to obtain a smaller subgraph, the methods restrict the number of edges and not only the number of vertices. The other streaming approach from the literature that meets this requirement is RE. Experiments on several social network datasets show that WES and RIES create sampled subgraphs that are more structurally similar to the full graph than does RE. While both methods perform well, the advantage of WES over RIES is that there is no need to set the average degree of the subgraph and the effect of the parameter setting is more predictable.

For many applications, new data is considered more relevant and may be preferred in a sample. This work also addressed the topic of temporally biased sampling on graph streams through the TWES algorithm. TWES, which also samples an edge stream in a single pass, can bias output towards newer edges and the degree of this bias is parameterized. Experiments show that TWES can produce subgraphs that are both temporally biased and structurally similar to those produced by WES. By adjusting parameters in real time as the stream is processed, both WES and TWES can sample subgraphs with targeted structural properties. This suggests that they can be useful in real-life applications.

The algorithms presented in this chapter produce a global sample of a graph. For certain applications, however, it may be useful to sample a local subgraph around a seed vertex or set of seed vertices of interest. Future work may explore the application of techniques used in WES and TWES to local streaming graph sampling.

CHAPTER 4
CREATING DYNAMIC GRAPHS FROM DATA STREAMS

4.1 Introduction

Dynamic graphs are used to represent relational data as the relationships change over time. Typically, a dynamic graph is a sequence of graph snapshots, each corresponding to the state of the graph at some particular point in time. Changes in the underlying data are then represented both by the addition of new edges as new relationships are added, the removal of old edges as old relationships disappear, or by the alternations of edge weights as existing relationships strengthen or weaken. Whenever dynamic graph analysis is performed, it is first necessary to decide how to add, remove, or modify weights of edges to create a dynamic graph from temporal data. In particular, the question of how to remove past data is not straightforward, especially when edges are derived from interaction data. Once interaction occurs, it is not explicitly reversed, but rather may decrease in relevance over time. Therefore, it is not obvious when to delete edges. This method of removing or de-emphasizing edges is important because the way in which it is done will affect the structure of the dynamic graph and how it changes and therefore the result of algorithmic analysis. Competing goals include preserving graph stability and detecting new changes.

This chapter addresses the problem of how to age past data when creating a dynamic graph from a stream of temporal data. By aging, we refer to any process that decreases the influence of historical data in a graph. The goals are to provide practitioners with quantitative comparisons of available methods and to provide and compare new alternative approaches. Section 4.2 presents two new methods, called *Active Vertex* and *Active Edge*, and discusses two existing methods from the literature, *Sliding Window* and *Weight Decay*. Through experiments on temporal graphs from five social networks in Section 4.3, these approaches are compared and contrasted to find patterns that consistently appear. This work was published in [99].

The work most similar to this is that of Saganowski *et al.* [74] and Oliveira *et al.* [75], who evaluate how two different ways of building dynamic graphs affect communities found and how they change. They consider the aggregate approach, in which edges are only added and never removed, and the sliding window method, in which older edges are removed. Both find that removing edges with the sliding window method causes more change and instability in community structure compared to aggregating. While this previous work studies only the sliding window approach, this chapter presents two newly developed methods and compares them to two previous techniques. In addition, this work examines aging performance on multiple datasets, whereas [75] and [74] each use only a single one. By using multiple datasets, it is possible to study which patterns occur consistently for a variety of data sources.

4.2 Aging Methods

This section describes and discusses methods for creating dynamic graphs from temporal data. Static graphs are created from a collection of actions or relationships. Deciding how these actions are transformed into edges and vertices of a static graph is not always straightforward. For example, in the case of a graph built from interactions between people, it is necessary to determine under what circumstances such communication should form an edge. One approach is to set a threshold for the frequency of communication over time before an edge is created. Different thresholds will produce very different graphs [60]. When forming a static graph of relationships between people based on common event attendance, various approaches may differently factor how many events two people attended and the number of people at each event [100], again with differing results.

Creating dynamic graphs poses additional complications. In addition to the problems found in the static case, it is necessary to decide when new data is added, when old data is removed, and how the two processes are combined.

As time passes, new actions take place and relationships change. A dynamic graph should reflect this change. In order for a graph to represent the evolving nature of the underlying data, new edges, representing new relationships, may be added, and old edges, representing old relationships, may be removed. Edge weights may be increased if supported by new data or decreased if the relationship loses relevance.

4.2.1 Definitions

It is necessary to distinguish between the stream of edges that represent interactions or events occurring over time and the graphs built from the stream. Let S be a stream of edges in time, where each $(u, v, weight, time) \in S$ is a tuple of the two endpoint vertices, the edge weight, and the time of the event represented by the edge. The edges occurring during a specific time interval are represented by $S_{l,k} = \{(u, v, weight, time) \in S \mid l < time \leq k\}$.

Let $G = (V, E)$ be a static graph where V are the vertices of G, and E are the edges. For our notation, each edge $(u, v) \in E$ has additional features: a weight $(u, v).weight$ and a timestamp $(u, v).time$ corresponding to the most recent time it appeared in S. Each vertex $v \in V$ has a timestamp $v.time$, corresponding to the latest time of any of its edges. G may be created from data in S using Algorithm 5.

A dynamic graph is then represented by a series of graph snapshots over time $\{G_1, G_2, \ldots, G_n\}$, where $G_t = (V_t, E_t)$ is the state of the dynamic graph at time index t. As time moves and the underlying data changes, vertices and edges will be added, removed, and modified to create a new graph snapshot. The time t may represent any discretized unit of time. It is not continuous time. For example, if months are used, G_1 is the graph after the first month, G_2 is the graph after the second month, and $S_{0,2}$ contains the edges with times within the first two months.

In this work we assume all graphs are undirected. Therefore, the vertices of each edge in

S are also undirected. To remove ambiguity, we assume that for all $(u, v) \in E_t$, $u < v$ and for all $(u, v, weight, time) \in S$, $u < v$. The aging algorithms presented in this chapter will therefore also process edges in an undirected manner. In practice the implementation of an undirected graph may vary. It is common for implementations to include both (u, v) in a list of edges of u and (v, u) in a list of edges of v to allow for faster accessing of all neighbors. However, we treat implementation issues as "under the hood". For implementations that require two directions, when an edge (u, v) is added, both directions (u, v) and (v, u) are added. Similarly, if an edge is removed or its weight or time features modified, this is done for both directions.

4.2.2 Sliding Window

The *Sliding Window* approach builds a dynamic graph by creating graph snapshots from intervals, or windows, of a stream of edges. Only the most recent data in the stream, inside the current window, is used to create a graph, while all previous data is forgotten. For example, a daily sliding window would create the most recent graph snapshot using only activity from the last 24 hours. For *Sliding Window*, each graph snapshot G_t is created as defined below, where λ is the window size and `ToGraph` is defined in Algorithm 5.

$$G_t = \texttt{ToGraph}(S_{t-\lambda,t}) \tag{4.1}$$

If the unit of time is one week, then $\lambda = 3$ means that the sliding window size is three weeks and the window shifts by one week, causing a $2/3$ overlap between consecutive windows.

Consecutive windows can overlap to varying degrees, or not at all, with a higher degree of overlap causing smoother and more gradual changes. *Sliding Window* is easily interpreted because it is clear what data each graph snapshot represents. The emphasis on new versus historical data can be adjusted with the window size. However, it does not distin-

```
Data: $S_{l,k}$
Result: $G$
initialize empty $G = (V, E)$;
for $(u, v, w, t) \in S_{l,k}$ do
    if $u \notin V$ then
        $V = V \cup u$;
    end
    if $v \notin V$ then
        $V = V \cup v$;
    end
    $u.time = t$;
    $v.time = t$;
    if $(u, v) \in G$ then
        $(u, v).weight += w$;
    else
        $E = E \cup (u, v)$;
        $(u, v).weight = w$;
    end
    $(u, v).time = t$;
end
```

Algorithm 5: `ToGraph`

guish well between temporary and persistent relationships. For example, graph vertices that correspond to two people who communicate regularly over time, but did not communicate in a given week will have no edge connecting them in the graph when using a weekly sliding window. Therefore, the resulting dynamic graph may change greatly between consecutive snapshots. This method is best suited for applications where a graph should represent only the most recent actions, not a balance of historic and new data, and it requires a careful choice of window size and overlap.

Sliding Window has been used to create dynamic graphs in many works. Non-overlapping windows are used to study the incremental k-clique clustering algorithm [101] and to test a framework for tracking community evolution on the Enron dataset [102]. Overlapping sliding windows have been used to create dynamic graphs to predict vertex centrality [103], to test the DENGRAPH algorithm for incremental community detection [104], and to study community dynamics [105, 106]. In [73], a sliding window of 60 days is used to create graphs of email exchanges. Unlike many other sliding window based approaches, instead of summing the number of edges to obtain a weight, the weight in [73] is set to the geometric rate of bilateral email exchange during the 60 day window. Clauset and Eagle [72] analyze a network of physical contact from the Reality Mining study and investigate the effect of the size of the window.

4.2.3 Edge Weight Decay

The second method discussed in this work is the edge weight decay approach in which, over time, the weight of old edges is decreased. The philosophy behind this approach is to treat new data as more important than old data, while also allowing stronger (higher weight) relationships to last longer than weaker (lower weight) ones. This is done by creating a new graph snapshot from a weighted combination of the previous graph and the new edges. Edges below a weight threshold ϵ are removed to eliminate relationships that are no longer relevant. Throughout the text we refer to this method as *Weight Decay*. Each graph snapshot created by *Weight Decay* is defined as:

$$G_t = \texttt{Decay}(G_{t-1}, \alpha, \epsilon) + \beta * \texttt{ToGraph}(S_{t-1,t}) \tag{4.2}$$

where `Decay` is defined in Algorithm 6.

In *Weight Decay*, edges that represent frequent communication or strong relationships will last for a longer time, which may create a dynamic graph with smoother transitions between snapshots compared to the *Sliding Window*. The parameters α and β control the relative importance of historic and new data, while ϵ represents the lowest weight edge that will remain in the graph. Compared to the sliding window approach, the interpretation of a graph snapshot based on the parameters is more difficult because there is no simple interpretation, such as “activity from the past week”.

Variations of this method have been used in the literature to create dynamic graphs.

```
Data: $G = (E, V)$, $\alpha$,$\epsilon$
Result: $G$
for $(u, v) \in E$ do
    $(u, v).weight = (u, v).weight * \alpha$;
    if $(u, v).weight < \epsilon$ then
        $E = E \setminus (u, v)$;
        if $degree(u) == 0$ then
            $V = V \setminus u$;
        end
        if $degree(v) == 0$ then
            $V = V \setminus v$;
        end
    end
end
```

Algorithm 6: `Decay`

Chen *et al.* predicts future links in a co-authorship network by exponentially decaying weights, to strike a balance between the the recency of a co-published paper and the number of co-published papers [94]. Cortes *et al.* summarize past behavior by using a linear combination of new and historical data and keeping only the top k edges for each vertex [95]. This same method is used in [96] to predict future data. Chan *et al.* predict future edges in a mobile network using exponential aging [97].

4.2.4 Active Graph

In addition to *Sliding Window* and *Weight Decay* from the literature, this chapter also proposes a new approach to creating a dynamic graph from a stream of edges. The idea behind the new approach stems from the fact that entities join and leave social networks and the amount of time a given entity or relationship is active in the network varies greatly. For example, many users join an online social network and soon become inactive [107, 108]. The edges representing their initial activity upon joining can likely be removed quickly because they no longer participate. Other users, however, remain active in the social network. Their information may be more important and therefore kept in the dynamic graph for longer. In such cases, it may be advantageous to retain even very old edges in the graph if they are connecting two very active users, so that all edge, or connection data for the active users is preserved

The two variations presented are the *Active Vertex* and *Active Edge* methods. *Active Vertex* keeps track of the last time each vertex was active (had a new edge in the stream S). Vertices whose last activity is within a specified window of time, τ_V, are considered active and we keep all edges whose two endpoint vertices are active. All other edges are removed. Each graph snapshot using *Active Vertex* is defined as:

$$G_t = \texttt{CheckActiveVertices}(G_{t-1} + \texttt{ToGraph}(S_{t-1,t}), t, \tau_V) \tag{4.3}$$

where `CheckActiveVertices` is defined in Algorithm 7.

For the second variation, *Active Edge*, we keep track of the last time an edge's weight was incremented (the last time an activity occurred between the two vertices). Edges whose last activity is within a specified window, τ_E, are considered active and their weights do not decrease. Inactive edges are removed from the graph regardless of their weights. Each graph snapshot using *Active Edge* is defined as:

$$G_t = \texttt{CheckActiveEdges}(G_{t-1} + \texttt{ToGraph}(S_{t-1,t}), t, \tau_E) \tag{4.4}$$

where `CheckActiveEdges` is defined in Algorithm 8.

The *Active Vertex* and *Active Edge* methods can be useful in representing the history of activity in active parts of the graph, while forgetting the inactive portions. They may also be useful in storing important parts of the graph when memory size is limited because edges are either preserved with cumulative weights, or removed completely. Similarly, they may be useful for visualization to represent accumulated relationships only of active vertices, thus reducing visual clutter.

```
Data: $G = (E, V)$,$t$,$\tau_V$
Result: $G$
for $(u, v) \in E$ do
    if $t - u.time > \tau_V$ or $t - v.time > \tau_V$ then
        $E = E \setminus (u, v)$;
    end
end
for $v \in V$ do
    if $v.time > \tau_V$ or $degree(v) == 0$ then
        $V = V \setminus v$;
    end
end
```

Algorithm 7: `CheckActiveVertices`

```
Data: $G = (E, V)$,$t$,$\tau_E$
Result: $G$
for $(u, v) \in E$ do
    if $t - (u, v).time > \tau_E$ then
        $E = E \setminus (u, v)$;
    end
end
for $v \in V$ do
    if $degree(v) == 0$ then
        $V = V \setminus v$;
    end
end
```

Algorithm 8: `CheckActiveEdges`

Table 4.1: Datasets and parameters used.

Graph	Vertices	Edges	Time Unit	Window Size
dblp	1,314,050	18,986,618	1 year	3 years
enron	87,273	1,148,072	4 months	1 year
slashdot	51,083	140,778	1 month	3 months
youtube	3,223,589	9,375,374	2/3 months	2 months
facebook	46,952	876,993	4 months	1 year

4.2.5 Setting Parameters for Each Aging Method

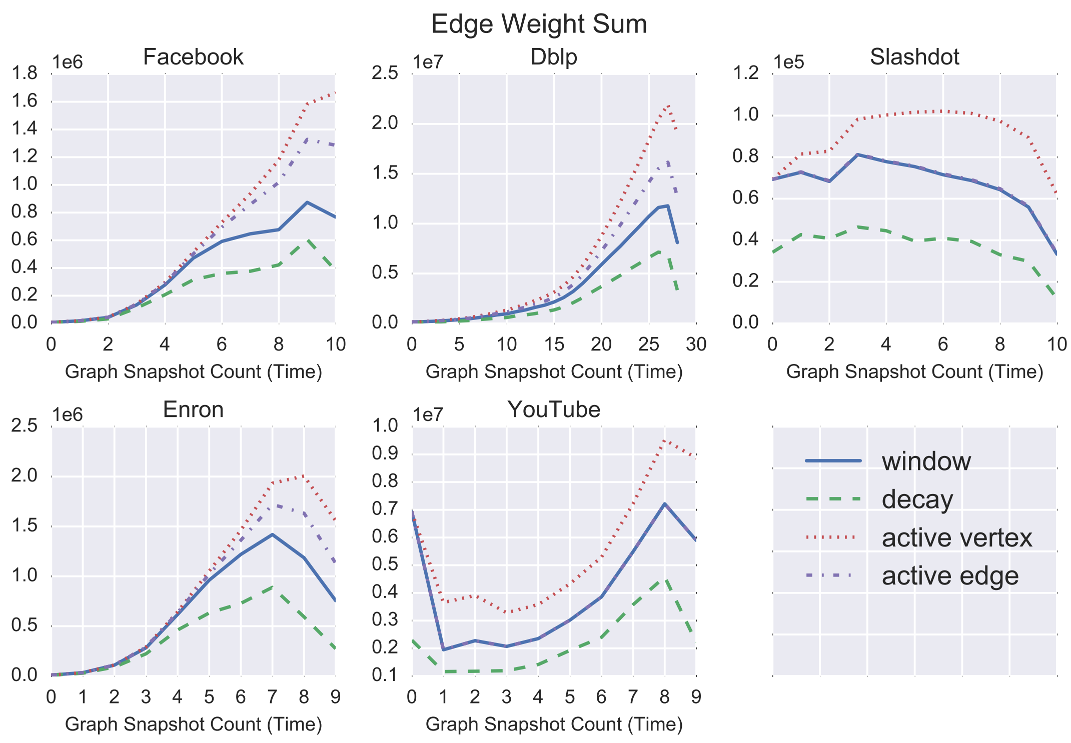
Section 4.3, compares the four methods described. To make a fair comparison, the various parameters are set to make all methods as similar as possible. To make *Weight Decay* with parameters α and ϵ similar to *Sliding Window* with parameter λ, we set $\alpha = \epsilon^{\frac{1}{\lambda}}$. An edge with weight 1 in *Weight Decay* method will persist for λ snapshots, just as it would in *Sliding Window*. Higher weighted edges will stay in the graph for longer. We also set $\beta = 1$ so that new data has the same weight as in the other methods.

For *Sliding Window*, only data within λ time of the current time is included in a graph snapshot. For *Active Vertex*, the threshold of a vertex being active τ_V is set equal to λ from *Sliding Window*. Similarly, for *Active Edge*, τ_E is set equal to λ. Note that setting $\tau_V = \lambda$ means that the set of vertices with degree greater than zero of a graph snapshot G_t produced by *Sliding Window* and by *Active Vertex* will be the same. Similarly, setting $\tau_E = \lambda$ causes the set of edges in a graph produced by *Sliding Window* and by *Active Edge* to be the same (the edge weights may differ though).

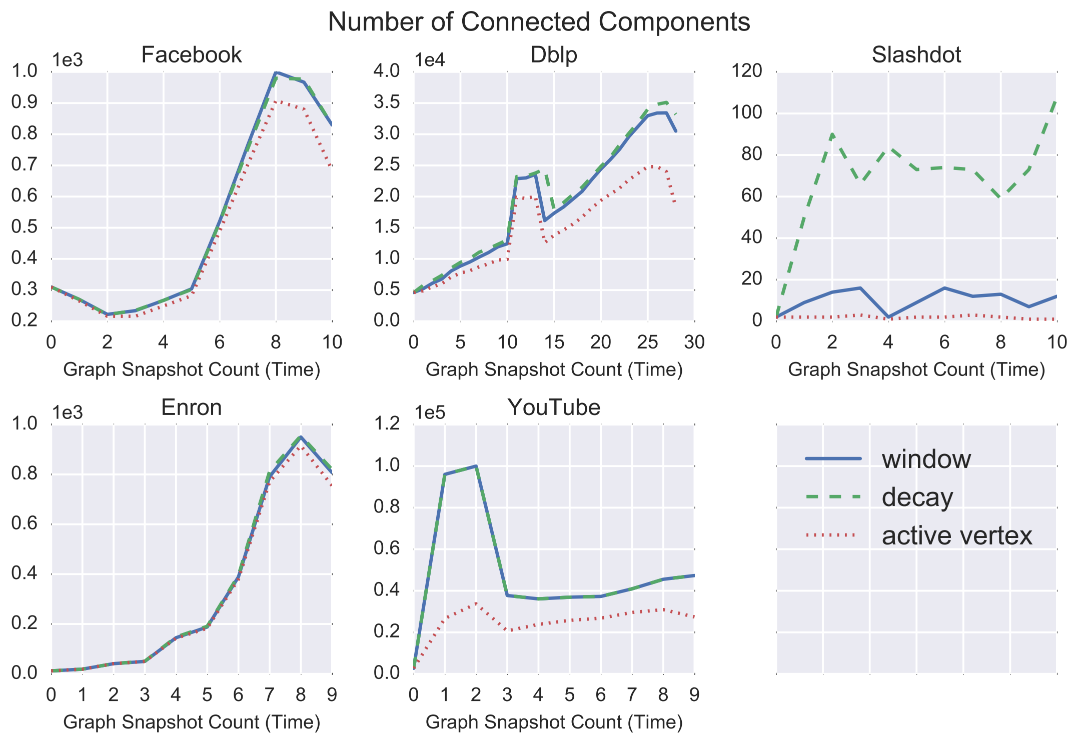
4.3 Results

4.3.1 Experimental Setup

The four methods for creating dynamic graphs are compared using the five social network datasets listed in Table 4.1. These five data sets are publicly available at the Koblenz Network Collection [59]. The *dblp* set describes co-authorship relationships in academic



(a) The sum of edge weights over time.



(b) The number of connected components over time.

Figure 4.1: Global properties of the dynamic graphs are shown over time. The x-axis shows the graph snapshot count.

papers. In the *enron* set the edges correspond to e-mails exchanged between Enron employees. The *slashdot* set includes an edge between users who reply to each others posts online. The *youtube* data contains friendship relationships on the website and the *facebook* set contains edges between users who post on each other's walls. For a proper comparison, the parameters of each method are set as described in Section 4.2.5. The unit of time is the same for each aging method and given in Table 4.1. For each dataset apart from *dblp*, the beginning and ending times with too few edges are removed and then the unit of time is set so that when $\lambda = 3$, approximately four non-overlapping windows fit into the time range. Because the *dblp* dataset had a much longer history, 30 years of data from 1984 to 2014 were used with a window size of 3 years based on the assumption that research relationships may change significantly over this time interval. For *Sliding Window*, we used $\lambda = 3$, yielding a $2/3$ overlap between consecutive windows. For example, each graph snapshot from *dblp* represented edges from three years and consecutive snapshots would have two years worth of edges in common. Parameters of other methods are then set as described in Section 4.2.5. Both τ_V and τ_E are set to equal λ. A small value of $\epsilon = 0.1$ is used and therefore $\alpha = 0.1^{\frac{1}{3}} \approx 0.464$.

The above parameters are used for results in Figures 4.1 – 4.8. In the experiments shown in Figures 4.9 – 4.12, the amount of overlap between consecutive windows of *Sliding Window* is varied, while keeping the number of months in each sliding window the same as shown in Table 4.1. The details are further explained in Section 4.3.4.

4.3.2 Global Properties

Based on the chosen parameters, *Sliding Window* and *Active Vertex* will result in graphs with the same number of vertices, while *Sliding Window* and *Active Edge* will produce graphs with both the same number of vertices and edges. *Weight Decay* will output graphs with at least as many vertices and edges as *Sliding Window*.

In Figure 4.1a, the sum of edge weights of the dynamic graphs is shown over time.

For all graphs, *Active Vertex* produces the highest edge weight sum. Interestingly, *Weight Decay*, although it stores more edges than *Sliding Window* and *Active Edge*, produces the lowest edge weight sum. This suggests that edge weights in the graphs produced by these datasets are generally low and fall below the removal threshold ϵ quickly.

Figure 4.1b shows the number of weakly connected components of the dynamic graphs over time. The weakly connected components of a graph are the maximal sets of vertices that have no edges to any other vertices. Only vertices with non-zero degree are considered. The effect of *Active Vertex* and *Weight Decay* is the opposite on graph connectivity. Figure 4.1b shows that *Weight Decay* produces graphs with a larger number of connected components, meaning that it creates a more disconnected graph. *Active Vertex*, on the other hand, produces the fewest connected components, meaning that the graph is more connected.

It is interesting to see that while the scores produced by different methods differ, they follow the same pattern by increasing and decreasing at the same time. This suggests that when the method parameters are matched using the method described in Section 4.2.5, similar types of changes can be detected by any of the four methods. For example, the points in time at which a graph disconnects will be approximately the same for each method, marked by an increase in the number of connected components.

4.3.3 Vertex Properties

Figures 4.2 – 4.7 show results for several vertex-level centrality scores over time. The x-axis of each shows the graph snapshot count and the y-axis of each represents how much the centrality scores have changed from the previous graph snapshot to the next. This is computed using two measures. The first is the Spearman rank correlation and the second is the Jaccard Index.

The Spearman rank correlation measures the strength of association between two ranked

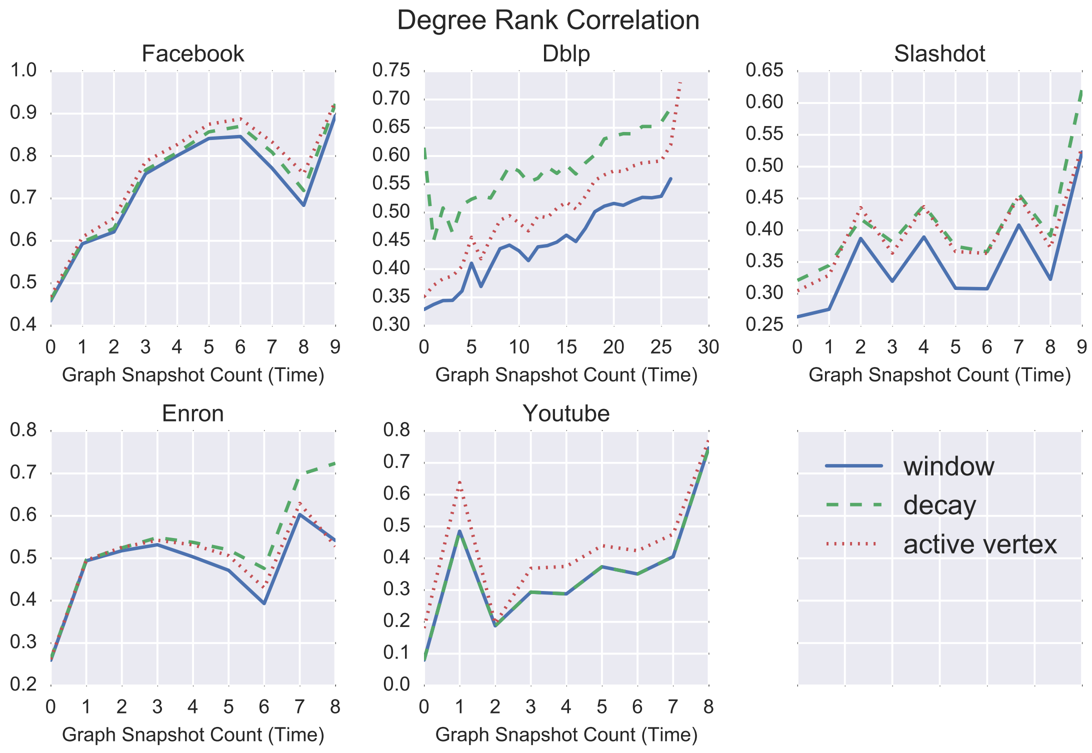


Figure 4.2: The Spearman's rank correlation of unweighted degrees is shown over time for each method.

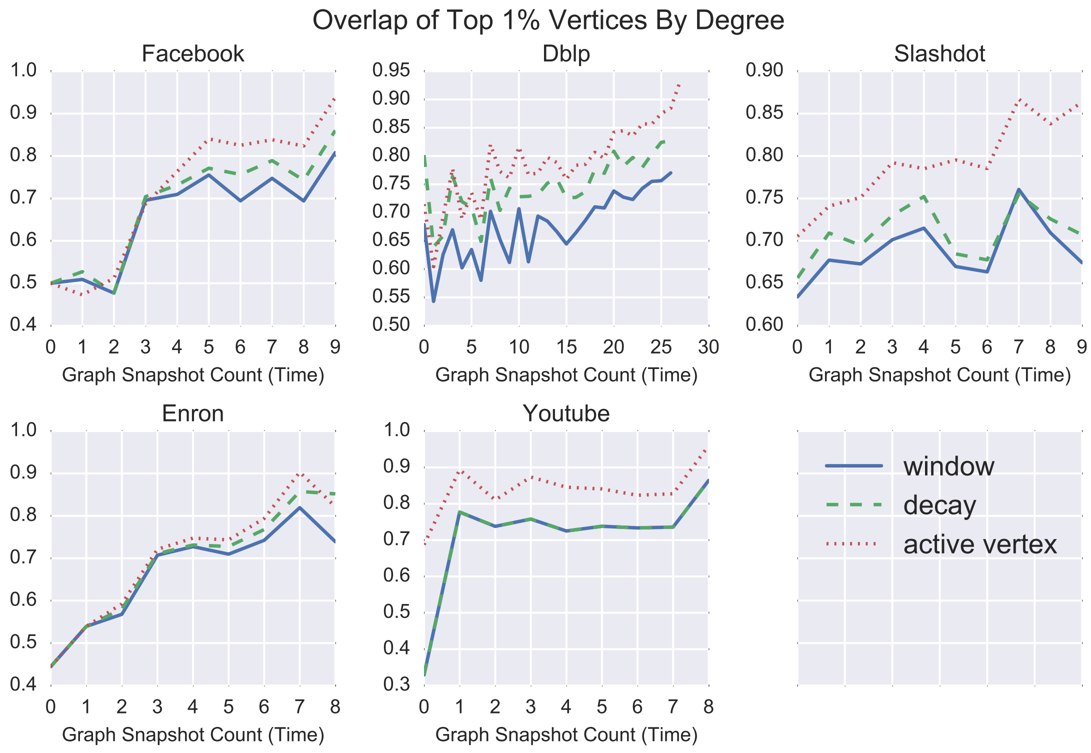


Figure 4.3: The overlap (Jaccard index) between the top 1% of vertices based on unweighted degree from consecutive snapshots is shown.

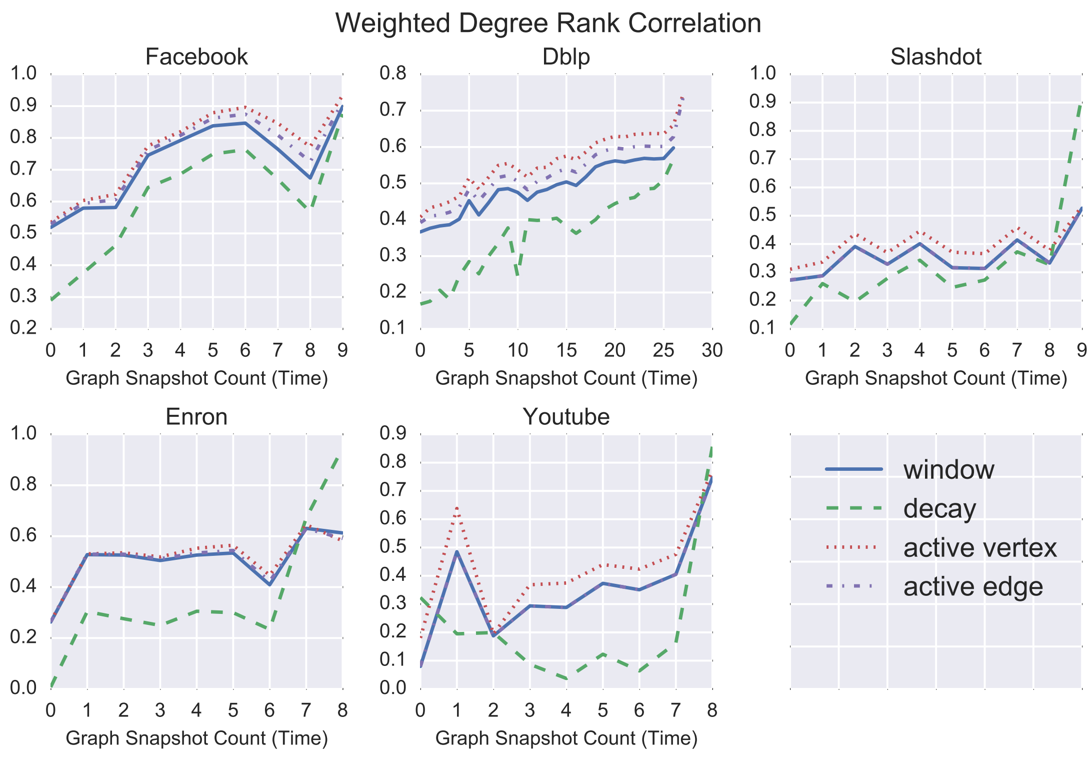


Figure 4.4: The Spearman's rank correlation of weighted degrees is shown over time for each method.

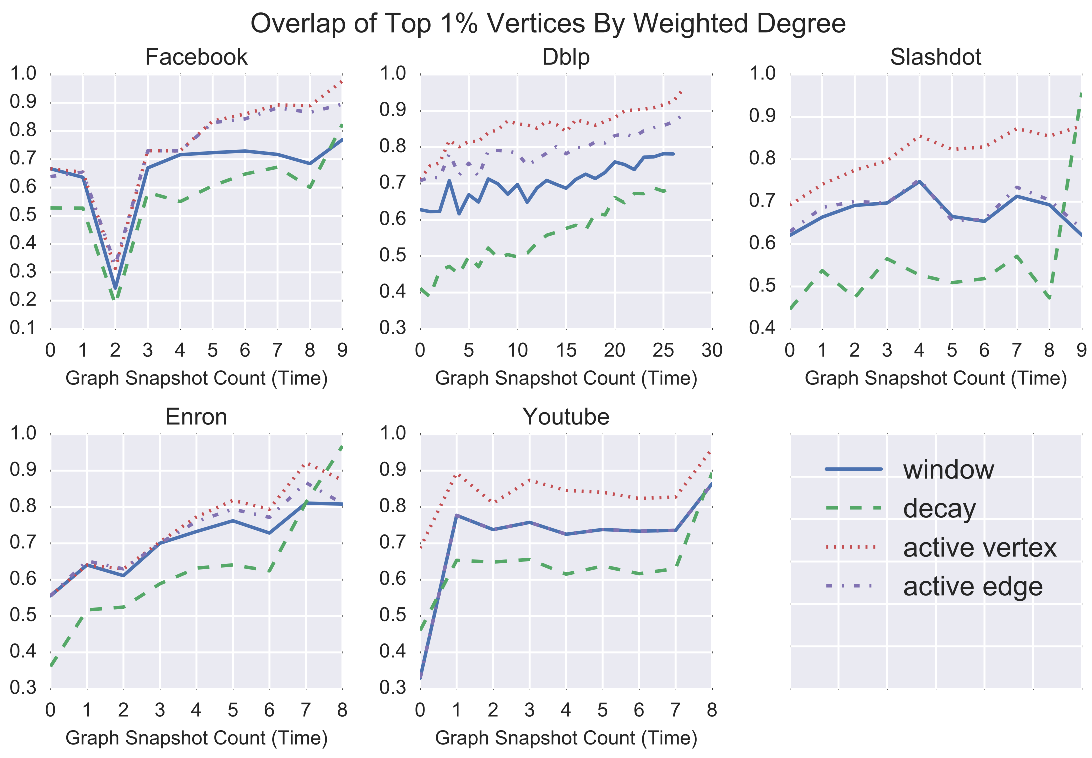


Figure 4.5: The overlap (Jaccard index) between the top 1% of vertices based on weighted degree from consecutive snapshots is shown.

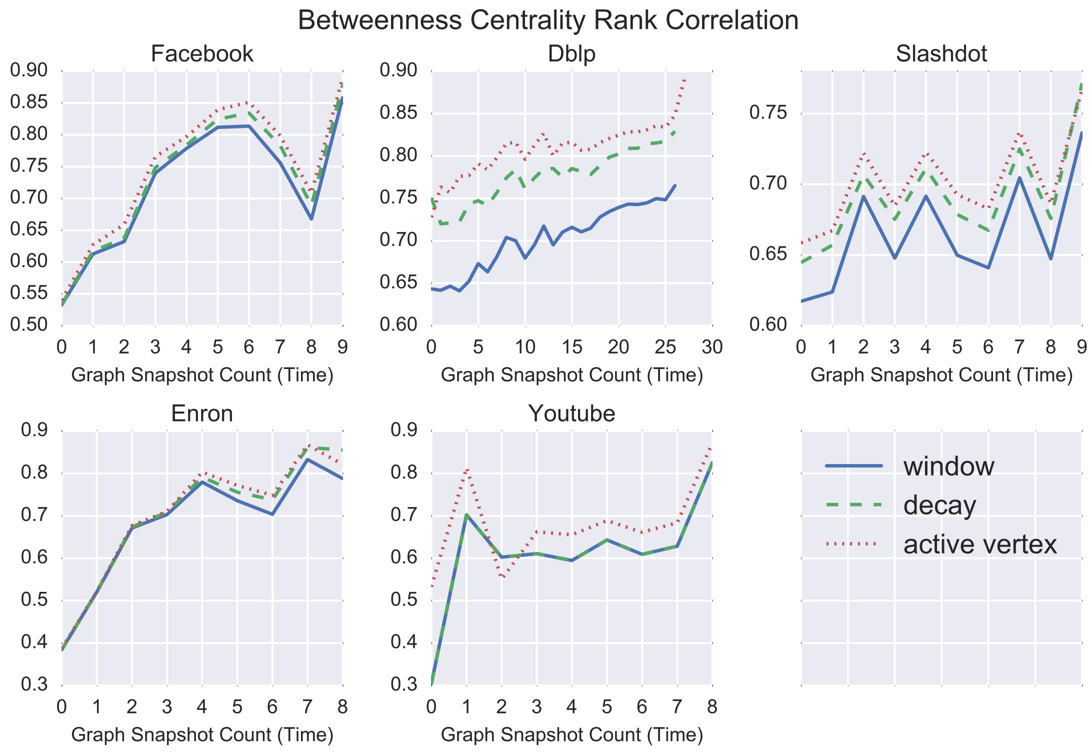


Figure 4.6: The Spearman's rank correlation of betweenness centrality is shown over time for each method.

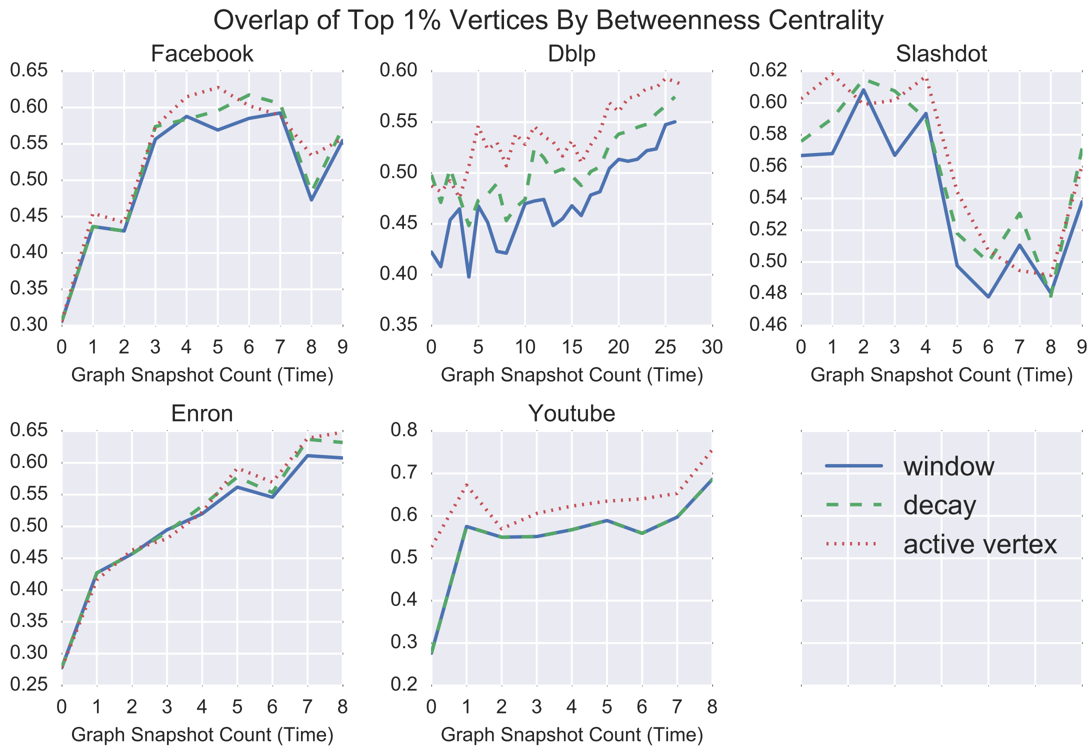


Figure 4.7: The overlap (Jaccard index) between the top 1% of vertices based on betweenness centrality from consecutive snapshots is shown.

variables. Here, it can be used to determine how much the ranking of vertices according to a particular property (such as degree) changes from one snapshot to the next. Given the vertex centrality scores in G_t a rank vector X_t can be constructed. For example, the vertex with highest centrality will have rank 1 and the vertex with second highest centrality will have a rank of 2. The Spearman rank correlation at time t is then computed by computing the Pearson correlation between X_{t-1} and X_t. The lengths of X_{t-1} and X_t are the same and the i^{th} element of each refers to the rank of the same vertex in the two consecutive snapshots. Values near 1 suggest that centrality properties of the graph have not changed much. Vertices what were highly central tend to remain so.

The Jaccard Index measures the overlap between two sets. Here it is used to determine how much the set of most central vertices has changed. For a given centrality measure, let Y_t be the set of vertices with values in the top 1%. The Jaccard measure of the top 1% central vertices at time t is then given by

$$\frac{|Y_{t-1} \cap Y_t|}{|Y_{t-1} \cup Y_t|} \tag{4.5}$$

While we use the Spearman correlation to compare the change in centrality among all vertices, the Jaccard Index is used to compare the change in vertices with highest centrality. In many applications, centrality is computed to find only these top vertices, which may be particularly influential. For both measures, higher values indicate more stable changes of the dynamic graph.

Figures 4.2 and 4.3 show the Spearman's rank correlation and Jaccard Index of unweighted degree centrality between consecutive snapshots. Higher scores show that *Active Vertex* produces dynamic graphs with least variability of vertex centrality over time, followed by *Weight Decay*. Note that for statistics using unweighted edges, results for *Active Edge* are the same as for *Sliding Window*. Figures 4.4 and 4.5 show the Spearman's rank correlation and Jaccard Index of weighted degree centrality between consecutive snapshots.

Again, *Active Vertex* produces dynamic graphs with the least centrality change. The results for *Weight Decay*, on the other hand, are very different. For this weighted statistic, *Weight Decay* results in dynamic graphs with the most centrality change over time. While the number of edges each vertex has is relatively stable, the edge weights change a great deal with *Weight Decay*. Finally, Figures 4.6 and 4.7 show results for betweenness centrality scores, with similar results as for the unweighted degree.

For all centrality measures, the score rank correlation coefficients and top 1% overlap are highest for *Active Vertex*. This means that *Active Vertex* reduces the variability of vertex centrality over time. It produces graphs that are less sensitive to quick changes. This can be useful for applications in which the graph should change gradually over time. On the other hand, *Sliding Window* has the lowest rank correlation scores and the lowest overlap of the top 1% of vertices. This means that *Sliding Window* produces dynamic graphs with a high variability of vertex centrality over time. High degree vertices will more easily become low degree vertices and vice versa. Because the graphs produced by *Sliding Window* may change quickly, it may be less suitable for applications that require gradual evolution. On the other hand, it may be more appropriate for creating graphs that are sensitive to recent changes.

As with the global properties, the local, vertex-level properties produced by each method increase and decrease approximately at the same time. The graphs produced by each method change in similar ways at similar points in time, even though the magnitude of change may be different.

The observation that *Active Vertex* and *Weight Decay* reduce the variability of vertex centrality compared to *Sliding Window* is even more pronounced when we consider only medium and high degree vertices instead of all vertices. Figure 4.8 provides a closer look into the rank correlation of betweenness centrality that was previously shown in Figure 4.6. Here we compare this measure when calculated for all vertices vs. when calculated only for vertices whose degree is greater than 9 in at least one of the two consecutive snapshots.

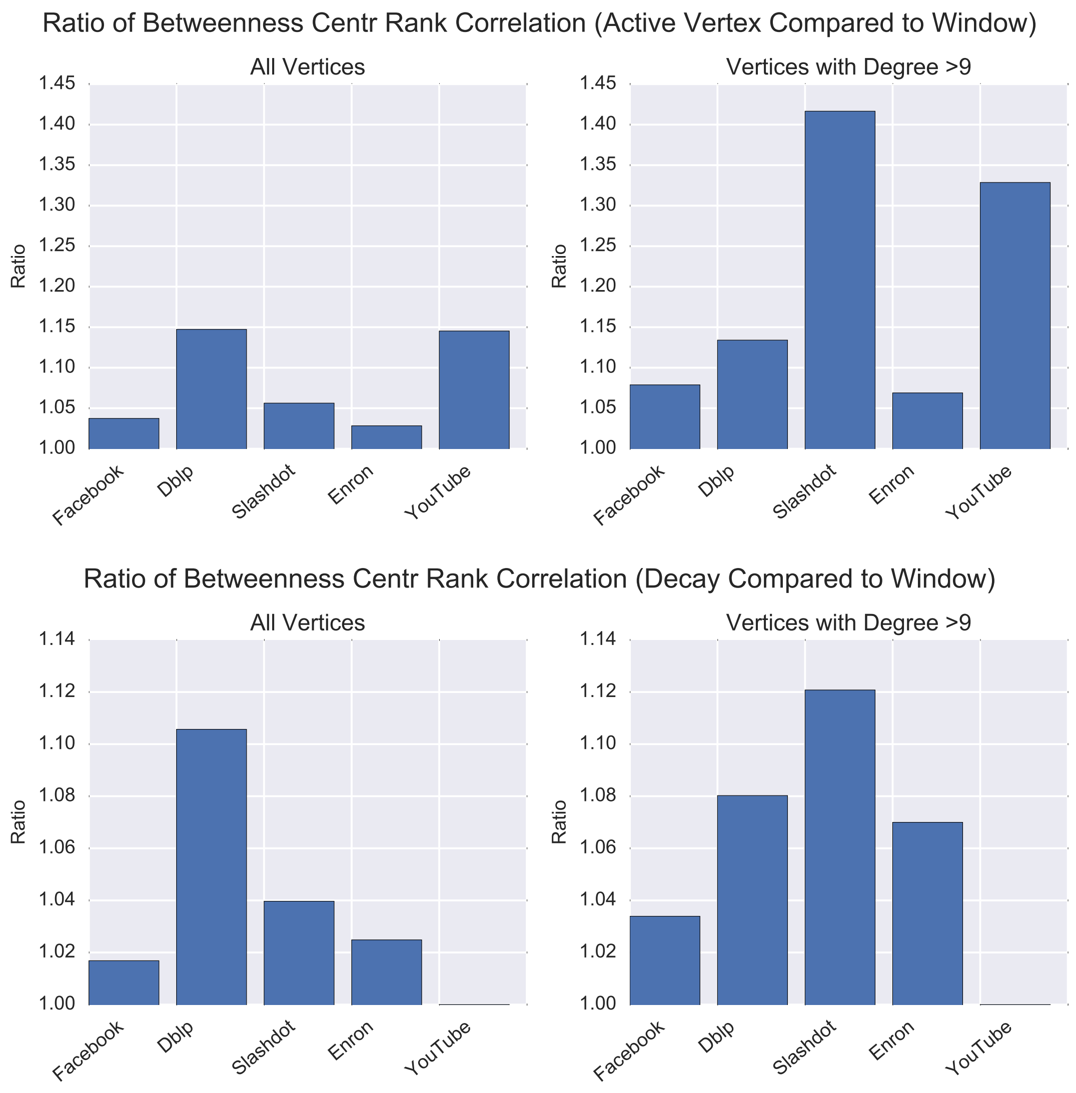


Figure 4.8: The ratio of the average betweenness centrality rank correlation of *Active Vertex* (top) and *Weight Decay* (bottom) compared to that of *Sliding Window* is shown. Plots on the left show rank correlations using all vertices and those on the right use only vertices with degree greater than 9 in either consecutive snapshot.

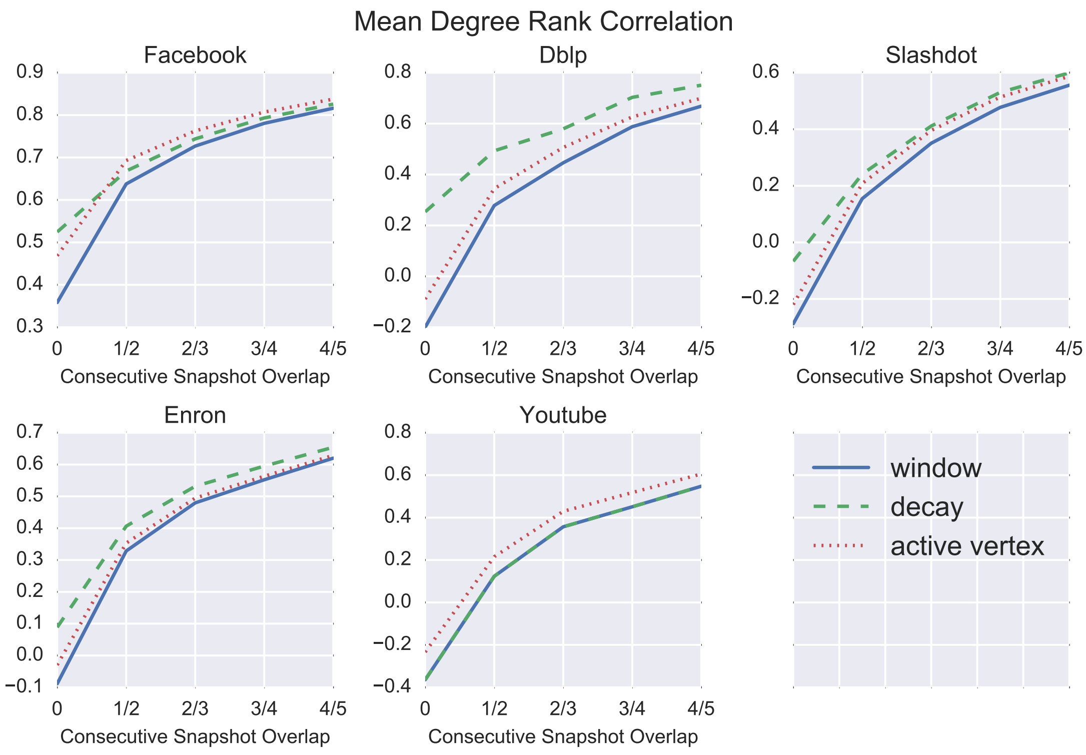


Figure 4.9: The Spearman's rank correlation of unweighted degree centrality is shown. The x-axis shows the overlap between consecutive snapshots produced by the sliding window method.

In the top plots, each bar shows the ratio of the average betweenness centrality rank correlation of *Active Vertex* compared to the average rank correlation of *Sliding Window*. Values above 1 indicate that *Active Vertex* produces higher rank correlations than *Sliding Window*. The higher the value, the greater the difference between aging methods. The bottom plots show the same ratio for *Weight Decay* compared to *Sliding Window*.

Plots on the left show rank correlations using all vertices and those on the right use only vertices with degree greater than 9 in either consecutive snapshot. It is clear that ratios are higher when the rank correlation uses only the vertices of degree greater than 9. This means that the variability of betweenness centrality of medium and high degree vertices is more affected by the choice of aging method than is the variability of low degree vertices.

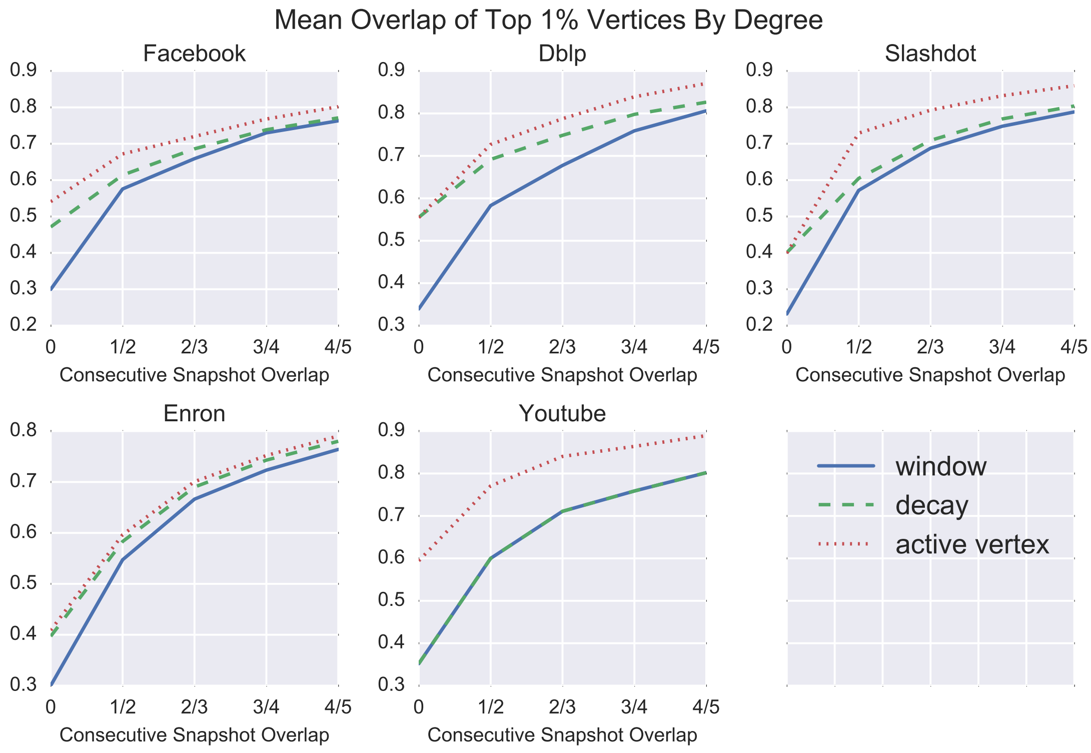


Figure 4.10: The overlap between the top 1% of vertices from consecutive snapshots based on unweighted degree centrality is shown. The x-axis shows the overlap between consecutive snapshots produced by the sliding window method.

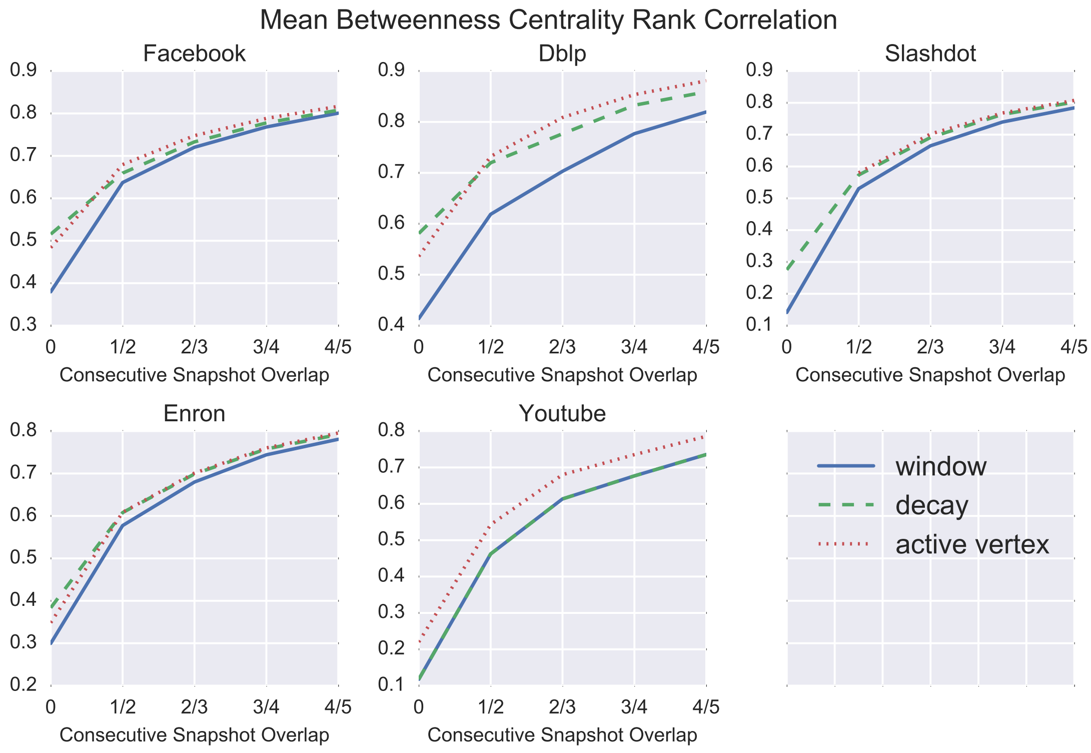


Figure 4.11: The Spearman's rank correlation of betweenness centrality is shown. The x-axis shows the overlap between consecutive snapshots produced by the sliding window method.

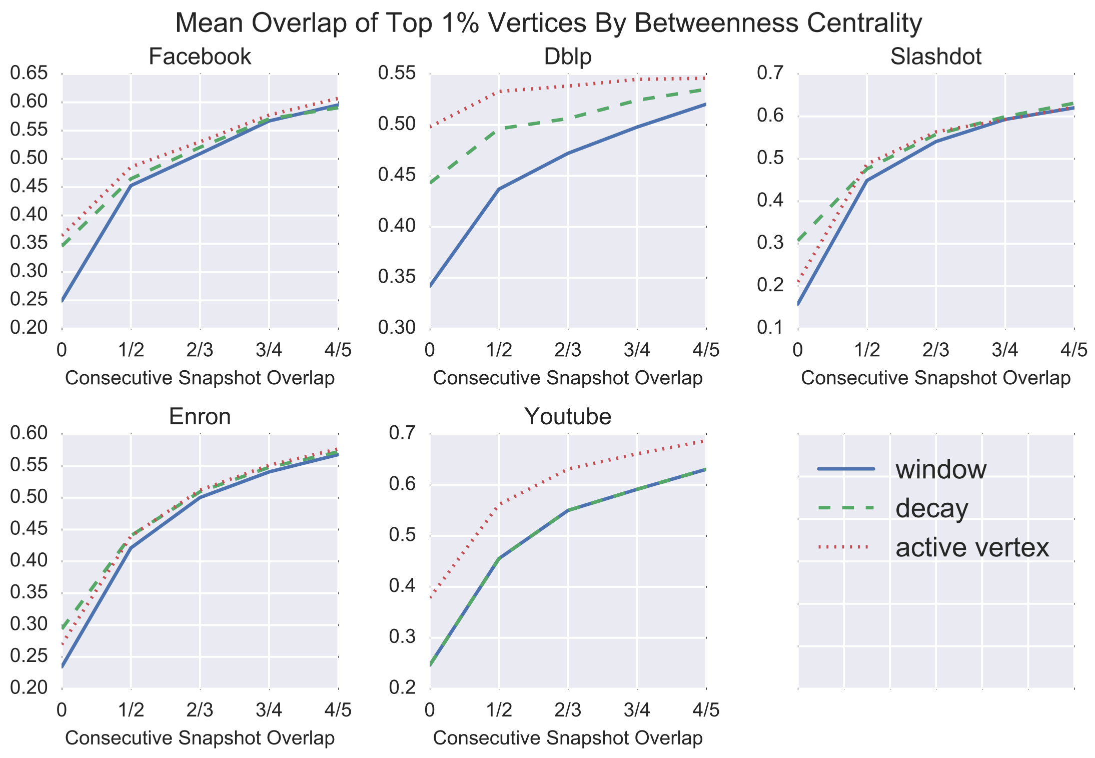


Figure 4.12: The overlap between the top 1% of vertices from consecutive snapshots based on betweenness centrality is shown. The x-axis shows the overlap between consecutive snapshots produced by the sliding window method.

4.3.4 Effect of Overlap

The experiments for Figures 4.2 – 4.8 used parameters based on consecutive snapshots of *Sliding Window* overlapping by $2/3$, as described in Section 4.3.1. This section explores how the average scores of different methods change as the amount of overlap between consecutive snapshots G_t and G_{t+1} increases. For *Sliding Window*, overlaps of 0, $1/2$, $2/3$, $3/4$, and $4/5$ between consecutive graph snapshots are tested, while keeping the range of time in each graph snapshot constant. For example, regardless of the amount of overlap, a snapshot of the YouTube graph represents edges from the last two months. Because setting $\lambda = h$ creates an overlap of $\frac{h-1}{h}$, increasing the overlap requires increasing λ. Therefore, in order to keep the range of time represented by each snapshot constant, when λ increases the unit of time must decrease (fewer new edges are added to the new snapshot).

The parameters of the other methods are set to match as described in Section 4.2.5. For *Active Vertex* and *Active Edge*, τ_V and τ_E are set equal to λ. For *Weight Decay*, $\epsilon = 0.1$ and values of $\alpha = \epsilon^{\frac{1}{\lambda}}$ are then 0.0999, 0.316, 0.464 0.562, and 0.630 for the overlap parameter values of 0, $1/2$, $2/3$, $3/4$, and $4/5$, respectively. The unit of time is the same for each method.

Figures 4.9 – 4.12 show how the average scores change as the amount of overlap between consecutive snapshots changes. For each plot, the x-axis shows the overlap between consecutive snapshots produced by *Sliding Window*. From left to right, there is less change between G_{t-1} and G_t. For each plot, the y-axis shows the average vertex centrality similarity for that amount of overlap. Figure 4.9 plots the Spearman rank correlation for unweighted degree, Figure 4.10 shows the Jaccard index of the top 1% vertices by unweighted degree, Figure 4.11 plots the Spearman rank correlation for betweenness centrality, and Figure 4.12 shows the Jaccard index of the top 1% vertices with greatest betweenness centrality.

Overall, the difference between *Weight Decay* scores and *Sliding Window* scores decreases as the amount of overlap increases. This means that when a large amount of data is added and removed between graph snapshots, *Weight Decay* reduces the variability over time of vertex centrality even more when compared to *Sliding Window*. However, the less the graph changes, the smaller the difference between the methods. The same holds to a lesser degree for *Active Vertex*.

4.4 Conclusion

This chapter addressed the question of aging data for the creation of dynamic graphs. In addition to previously used methods, a new approach was proposed, based on the concept of active vertices and edges. The new approach may allow us to better preserve in the graph information that is relevant to its most prominent or significants actors. By analyzing several global and vertex-level graph properties, we find the differences and similarities between dynamic graphs created by each aging approach.

The *Active Vertex* and *Weight Decay* methods each reduce the variability of vertex centrality scores over time, especially for medium and high degree vertices, as compared to *Sliding Window* and *Active Edge*. This difference is greater when more changes are accumulated between consecutive graph snapshots. In practice, *Active Vertex* or *Weight Decay* may be more useful if graph stability is preferred, while *Sliding Window* or *Active Edge* may be chosen if a faster reflection of changes in the underlying data is needed. The choice of method matters more when more data is added between snapshots. However, *Active Vertex* and *Weight Decay* have opposite effects on graph connectivity. *Active Vertex* decreases the number of connected components, while *Weight Decay* increases it.

This means that *Active Vertex* may be more suitable if it is desirable to avoid graph fragmentation into several mutually disconnected parts. This is consistent with our original motivation to introduce the *Active Vertex* approach, which was to avoid deleting even very old edges if they connect continually active vertices. Keeping such edges can prevent the breaking up of the graph into disjoint components and will preserve information about old associations between a networks important entities. In practical terms it may useful in the tracking, for example, of criminal groups, when even very old connections between active entities may prove very significant.

Despite these differences, if each method's parameters are carefully chosen, they all produce dynamic graphs with very similar patterns. The dynamic graphs produced by each method experience similar types of changes at approximately the same time. This is an important consideration for applications where the goal is to monitor the graph and detect change-points in time.

The effects on the graph properties studied here will help practitioners understand the consequences of choosing a particular method of removing old data. Because the topic of aging data in dynamic graphs was previously largely unexplored, the focus here was on basic graph properties. Future work should study the effect on more complex graph measures, such as community evolution. The method chosen to age data will likely have a strong effect both on the communities found and on how much they change between consecutive snapshots.

CHAPTER 5
LOCAL COMMUNITY DETECTION ON DYNAMIC GRAPHS

5.1 Introduction

Previous chapters have discussed issues that must be addressed before graph analysis takes place: namely, how to collect edges with sampling and how to build dynamic graphs that represent changes in relationships over time through aging. This chapter focuses on the analysis of graphs through local community detection. As discussed in Section 2.2, local community detection, also referred to as seed set expansion, is the task of finding a community relevant to a chosen seed vertex or set of seed vertices. The contribution of this chapter is a new algorithm for local community detection on dynamic graphs, Ordered Dynamic Seed Expansion (ODSE), which maintains a community over time by incrementally updating it as the underlying graph changes. The algorithm outputs high quality communities that are similar to those found when using a standard static algorithm. It works well both when beginning with an already existing graph and in the fully streaming case when starting with no data. The dynamic approach is also faster than re-computation when low latency updates are needed. This work was published in [109] and [110].

5.1.1 Problem Statement

The new dynamic local community detection algorithm, ODSE, incrementally updates local communities when the underlying graph changes. Since incremental updates are faster than re-computation, this method can be used to improve performance for any application of seed set expansion, such as those described in Section 2.2. We begin with an initial graph G and find an initial local community C using a static seed expansion method. Next, a sequence of updates is applied to G and after each such update, the new algorithm incrementally updates C to reflect changes in graph structure. Each graph update is of the form $(u, v, \Delta\omega)$, where u and v are edge endpoint vertices and $\Delta\omega$ is an increment or a decrement in edge weight. An edge insertion is represented by a weight increment to a non-existent edge, while a deletion is represented by a decrement of the edge weight to 0.

The goal of ODSE is to output a community that is similar to what would have been returned by re-running a static, greedy algorithm whenever the graph changes. This static algorithm is described in Algorithm 9 and is the approach used in [32, 31, 5]. The community is iteratively expanded by adding the neighboring vertex that maximizes the chosen fitness function. The algorithm terminates when there exists no vertex whose inclusion in the community increases the fitness score. In Algorithm 9, $seed$ represents the initial set of seed vertices, $f(C)$ the fitness score for a community C, and $N(C)$ the set of vertices not in C with at least one neighbor in C. For the experiments of Section 5.4 the fitness metric $f(C)_{MONC}$ from Equation 2.5 is used, although the approach will work for other appropriate fitness functions as well. This metric was chosen because unlike modularity, it is local in nature. Note that, as for any greedy method, the output of Algorithm 9 depends on the particular order in which the set $N(C)$ is traversed in the for loop, and does truly optimize the fitness metric over all possible expansions of C.

5.1.2 Sources of Edge Updates

As described above, the input to ODSE is a sequence of edge insertions, deletions, or edge weight modifications. These updates may come from a dynamic graph created through one of the aging methods described in Chapter 4. Recall that a dynamic graph is represented by a sequence of graph snapshots $\{G_0, \ldots, G_t\}$, each snapshot representing the state of the graph at that point in time. Once a dynamic graph is created, edge updates are then the differences between consective snapshots G_t and G_{t+1}. If G_{t+1} contains an edge not in G_t, an update $(u, v, \Delta\omega)$, where $\Delta\omega > 0$, is necessary. Similarly, if G_{t+1} is missing an edge contained in G_t, an update $(u, v, \Delta\omega)$, where $\Delta\omega < 0$, is formed. Typically, there will be

```
Data: graph $G$ and seed set $seed$
$C = seed$;
while progress do
    $maxscore = -1$;
    $maxvtx = -1$;
    for $v \in N(C)$ do
        $s(v) = f(C \cup v) - f(C)$;
        if $s(v) > maxscore$ then
            $maxscore = s(v)$;
            $maxvtx = v$;
        end
    end
    if $maxscore > 0$ then
        $C = C \cup maxvtx$;
    end
end
```

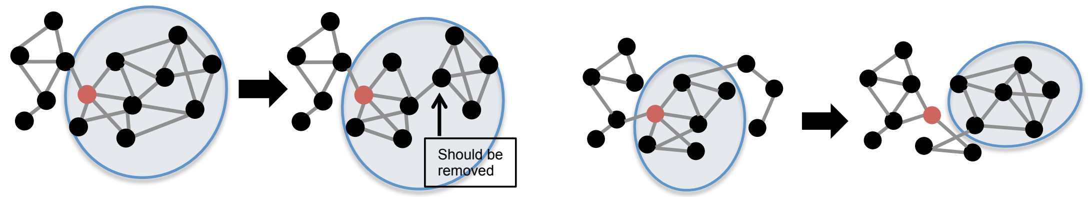
Algorithm 9: Static Seed Expansion (SSE)

multiple edge changes between consective snapshots. ODSE may update communities after every single edge update or collect all changes that occur between consecutive snapshots as a batch and take this batch as input.

If a dataset with timestamps has already been collected, the edges with timestamps can be used to create the entire dynamic graph. Then, ODSE can be applied to get local communities for all graph snapshots. In some real-life applications, the full dataset is not available at the time of computation. As new data is generated, the dynamic graph changes and new edge insertions and deletions are formed. These edge updates are then input to ODSE. Because it is faster than recomputing, ODSE is more likely to be able to keep up with a fast pace of updates.

5.2 Motivation

To motivate the new ODSE algorithm presented in this chapter, a simple, alternative algorithm is first considered and the problems it may run into are discussed. The detection of global communities in dynamic graphs has been studied extensively and many algorithms



(a) Undetected community splitting. (b) Undetected seed migration.

Figure 5.1: Shortcomings of the Simple Dynamic Seed Expansion (SDSE) algorithm. Undesired community evolution shown left to right.

have been presented in the literature. The MIEN algorithm [47] updates communities initially found with the static Louvain method [16] using a simple update procedure. All vertices incident to newly removed, inserted, or updated edges are removed from their currently assigned communities and placed in their own singleton communities. Using this new assignment as a starting point, the iterative Louvain method is restarted. A similar approach is used in the algorithm presented in [50], which updates communities found with a parallel version of CNM [15]. In [50], only vertices that are endpoints of inserted inter-community edges or deleted intra-community edges are removed from their clusters and placed into single communities. The reasoning is that inserting inter-community edges and deleting intra-community edges weakens the current community structure, and thus changes may need to occur, while deleting inter-community edges or inserting intra-community ones strengthens it. This method can naturally be extended to the local community problem. It is a simple, first-pass attempt which can function as a comparison for ODSE. We apply the method from [50] to local communities and call this method Simple Dynamic Seed Expansion (SDSE).

SDSE works as follows. If an edge is inserted or incremented on the border of the local community, the member vertex is removed. If an intra-community edge is deleted or decremented, both vertices are removed. Then, Static Seed Expansion is restarted to allow any vertices to be added to C. This last step allows the vertices that were just removed to be re-added if this increases the fitness score. This process is shown in Algorithm 10 for a single edge update. It is naturally extended to processing batches of multiple edge updates

```
Data: edge $(u, v, \Delta\omega)$
if $u \in C$ and $v \notin C$ and $\Delta\omega > 0$ and $u \neq seed$ then
    $C = C \setminus u$;
else if $u \in C$ and $v \in C$ and $\Delta\omega < 0$ then
    if $u \neq seed$ then
        $C = C \setminus u$;
    end
    if $v \neq seed$ then
        $C = C \setminus v$;
    end
while progress do
    $maxscore = -1$;
    $maxvtx = -1$;
    for $v \in N(C)$ do
        $s(v) = f(C \cup v) - f(C)$;
        if $s(v) > maxscore$ then
            $maxscore = s(v)$;
            $maxvtx = v$;
        end
    end
    if $maxscore > 0$ then
        $C = C \cup maxvtx$;
    end
end
```

Algorithm 10: Simple Dynamic Seed Expansion (SDSE)

at once. Each edge update is processed as above before Static Seed Expansion is restarted.

Unfortunately, SDSE has shortcomings. For example, the community may split apart and the algorithm may not be able to detect this because the neighbors of removed vertices remain in the community. This is illustrated in Figure 5.1a, where the previously correctly detected community (shown on the left) has actually split into two natural communities (shown on the right) because of the deletion of five intra-community edges. The desired action of the algorithm would be to remove some vertices from the community, so that the resulted updated community is well-connected. However, SDSE may fail to prune the community correctly. In the set of vertices that has split off and should be removed, most neighbors of each vertex are also in the community and therefore no vertex will be removed. Even if we evaluate multiple vertices at once for removal, the same problem may occur if the set that has split off is large enough.

SDSE may fail even when it outputs a valid community in the graph. This is because seed set expansion differs from global community detection in an important way: the local community is chosen for a particular seed set. The task is not simply to find any good community in the graph, but rather the appropriate community for the seed. Changes to the graph may shift the community C to one not centered around the original seed, as shown in Figure 5.1b. On the left hand side of the Figure, the detected community includes the seed vertex, shown in red. On the right hand side, the original seed has been removed from the community because of the deletion of one intra-community edge and insertion of one border edge. While C may still have a good fitness score, it may not be a local community of the seed and would not be produced by a complete re-computation using Static Seed Expansion.

Given these considerations, quality evaluation for an updated community of a seed is more difficult than for general communities. We must consider not only the degree to which the chosen set of vertices resembles a community, but also whether it is a good community for the particular seed. A static seed set expansion algorithm detects a community for the seed set using full information. Thus, one method of determining quality is to use the community found using Static Seed Expansion as a baseline and consider an incremental updating algorithm to be successful if it produces similar results.

5.3 Dynamic Seed Set Expansion Algorithm

5.3.1 Algorithm Overview

This section presents Ordered Dynamic Seed Expansion (ODSE), the new dynamic algorithm for seed set expansion. ODSE updates a local community C that was produced by Static Seed Expansion (SSE) (Algorithm 9). SSE iteratively adds to C the vertex that most increases the fitness score. Therefore, the member vertices of C can be arranged as a sequence, in the order in which they were added, and this induces a sequence of increasing fitness scores. The new algorithm ODSE works by treating C as this sequence of members. When the graph changes, the sequence is pruned as needed so that the resulting fitness score sequence remains monotonically increasing. This approach will not suffer from the pitfalls shown in Figures 5.1a and 5.1b. In the case of Figure 5.1a, the set of four vertices that are barely connected to the natural community would not be included. The first of those four vertices to appear in the vertex sequence would cause a drop in fitness score. In the case of Figure 5.1b, none of the vertices connect to the seed, so their inclusion would also cause a drop in fitness score.

Let m_i denote the i^{th} vertex added as a member of the community in Algorithm 9 and $M_i = \{m_j \mid j \leq i\}$. M_i thus is a set of the first i vertices added to community C and it has an interior edge weight sum of $k_{in\,i}$, a border edge weight sum of $k_{out\,i}$, and a fitness score of s_i. Note that $k_{in\,i}$ is equal to $k_{in}^{M_i}$ and $k_{out\,i}$ is equal to $k_{out}^{M_i}$ as in Equations 2.1 and 2.2. If the i^{th} vertex of the community m_i is vertex v, then we say that v has position i or $\rho(v) = i$ in the community. The entire sequence of members is refered to as (m). The corresponding sequences of interior edge weights, border edge weights, and fitness scores are refered to by (k_{out}), (k_{in}), and (s). Here end is used to represent the last position in the sequences,

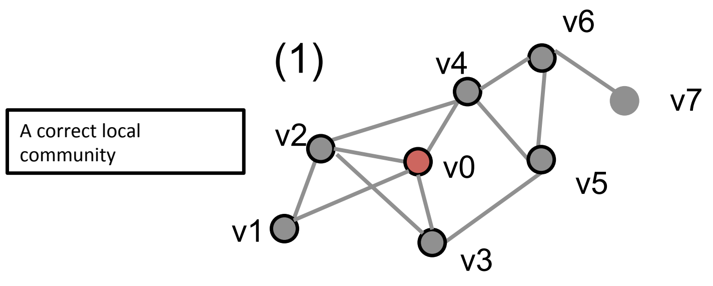
Table 5.1: Community sequences (m),(k_{in}),(k_{out}), and (s)

position	0	1	2	...	n
members	m_0	m_1	m_2	...	m_{end}
inner edges	k_{in0}	k_{in1}	k_{in2}	...	k_{inend}
border edges	k_{out0}	k_{out1}	k_{out2}	...	k_{outend}
fitness score	s_0	s_1	s_2	...	s_{end}

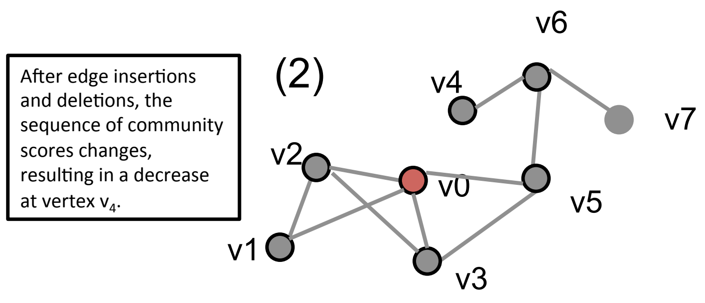
so that $(m) = \{m_0, \ldots m_{end}\}$ and M_{end} is the current community, which we also call C. Table 5.1 shows the sequences (m),(k_{in}),(k_{out}), and (s).

The dynamic algorithm works as follows. In phase A, we start with the initial graph and find an initial community with Static Seed Expansion (Algorithm 9) to produce the sequences (m),(k_{in}),(k_{out}), and (s). In phase B, a stream of graph updates is applied. With each graph update, ODSE updates the community by modifying (m), (k_{in}),(k_{out}), and (s) in order to make sure that (s) remains monotonically increasing. That is, we require the updated community to contain vertices that, if added one by one as in the static algorithm, result in an increasing sequence (s). This helps the resulting community to remain relevant to the source seed. ODSE detects any decreases in (s) and removes vertices from (m) to eliminate any such decrease. Note that a change in (m) will cause a change in (k_{in}) and (k_{out}), which will in turn modify (s). Removing a vertex from (m) (and thus from C), makes some previously internal edges into border edges, so that (k_{in}) and (k_{out}) must be modified. The fitness scores then change so (s) is updated. After ODSE removes vertices from the community, it may add new vertices using the Static Seed Expansion method of Algorithm 9.

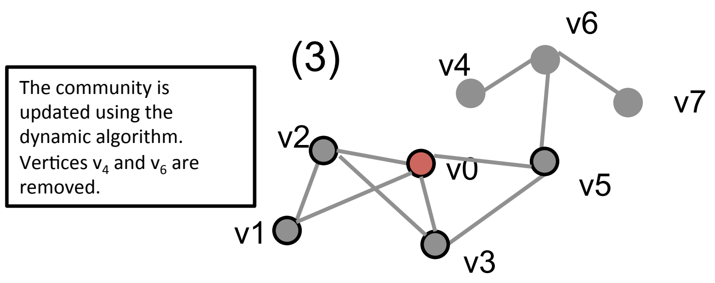
This process is shown in Figure 5.2. The seed vertex v_0 is in red and all members of the community have a black border. The top image shows a community centered around the seed vertex. The order of the community members is shown along with the corresponding scores. The second image shows the state of the community after edges have been inserted into and removed from the graph. The members of the community still remain the same,



Position	0	1	2	3	4	5	6
Member	v_0	v_1	v_2	v_3	v_4	v_5	v_6
Score	0.25	0.5	0.7	0.85	0.88	0.95	1


(2)
After edge insertions and deletions, the sequence of community scores changes, resulting in a decrease at vertex v4.
v6
v4
v7
v2
v0
v5
v1
v3

Position	0	1	2	3	4	5	6
Member	v_0	v_1	v_2	v_3	v_4	v_5	v_6
Score	0.25	0.5	0.78	0.92	0.85	0.94	0.88


(3)
The community is updated using the dynamic algorithm. Vertices v4 and v6 are removed.
v6
v4
v7
v2
v0
v5
v1
v3

Position	0	1	2	3	4
Member	v_0	v_1	v_2	v_3	v_5
Score	0.25	0.5	0.78	0.92	1

Figure 5.2: The process of storing and maintaining the sequences (m),(k_{in}),(k_{out}), and (s) for a community is shown. The seed vertex v_0 is in red and all members of the community have a black border.

but the number of internal and border edges has changed. As a result, the sequence of scores is no longer increasing and the community is inappropriate. In the bottom image, ODSE has been applied to update the community. Vertices v_4 and v_6 are removed and the sequence of scores is once again increasing.

5.3.2 Algorithm Details

For each batch of edge updates, the following four steps are performed. Some may be omitted depending on particular properties of edges being updated. Further details are given later.

1. The sequences (k_{in}),(k_{out}), and (s) are updated to reflect new internal and border edges.

2. Vertices that are endpoints of an updated edge are checked for removal. If a vertex z is removed, the community members are further pruned. When members are removed from C, (m), (k_{in}),(k_{out}), and (s) are updated.

3. The sequence (s) is scanned to check that the fitness scores are still monotonically increasing. If a dip exists at position i, all sequences are truncated after position i. We set end=$i-1$ so that $(m) = \{m_0, \ldots, m_{i-1}\}, (k_{in}) = \{k_{in0}, \ldots, k_{ini-1}\}, (k_{out}) = \{k_{out0}, \ldots, k_{outi-1}\}, (s) = \{s_0, \ldots, s_{i-1}\}$.

4. Algorithm 9 is restarted to check whether neighboring vertices in $N(C)$ should be added to the community.

The four steps of ODSE are given in Algorithm 11. To make subsequent discussion easier to interpret, three simplifying assumptions are made. First, as edges are undirected, the order of vertices in an edge update $(u, v, \Delta\omega)$ is arbitrary. Therefore, we only consider $\rho(u) < \rho(v)$. Second, we assume one seed vertex, referred to as $seed$. The algorithm can handle any number of seed vertices, though this only makes sense if there is prior

```
**Data**: edge $(u, v, \Delta\omega)$
**//Step 1**
**if** $u \in C$ *and* $v \in C$ **then**
    **for** $p = \rho(u)$ *to* $\rho(v) - 1$ **do**
        $k_{outp} += \Delta\omega$;
        update $s_p$;
    **end**
    **for** $p = \rho(v)$ *to* $end$ **do**
        $k_{inp} += \Delta\omega$;
        update $s_p$;
    **end**
**else if** $u \in C$ *and* $v \notin C$ **then**
    **for** $p = \rho(u)$ *to* $end$ **do**
        $k_{outp} += \Delta\omega$;
        update $s_p$;
    **end**
**//Step 2**
**if** $\Delta\omega < 0$ *and* $u \in C$ *and* $v \in C$ **then**
    Queue $\leftarrow v$
**else if** $\Delta\omega > 0$ *and* $u \in C$ *and* $v \notin C$ *and* $u \neq seed$ **then**
    Queue $\leftarrow u$
**else if** $\Delta\omega > 0$ *and* $u \in C$ *and* $v \in C$ *and* $u \neq seed$ **then**
    Queue $\leftarrow u$
**while** *Queue not empty* **do**
    $x \leftarrow$ Queue;
    **if** $x \in C$ *and* $s_{\rho(x)-1} \geq s_{\rho(x)}$ **then**
        remove $x$ from $C$ and update $(m)$,$(k_{in})$,$(k_{out})$, and $(s)$;
        **for** *neighbors* $z$ *of* $x$ **do**
            **if** $z \in C$ *and* $\rho(z) > \rho(x)$ **then**
                Queue $\leftarrow z$;
            **end**
        **end**
    **end**
**end**
**//Step 3**
**for** $i = max(\rho(u), 1)$ *to* $end$ **do**
    **if** $s_{i-1} \geq s_i$ **then**
        $end \leftarrow i - 1$;
        $C = M_{i-1}$;
        break;
    **end**
**end**
**//Step 4**
Check for new members using Static Seed Expansion process;
```

Algorithm 11: Ordered Dynamic Seed Expansion (ODSE)

knowledge that all those vertices will belong to the same community. Third, we only consider a single edge update, even though the algorithm can handle batches of more than one update. To process a batch, several edge updates are accumulated before updating the graph and the community. Step 1 is first performed for each update in the batch, then step 2 is performed for each update, and finally steps 3 and 4 are only executed once per batch. Because steps 3 and 4 of the algorithm can be performed once per batch, accumulating a larger set of updates before processing results in a faster running time. On the other hand, if larger batches are used, the community is not updated as frequently. Using a larger batch size can be thought of as a compromise between a fully dynamic algorithm, which updates results immediately, and infrequently using the static algorithm to recompute communities.

The complexity of obtaining one community with Static Seed Expansion is $\mathcal{O}(n^2d)$, where n is the final community size and d is the average vertex degree, though this is an overestimate for graphs whose vertices share many neighbors. In each iteration, in order to determine the best candidate for addition into the community, all vertices neighboring the current set (the border set) are checked and the corresponding change in fitness score is computed. With n current member vertices of average degree d, there may be nd distinct border set vertices to check, resulting in a time complexity of $\mathcal{O}(nd)$. In reality, however, many member vertices will have the same neighbors, so not all nd are distinct. By maintaining a list of current border set vertices, such as with a hash map, it is only necessary to process unique neighbors in each iteration. Thus, in practice, the time complexity of checking neighboring vertices in each iteration may be less than $\mathcal{O}(nd)$. Assuming the community and border set are each represented with a hash map, adding a vertex v has worst case $\mathcal{O}(d)$ complexity because each of v's neighbors may need to be either added to the border set or have their count of edges touching the community updated. To obtain a final community of size n, n iterations must be completed.

In the worst case, ODSE must recompute a large portion of the community. Because this re-computation is performed with the static expansion, the worst case time complexity is $\mathcal{O}(n^2 d)$ as well. In practice, many of the updates result in no decrease in the fitness score sequence so that only a scan of sequences (m),(k_{in}),(k_{out}), and (s) is needed. In this case the complexity becomes $\mathcal{O}(n)$. The time complexity of each step is given next. Step 1 updates the values of (k_{in}), (k_{out}), and (s) by iterating once over each. The complexity is $\mathcal{O}(n)$ where the n is the size of the community. In step 2, if no vertices are removed, the complexity is $\mathcal{O}(1)$. The complexity of removing a vertex z with degree d in step 2 is $\mathcal{O}(d)$ because each neighbor of z must either also be checked for removal if it is a current community member or else have its count of edges touching the community decreased. With a community size of n, at most n vertices can be removed in step 2, for a worst case $\mathcal{O}(nd)$ complexity. Step 3 requires a scan of (s) and takes $\mathcal{O}(n)$ time. Step 4 uses Algorithm 9 and therefore has a worst case complexity of $\mathcal{O}(n^2 d)$. The data structures required are a representation of the community and of the set of border vertices, both of which may be, for example, a hash map. The sequences (m),(k_{in}),(k_{out}), and (s) are each an array with length n. Additional details of each step are given next.

Step 1: First, (k_{in}), (k_{out}), and (s) are updated to reflect new edges internal to and on the border of the community. The input is $(u, v, \Delta\omega)$, where u and v are vertices and $\Delta\omega$ the corresponding change in weight.

Step 2: Due to the update in step 1, (s) may no longer be monotonically increasing. Step 2 is only performed if there is a specific candidate vertex for removal, which will always be an endpoint of an updated edge. An edge update $(u, v, \Delta\omega)$, can cause the removal of one of its endpoints z in three cases. Recall the simplifying assumption that $\rho(u) < \rho(v)$. The first case is a an edge decrement with $u \in C$, $v \in C$; then the edge vertex to be examined is $z = v$. The second is an edge increment with $u \in C$ and $v \notin C$; then the vertex to be examined becomes $z = u$. The third case is an edge increment with in $u \in C$ and $v \in C$. Then the vertex to be examined will be $z = u$. For each case, if $s_{\rho(z)} \leq s_{\rho(z)-1}$, z will be removed.

This third case may seem counter-intuitive because an intra-community edge is incremented, densifying the community. However, we must maintain an increasing sequence of fitness scores (s). As u was added to C before v, the edge between u and v is a border edge at position $\rho(u)$, and becomes internal only starting at position $\rho(v)$. Thus, by incrementing it, the sum of border edges $k_{out \rho(u)}$ increases. If, due to this increase, $s_{\rho(u)} \leq s_{\rho(u)-1}$, then u would have to be removed from C. In a later step, u could be re-added to C, but it would have to be removed from its current position due to causing a non-increase of (s).

If a candidate z is removed, (m) is updated and values of (k_{in}), (k_{out}), and (s) for $\rho(z) \leq i \leq end$ must be recalculated to reflect the fact that edges of z are no longer inside C. For each edge (z, x), if $x \in C$, the edge changes from an internal community edge (contributing to (k_{in})), to a border edge (contributing to (k_{out})). If $x \notin C$, the edge changes from a border edge to an edge with no influence on the fitness score. Only sequence elements after position $\rho(z)$ must be updated because earlier elements reflect a state in which z was not yet in the community. The positions of all member vertices in the sequences must be updated to reflect the fact that an element has been removed.

The removal of a vertex z from C in step 2 may cause other vertices in C to be removed as well. Candidate vertices are neighbors of z that were added to C after z. Let x be such a neighbor. At the time of x's inclusion in C, adding x increased the fitness score by increasing (k_{in}), which was due to x having neighbors already in the community. However, at least one such neighbor was z, which is now no longer in C. Thus, it is possible that without z in C, x would not have enough neighbors in C to be added. We can check this by testing if $s_{\rho(x)-1} \geq s_{\rho(x)}$. Therefore, if z is removed from C, all such neighbors x of z in C are also checked. If any neighbor x of z is removed, then we must in turn check neighbors of x that were added to C after x. In order to perform the entire pruning process, a selective breadth first search beginning from z is performed, as in step 2 of Algorithm 11. Note that neighbors of z added to C before z ($\rho(x) < \rho(z)$) need not be checked because they were added to C without the assistance of z. We know that their addition improved the community fitness score even without accounting for z so their corresponding fitness scores are guaranteed to be increasing also after the removal of z.

Step 3: Next we scan all of (s) to check if values are still monotonically increasing. If $s_{i-1} \geq s_i$, we truncate (m),(k_{in}),(k_{out}), and (s) at position $i-1$ and set $end = i-1$. The community is now $C = M_{i-1}$ with fitness score s_{i-1}.

Step 3 differs from step 2 because instead of a selective pruning, all of (m) after the chosen position is deleted. Step 2 is performed only when there is a specific candidate vertex to check for removal from C. Step 3 can check all vertices and is necessary because after step 2, (s) may still not be monotonically increasing. For example, let the update be $(u, v, \Delta\omega)$, with $u \in C$, $v \in C$, $\rho(u) < \rho(v)$, and $\Delta\omega > 0$. By incrementing an intra-community edge, $k_{\rho(u),in}$ increases and the set $M_{\rho(v)}$ becomes denser. Thus, any vertex added after position $\rho(v)$ may no longer increase the fitness score and all scores after position $\rho(v)$ must be scanned for a dip.

Although step 3 could replace the previous step entirely, it is still beneficial to perform step 2 because it can reduce computation. In step 2, we inspect for removal only endpoints of updated edges and then iteratively neighbors of all vertices removed in the process. These are the community members most likely needing to be removed. This is why running step 2 is advantageous before executing step 3. After step 2 has been run, the likelihood of finding non-monotonically increasing members in C is lower, and step 3 will need to prune the community less often, reducing computation.

Step 4: Finally, new vertices can be added to the community. Vertices neighboring C are checked for inclusion by running the loop in Algorithm 9. For every vertex added to C, a new entry is appended to (m),(k_{in}),(k_{out}), and (s).

5.3.3 Fully Streaming Version

ODSE, as described above, begins with an initial existing graph and an initial community for this graph. However, the method can be extended to work on a fully streaming graph. Instead of starting with an initial graph and running Static Seed Expansion, it is possible to

Table 5.2: Datasets used as test graphs with the number of edges and vertices and the size of the sliding window.

Graph	**Vertices**	**Edges**	**Sliding Window**
facebook	46,952	876,993	292,331
slashdot	51,083	140,778	46,926
contact	274	28,244	9,414
digg	30,398	87,627	29,209
ucirvine	1,899	59,835	11,240
manufacturing	167	82,927	9,214

begin with an empty graph. ODSE can then build and maintain a community from scratch. Given a seed vertex v (or set of seed vertices), the community will be initialized containing only v (or the set of seeds), with no interior or border edges. After each edge insertion or deletion (or batch of such updates), ODSE will update the community as explained above. The community will begin to grow as edges are inserted around v. We show results for ODSE both when beginning with an initial graph and for a fully streaming graph in Section 5.4.

5.4 Results

5.4.1 Experimental Setup

We test Ordered Dynamic Seed Expansion on six social network graphs, listed in Table 5.2. All datasets were obtained from the Koblenz Network Collection [59]. The *facebook* dataset represents wall posts between users. In the *slashdot* and *digg* datasets, an edge occurs when one user replies to another in a thread. The *contact* dataset represents contact between users carrying wireless devices. In the *ucirvine* dataset, vertices are users and forums, with an edge occurring when a user posts on a forum. Finally, *manufacturing* represents email contact. As these graphs represent social interactions, they are likely to display group structure. These graphs were chosen because they contain timestamped data, allowing us to track real community evolution. We can insert and remove edges in the order given by timestamps.

We perform two types of experiments with each of these real social networks. In the first type, an initial graph is formed out of the first one third of edges. Static Seed Expansion is run on this initial graph as in Algorithm 9. The remaining two thirds of edges are streamed in as edge insertions or edge weight increments. Edge deletions and edge weight decrements are created by removing old edges with a sliding window approach. Edges are inserted and removed in the same timestamped order, but with edge deletions lagging by a gap. This gap is the window size, which is given in Table 5.2. The update stream ends when no new edges can be inserted. The removal of old interactions as new interactions are added allows communities to evolve. For all graphs except *manufacturing*, the sliding window gap is set to one third of the edges. Because the the *manufacturing* graph is very dense, we set both the initial number of edges and the sliding window gap to one ninth, instead of one third, of the edge count.

The second type of experiment performed on each graph ODSE in a fully streaming manner. Instead of beginning with an existing initial graph, we begin with an empty graph and process all edge insertions and deletions as a stream. Because there is no initial community for the dynamic algorithm to begin with, this approach is more challenging. Any community must be incrementally built.

For seed vertices, we chose from each of the *facebook*, *slashdot*, *digg*, and *ucirvine* graphs 100 random vertices whose degree was in the top 75^{th} percentile for the given graph and 100 random vertices whose degree was in the top 99^{th} percentile. Both medium and high degree vertices were chosen to allow variety in the experiments. We did not choose low degree vertices because the graphs tended to have skewed degree distributions so vertices with low degree percentiles appeared only a few times in the dataset. For the *contact* and *manufacturing* graphs, because the total number of vertices was small, we simply chose 100 random vertices as seeds. We use the fitness function f_{MONC}, defined in Section 2.2, with parameter $\alpha = 1.0$ and $\alpha = 0.8$. A smaller α allows for larger communities. A value of $\alpha = 1.0$ is recommended [31, 30] and we used $\alpha = 0.8$ to also obtain slightly large communities. Two α parameters were chosen to evaluate results for different types of communities. Our results consist of the two experiment types for all seed vertices of the six datasets with both α parameters. The code was implemented in C and run on an 8 core Intel i7-2600K CPU at 3.40GHz.

5.4.2 Quality of Communities

In order to compare the communities output by ODSE to those from Static Seed Expansion, we repeatedly re-run SSE as a graph is updated. This means that at any point in time (after each number of graph updates) for each seed vertex of each graph, we have the community computed with ODSE and the community computed by SSE. The community obtained by SSE can serve as a reference for comparison purposes. Of course, in many real-life datasets there may be more than one good local community for a seed vertex, but in the absence of real ground truth, using the results of the static algorithm is suitable.

The algorithm performance is measured by four metrics. The first is the ratio of the fitness scores in the dynamic algorithm vs. those obtained by re-computation. The second is the ratio of the size of the community output by the two methods. Because ODSE maintains vertices that induce increasing fitness scores, the output will be relevant to the seed. Therefore even if the vertex members of the two sets differ, as long as the scores and sizes are similar, we can say that the communities are comparable in quality. Communities in real-life graphs are often overlapping, so there may be multiple sets for an algorithm to return. The third and fourth metrics used are the precision and recall, which compare the overlap between the members of communities output by the dynamic algorithm and those output by SSE. For a given graph update, let C_U be the community produced by ODSE and C_R be the community output by SSE. Then Equations 5.1 and 5.2 give precision and recall.

$$precision = \frac{|C_U \cap C_R|}{|C_U|} \tag{5.1}$$

$$recall = \frac{|C_U \cap C_R|}{|C_R|} \tag{5.2}$$

Table 5.3 shows the mean score ratio, size ratio, precision, and recall for each graph. In both tables, the top section shows results when the first third edges are used to form an initial graph before updating with a dynamic algorithm. The bottom section shows results when starting with an empty graph. A batch size of 1 is used, which means that communities are updated after each edge.

Table 5.3a shows results for ODSE. While both the fitness score and community size tend to be higher for the dynamic algorithm, the values are near 1 for most graphs, showing similar quality. Recall is higher than precision, which makes sense given that community sizes of the dynamic method are larger. The fact that average recall is high, with most values at or above 0.9, means that all relevant vertices are returned, which may be important for many applications. At the same time, the size of the community is not on average much larger, so not many additional vertices are returned. While precision is not as high as recall, the average is above 0.8 for half the graphs and above 0.7 for most graphs. As mentioned before, the lack of perfect overlap does not signify poor quality because a different equally good community may be returned. The fact that the fitness function score is high with community size similar to re-computation indicates good results.

The results of ODSE are slightly closer to those of SSE when beginning with an initial graph, seen in the top half of Table 5.3a, compared to the fully streaming case, seen in the bottom half. This result intuitively makes sense as the dynamic algorithm does not start with a pre-computed community in the latter case. However, the results are still good in the fully streaming case. The recall and precision remain fairly high, the score ratio is close to 1 for all graphs, and the size ratio is below 2 for all but two graphs. This shows that ODSE still works well when applied in a fully streaming manner.

Table 5.3b shows the results of SDSE described in Section 5.2. It is clear that this simple approach grows very large communities with low precision. However, the problem

Table 5.3: The average score ratio, size ratio, precision, and recall for each graph with a batch size of 1. Recomputing with SSE is used as the baseline. The bottom section of each table shows results for the fully streaming case.

(a) Results for Ordered Dynamic Seed Expansion (ODSE)

Ordered Dynamic Seed Expansion					
Type	**Graph**	**Score Ratio**	**Size Ratio**	**Precision**	**Recall**
Starting With Initial Graph	*facebook*	1.07	1.56	0.72	0.86
	slashdot	1.02	1.17	0.84	0.90
	contact	1.09	1.23	0.98	0.99
	digg	1.06	1.33	0.77	0.92
	ucirvine	1.14	2.13	0.67	0.81
	manufacturing	1.23	2.22	0.88	0.95
Fully Streaming	*facebook*	1.08	1.83	0.63	0.84
	slashdot	1.02	1.20	0.81	0.90
	contact	1.12	1.30	0.98	0.99
	digg	1.07	1.41	0.75	0.92
	ucirvine	1.27	3.94	0.59	0.80
	manufacturing	1.29	2.75	0.85	0.94

(b) Results for Simple Dynamic Seed Expansion (SDSE) from Section 5.2

Simple Dynamic Seed Expansion					
Type	**Graph**	**Score Ratio**	**Size Ratio**	**Precision**	**Recall**
Starting With Initial Graph	*facebook*	1.69	29.95	0.28	0.78
	slashdot	1.44	15.82	0.34	0.86
	contact	2.20	6.00	0.94	0.99
	digg	1.57	33.00	0.21	0.84
	ucirvine	2.63	33.53	0.23	0.86
	manufacturing	3.49	22.06	0.51	0.98
Fully Streaming	*facebook*	2.44	196.21	0.17	0.78
	slashdot	1.95	173.86	0.25	0.85
	contact	2.61	7.98	0.90	0.99
	digg	2.36	212.98	0.12	0.84
	ucirvine	3.16	56.60	0.12	0.89
	manufacturing	3.53	23.30	0.48	0.94

Table 5.4: The average score ratio, size ratio, precision, and recall across all graphs binned by community size. A batch size of 1 is used. Recomputing with SSE is used as the baseline. The bottom section of each table shows results for the fully streaming case.

(a) Results for Ordered Dynamic Seed Expansion (ODSE)

Ordered Dynamic Seed Expansion					
Type	**Size Range**	**Score Ratio**	**Size Ratio**	**Precision**	**Recall**
Starting With Initial Graph	1-4	1.00	1.02	0.98	0.99
	5-9	1.04	1.27	0.84	0.94
	10-24	1.05	1.34	0.78	0.89
	15-49	1.07	1.44	0.71	0.83
	50-99	1.14	1.98	0.73	0.84
	100-499	1.41	3.34	0.76	0.91
	500+	1.99	11.35	0.52	0.92
Fully Streaming	1-4	1.00	1.02	0.98	0.99
	5-9	1.04	1.31	0.84	0.95
	10-24	1.05	1.44	0.74	0.88
	15-49	1.08	1.64	0.67	0.83
	50-99	1.16	2.17	0.67	0.82
	100-499	1.58	5.23	0.69	0.90
	500+	2.77	25.87	0.32	0.91

(b) Results for Simple Dynamic Seed Expansion (SDSE) from Section 5.2

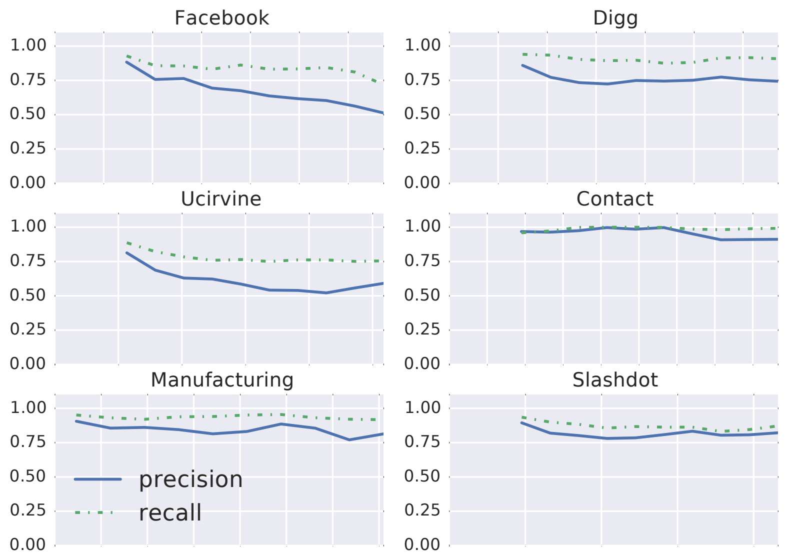
Simple Dynamic Seed Expansion					
Type	**Size Range**	**Score Ratio**	**Size Ratio**	**Precision**	**Recall**
Starting With Initial Graph	1-4	1.02	1.04	0.99	1.00
	5-9	1.12	2.01	0.65	0.93
	10-24	1.20	3.05	0.49	0.88
	15-49	1.22	4.67	0.35	0.82
	50-99	1.32	7.65	0.27	0.79
	100-499	2.85	23.46	0.30	0.87
	500+	3.64	61.95	0.10	0.90
Fully Streaming	1-4	1.03	1.07	0.98	1.00
	5-9	1.22	2.22	0.61	0.92
	10-24	1.22	3.49	0.46	0.90
	15-49	1.24	5.06	0.31	0.81
	50-99	1.33	9.63	0.22	0.79
	100-499	2.87	24.25	0.27	0.87
	500+	3.76	85.00	0.06	0.92

is not restricted to the large size of the clusters. For half the graphs, the recall of SDSE is lower than that of ODSE. Therefore, the communities also contain fewer of the ground truth members compared to ODSE. When run with a small batch size, ODSE performs better. With a large enough batch size, we expect that the quality of SDSE would improve because most vertices would be removed from the community at each update. However, that would be very similar to re-computing from scratch with SSE.

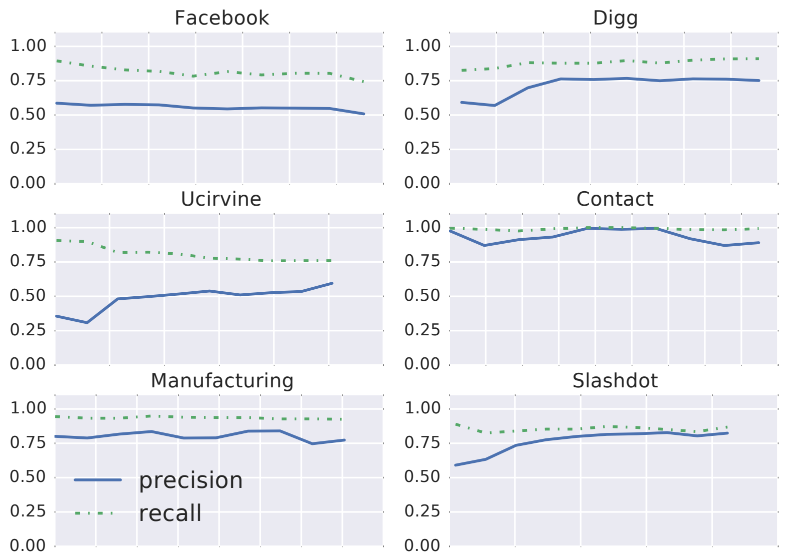
Table 5.4 shows the average score ratio, size ratio, precision, and recall of both dynamic approaches (again with the results of SSE as a baseline) for different community sizes. Averages are taken across communities output for all seed vertices on all six graphs. The runs for each seed vertex on each graph are divided by the average size of the community output by the dynamic algorithm. If the size of a community for a seed vertex is on average 20, the values for that experiment will contribute to the $10-24$ size bin in Table 5.4. Again, results are shown separately for the case when we start with an initial graph and the fully streaming case where we begin with an empty graph.

The results in Table 5.4a show that when ODSE outputs smaller communities, the results are more similar to the output of SSE. However, the quality of the results remains high up to a size of 500 so ODSE performs well on a wide range of sizes. Table 5.4b shows these statistics for SDSE. Even for the same community sizes, the precision of SDSE is much worse than that of ODSE.

Figure 5.3a shows the average precision and recall of ODSE, compared to SSE, against the number of updates applied to the graph. We begin with the first third of the edges as an initial graph before streaming updates. A batch size of 1 is used. Averages are taken across multiple independent expansions, each with its own seed. The x-axis represents the number of insertions and deletions applied to the graph and the y-axis shows the average precision or recall of communities output by the dynamic algorithm after that many updates. Thus, left to right shows how average precision and recall change as the graph changes. For each graph, all edges are inserted and edges are removed with a sliding window. The number of



(a) Starting with one third of the dataset as the initial graph before streaming updates.



(b) Fully streaming version: starting with an empty graph and streaming the entire graph as updates.

Figure 5.3: The precision and recall of ODSE, using results of re-computation with SSE as a baseline, are shown over time for all graphs.

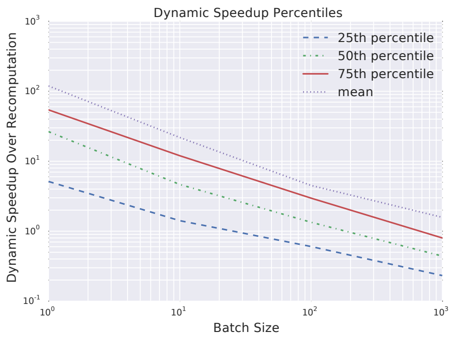
edge insertions and size of the deletion sliding window can be found in Table 5.2. Apart from *facebook*, there is no downward trend in either precision or recall of our approach as the number of insertions and deletions increases. This shows that we can use ODSE for a large number of updates before a static recalculation should be applied. In fact, for these datasets, there is no indication that a static re-computation would be necessary.

Figure 5.3b shows the precision and recall of ODSE for the fully streaming case when we begin with an empty graph. For several of the graphs, the precision and recall are lower in the fully streaming case, which makes sense given that ODSE begins with no initial community. However, the values are still high in most cases with no downward trend. In fact, for the *digg*, *ucirvine*, and *slashdot* graphs, the precision increases at the beginning.

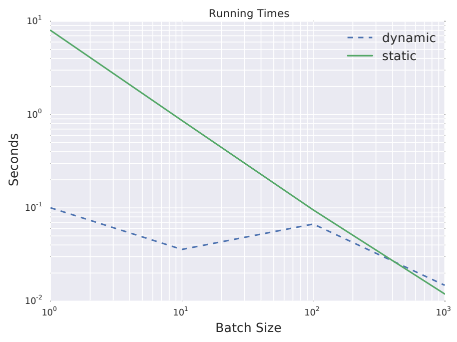
5.4.3 Performance Results

In this section we evaluate the performance of using various batch sizes of updates with ODSE. Using a batch size of x means that x edge updates are accumulated before applying them to the graph and updating the community. A smaller batch size provides updated results more frequently. When working with Static Seed Expansion, in order to produce updated results for each batch, the algorithm must be re-run. For example, if there are 2000 edge insertions and deletions, then using a batch size of 1 would require 2000 batches to be processed, while using a batch size of 10 would require processing 200 batches. With a batch size of 10 and 200 batches, our algorithm, or the static algorithm, would be run 200 times, each time processing 10 updates.

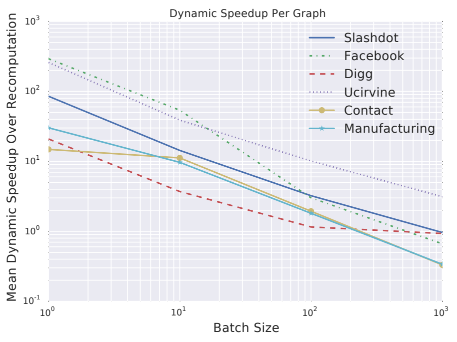
In Figure 5.4, we compare the running times of ODSE and SSE. For each seed set of each graph, the running time is measured as the total time taken to process all edge insertions and deletions. This is not the time of processing a single batch, but the time to process all batches. To fairly compute the running time of the static algorithm, we only recompute with SSE when an edge update occurs that may affect the community result. Many edge updates will affect vertices not related to the community and we need not update



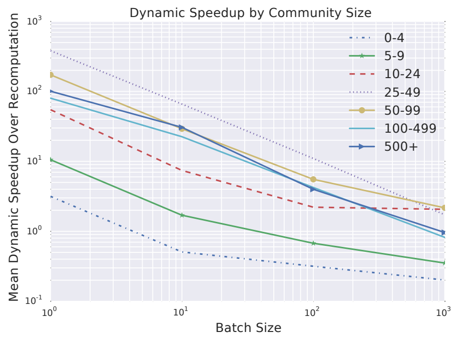
(a) The speedup mean, median, and quartiles are plotted for all runs on all graphs.



(b) The mean running time over all seed sets is shown for both ODSE and SSE.



(c) The mean speedup is given for each of the six graphs.



(d) The mean speedup is shown for different community sizes.

Figure 5.4: Subfigures a, c, and d show the speedup of ODSE compared to re-computing with SSE. A speedup of x means that using ODSE is x times faster than recomputing with SSE. Subfigure b shows the running time.

the community in those cases. When accumulating a batch, we only count edge insertions with at least one endpoint vertex in the current community. We count edge deletions with at least one endpoint vertex either in the community or with neighbors in the community.

Figure 5.4a shows the mean, median, and quartiles of the speedup of ODSE over SSE. The speedup is the ratio of the running time of SSE over the running time of ODSE. This speedup ratio is computed for every seed set of every graph. A speedup of x means that using ODSE is x times faster than SSE, so higher values are better. The x-axis shows the batch size used and the y-axis shows the speedup, both on a log scale.

It is clear that the advantage of ODSE is greatest for small batch sizes. This is expected because the total running time of SSE decreases proportionally as the batch size increases. When the batch size is increased by a factor of x, there are x times fewer batches and re-computation occurs x times less frequently. Because the running time of SSE does not depend on the batch size, when the number of batches decreases by a factor of x, the total running time also decreases by a factor of x. ODSE, however, performs more work as the batch size increases. Steps 3 and 4 are only run once per batch, but the decrease is not by a factor of x as some steps must occur the same number of times, once per edge update, regardless of the batch size. This can be seen in Figure 5.4b, which shows the mean running time, across all seed sets of all graphs.

For batch sizes of 1, 10, and 100, using ODSE is faster than using SSE. Of course, ODSE performs less work and solves a slightly different problem because it updates the results instead of computing from scratch. However, for applications where it is desirable to continually know the current community as the underlying graph changes, the community must be updated when the graph is modified. In such cases, the comparison of the dynamic and static algorithms is warranted and fair. The updated output can be obtained either by re-computation with SSE or incrementally with ODSE.

Figure 5.4c shows the mean dynamic speedup over re-computation for each graph. For a batch size of 1, we achieve mean speedups of two orders of magnitude on some graphs.

ODSE is faster for batch sizes of up to and including 100. Figure 5.4d shows the mean dynamic speedup over re-computation for different community sizes. Each point shows the mean using all expansions with an average community size in the specified range. ODSE performs relatively better for communities of size 25 and up. This is not surprising because SSE will take longer to recompute a large community.

5.5 Community Operations

Tracking the evolution of communities through operations is an important topic when studying dynamic graphs [53, 38]. When dealing with global clusters, it is typical to compare the overlap of communities found at different times in order to detect continuing, growing, shrinking, merging, splitting, appearing, and disappearing communities. The focus of our work has been to present an algorithm that maintains a local community over time by incrementally updating. However, we will briefly discuss potential approaches to detecting community operations to motivate future work.

First, it is necessary to address the number of seed vertices used to expand a single community and the implications in a dynamic context. Although each seed set consists of a single vertex in our experiments, ODSE can be run with a seed set of multiple vertices of interest. However, because a single community is found for the seed set, it only makes sense to use multiple vertices if there is reason to believe that there exists some community containing all of them. In a dynamic graph the seed vertices should remain in a single community over time. Therefore, for the community operations addressed below, we limit our discussion to the use of one seed vertex per community.

We can track the evolution of individual local communities and detect interactions between them. A community can grow, shrink, or disappear, all of which can be detected by the size of the community. The volatility of a single cluster could be measured by comparing the members of consecutive times.

Independent local communities can merge and split by increasing and decreasing their

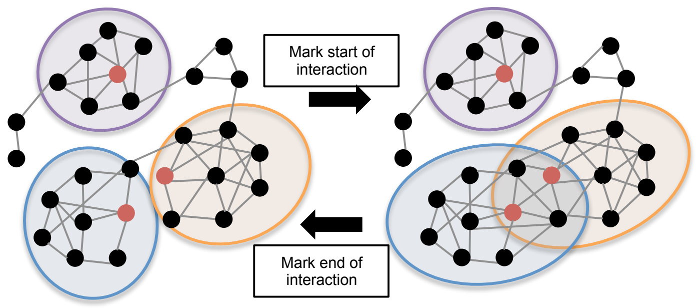


Figure 5.5: This figure illustrates tracking key vertex interactions using seed set expansion.

overlap. If, for example, two seed vertices have highly overlapping communities, then this indicates similarity or interaction between them. A metric based on overlap, such as the Jaccard index, and a threshold could be used to determine when the communities of two seeds have merged or split. A simpler option marks a merge, or beginning of interaction, when the community of one seed includes another seed vertex. A split, or end of interaction, is marked when when the latter seed is no longer included in the former seed's community. This is shown in Figure 5.5, where vertices of interest are used as seeds and shown in red. Communities are marked with shaded ovals. Left to right marks the beginning of interaction between two seed vertices when the community of one seed grows to include another seed. Right to left marks the end of an interaction.

5.6 Conclusion

We have presented Ordered Dynamic Seed Expansion, a new algorithm that incrementally updates the local community of a seed set when the underlying graph changes. For a variety of social networks with timestamps, ODSE produces communities that have both high fitness scores and high overlap with the communities produced by Static Seed Expansion. ODSE works well both when beginning with an initial existing graph and in a fully streaming manner when beginning with no initial data. The dynamic method is faster than Static Seed Expansion, which must be re-run whenever the graph is updated, and the performance improvement is greatest when low latency updates are needed. The speedup achieved varies based on the size of a local community, with ODSE performing relatively better on large communities. It is easily parallelized across independent expansions. This chapter also discussed an approach for tracking vertex interaction over time using local communities.

CHAPTER 6
A LOCAL MEASURE OF COMMUNITY CHANGE IN DYNAMIC GRAPHS

6.1 Introduction

Dynamic graphs represent relational data over time and can be used to discover how these relationships change. Recall that a dynamic graph can be represented as a sequence of graph snapshots $\{G_0, \ldots, G_t\}$, where each snapshot $G_t = (V_t, E_t)$ represents the state at that point in time. A simple extension of static graph analysis to dynamic graphs involves computing a standard static measure on each graph snapshot. For example, the centrality value of each vertex may be computed, or communities can be found and a selected fitness score may be evaluated for all clusters currently identified. However, the creation of a dynamic graph also allows for new types of dynamic analysis that study how the graph changes over time.

This chapter presents a new local, vertex-level measure of community change. In a dynamic graph, communities may evolve over time in a variety of ways and vertices may move between communities. Here, we focus on finding vertices that experience a particular type of change in their local community behavior, and refer to them as allegiance switching vertices. For example, consider a paper co-authorship network, in which vertices represent researchers and an edge indicates that two researchers have co-authored a paper within the time period under consideration. A community in such a graph may represent a group of researchers who collaborate with each other if the clusters are small, or researchers who are working on the same topic if the clusters are larger. In such a graph, we may wish to detect researchers who move to a different lab or university or change the field in which they publish. The contribution of this chapter is a new local measure of community change, with the following properties: (1) sensitivity: it detects vertices that have a change in their community and (2) stability: the community change detected is related to the interactions (edges) of the particular vertex and not only caused by global community shifts. This work was published in [111].

6.2 Allegiance Changing Vertices

6.2.1 Motivation

Typically, dynamic community analysis is done by first finding communities on each snapshot of a dynamic graph and then matching communities found in different snapshots using their overlap, as in [53, 54, 55]. If communities $C_{i,t_1} \in G_{t_1}$ and $C_{j,t_2} \in G_{t_2}$ share a high proportion of the same vertices, then they are considered to have a high overlap and may be matched, meaning that C_{j,t_2} is a continuation of C_{1,t_1}. If $C_{h,t_3} \in G_{t_3}$ is also matched as a continuation of $C_{j,t_2} \in G_{t_2}$ the three clusters may then be considered to be manifestations at different points in time of the same community of vertices. By considering the overlap, it may also be possible to detect a community splitting into multiple components or multiple clusters joining together. Once this matching process is performed, it could be possible to detect vertex level events, such as when a vertex moves from one community to another [55].

However, using such a global approach to detect vertex changes may produce unintended results if the communities are not stable over time. In a dataset with a very smooth community evolution, a global approach may work well; in practice however, a graph may greatly change between two snapshots, causing the communities to change drastically as well. In that case, a vertex may be flagged as moving between communities only because the community structure was unstable. A re-arrangement of community composition may prevent a high enough overlap between its communities at consecutive points in time to allow them to match.

The global matching approach does not ensure that the detected community change of a vertex is local and not only a global shift. Suppose vertex v has edges only to vertices w

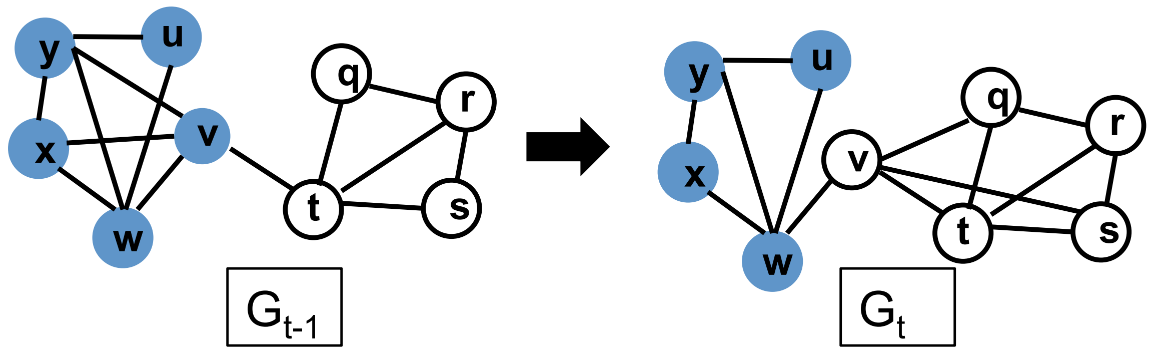


Figure 6.1: As the graph changes over time, vertex v has a high local measure of community change. Its neighbor set is $N_{t-1}(v) \cup N_t(v) = \{y, x, w, q, s, t\}$. The set of neighbors that leave v's community is $L_t(v) = \{y, x, w\}$. The set of neighbors that join v's community is $J_t(v) = \{q, s, t\}$. The set of neighbors that stay in the same community is $S_t(v) = \emptyset$.

and u. If w and u both move to a new community, v will likely move along with them and exhibit a global change in community membership. However, the behavior of v will not have changed (as it is still connected to the same two neighbors) and it will remain in the same community as its neighbors.

We refer to such events, when a vertex changes communities, as allegiance changes and the vertices that undergo such events are allegiance changing vertices. The detected community change of a vertex v should be local: caused by a change in v's actions (its edges) and not only a global shift.

6.2.2 Proposed Method

Given a dynamic graph, for each snapshot G_t, a set of non-overlapping communities $\{C_{1,t}, \ldots, C_{r_t,t}\}$ is detected. Let $C_t(v)$ represent the community that contains vertex v as its member in G_t and $N_t(v)$ the set of neighbors of vertex v in G_t. Using this community decomposition three sets are defined for each vertex $v \in V_t$. $S_t(v)$ is the set of neighbors of v that were in the same community as v at the previous snapshot G_{t-1} and stay in the same community as v in G_t. $L_t(v)$ is the set of neighbors of v that were in the same community in G_{t-1}, but are no longer in G_t. $J_t(v)$ is the set of neighbors of v that were not in the same community as v in G_{t-1}, but are in G_t. Formally these are defined in Equations 6.1, 6.2, and 6.3 respectively.

$$S_t(v) = \{w | w \in N_{t-1}(v) \cup N_t(v) \wedge C_{t-1}(v) = C_{t-1}(w) \wedge C_t(v) = C_t(w)\} \tag{6.1}$$

$$L_t(v) = \{w | w \in N_{t-1}(v) \cup N_t(v) \wedge C_{t-1}(v) = C_{t-1}(w) \wedge C_t(v) \neq C_t(w)\} \tag{6.2}$$

$$J_t(v) = \{w | w \in N_{t-1}(v) \cup N_t(v) \wedge C_{t-1}(v) \neq C_{t-1}(w) \wedge C_t(v) = C_t(w)\} \tag{6.3}$$

A large change in v's neighborhood at time t occurs when three conditions occur simultaneously: (1) the set $L_t(v)$ is large, indicating that many vertices with which v interacted and which were in the same community as v are no longer in the same community as v; (2) the set $J_t(v)$ is large, indicating that v is now connected to vertices which were not previously in its community; and (3) the set $S_t(v)$ is relatively small, so that a low percentage of v's neighborhood was and continues to be in the same community. Using $S_t(v)$, $L_t(v)$, and $J_t(v)$, two measures of community change severity $\alpha_t(v)$ and $\beta_t(v)$ are defined below.

$$\alpha_t(v) = \frac{|L_t(v)|}{|S_t(v) \cup L_t(v)|} \tag{6.4}$$

$$\beta_t(v) = \frac{|J_t(v)|}{|S_t(v) \cup J_t(v)|} \tag{6.5}$$

Figure 6.1 shows an example. Intuitively, a high value of $\alpha_t(v)$ indicates that a large percentage of neighbors of v have left v's community (or v has left theirs). Similarly, a high value of $\beta_t(v)$ means that a large percentage of v's neighbors were not in the same community as v at time $t-1$, but joined v's community at time t (or v joined their community). Therefore, vertices v with high values $\alpha_t(v)$ and $\beta_t(v)$ have experienced a large change in their local neighborhood.

The focus of this community change characterization is on the community membership of a vertex v relative to its neighbor set because the neighbors represent other entities v has recently interacted with. With a larger number of hops, any relationship becomes much weaker and more difficult to characterize. Vertex v may be in the same community as many vertices with which it does not interact or which are not even in a two hop neighborhood. Any community change associated with such distant vertices should not affect the characterization of v's allegiance. Therefore, our measure of community change is local.

6.2.3 Setting a Threshold

In order to mark a vertex as allegiance switching, it needs to have both high values of $\alpha_t(v)$ and $\beta_t(v)$. To combine the two values, both of which fall between 0 and 1, we can draw from work from the field of fuzzy logic, where the logical conjunction of two values ("x and y") is represented by taking the minimum ($min(x,y)$), using Gödel's t-norm, or by taking the product ($x * y$), using the Product t-norm [112, 113]. In experiments shown in Section 6.3, both the minimum and the product of $\alpha_t(v)$ and $\beta_t(v)$ are used to demonstrate two different approaches of combining the two values, although other methods could be used as well.

As in any extreme value problem, there can be various ways of setting a threshold, depending on the application. A fixed threshold is easily interpreted and can be set by the end user. For example, for a given application, it may be determined that both $\alpha_t(v)$ and $\beta_t(v)$ need to be above 0.8. On the other hand, it can be advantageous to use a threshold that adjusts to the distribution of scores for a particular dataset. One option is to choose vertices with a score within a a top percentile range. Another is to use the z-score and choose only values that are at least a certain number of standard deviations above the mean.

6.2.4 Alternative Approaches

In this section we consider two possible alternative approaches to detecting allegiance changing vertices. The first is a global approach, which matches communities in consecutive snapshots. For each pair of consecutive graph snapshots G_{t-1} and G_t, we compute communities using the static Louvain algorithm [16]. Two communities, $C_{j,t-1} \in G_{t-1}$ and $C_{i,t} \in G_t$, are matched as equal if the Jaccard index $|C_{j,t-1} \cap C_{i,t}|/|C_{j,t-1} \cup C_{i,t}| \geq 0.75$. A vertex is flagged as allegiance switching if $C_{t-1}(v)$ and $C_t(v)$ are not matched. Experiments in Section 6.3 compare the vertices flagged with the new local approach to those flagged with this baseline global approach. Results show that this global approach identifies a very large percent of vertices as community switching. This occurs because communities change a lot between consecutive snapshots and therefore cannot be matched. The success of the global approach may be increased by using a community detection algorithm that tries to detect communities with smooth transitions by taking into account historical data [42, 39]; however, such algorithms can be more computationally expensive.

Another possible alternative approach might be to compare the neighbors of a vertex v at time t to those at time $t-1$ and count how many have changed. However, doing so would not capture community allegiance changing behavior. For example, consider the co-authorship network example mentioned in Section 6.1. From year to year, a researcher may change with whom he co-authors papers without changing his field or even his work group. In the time represented by G_{t-1} he may co-author with colleagues w and u and during the time represented by G_t co-author with colleagues y and z. In this case, the set of neighboring vertices would change completely with no overlap. However, if w, u, y, and z all belong to the same lab or department and work together, then we would not consider v to have changed behavior because he would be working within the same group. Therefore, simply looking at the change in vertex's neighbors will not capture the desired community changing behavior.

6.3 Experiments

6.3.1 Synthetic Graphs

This section evaluates performance of the new local approach using synthetic graphs built with a stochastic block model, and provides insights into dependence of the algorithm on its parameter setting. The method is also compared with the global alternative described in Section 6.2.4. In an assortative stochastic block model, vertices within the same community are connected by an edge with probability p, while vertices in different communities are connected with probability q, where $p > q$ is necessary to generate community structure. The higher p and the lower q, the stronger the community structure is and the easier it is to discover.

In the first experiment, dynamic graphs are generated by stochastic block model with 2 communities, 1000 vertices, $p = 0.3$, and $q = 0.01$. Each dynamic graph is composed of two snapshots. After G_1 is generated as described, 1% of vertices are chosen and moved to a different community before generating G_2. Both the global approach and our local measure are able to identify all of the vertices that moved between communities because the community structure is clear and only changes a little. The local approach assigns high scores to all vertices that were moved from one community to the other. After 20 runs, each on a randomly generated graph, vertices that moved between clusters all had high allegiance changing scores, where $\alpha_t(v) * \beta_t(v)$ values ranged from 0.93 to 1, while their $min(\alpha_t(v), \beta_t(v))$ values ranged from 0.96 to 1. To select which vertices will be flagged as allegiance changing by the local method, the threshold for this experiment is chosen using the z-score. Figure 6.2 shows how the false positive rate, the proportion of vertices incorrectly flagged as moving communities, varies as the z-score threshold changes. It is clear that combining $\alpha_t(v)$ and $\beta_t(v)$ scores yields lower false positives, with the use of $\alpha_t(v) * \beta_t(v)$ giving best results. In this example, vertices cleanly move from one community to another and both the global and local methods can detect the change.

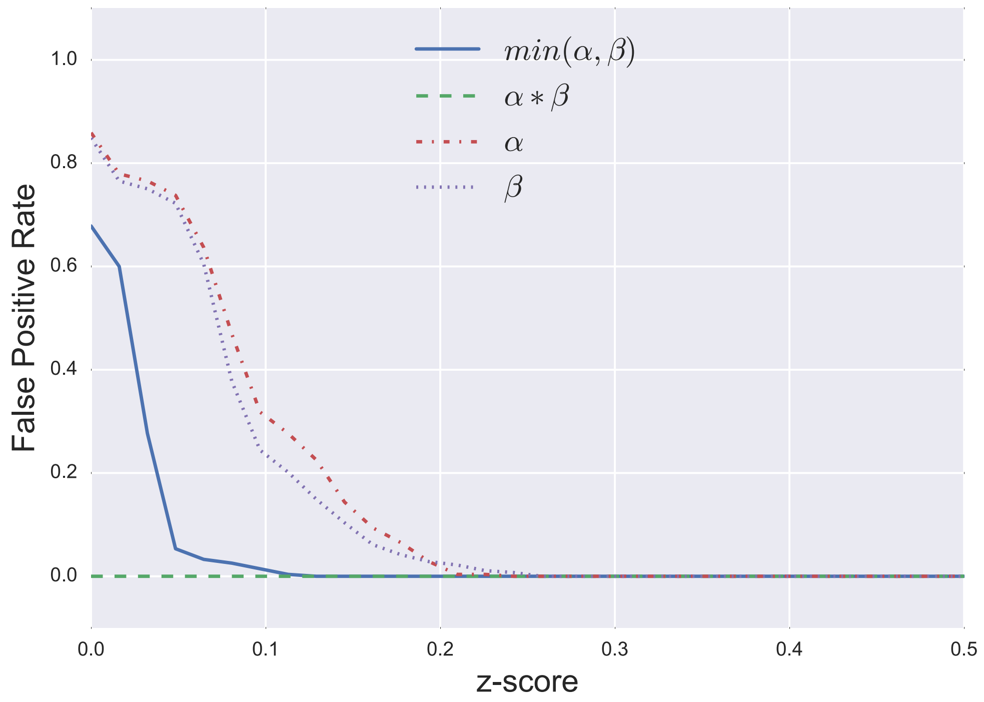


Figure 6.2: The false positive rate is shown for the simple non-hierarchial stochastic block model experiment. Combining $\alpha_t(v)$ and $\beta_t(v)$ scores yields fewer false positives.

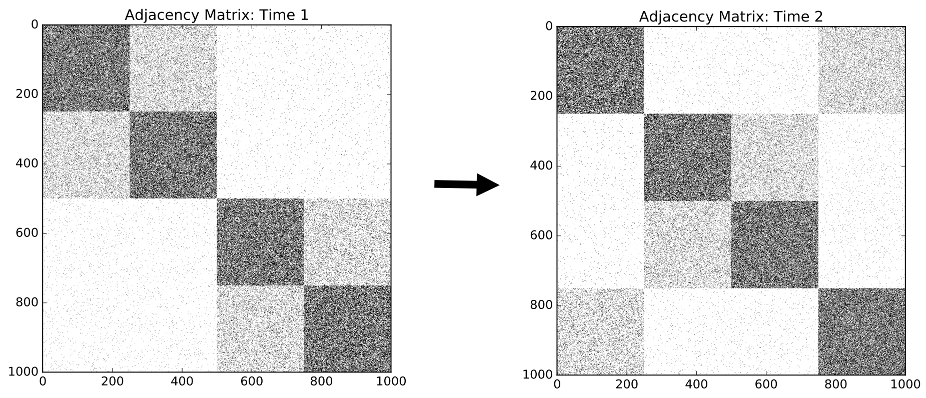


Figure 6.3: The rearrangement of communities in a hierarchical stochastic block model

However, in graphs created from real data, communities do not always persist over time and when changes do occur, they are often not as clean as a simple joining of multiple communities into one or splitting apart cleanly. Instead, they may break apart and re-arrange. Such changes are simulated using a hierarchical block model. Each top level community is composed of multiple, more tightly connected, sub-communities. Vertices within the same sub-community are connected with probability p_1, vertices within different sub-communities, but the same community, share an edge with probability p_2, and all others are connected with probability q, where $p_1 > p_2 > q$. Community change is then created by keeping each sub-community intact, but re-arranging them into different top level communities. This process is shown in Figure 6.3. The left and right images show a representation of G_1 and G_2, respectively, as adjacency matrices. If the i^{th} and j^{th} vertices of the graph are connected with an edge, entry $A[i, j]$ of the corresponding adjacency matrix is filled in. Therefore, the degree of shading in different regions of the matrix corresponds to the percentage of vertices that are connected with edges. The darker the region, the higher the connectedness between vertices. This way, the hierarchical community structure and its re-arrangement can be visualized.

This example demonstrates the limitation of the global approach, which marks all vertices as community changing, despite the fact that the sub-communities remain the same.

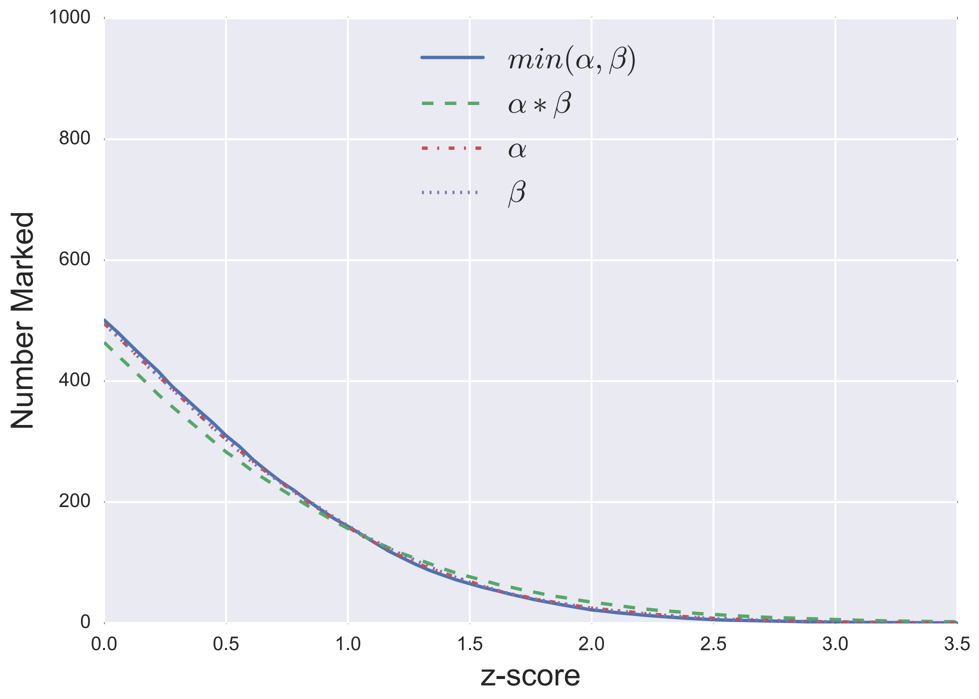
As shown in Figure 6.3, each snapshot has two communities and each community has two sub-communities. Therefore, the overlap between any pair of communities in G_1 and G_2 is 0.5 using the the Jaccard index. Therefore, no community from G_1 will be matched with one from G_2 and the global method will mark every vertex as community changing, despite the fact that the four sub-communities in both graphs remain the same.

The local metric scores $\alpha_t(v)$ and $\beta_t(v)$, however, remain low for all vertices. Figure 6.4 shows the number of vertices flagged as allegiance changing as the score threshold increases. Lower numbers on the y-axis are better. As sub-communities do not change at all, here the desired behavior is not to detect allegiance changes at all. We consider flagging vertices as allegiance changing based on four values: $\alpha_t(v)$, $\beta_t(v)$, $min(\alpha_t(v), \beta_t(v))$, and $\alpha_t(v) * \beta_t(v)$. The top plot of Figure 6.4 shows how the number of flagged varies as the threshold for each of these scores is set based on the z-score. The bottom plot shows how the number flagged varies as the threshold is set based on the raw value. Combining the $\alpha_t(v)$ and $\beta_t(v)$ values by using either $min(\alpha_t(v), \beta_t(v))$ or $\alpha_t(v) * \beta_t(v)$ yields better results because fewer vertices are incorrectly flagged as allegiance changing.

In this synthetic example, while the communities are globally unstable from one snapshot to the next, large portions of the communities remain the same. The global approach only detects this global change, while the local measure detects that each vertex experienced little change in its own community behavior.

6.3.2 Real-Life Datasets

The proposed local allegiance change metric is also evaluated on three dynamic graphs built from datasets from the Koblenz Network Collection [59], listed in Table 6.1. The dynamic graphs are created using the sliding window method described in Chapter 4, with window size and overlap as listed in Table 6.1. A window size of 6 months with 4 month overlap means that each snapshot contains edges from a 6 month period and consecutive snapshots overlap by 4 months. In the experiments below, communities are found using the



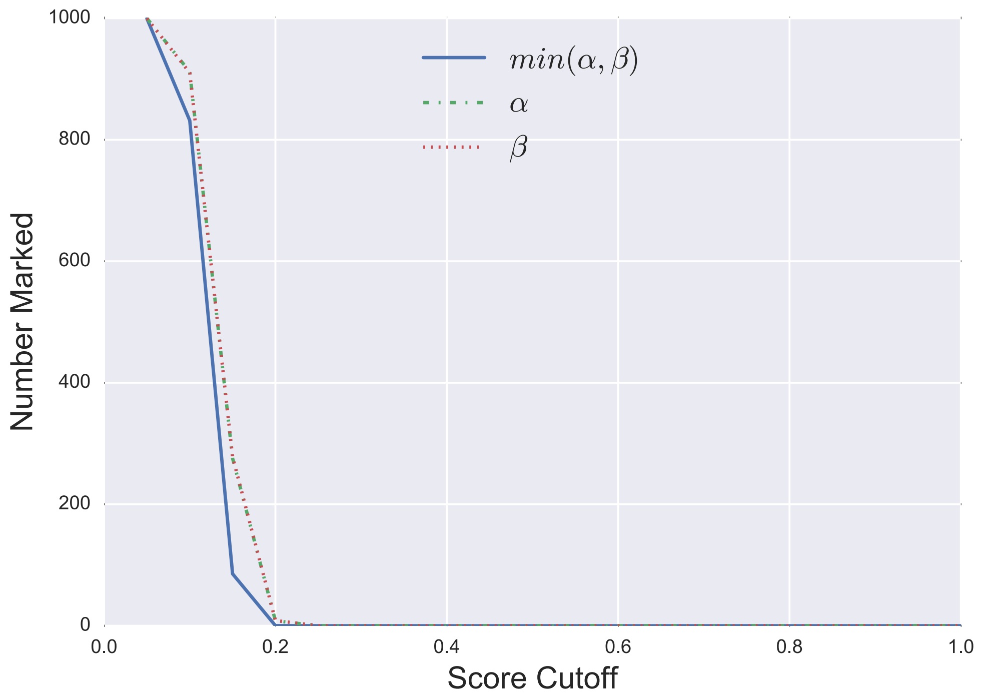


Figure 6.4: The number of vertices incorrectly flagged as community changing in the hierarchical stochastic block model experiment is shown. A z-score and raw value threshold is used on the x-axis on the top and bottom plots, respectively.

Table 6.1: A description of the graphs and sliding window used to build snapshots is shown.

Graph	Vertices	Edges	Snapshot Window	Window Overlap
dblp	1,314,050	18,986,618	6 years	4 years
youtube	3,223,589	9,375,374	3 months	2 months
facebook	46,952	876,993	6 months	4 months

Table 6.2: The average percentage of vertices flagged is shown for all three datasets.

Graph	Global $\%$	$min(\alpha_t(v), \beta_t(v))$ $\%$	$\alpha_t(v) * \beta_t(v)$ $\%$
dblp	64.6	2.5	2.5
youtube	51.4	4.1	4.0
facebook	77.5	6.3	6.2

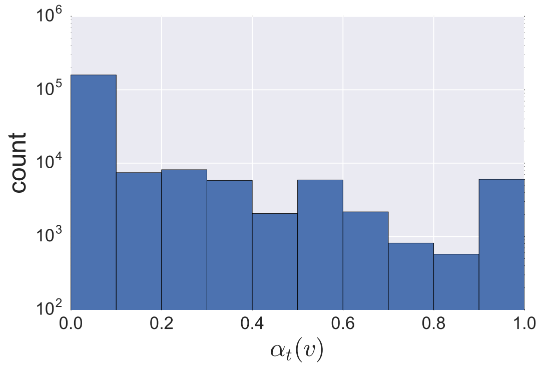
popular Louvain algorithm [16], although any other suitable algorithm can be used, as the approach is agnostic to it.

By running the Louvain algorithm on each snapshot and computing the overlap of communities in consecutive snapshots, communities can be matched as described in Section 6.2.4. To increase the number of communities matched, the overlap was computed using only vertices with non-zero degree in both consecutive snapshots. Many communities, especially larger ones, were not matched. This does not mean that all non-matched communities necessarily dissolve completely. Rearranged communities, such as in the stochastic block model example, will also fail to match. The smoothness of community evolution will of course depend on both the particular community detection algorithm and sliding window used. However, results suggest that we cannot rely on the dataset in question to have smooth transitions between communities of different snapshots, which makes the global metric sensitive to many factors. In contrast, the local allegiance change measure does not require stable clusters between snapshots because it only considers the community of a vertex relative to that of its neighbors. Clusters can be locally stable, without being globally stable.

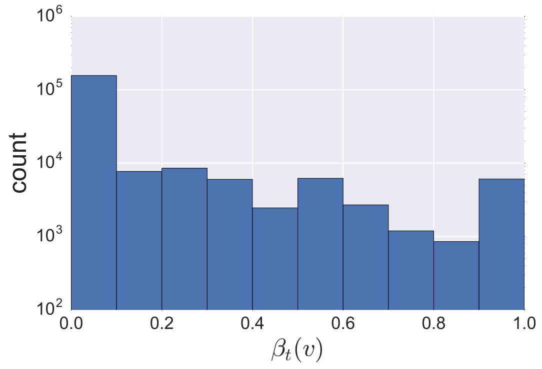
Table 6.2 shows the average (over all snapshots) percentages of vertices flagged by the local measure using both the minimum and product to combine $\alpha_t(v)$ and $\beta_t(v)$. Minimum

Table 6.3: The average percentage of vertices flagged using varying overlaps of the *dblp* dataset is shown. Each snapshot contains data from 6 years and consecutive snapshots overlap by the amount shown.

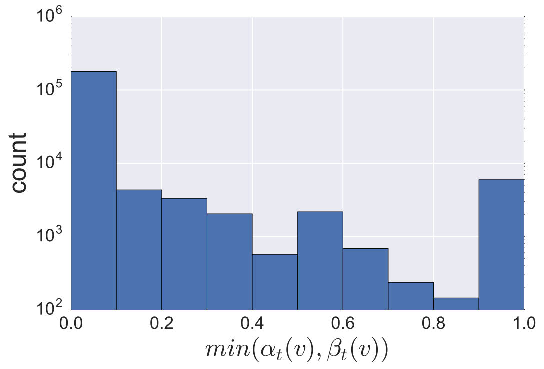
Overlap	Global %	$min(\alpha_t(v), \beta_t(v))$ %	$\alpha_t(v) * \beta_t(v)$ %
0 years	77.0	19.5	18.7
2 years	67.2	7.0	6.7
4 years	64.6	2.5	2.5
5.5 years	56.8	0.9	0.9



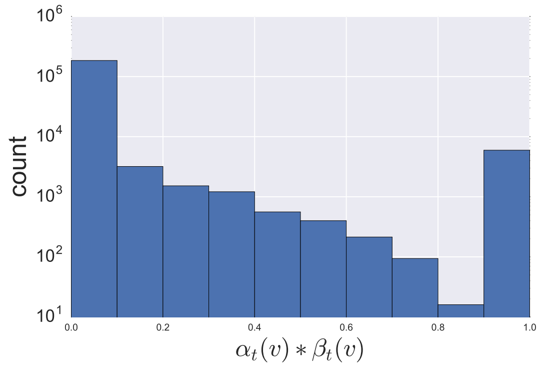
(a) Histogram of the $\alpha_t(v)$ scores.



(b) Histogram of the $\beta_t(v)$ scores.



(c) Histogram of the $min(\alpha_t(v), \beta_t(v))$ scores.



(d) Histogram of the $\alpha_t(v) * \beta_t(v)$ scores.

Figure 6.5: Histograms of individual and combined metrics for one snapshot of the *dblp* graph.

has a fixed threshold of 0.8 and product uses the top 1% score. The percentage flagged by the alternative global metric is also shown. For all datasets, we see that the global metric flags many more vertices than either local metric. Almost all vertices flagged by the local method were also flagged by the global approach.

As discussed in Chapter 4, increasing the overlap between consecutive snapshots of a dynamic graph will lead to smoother evolution, including more gradual changes in community structure. This will naturally affect the number of communities matched and therefore the number of vertices flagged as community changing. This effect is explored in Table 6.3, which shows the percentage of vertices flagged as allegiance changing for various overlap intervals for the *dblp* dataset. As expected, as the overlap between consecutive snapshots increases, the percentage of vertices flagged as allegiance changing decreases. Nevertheless, even for a high overlap, the global method flags over half the vertices. Therefore, for analysts wishing to tag allegiance changing vertices, the local metrics provide a much smaller set of candidates than their global counterpart.

Finally, as with the synthetic experiments, we again consider the effect of combining $\alpha_t(v)$ and $\beta_t(v)$ values Figure 6.5 shows distributions of the scores for $\alpha_t(v)$, $\beta_t(v)$, $\alpha_t(v) * \beta_t(v)$, and $min(\alpha_t(v), \beta_t(v))$ for one snapshot of the *dblp* graph. This histograms show that outliers can be spotted more clearly by combining $\alpha_t(v)$ and $\beta_t(v)$ values. Figure 6.5d especially shows a clear trend of a decreasing number of vertices as the score increases, and then a spike in outlying vertices with a very high score.

6.4 Conclusion

This chapter presented a new local metric for identifying vertex-level community changes. Experiments on synthetic graphs show that when communities change very little, both the global and our local measures correctly detect vertices that switch communities. However, when the communities of a graph change between snapshots in unexpected ways, the global method may be unreliable, often flagging a majority of vertices, while the local method does not. Experiments on graphs representing real social networks show that the new local approach flags far fewer vertices of interest compared to the global alternative. The results suggest that the global approach is flawed when communities do not evolve smoothly between snapshots. While there is an increasing interest in dynamic networks and dynamic community detection, finding community allegiance changing vertices is a relatively new area. Measuring vertex level community-oriented changes pinpoints vertices with interesting behavior, allowing for further investigation. The measure applies to any community detection algorithm and could be extended to overlapping community detection methods.

CHAPTER 7
CONCLUSION

This dissertation addressed several challenges of performing graph analysis on streaming, relational datasets. Analyzing many real-life graphs is difficult due to the large volume of data that is continually created. The size of the full dataset may be too large to store or large enough that running computationally intensive analytics becomes infeasible. A solution to this is to sample a smaller subgraph that is structurally similar to the graph created from the full dataset. Further analysis can then be performed on the sampled subgraph. Chapter 3 contributed three new algorithms for streaming graph sampling: WES, RIES, and TWES. These methods sample an edge stream in a single pass without requiring any information about its properties. Therefore, they can be used to tap into a real-time stream of edges as they are generated. The algorithms create samples with a specified number of edges and are therefore appropriate to use when there is limited memory to store the dataset. Experiments on several social network datasets show that the new sampling algorithms produce better subgraphs than the RE method from the literature. This means that the subgraphs sampled with WES, RIES, and TWES are more structurally similar to the full graph than those created by RE are. WES and RIES both produce non-temporally biased samples. The advantage of WES over RIES is that the effect of the parameter setting on the structure of the sample is more predictable. TWES produces a temporally biased sample. Newer edges with higher timestamps are more likely to be included, making TWES useful for the applications in which new data is considered more relevant and may be preferred. By adjusting their parameters as a stream is processed, both WES and TWES can sample subgraphs with targetted structural properties, suggesting that they can be useful in real-life applications.

Chapter 4 contributed to the topic of dynamic graph creation. Before dynamic graph analysis is performed, a dynamic graph must first be created from temporal edge data. To do so, it is necessary to decide how to add new data and remove old data. In particular, it is not obvious how to remove past data, especially when edges are derived from interactions. Once an interaction occurs, it is not explicitly reversed, but rather may decrease in relevance over time. Chapter 4 evaluated several methods of aging data to create dynamic graphs. In addition to methods used in the literature, a new approach was proposed, based on the concept of active vertices and edges. The differences and similarities between dynamic graphs created by each aging approach were evaluated using several global and vertex-level graph properties. Because the topic of aging data to create dynamic graphs was previously largely unexplored, the focus in this work was on basic graph properties. Future work should evaluate the effect on more complex graph measures, such as community evolution. The method chosen is likely to affect both the communities found and how much they change.

While Chapters 3 and 4 addressed steps that are taken before analysis is performed, Chapters 5 and 6 made contributions to community based graph analysis. Chapter 5 presented Ordered Dynamic Seed Expansion, a new dynamic algorithm for local community detection. Ordered Dynamic Seed Expansion is useful when a dynamic graph quickly changes and low latency community updates are needed. Because it incrementally updates, computation is reduced. Experiments show that using Ordered Dynamic Seed Expansion is indeed faster than recomputing with Static Seed Expansion. The quality of Ordered Dynamic Seed Expansion also remains high even after a large number of graph updates have occurred. Finally, Chapter 6 focused on analyzing community changes over time. It contributed a new measure to detect when vertices change communities. This measure uses changes in a vertex's neighborhood to flag a community change. Because the measure is local to a vertex's neighborhood, it is particularly suitable when the communities found are unstable over time. Tracking community changes can reveal vertices with interesting behavior, allowing for further investigation.

Together, these contributions further the study of streaming edge data through graph analysis. The work presented addresses challenges arising in multiple stages of the process, improving our ability to learn from the vast and ever-growing amounts of data at our disposal.

REFERENCES

[1] *Internet live stats – internet usage and social media statistics*, `http://www.internetlivestats.com/`, Feb. 2018.

[2] D. Ediger, K. Jiang, J. Riedy, D. A. Bader, and C. Corley, "Massive social network analysis: Mining twitter for social good," in *Proceedings of the 39th International Conference on Parallel Processing (ICPP)*, IEEE, 2010, pp. 583–593.

[3] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph evolution: Densification and shrinking diameters," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 1, no. 1, p. 2, 2007.

[4] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3, pp. 75–174, 2010.

[5] F. Havemann, M. Heinz, A. Struck, and J. Gläser, "Identification of overlapping communities and their hierarchy by locally calculating community-changing resolution levels," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2011, no. 01, P01023, 2011.

[6] M. E. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, no. 2, p. 026 113, 2004.

[7] L. Hagen and A. B. Kahng, "New spectral methods for ratio cut partitioning and clustering," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 9, pp. 1074–1085, 1992.

[8] J. Shi and J. Malik, "Normalized cuts and image segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 8, pp. 888–905, 2000.

[9] U. Von Luxburg, "A tutorial on spectral clustering," *Statistics and Computing*, vol. 17, no. 4, pp. 395–416, 2007.

[10] M. Girvan and M. E. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[11] A. Clauset, M. E. Newman, and C. Moore, "Finding community structure in very large networks," *Physical Review E*, vol. 70, no. 6, p. 066 111, 2004.

[12] L. Danon, A. Díaz-Guilera, and A. Arenas, "The effect of size heterogeneity on community identification in complex networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2006, no. 11, P11010, 2006.

[13] K. Wakita and T. Tsurumi, "Finding community structure in mega-scale social networks," in *Proceedings of the 16th international conference on World Wide Web*, ACM, 2007, pp. 1275–1276.

[14] P. Schuetz and A. Caflisch, "Efficient modularity optimization by multistep greedy algorithm and vertex mover refinement," *Physical Review E*, vol. 77, no. 4, p. 046 112, 2008.

[15] E. J. Riedy, H. Meyerhenke, D. Ediger, and D. A. Bader, "Parallel community detection for massive graphs," in *International Conference on Parallel Processing and Applied Mathematics*, Springer, 2011, pp. 286–296.

[16] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, P10008, 2008.

[17] W. E. Donath and A. J. Hoffman, "Lower bounds for the partitioning of graphs," *IBM Journal of Research and Development*, vol. 17, no. 5, pp. 420–425, 1973.

[18] M. Fiedler, "Algebraic connectivity of graphs," *Czechoslovak Mathematical Journal*, vol. 23, no. 2, pp. 298–305, 1973.

[19] M. E. Newman, "Modularity and community structure in networks," *Proceedings of the National Academy of Sciences*, vol. 103, no. 23, pp. 8577–8582, 2006.

[20] J. Dahlin and P. Svenson, "Ensemble approaches for improving community detection methods," *ArXiv preprint arXiv:1309.0242*, 2013.

[21] R. Kanawati, "Yasca: An ensemble-based approach for community detection in complex networks," in *International Computing and Combinatorics Conference*, Springer, 2014, pp. 657–666.

[22] C. L. Staudt and H. Meyerhenke, "Engineering high-performance community detection heuristics for massive graphs," in *Proceedings of the 42nd International Conference on Parallel Processing (ICPP)*, IEEE, 2013, pp. 180–189.

[23] M. Ovelgönne and A. Geyer-Schulz, "An ensemble learning strategy for graph clustering.," *Graph Partitioning and Graph Clustering*, vol. 588, p. 187, 2012.

[24] J. Xie, S. Kelley, and B. K. Szymanski, “Overlapping community detection in networks: The state-of-the-art and comparative study,” *ACM Computing Surveys (CSUR)*, vol. 45, no. 4, p. 43, 2013.

[25] I. Derényi, G. Palla, and T. Vicsek, “Clique percolation in random networks,” *Physical review letters*, vol. 94, no. 16, p. 160 202, 2005.

[26] T. Evans and R Lambiotte, “Line graphs of weighted networks for overlapping communities,” *The European Physical Journal B*, vol. 77, no. 2, pp. 265–272, 2010.

[27] U. N. Raghavan, R. Albert, and S. Kumara, “Near linear time algorithm to detect community structures in large-scale networks,” *Physical Review E*, vol. 76, no. 3, p. 036 106, 2007.

[28] J. Xie and B. K. Szymanski, “Labelrank: A stabilized label propagation algorithm for community detection in networks,” in *IEEE 2nd Network Science Workshop (NSW)*, 2013, pp. 138–143.

[29] ——, “Towards linear time overlapping community detection in social networks,” in *Advances in Knowledge Discovery and Data Mining*, Springer, 2012, pp. 25–36.

[30] A. Lancichinetti, S. Fortunato, and J. Kertész, “Detecting the overlapping and hierarchical community structure in complex networks,” *New Journal of Physics*, vol. 11, no. 3, p. 033 015, 2009.

[31] C. Lee, F. Reid, A. McDaid, and N. Hurley, “Detecting highly overlapping community structure by greedy clique expansion,” in *4th SNA-KDD Workshop*, 2010, 3342.

[32] A. Clauset, “Finding local community structure in networks,” *Physical Review E*, vol. 72, no. 2, p. 026 132, 2005.

[33] J. Riedy, D. A. Bader, K. Jiang, P. Pande, and R. Sharma, “Detecting communities from given seeds in social networks,” Georgia Institute of Technology, Tech. Rep., 2011.

[34] J. P. Bagrow and E. M. Bollt, “Local method for detecting communities,” *Physical Review E*, vol. 72, no. 4, p. 046 108, 2005.

[35] R. Andersen, F. Chung, and K. Lang, “Local graph partitioning using pagerank vectors,” in *47th Annual IEEE Symposium on Foundations of Computer Science*, IEEE, 2006, pp. 475–486.

[36] R. Andersen and K. J. Lang, “Communities from seed sets,” in *Proceedings of the 15th International Conference on World Wide Web*, ACM, 2006, pp. 223–232.

[37] T. Aynaud, E. Fleury, J.-L. Guillaume, and Q. Wang, "Communities in evolving networks: Definitions, detection, and analysis techniques," in *Dynamics On and Of Complex Networks, Volume 2*, Springer, 2013, pp. 159–200.

[38] R. Cazabet and F. Amblard, "Encyclopedia of social network analysis and mining," in, R. Alhajj and J. Rokne, Eds. Springer New York, 2014, ch. Dynamic Community Detection, pp. 404–414.

[39] C. Tantipathananandh, T. Berger-Wolf, and D. Kempe, "A framework for community identification in dynamic social networks," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge Discovery and Data Mining*, 2007, pp. 717–726.

[40] P. J. Mucha, T. Richardson, K. Macon, M. A. Porter, and J.-P. Onnela, "Community structure in time-dependent, multiscale, and multiplex networks," *Science*, vol. 328, no. 5980, pp. 876–878, 2010.

[41] M. B. Jdidia, C. Robardet, and E. Fleury, "Communities detection and analysis of their dynamics in collaborative networks.," in *ICDIM*, 2007, pp. 744–749.

[42] D. Chakrabarti, R. Kumar, and A. Tomkins, "Evolutionary clustering," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2006, pp. 554–560.

[43] Y.-R. Lin, Y. Chi, S. Zhu, H. Sundaram, and B. L. Tseng, "Analyzing communities and their evolutions in dynamic social networks," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 3, no. 2, p. 8, 2009.

[44] H. Ning, W. Xu, Y. Chi, Y. Gong, and T. S. Huang, "Incremental spectral clustering by efficiently updating the eigen-system," *Pattern Recognition*, vol. 43, no. 1, pp. 113–127, 2010.

[45] T. Aynaud and J.-L. Guillaume, "Static community detection algorithms for evolving networks," in *WiOpt'10: Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks*, 2010, pp. 508–514.

[46] J. Shang, L. Liu, F. Xie, Z. Chen, J. Miao, X. Fang, and C. Wu, "A real-time detecting algorithm for tracking community structure of dynamic networks," *ArXiv preprint arXiv:1407.2683*, 2014.

[47] T. N. Dinh, Y. Xuan, and M. T. Thai, "Towards social-aware routing in dynamic communication networks," in *2009 IEEE 28th International Performance Computing and Communications Conference (IPCCC)*, IEEE, 2009, pp. 161–168.

[48] R. Aktunc, I. H. Toroslu, M. Ozer, and H. Davulcu, “A dynamic modularity based community detection algorithm for large-scale networks: Dslm,” in *Proceedings of the IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, 2015, pp. 1177–1183.

[49] L. Waltman and N. J. van Eck, “A smart local moving algorithm for large-scale modularity-based community detection,” *The European Physical Journal B*, vol. 86, no. 11, pp. 1–14, 2013.

[50] J. Riedy and D. A. Bader, “Multithreaded community monitoring for massive streaming graph data,” in *27th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, 2013, pp. 1646–1655.

[51] M. Takaffoli, R. Rabbany, and O. R. Zaïane, “Incremental local community identification in dynamic social networks,” in *Proceedings of the IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, 2013, pp. 90–94.

[52] J. Chen, O. R. Zaiane, and R. Goebel, “Detecting communities in large networks by iterative local expansion,” in *International Conference on Computational Aspects of Social Networks (CASON)*, IEEE, 2009, pp. 105–112.

[53] M. Spiliopoulou, “Evolution in social networks: A survey,” in *Social Network Data Analytics*, Springer, 2011, pp. 149–175.

[54] D. Greene, D. Doyle, and P. Cunningham, “Tracking the evolution of communities in dynamic social networks,” in *Proceedings of the 2010 International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, IEEE, 2010, pp. 176–183.

[55] S. Asur, S. Parthasarathy, and D. Ucar, “An event-based framework for characterizing the evolutionary behavior of interaction graphs,” *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 3, no. 4, p. 16, 2009.

[56] J. Hopcroft, O. Khan, B. Kulis, and B. Selman, “Tracking evolving communities in large linked networks,” *Proceedings of the National Academy of Sciences*, vol. 101, no. suppl 1, pp. 5249–5253, 2004.

[57] G. Palla, A.-L. Barabási, and T. Vicsek, “Quantifying social group evolution,” *Nature*, vol. 446, no. 7136, pp. 664–667, 2007.

[58] J. Leskovec and A. Krevl, *Snap datasets: Stanford large network dataset collection*, `http://snap.stanford.edu/data`, Jun. 2014.

[59] J. Kunegis, “Konect: The koblenz network collection,” in *Proceedings of the 22nd International Conference on World Wide Web*, ACM, 2013, pp. 1343–1350.

[60] M. De Choudhury, W. A. Mason, J. M. Hofman, and D. J. Watts, “Inferring relevant social networks from interpersonal communication,” in *Proceedings of the 19th International Conference on World Wide Web*, ACM, 2010, pp. 301–310.

[61] I.-H. Ting, H.-J. Wu, and P.-S. Chang, “Analyzing multi-source social data for extracting and mining social networks,” in *Proceedings of the International Conference on Computational Science and Engineering*, IEEE, vol. 4, 2009, pp. 815–820.

[62] N. Jeners, P. Nicolaescu, and W. Prinz, “Analyzing tie-strength across different media,” in *OTM Confederated International Conferences “On the Move to Meaningful Internet Systems”*, Springer, 2012, pp. 554–563.

[63] J. Wiese, J.-K. Min, J. I. Hong, and J. Zimmerman, “You never call, you never write: Call and sms logs do not always indicate tie strength,” in *Proceedings of the 18th ACM conference on Computer Supported Cooperative Work & Social Computing*, 2015, pp. 765–774.

[64] D. J. Crandall, L. Backstrom, D. Cosley, S. Suri, D. Huttenlocher, and J. Kleinberg, “Inferring social ties from geographic coincidences,” *Proceedings of the National Academy of Sciences*, vol. 107, no. 52, pp. 22 436–22 441, 2010.

[65] J. Cranshaw, E. Toch, J. Hong, A. Kittur, and N. Sadeh, “Bridging the gap between physical location and online social networks,” in *Proceedings of the 12th ACM international conference on Ubiquitous computing*, 2010, pp. 119–128.

[66] T. M. T. Do and D. Gatica-Perez, “Human interaction discovery in smartphone proximity networks,” *Personal and Ubiquitous Computing*, vol. 17, no. 3, pp. 413–431, 2013.

[67] T. M. T. Do, K. Kalimeri, B. Lepri, F. Pianesi, and D. Gatica-Perez, “Inferring social activities with mobile sensor networks,” in *Proceedings of the 15th ACM on International Conference on Multimodal Interaction*, 2013, pp. 405–412.

[68] S. Mardenfeld, D. Boston, S. J. Pan, Q. Jones, A. Iamntichi, and C. Borcea, “Gdc: Group discovery using co-location traces,” in *Proceedings of the IEEE 2nd International Conference on Social Computing (SocialCom)*, 2010, pp. 641–648.

[69] N. Eagle, A. S. Pentland, and D. Lazer, “Mobile phone data for inferring social network structure,” in *Social Computing, Behavioral Modeling, and Prediction*, Springer, 2008, pp. 79–88.

[70] C. Ma, J. Cao, L. Yang, J. Ma, and Y. He, "Effective social relationship measurement based on user trajectory analysis," *Journal of Ambient Intelligence and Humanized Computing*, vol. 5, no. 1, pp. 39–50, 2014.

[71] Y. Lin, X. Jia, M. Lin, S. Gregory, H. Wan, and Z. Wu, "Inferring high quality co-travel networks," *ArXiv preprint arXiv:1305.4429*, 2013.

[72] A. Clauset and N. Eagle, "Persistence and periodicity in a dynamic proximity network," *ArXiv preprint arXiv:1211.7343*, 2012.

[73] G. Kossinets and D. J. Watts, "Empirical analysis of an evolving social network," *Science*, vol. 311, no. 5757, pp. 88–90, 2006.

[74] S. Saganowski, P. Bródka, and P. Kazienko, "Influence of the dynamic social network timeframe type and size on the group evolution discovery," in *Proceedings of the International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, IEEE, 2012, pp. 679–683.

[75] M. Oliveira, A. Guerreiro, and J. Gama, "Dynamic communities in evolving customer networks: An analysis using landmark and sliding windows," *Social Network Analysis and Mining*, vol. 4, no. 1, pp. 1–19, 2014.

[76] J. Leskovec and C. Faloutsos, "Sampling from large graphs," in *Proceedings of the 12th ACM SIGKDD International conference on Knowledge Discovery and Data Mining*, 2006, pp. 631–636.

[77] S. H. Lee, P.-J. Kim, and H. Jeong, "Statistical properties of sampled networks," *Physical Review E*, vol. 73, no. 1, p. 016 102, 2006.

[78] A. S. Maiya and T. Y. Berger-Wolf, "Sampling community structure," in *Proceedings of the 19th International Conference on World Wide Web*, ACM, 2010, pp. 701–710.

[79] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: Densification laws, shrinking diameters and possible explanations," in *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, 2005, pp. 177–187.

[80] G. Lindner, C. L. Staudt, M. Hamann, H. Meyerhenke, and D. Wagner, "Structure-preserving sparsification of social networks," in *Proceedings of the IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, 2015, pp. 448–454.

[81] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software (TOMS)*, vol. 11, no. 1, pp. 37–57, 1985.

[82] C. C. Aggarwal, Y. Zhao, and S. Y. Philip, "Outlier detection in graph streams," in *Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE)*, 2011, pp. 399–409.

[83] S. Tabassum and J. Gama, "Sampling massive streaming call graphs," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016, pp. 923–928.

[84] N. K. Ahmed, F. Berchmans, J. Neville, and R. Kompella, "Time-based sampling of social network activity graphs," in *Proceedings of the 8th Workshop on Mining and Learning with Graphs*, ACM, 2010, pp. 1–9.

[85] N. K. Ahmed, J. Neville, and R. Kompella, "Space-efficient sampling from social activity streams," in *Proceedings of the 1st International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, ACM, 2012, pp. 53–60.

[86] ——, "Network sampling: From static to streaming graphs," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 8, no. 2, 7:1–7:56, 2014.

[87] C. E. Tsourakakis, U Kang, G. L. Miller, and C. Faloutsos, "Doulion: Counting triangles in massive graphs with a coin," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2009, pp. 837–846.

[88] N. K. Ahmed, N. Duffield, J. Neville, and R. Kompella, "Graph sample and hold: A framework for big-graph analytics," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2014, pp. 1446–1455.

[89] A. Zakrzewska and D. A. Bader, "Streaming graph sampling with size restrictions," in *Proceedings of the IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, 2017, pp. 282–290.

[90] C. C. Aggarwal, "On biased reservoir sampling in the presence of stream evolution," in *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB Endowment, 2006, pp. 607–618.

[91] A. Pavan, K. Tangwongsan, S. Tirthapura, and K.-L. Wu, "Counting and sampling triangles from a graph stream," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, pp. 1870–1881, 2013.

[92] H. Jowhari and M. Ghodsi, "New streaming algorithms for counting triangles in graphs," in *International Computing and Combinatorics Conference*, Springer, 2005, pp. 710–716.

[93] P. S. Efraimidis and P. G. Spirakis, "Weighted random sampling with a reservoir," *Information Processing Letters*, vol. 97, no. 5, pp. 181–185, 2006.

[94] H.-H. Chen, L. Gou, X. L. Zhang, and C. L. Giles, "Predicting recent links in foaf networks," in *Social Computing, Behavioral-Cultural Modeling and Prediction*, Springer, 2012, pp. 156–163.

[95] C. Cortes, D. Pregibon, and C. Volinsky, "Computational methods for dynamic graphs," *Journal of Computational and Graphical Statistics*, 2012.

[96] S. Hill, D. K. Agarwal, R. Bell, and C. Volinsky, "Building an effective representation for dynamic networks," *Journal of Computational and Graphical Statistics*, 2012.

[97] S.-Y. Chan, P. Hui, and K. Xu, "Community detection of time-varying mobile social networks," in *Complex Sciences*, Springer, 2009, pp. 1154–1159.

[98] G. Cormode, V. Shkapenyuk, D. Srivastava, and B. Xu, "Forward decay: A practical time decay model for streaming systems," in *Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE)*, 2009, pp. 138–149.

[99] A. Zakrzewska and D. A. Bader, "Aging data in dynamic graphs: A comparative study," in *Proceedings of the IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, 2016, pp. 1055–1062.

[100] S. P. Borgatti and D. S. Halgin, "Analyzing affiliation networks," *The Sage Handbook of Social Network Analysis*, pp. 417–433, 2011.

[101] D. Duan, Y. Li, R. Li, and Z. Lu, "Incremental k-clique clustering in dynamic social networks," *Artificial Intelligence Review*, vol. 38, no. 2, pp. 129–147, 2012.

[102] M. Takaffoli, F. Sangi, J. Fagnan, and O. R. Zaïane, "A framework for analyzing dynamic social networks," *Applications of Social network Analysis (ASNA)*, 2010.

[103] H. Kim, J. Tang, R. Anderson, and C. Mascolo, "Centrality prediction in dynamic human contact networks," *Computer Networks*, vol. 56, no. 3, pp. 983–996, 2012.

[104] T. Falkowski, A. Barth, and M. Spiliopoulou, "Studying community dynamics with an incremental graph mining algorithm," *Proceedings of the 14th Americas Conference on Information Systems (AMCIS)*, pp. 1–11, 2008.

[105] T. Falkowski, J. Bartelheimer, and M. Spiliopoulou, "Mining and visualizing the evolution of subgroups in social networks," in *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence*, 2006, pp. 52–58.

[106] T. Falkowski, J. Bartelheimer, and M. Spiliopoulou, "Community dynamics mining.," in *ECIS*, 2006, pp. 318–329.

[107] X. Zhao, A. Sala, C. Wilson, X. Wang, S. Gaito, H. Zheng, and B. Y. Zhao, "Multi-scale dynamics in a massive online social network," in *Proceedings of the ACM Conference on Internet Measurement Conference*, ACM, 2012, pp. 171–184.

[108] J. Leskovec, L. Backstrom, R. Kumar, and A. Tomkins, "Microscopic evolution of social networks," in *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2008, pp. 462–470.

[109] A. Zakrzewska and D. A. Bader, "A dynamic algorithm for local community detection in graphs," in *Proceedings of the IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, 2015, pp. 559–564.

[110] ——, "Tracking local communities in streaming graphs with a dynamic algorithm," *Social Network Analysis and Mining*, vol. 6, no. 1, p. 65, 2016.

[111] A. Zakrzewska, E. Nathan, J. Fairbanks, and D. A. Bader, "A local measure of community change in dynamic graphs," in *Proceedings of the IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, 2016, pp. 349–353.

[112] P. Hájek, "Basic fuzzy logic and bl-algebras," *Soft Computing*, vol. 2, no. 3, pp. 124–128, 1998.

[113] P. Hájek, L. Godo, and F. Esteva, "A complete many-valued logic with product-conjunction," *Archive for Mathematical Logic*, vol. 35, no. 3, pp. 191–208, 1996.