GRAPHLET BASED NETWORK ANALYSIS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Md Mahmudur Rahman

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2016

Purdue University

West Lafayette, Indiana

To the loving memory of my grandfather.

ACKNOWLEDGMENTS

First, I thank my advisor Dr. Mohammad Al Hasan for his guidance and continuous support during my Ph.D. program. His advices and consistent encouragement have made this journey both enlightening and delightful. He introduced me to the research in graph mining and graphlet analysis, which opened up a plethora of research possibilities in front of me. He took the responsibility upon himself to teach me how to critically think about a research problem, how to design experiments to validate the contribution and finally, how to use written words to present the research contribution. I also like to thank him for giving me freedom to explore and choose research problems which appeal to my interest and match my skill-set.

Next, I also thank Mansurul Alam Bhuiyan, an inspirational figure of our research group. Through out the last six years, his dedication and hard work has been pivotal. We collaborated in several research projects; in all of them he has been instrumental in idea generation, validation and development. His unique ability to organize large volume of source code made the project development, a smooth and delightful endeavor. Most importantly, he was always there to help.

My special thanks and gratitude goes to Dr. Jennifer Neville, Dr. Alex Pothen and Dr. Dan Goldwasser for their guidance, encouragement and suggestions. I like to thank Dr. Naveen Ramakrishnan from Bosch, for being an ideal mentor when I was an intern at Bosch research lab in year 2014. Under his mentor-ship I got the resources and independence to explore several distributed and parallel frameworks for scalable network analysis. A special note of appreciation goes to Dr. Faizan Javed form CareerBuilder, for being an inspirational mentor when I was an intern at their lab in year 2015. He introduced me to a rich network data-set and several research problems associated with such a data-set. He gave me abundant freedom to choose research problems which matched my interest.

The acknowledgement will be incomplete without mentioning faculty members of CS, IUPUI from whom I learned a lot. I specially remember Dr. Yuni Xia, for teaching me algorithms in database systems and data mining methodologies. Beside being an inspirational teacher to me, she served in my qualifier exam committee, preliminary exam committee and final dissertation exam committee. I thank her for her selfless and spontaneous involvement in every step of my journey to Ph.D. graduation. I also like to thank Dr. Rajeev R. Raje for teaching me theory of programming languages and distributed computing. I am also specially grateful to the departmental staff, Nicole, Joan and Katherine, for their continuous and whole hearted effort to make us feel right at home in the department.

Finally, I express my gratitude to all my family members. Specially, my parents, my younger brother, my younger sister and my wife; all of whom have been sources of inspiration to me in every step of my life. My parents' encouragement has been my greatest strength which enabled me to keep pursuing the hardest of objectives in the harshest of times. I like to remember the companionship of my wife Amrin and thank her for continuous patience and love in these stressful times.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

Table Page

Table Page

LIST OF FIGURES

Figure Page

ABSTRACT

Rahman, Md Mahmudur PhD, Purdue University, December 2016. Graphlet Based Network Analysis. Major Professor: Dr Mohammad Al Hasan.

The majority of the existing works on network analysis, study properties that are related to the global topology of a network. Examples of such properties include diameter, power-law exponent, and spectra of graph Laplacians. Such works enhance our understanding of real-life networks, or enable us to generate synthetic graphs with real-life graph properties. However, many of the existing problems on networks require the study of local topological structures of a network. Graphlets which are induced small subgraphs capture the local topological structure of a network effectively. They are becoming increasingly popular for characterizing large networks in recent years.

Graphlet based network analysis can vary based on the types of topological structures considered and the kinds of analysis tasks. For example, one of the most popular and early graphlet analyses is based on triples (triangles or paths of length two). Graphlet analysis based on cycles and cliques are also explored in several recent works. Another more comprehensive class of graphlet analysis methods works with graphlets of specific sizes—graphlets with three, four or five nodes ($\{3, 4, 5\}$-*Graphlets*) are particularly popular. For all the above analysis tasks, excessive computational cost is a major challenge, which becomes severe for analyzing large networks with millions of vertices. To overcome this challenge, effective methodologies are in urgent need. Furthermore, the existence of efficient methods for graphlet analysis will encourage more works broadening the scope of graphlet analysis.

For graphlet counting, we propose edge iteration based methods (ExactTC and ExactGC) for efficiently computing triple and graphlet counts. The proposed methods compute local graphlet statistics in the neighborhood of each edge in the network

and then aggregate the local statistics to give the global characterization (transitivity, graphlet frequency distribution (GFD), etc) of the network. Scalability of the proposed methods is further improved by iterating over a sampled set of edges and estimating the triangle count (ApproxTC) and graphlet count (Graft) by approximate rescaling of the aggregated statistics. The independence of local feature vector construction corresponding to each edge makes the methods embarrassingly parallelizable. We show this by giving a parallel edge iteration method ParApproxTC for triangle counting.

For graphlet sampling, we propose Markov Chain Monte Carlo (MCMC) sampling based methods for triple and graphlet analysis. Proposed triple analysis methods, Vertex-MCMC and Triple-MCMC, estimate triangle count and network transitivity. Vertex-MCMC samples triples in two steps. First, the method selects a node (using the MCMC method) with probability proportional to the number of triples of which the node is a center. Then Vertex-MCMC samples uniformly from the triples centered by the selected node. The method Triple-MCMC samples triples by performing a MCMC walk in a triple sample space. Triple sample space consists of all the possible triples in a network. MCMC method performs triple sampling by walking form one triple to one of its neighboring triples in the triple space. We design the triple space in such a way that two triples are neighbors only if they share exactly two nodes. The proposed triple sampling algorithms Vertex-MCMC and Triple-MCMC are able to sample triples from any arbitrary distribution, as long as the weight of each triple is locally computable.

The proposed methods are able to sample triples without the knowledge of the complete network structure. Information regarding only the local neighborhood structure of currently observed node or triple are enough to walk to the next node or triple. This gives the proposed methods a significant advantage: the capability to sample triples from networks that have restricted access, on which a direct sampling based method is simply not applicable. The proposed methods are also suitable for dynamic and large networks. Similar to the concept of Triple-MCMC, we propose Guise for sampling graphlets of sizes three, four and five ($\{3, 4, 5\}$-*Graphlets*). GUISE samples graphlets, by performing a MCMC walk on a graphlet sample space, containing all the graphlets of sizes three, four and five in the network.

Despite the proven utility of graphlets in static network analysis, works harnessing the ability of graphlets for dynamic network analysis are yet to come. Dynamic networks contain additional time information for their edges. With time, the topological structure of a dynamic network changes—edges can appear, disappear and reappear over time. In this direction, predicting the link state of a network at a future time, given a collection of link states at earlier times, is an important task with many real-life applications. In the existing literature, this task is known as link prediction in dynamic networks. Performing this task is more difficult than its counterpart in static networks because an effective feature representation of node-pair instances for the case of a dynamic network is hard to obtain.

We design a novel graphlet transition based feature embedding for node-pair instances of a dynamic network. Our proposed method GRATFEL, uses automatic feature learning methodologies on such graphlet transition based features to give a low-dimensional feature embedding of unlabeled node-pair instances. The feature learning task is modeled as an optimal coding task where the objective is to minimize the reconstruction error. GRATFEL solves this optimization task by using a gradient descent method. We validate the effectiveness of the learned optimal feature embedding by utilizing it for link prediction in real-life dynamic networks. Specifically, we show that GRATFEL, which uses the extracted feature embedding of graphlet transition events, outperforms existing methods that use well-known link prediction features.

1 INTRODUCTION

Graph (or network) is a form of data representation that is used to capture and represent complex relations in a wide variety of disciplines, including social science [1], system sciences [2], and bioinformatics [3]. The relationship (derived by the underlying phenomena), can be modeled by analyzing the structure of network. Decades of works in this direction led to the discovery of various non-random properties of real life networks, such as, scale-free-ness by Barabasi et al. [4], small diameter by Watts et al. [5], and graph densification with shrinking diameter by Leskovec et al. [6]. Besides, scientists also invented various graph generation models [7] which help the development of synthetic graphs having metrics that are similar to those of real-life networks; for example, Erdös-Rènyi model [8,9], Barabàsi-Albert (BA) model [4], and Watts and Strogatz model [5].

Existing works contribute to our general understanding of the structure of a real-life network, but most of these works do not include small substructures in their analysis. Thus, they fall short in providing the sketch of the topological building blocks that are prevalent in a network. Apparently, the knowledge of topological building blocks can be crucial for solving some of the most important tasks involving networks—examples include identification of anomalous nodes for detecting threat [10], discovery of hidden groups [11], link prediction [12], and the mapping of a structural block to a functional unit [13]. Specifically, it is well known that, in biological networks small substructures play a unique role in carrying out the functionalities that are performed over the networks [14].

Some works exist that attempted to model the topological context of the nodes in a network by finding a fingerprint of the network that is based on small topological templates. Such a fingerprint encodes the "normal behavior", and is useful when solving tasks that aim to discover nodes or communities (a set of well-connected nodes) that reside in an unusual topological context. The simplest of such fingerprints can be the local count (frequency) of triangles incident to the nodes in the network [15]. Other motivation for obtaining triangle frequency is that it can be used for computing other metrics namely, *local clustering coefficient*, *global clustering coefficient* and *transitivity ratio* [5, 16]. In the social science, triangles have been used to identify various interesting behaviors [17, 18]. Triangle counting also helps identifying the common topics on the Web [19] and spam detection [20]. Being the simplest topological fingerprint, triangle counting has limited capability (power of expression) for effective analysis of networks.

A more complex and powerful example of such fingerprints that goes beyond triangle can be a frequency histogram of various topological structures that have more than three vertices. N. Przulj et al. [21] is the first to propose such a fingerprint for characterizing biological networks. In their works, they consider the complete set of graph topologies with three, four and five vertices and name them as graphlet [1] (for a preview of these graphlets, see Figure 1.1). For a given network, they count the frequencies of various graphlets in the network for designing a fingerprint that they call *graphlet frequency distribution* (GFD). Since then, graphlet frequencies have been used for comparing structures of different biological networks [3], characterizing biological networks using graphlet degree distribution [3], obtaining a structural to functional mapping for biological networks [13], and for relating various kinds of graphlets with different information cascades [23]. Frequencies of various graphlets have also been used for designing effective graph kernels [24, 25].

However, huge computation cost is typically the deterrent which prevents the popular adoption of graphlet frequencies for analyzing large graphs in this domain. In the few exceptions that we have, such as [23], [26] and [25], the high computation cost is dealt with a compromise that considers graphlets that have up to four vertices. N. Przulj's team built a software called GraphCrunch2 [27], which counts the frequencies of all graphlets with three, four, and five vertices. We attempt to use GraphCrunch2

[1] In [22], the term "graphlet" has also been used for describing wavelet decomposition of graphs, our work is not related to this definition.

for obtaining graphlet frequencies on some moderate-sized social network graphs; for the majority of these graphs, the attempt was *unsuccessful*. For instance, we ran GraphCrunch2 on roadNet-PA data set (1,088,092 vertices, 1,541,898 edges) and soc-sign-Slashdot081106 data set (77,357 vertices, 468,554 edges) for nearly 80 hours on all cores [2] of a quad-core, 2.1 GHz machine; neither of the processes finish more than 40% of the computation as reported on the software's status bar. Typical graphs in the social network domain have millions of vertices and millions of edges; the counting of various graphlet frequencies in such graphs will take months, if not years.

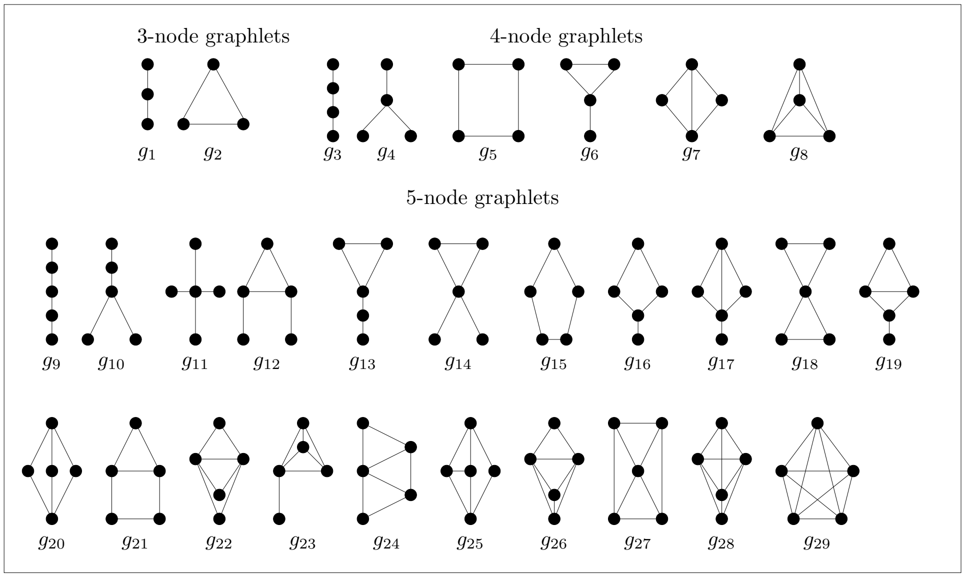


Figure 1.1.: All 3,4,5-node graphlets

1.1 Contribution of This Dissertation

In this dissertation, we propose edge iteration based algorithms for efficient counting of triples and graphlets. We also propose MCMC based methods for triple and graphlet sampling. Finally, we give a novel method for graphlet transition based feature embedding for node-pair instances in a dynamic network. In the following paragraphs, we give details of our contributions.

[2] GraphCrunch2 is a parallel library that uses all the available cores in a computer

First, for efficient graphlet counting we give edge iterator based method ExactGC[3]. The exact graphlet counting is obtained by aggregating the local graphlet counts of all the edges. Local graphlet count stands for the count of graphlets for which the edge is a part. But, estimated graphlet count is enough for many real life applications of network analysis. Inspired by the vast applicability, we adopt edge iterator based algorithms to obtain efficient estimation of graphlet count—thus further improving the scalability [28–30]. The estimation is achieved by sampling a subset of edges in the network and aggregating the local graphlet counts of those selected edges. The aggregated result is then appropriately normalized with respect to the relative sample size—giving estimated frequency of graphlet. We also propose a distributed solution for graphlet counting method using Spark [31], which further increases the scalability of the estimation method by using the power of distributed computation.

Secondly, we propose Markov Chain Monte Carlo (MCMC) sampling based methods for triple and graphlet analysis. Proposed triple sampling methods, Vertex-MCMC and Triple-MCMC [32], estimate triangle count and network transitivity. Vertex-MCMC samples triples in two steps: 1) sample a node using the MCMC method, 2) samples from the triples centered by the selected node in step 1. The method Triple-MCMC samples triples by performing an MCMC walk in a triple sample space. Triple sample space consists of all the possible triples in the network. MCMC method performs triple sampling by walking form one triple to one of its neighboring triples in the triple space. We design the triple space in such a way that two triples are neighbors only if they share exactly two nodes. Similar to the concept of Triple-MCMC, another proposed method Guise [33,34] samples graphlets ($\{3, 4, 5\}$-*Graphlets*), by performing an MCMC walk on a graphlet sample space, containing all the graphlets of sizes three, four and five in the network. The proposed MCMC methods are able to sample graphlets without the knowledge of the complete network structure. Information regarding only the local neighborhood structure of currently observed graphlet are enough to walk to the next graphlet. This gives the

[3] Similar discussion goes for triangle counting method ExactTC

proposed methods a significant advantage: the capability to sample graphlets from networks that have restricted access[4], on which a direct sampling based method is simply not applicable. The proposed methods are also suitable for dynamic and large networks.

Finally, this dissertation also investigates the potential of graphlet based analysis for dynamic networks. In dynamic network domain, a network changes the structure (by adding/removing nodes and/or edges) with time. We introduce the novel concept of Graphlet Transition Events (GTE) for feature representation of edges in a dynamic network [35]. GTE is defined as the transition of an existing graphlet of a network into another graphlet as a result of addition of one edge into that network. Our proposed method GraTFEL, uses automatic feature learning methodologies on GTE based features to give a low-dimensional feature embedding of unlabeled node-pair instances. The feature learning task is modeled as an optimal coding task where the objective is to minimize the reconstruction error. GraTFEL solves this optimization task by using a gradient descent method. We validate the effectiveness of the learned optimal feature embedding by utilizing it for link prediction in real-life dynamic networks. Specifically, we show that GraTFEL, which uses the extracted feature embedding of GTEs, outperforms existing methods that use well-known link prediction features.

1.2 Organization of The Dissertation

The bulk of the dissertation includes several research articles that we published over the period of my doctoral study. In Chapter 2, we discuss background materials for graphlet based network analysis. In Chapter 3, we discuss related works in triple and graphlet analysis. We also discuss related link prediction researches in static and dynamic network setups. Exact triangle and graphlet counting algorithms are presented in Chapter 4. The discussion is continued to Chapter 5 for

[4] A restricted access network has an access model, where one can not arbitrarily access a network. Beginning from an initial node, this model allows us to explore the network following the edges of currently visible nodes. The size of the set of currently visible nodes is limited (in our case it is 5).

presenting approximate solutions for triangle and graphlet counting. The main concept of multi-core triangle counting solution is published in the proceedings of IEEE International Conference on Big Data, 2013 [28]. Sequential graphlet counting algorithm Graft is published in the proceedings of the Conference of Information and Knowledge Management (CIKM), 2012 [30] and an extended version is published in the journal of IEEE Transactions on Knowledge and Data Engineering (TKDE), 2014 [29]. An ongoing work on distributed graphlet counting algorithms using Spark distributed computation framework is also presented in Chapters 4 (ExactSpark) and 5 (AppSpark).

The main concept of MCMC based triple sampling for estimation of triple based network metrics is published in proceedings of CIKM, 2014 [32]. It is described in Chapter 6. The MCMC based graphlet sampling method Guise for estimation of graphlet based metrics is presented in Chapter 7. This work is published in the proceedings of International Conference on Data Mining (ICDM), 2012 [33]. An extended version of Guise is also published in journal of Knowledge and Information Systems, 2014 [34].

The novel method of link prediction using graphlet transition event (GTE) is accepted for presentation in The European Conference on Machine Learning and Principles and Practice of Knowledge Discovery (ECML-PKDD), 2016 [35]. Application of graphlet analysis including the link prediction method GraTFEL is presented in Chapter 8. Future works and conclusion is discussed in Chapter 9.

2 BACKGROUND

In this chapter, we discuss the background materials for graphlet based analysis. We give preliminary discussion about static networks, restricted access network, triples, triple based metric *transitivity* in Sections 2.1-2.4. We also discuss induced subgraphs, graphlet, graphlet based metric GFD in Sections 2.5-2.7. Dynamic network is discussed in Section 2.8. We introduce graphlet transition event (GTE) in dynamic network setup in Section 2.9. The link prediction problem in dynamic network setup is defined in Section 2.10. Finally, we discuss Markov chain Monte Carlo (MCMC) sampling in Section 2.11.

2.1 Networks

A **Network (Graph)** is a combinatorial structure that represents a set of binary relations among a set of objects. The set of objects are called the vertices and the set of relations are called the edges. Let $G(V,E)$ is a graph, where $V = \{v_1, v_2, ..., v_n\}$ is the set of vertices and $E = \{e_1, e_2, ..., e_m\}$ is the set of edges. Each edge $e \in E$ can be denoted by a pair of vertices (v_i, v_j) where, $v_i, v_j \in V$. A graph without a self-loop or multi edge is a simple graph. In this dissertation, we consider simple and undirected graphs. We use the terms **Network** and **Graph** interchangeably. We use n to define the number of vertices in G, $d(v)$ to define the degree of a node v, and d_{max} to denote the maximum degree value for a vertex over the entire graph.

2.2 Restricted Access Networks

A restricted access network can be explored only by following edges of the network. Contrary to full access network, which allows to access adjacency list of an arbitrary node in the network; a restricted access network allows to access adjacency list(s) of currently observed node(s) in set $\mathcal{W}$ only. Which means, to check if an edge (u, v) is existent on a restricted network, at least one of the contributing nodes must be in $\mathcal{W}$ ($u \in \mathcal{W}$ or $v \in \mathcal{W}$). The size of the observing node set $|\mathcal{W}|$ is limited by a restriction parameter $G.ViewSize$. Restricted access network with $G.ViewSize = 5$ is sufficient to perform MCMC sampling for analysis of $\{3, 4, 5\}$-*Graphlets*. The nodes in $\mathcal{W}$ are normally connected and addition of a new node in the $\mathcal{W}$ generally means removal of another node–thus maintaining the size restriction for $\mathcal{W}$.

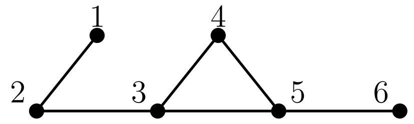
The motivation for a restricted network comes from real-life consideration. For example, Web graph is too big to be stored in memory/disk. But, crawling a Web graph is feasible by following the hyper-links (edges). Under this setting, we can sample a set of triples (from a uniform distribution) alongside crawling so that we can approximate the transitivity of the Web graph. Clearly, without storing the entire network, we have no knowledge of the number of vertices, or edges in this network, let alone the number of triples. Also, for a hidden network, access to an arbitrary node in the network is prohibited for security reason. The desired node can only be accessed from another node which is one-hop away from it.

2.3 Triples and Triangles

A triple (u, v, w) at a vertex v is a path of length two for which v is the center vertex. If the other two vertices (u and w) are also connected by an edge, the triple is called a closed triple (triangle), otherwise it is called an open triple. A triangle actually contains three closed triples, one centered on each of its vertices.

We use the symbol Π_v to represent the set of triples that are centered at the vertex v. The set of triples in a graph $G = (V, E)$ is Π, which is the union of the set of triples at each of its node, i.e., $\Pi = \bigcup_{v \in V} \Pi_v$. Based on whether the triple is open or closed (in terms of its induced embedding in the graph G), we can partition the set Π into $\Pi^{\angle}$ (open triples) and $\Pi^{\triangle}$ (closed triples). Note that, each of the nodes of a triangle in a graph G contributes one distinct triple in the set $\Pi^{\triangle}$. To represent the set of open and closed triples centered at a vertex v, we will use $\Pi_v^{\angle}$ and $\Pi_v^{\triangle}$, respectively. If $\delta(G)$ is the number of triangles in the graph G, then

$$\delta(G) = \frac{1}{3}|\Pi^{\triangle}| = \frac{1}{3}\sum_{v \in V} |\Pi_v^{\triangle}| \tag{2.1}$$



$\Pi^{\angle}$: $\{(1,2,3),(2,3,4),(2,3,5),(3,5,6),(4,5,6)\}$
$\Pi^{\triangle}$: $\{(3,4,5),(4,3,5),(3,5,4)\}$

Figure 2.1.: Open and closed triples in a graph

Example: Graph in Figure 2.1 has eight triples. Five of them are open and the remaining three are closed. Also the graph has exactly one triangle. ■

If the degree of each of the vertices in known, the total number of triples can be computed efficiently as below:

$$|\Pi| = \sum_{v \in V} |\Pi_v| = \sum_{v \in V} \binom{d(v)}{2} \tag{2.2}$$

Lemma 1 *Given a graph $G(V,E)$, the total number of triples in G is equals to $\sum_{v \in V} \binom{d(v)}{2}$*

PROOF*:*

A triple at a node v is a path of length two for which v is the center. For constructing such a path, we can select two different vertices from the adjacency list of v. There are $\binom{d(v)}{2}$ such choices. Therefore, by summing the partial triples over each $v \in V$, we get the total number of triples in G. ■

2.4 Transitivity

Newman, Watts and Strogatz [36] defined the transitivity of a graph G (say, $\gamma(G)$) as the fraction that represents the number of closed triples divided by the number of all the triples over the entire network.

$$\gamma(G) = \frac{|\Pi^{\triangle}|}{|\Pi|} = \frac{|\Pi^{\triangle}|}{|\Pi^{\angle}| + |\Pi^{\triangle}|} \tag{2.3}$$

2.5 Induced Sub-Graphs

A graph $G' = (V', E')$ is a **subgraph** of G if $V' \subseteq V$ and $E' \subseteq E$. A graph $G' = (V', E')$ is a **vertex-induced subgraph** of G if $V' \subseteq V$ and $E' \subseteq E$ and $\{e = (v_a, v_b) : v_a, v_b \in V', e \in E, e \notin E'\} = \phi$. A vertex-induced subgraph is a subset of the vertices of a graph G together with any edges whose both endpoints are in this subset. In this dissertation we will refer to vertex-induced subgraph as "induced subgraph". Two graphs G and G' are **isomorphic**, denoted by $G \cong G'$, if there exists a structure-preserving (both adjacency and non-adjacency preserving) bijection $f : V_G \to V_{G'}$; such a function f is called an isomorphism from G to G'. An **embedding** of a graph G' in another graph G is a subgraph S of G, such that S and G' are isomorphic; when the subgraph S is a vertex-induced subgraph of G, the embedding is called an **induced embedding**.

Example: In Figure 2.2, graph (b) is a subgraph, but not an induced subgraph, of the graph (a); On the other hand, graph (c) is an induced subgraph of the graph (a). The induced-subgraph consisting with the vertices $\{4, 5, 6, 7\}$ in the graph (a) is an induced embedding of the graph (c). ■

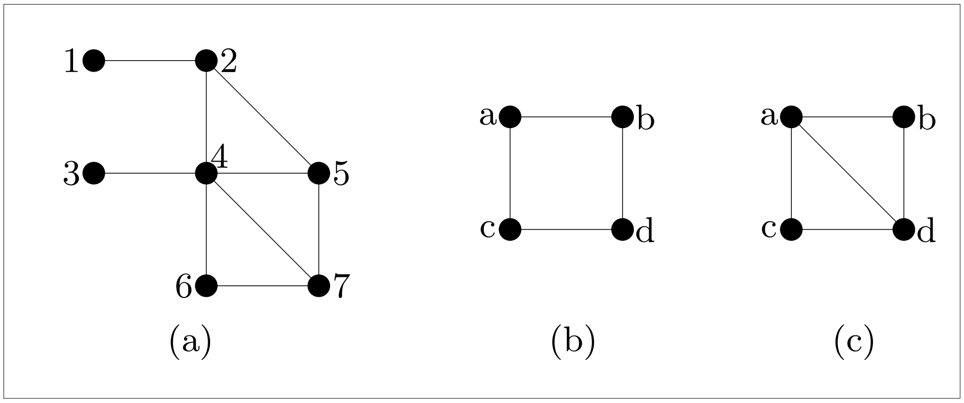


Figure 2.2.: Sub-graph and vertex induced sub-graph

2.6 Graphlets

Graphlets can be defined as small, connected, non-isomorphic, induced sub-graphs of a large network. In this dissertation, we work with all possible graphlets having k vertices; where, $k \in \{3, 4, 5\}$. We refer a graphlet with k vertices, as k-*Graphlet*; Note that, 1-*Graphlet* is simply a vertex, and 2-*Graphlet* is simply an edge. The frequency of a graphlet g_i in a graph G is the total number of distinct embedding of that graphlet g_i in the graph G. For triple analysis we use only 3-*Graphlet*s (g_1 and g_2 in Figure 1.1)

Example: Figure 1.1 shows all the graphlets that have between 3 and 5 vertices; there are 29 graphlets in this set. They are referred as g_i or *type-i*, where i varies from 1 to 29. We denote the specific embedding of a *k-Graphlet* by a sorted arrangement of its vertex ids, i.e. $\langle id_1, id_2, \ldots, id_k \rangle$, where $id_1 < id_2 < \ldots < id_k$. In Figure 2.2(a) vertex ids 1, 2, 3, and 4 construct a distinct embedding of the graphlet g_3 and using the above notation we can write this embedding as $\langle 1, 2, 3, 4 \rangle$. The frequency of graphlet g_2 (triangle) in the same graph is 3. ■

2.7 Graphlet Frequency Distribution (GFD)

Graphlet Frequency Distribution (GFD) of a graph G is a vector that characterize the relative frequencies of various graphlets in the graph G. The frequencies of 5-*Graphlet* is much smaller than those of 4-*Graphlet*s or 3-*Graphlet*s, so, we compare the frequencies in GFD in a logarithm scale. To construct GFD, first we compute frequencies of all the graphlets of size 3, 4, and 5 (shown in Figure 1.1) by enumerating all their distinct induced embeddings. Let, $f(g_i)$ be the frequency of graphlet g_i where $i \in \{1 \dots 29\}$. Second, we calculate the normalized frequency of each $f(g_i)$ by dividing it with $\sum_{i=1}^{29} f(g_i)$, and take logarithm (10-base) of the normalized frequency. The resulting size 29 vector is called the GFD. It can happen that an input graph does not contain any embedding of one or more graphlets. Since $log\ (0)$ is undefined, we use the following definition of GFD, where we add 1 to each of the frequencies; thus each entry in the GFD becomes,

$$GFD_i = log\left(\frac{f(g_i) + 1}{\sum_{j=1}^{29} f(g_j) + 29}\right); \forall 1 \le i \le 29 \tag{2.4}$$

Example: In the graph in Figure 2.2(a), the frequency of each type of graphlets are $\langle 11, 3, 5, 3, 0, 6, 2, 0, 1, 2, 0, 2, 1, 2, 0, 0,\ 2, 0, 1, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0 \rangle$. Here, 11 is the count of g_1, 3 is the count of g_2, and so on. Using the process discussed above, the GFD of this graph is:

$\langle -0.8, -1.3, -1.1, -1.3, -1.9, -1, -1.4, -1.9, -1.6, -1.4,$
$-1.9, -1.4, -1.6, -1.4, -1.9, -1.9, -1.4, -1.9, -1.6,$
$-1.9, -1.9, -1.9, -1.9, -1.2, -1.9, -1.9, -1.9, -1.9, -1.9 \rangle$ ■

2.8 Dynamic Networks

Let $G(V, E)$ be an undirected network, where V is the set of nodes and E is the set of edges $e(u, v)$ such that $u, v \in V$. A dynamic network is represented as a sequence of snapshots $\mathbb{G} = \{G_1, G_2, \dots, G_t\}$, where t is the number of time stamps for which we have network snapshots and $G_i(V_i, E_i)$ is a network snapshot at time stamp $i : 1 \leq i \leq t$ (see Figure 2.3). In this dissertation, we assume that the vertex set remains the same across different snapshots, i.e., $V_1 = V_2 = \cdots = V_t = V$. However, the edges appear and disappear over different time stamps. We also assume that, in addition to the link information, no other attribute data for the nodes or edges are available. We use n to denote number of nodes ($|V|$), and m to denote all node-pairs $\binom{|V|}{2}$ in the network.

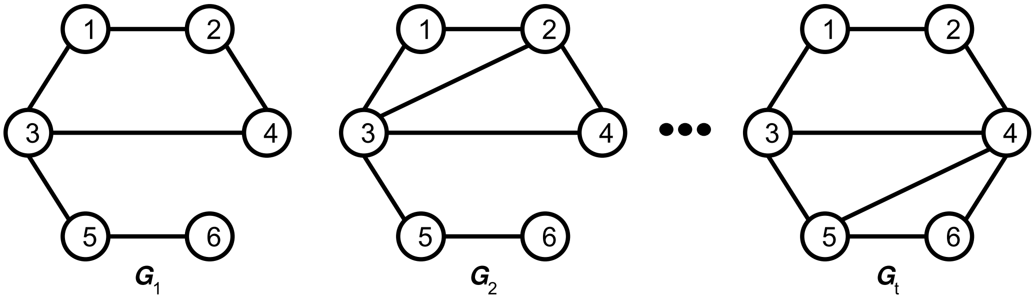


Figure 2.3.: A toy dynamic network with t snapshots. First two and last snapshots are given in this figure.

Example: Figure 2.3 shows an example dynamic network with t time stamps. There are six vertices in each time stamp but the number of edges changes across different time stamps. Here, $n = 6$ and $m = 15$. ■

2.9 Graphlet Transition Event (GTE)

Considering local topology around a node in a given network, the node can be a part of many different graphlets. Adding an edge in the local neighborhood of a node changes the neighborhood graphlet configuration of that node. To capture this we introduce a novel concept, namely **Graphlet Transition Event (GTE)**, which is an atomic event of a dynamic network. GTE is defined as the transition of an existing graphlet of a network into another graphlet as a result of addition of one edge into that network. For illustration, see Figure 2.4, three different GTEs are triggered as a result of adding the edge $(2,3)$ into the network G_1.

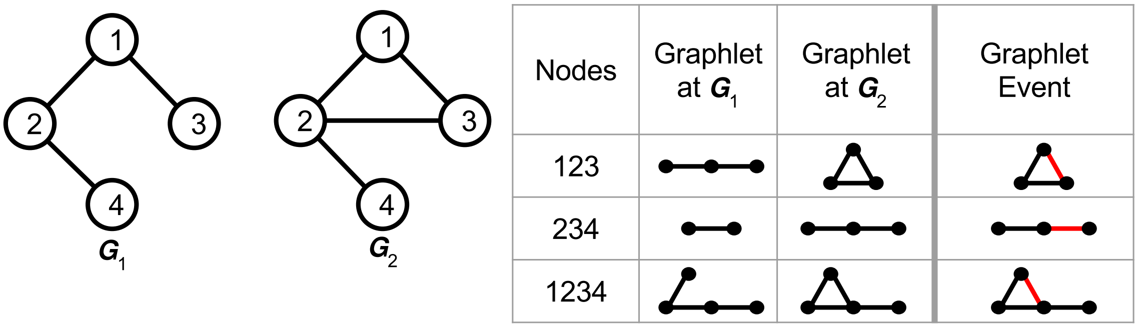


Figure 2.4.: A toy dynamic network. G_1 and G_2 are two snapshots of the network. Three different types of graphlet events are observed.

2.10 Link Prediction In Dynamic Network

Given a sequence of snapshots $\mathbb{G} = \{G_1, G_2, \dots, G_t\}$ of a network, and a pair of nodes u and v, the link prediction task on a dynamic network predicts whether u and v will have a link in G_{t+1}. Note that, we assume no link information regarding the snapshot G_{t+1} is available, except the fact that G_{t+1} contains the identical set of vertices. We use graphlet transition event as features for the link prediction task on a dynamic network.

2.11 Markov Chains and MCMC Sampling

Markov chain Monte Carlo (MCMC) sampling [37] is a class of algorithms for sampling from a desired probability distributions by constructing a Markov chain such that the stationary distribution of the Markov chain is the same as the desired distribution. The state of the chain after a suitable number of steps can be used as a sample of the desired distribution.

If we consider a random variable X that has a range of values (states) that are defined in a state space $\mathcal{S}$ and assume that X_t denotes the value(state) of X at time t(discrete). This random variable is called a **Markov process** if the transition probabilities between a pair of states in $\mathcal{S}$ depends only on the current value(state) of X. A **Markov Chain** is the sequence of Markov process over the state space $\mathcal{S}$. The transition probabilities can be expressed in a matrix, T, called *Transition Probability Matrix*. Each state in $\mathcal{S}$ occupies exactly one row and one column of T, in which the entry $T(i,j)$ is the probability of transition from state i to state j. For all $i,j \in \mathcal{S}$, we have $0 \leq T(i,j) \leq 1$, and $\sum_j T(i,j) = 1$.

A Markov chain is said to reach a stationary distribution π, when the probability of being in any particular state is independent of the initial condition. This scenario can be indicated by the condition

$$\pi = \pi T \tag{2.5}$$

π is a row vector of size $|T|$. Thus, the stationary distribution is the left eigen-vector of the matrix T with an eigenvalue of 1. We use $\pi(i)$ to denote the i'th component of this vector. A Markov chain in reversible if it satisfies the *reversibility condition* below:

$$\pi(i)T(i,j) = \pi(j)T(j,i), \forall i,j \in \mathcal{S} \tag{2.6}$$

The above condition is the sufficient condition for π to be a stationary distribution of the Markov chain. A Markov chain is ergodic if it has a stationary distribution.

2.11.1 Metropolis-Hastings (MH) Algorithm

MH algorithm is a variant of MCMC algorithm; its goal is to draw samples from some distribution $\pi(x)$, called the *target distribution*, where, $\pi(x) = f(x)/K$; here K is a normalizing constant which may not be known and difficult to compute. MH algorithm can be used together with a random walk to perform Markov Chain Monte

Carlo (MCMC) sampling. For this, the MH algorithm draws a sequence of samples from the target distribution as follows:

i It picks an initial state (say, x) satisfying $f(x) > 0$.

ii From current state x, it samples a state y using a distribution $q(x, y)$, referred as *proposal distribution.*

iii Then, it calculates the *acceptance probability* $\alpha(x, y)$ (Equation 2.7) and accepts the proposal move to y with probability $\alpha(x, y)$. The process continues until the Markov chain reaches to a stationary distribution.

$$\alpha(x, y) = \min\left(\frac{\pi(y)q(y, x)}{\pi(x)q(x, y)}, 1\right) = \min\left(\frac{f(y)q(y, x)}{f(x)q(x, y)}, 1\right) \tag{2.7}$$

3 RELATED WORKS

In this chapter, we discuss several research works that are relevant to the works presented in this dissertation. The related works are partitioned into three different sections: triple analysis, graphlet analysis and link prediction.

3.1 Triple Analysis

Being one of the most important building blocks of social networks, triples have been studied in many recent works. First introduced by Holland *et al.* [38], *transitivity* (also known as *clustering coefficient*) measures the degree to which nodes in a network tend to cluster together. The importance of transitivity and its obvious connection to the triangles in a network also contributed to the regained interest in the seemingly simple task of triangle counting. Since then, multitude of applications of triple analysis have been demonstrated. To mention a few, the distribution of triangles is used to uncover hidden thematic structure in the World Wide Web [19]. Triangle count is used for query plan optimization of database [39]. The distribution of the local number of triangles can be used to create successful spam filters and the same can also be used as features for assessing content quality in social networks [20]. Contrast between (degree) homogeneous triangles and heterogeneous triangles has been shown to be quite useful in characterizing networks [40]. They also give a quantifiable method for evaluating graph generation models.

3.1.1 Triangle Counting

The best practical algorithm for triangle counting in a network is an EdgeIterator algorithm. An EdgeIterator algorithm iterates over all edges and compares the neighborhoods of the two incident nodes. For an edge (u, w) the nodes $\{u, v, w\}$ induce a triangle if and only if node v is present in both neighborhoods $adj(u)$ and $adj(w)$. The asymptotic running time $\mathcal{O}(m^{3/2})$ is achieved by using hashed container for the neighborhoods [41]. Here, m is the number of edges in the network.

Although the best practical algorithm has good asymptotic running time $\mathcal{O}(m^{3/2})$, the best asymptotic solution is achieved by Alon *et al.* [42]. Alon *et al.* proposed a fast matrix multiplication based algorithm for finding and counting simple cycles of a given length $k \geq 3$. Triangle counting is a special case of proposed algorithm where $k = 3$. The time complexity of the algorithm is $\mathcal{O}(m^{\frac{2\gamma}{\gamma+1}})$, where γ is the cost of matrix multiplication. Asymptotic running time $\mathcal{O}(m^{1.41})$ is achieved by using best known matrix multiplication algorithm, for which the value of γ is 2.37. However, the methods that are based on matrix multiplication require large amount of memory, and hence they are not suitable for counting triangles in very large networks.

Distributed solution proposed by Suri and Vassilvitskii [43] elevates the scalability of triangle counting solution. Authors adapt any sequential triangle counting algorithm to the MapReduce setting. The algorithm creates overlapping partitions of the graphs so that each triangle is present in at least one of the partitions. The total amount of work spent on finding all of the triangles remains same as EdgeIterator algorithm, $\mathcal{O}(m^{3/2})$. There are simply more machines with each doing less work. The algorithm effectively takes any triangle counting algorithm that works on a single machine and distributes the computation. One weakness of the solution is that, the number of partitions can be very large for a large network. And many triangles can appear in multiple partitions; incurring redundant computational efforts.

3.1.2 Approximate Triangle Counting

In this section, we discuss approximate triangle counting algorithms. First, we discuss methods which use uniform sample of triples from a given network to estimate transitivity and triangle count. Secondly, we shade light on triangle counting methods which use trace of A^3 (here A is adjacency matrix). Third, we discuss network sparsification based triangle counting methods, which use edge sampling to construct a sparse sample of the given network. Theses methods performs exact triangle counting on the sparse network and normalize the count appropriately to obtain triangle count of the original network. Finally, we discuss methods which uses streaming model of network access to perform triangle counting in networks. Networks with streaming access model are observed (by algorithms) as streams of edges.

For many large networks, it is often more desirable to compute approximate triangle count using reasonable resources (memory and/or execution time). One of the earliest works for uniform triple sampling was proposed by Schank *et al.* [44]. The uniform distribution of sampled triples is achieved by first sampling the center vertex (of triple) from an appropriately weighted distribution and then uniformly sampling two neighboring vertices of the center vertex. After sampling a triple Schank *et al.* determine if the triple is open (path of size 2) or closed (triangle). The sampled triples are then used to approximate the transitivity (and triangle count) of the network. Similar methods are used by a more recent work for estimating many variants of clustering coefficients and for estimating triangle count [45, 46]

EigenTriangle, a linear algebraic method for triangle counting is proposed by Tsourakakis [47]. The proposed method iteratively computes eigenvalues λ_i of the adjacency matrix A of the given network in descending order of magnitude, and use them to approximate the triangle count ($\Delta = \frac{1}{6}\sum_{i=1}^{n}\lambda_i^3$). The algorithm terminates when the smallest eigenvalue contributes very little to the total number of triangles. Tsourakakis extends the core idea of proposed method EigenTriangle to give method FastSVD [48]. The method FastSVD first takes a projection A' of adjacency matrix A by sampling nodes with probability $\frac{d_i}{2m}$ where d_i is degree of node i and m is the number of edges in the network. The triangle count is then approximated using top singular values and corresponding singular vectors of matrix A'. FastSVD increases the scalability of EigenTriangle by taking projection of adjacency matrix of a large network.

As explained above, EigenTriangle approximates triangle count of a network by estimating the trace of A^3. Each iteration of EigenTriangle uses *Lanczos* method to find the next largest magnitude of eigenvalues. The limitations of the proposed method includes, the hardness to determine how many eigenvalues need to be computed and the absence of approximation guarantee. Avron [49] overcomes the issues, by estimating trace of A^3 using standard Monte-Carlo simulation for estimating the trace of implicit matrix A^3. Method proposed in [49] gives (ϵ, δ)-approximation guarantee with $\mathcal{O}(\epsilon^{-2} log(1/\delta)\rho(G)^2)$. Here, $\rho(G)$ a measure of the triangle sparsity is not necessarily small. Additionally, each sample requires $\mathcal{O}(m)$ time.

[50] proposes $DOULION$, which uses a probabilistic sparsification technique to obtain a sample of the original network (a more sparse network). The approximate triangle counting is obtained by extrapolating the exact triangle count of the sampled network. For a user defined p, $DOULION$ iterates over each of the edges of the input graph and sparsify it by removing an edge with a probability of $1-p$; then it executes any exact triangle counting algorithm on the sparse graph and divide the result by p^3 to approximate the triangle count in the original graph. Authors of this work also offer a Hadoop (An implementation of MapReduce [51]) based solution for this work. Another Hadoop based triangle counting algorithm is proposed by Pagh *et al.* [52], which uses random coloring of nodes to sample subgraph G' from a given network G. The algorithm first randomly colors each node with color $i \in \{1 \dots N\}$, here N is an integer. An edge is called *monochromatic* if both its endpoints have the same color. G' is constructed using only *monochromatic* edges. Then the algorithm executes any exact triangle counting algorithm on G' and multiply the result with N^2 to approximate the triangle count in the original graph. Degree-based vertex partitioning method proposed by Kolountzakis *et al.* [53], extends the method $DOULION$. This method partitions the set of vertices into a high degree and a low degree subsets and treats each set appropriately. The algorithm has running time $\mathcal{O}(m + \frac{m^{3/2} logn}{t\epsilon^2})$ and an $(1 \pm \epsilon)$ approximation.

All the triangle counting methods discussed above uses direct access networks. Direct access allows the algorithms to query the existence of an arbitrary edge in constant time. But, a growing set of real life networks can be accessed only as streams of edges. Buriol *et al.* [54] propose a collection of streaming algorithms for triple sampling and triangle counting approximation. Their proposed method, 3-pass-incident-stream, is actually similar to the sampling methods of [44–46]. Buriol *et al.* also consider another 3-pass method for arbitrary edge streaming; it samples triples by first sampling an edge, and then sampling a vertex, both uniformly. A triple that is obtained this way belongs to one of the following sets exclusively: disconnected triples (set T_1), connected open triples (set T_2), or triangles (set T_3). From the size of each of these sets, the authors find an approximation of the triangle count in a graph.

Three-pass algorithm proposed by Jowhari *et al.* [55] gives (ϵ, δ)-approximation[1] guaranty. At first pass of the algorithm a vertex u is sampled with probability $\binom{d_u}{2}/D$. Here, d_u is degree of vertex u and $D = \sum_{i \epsilon V} \binom{d_i}{2}$ is total number of connected triples in network $G(V, E)$. Then at second pass, the algorithm samples a pair of vertices (v, w) randomly and uniformly from the neighbors of u. The final pass checks if there exists an edge between v and w. A triple obtained this way belongs to one of the following sets exclusively: connected open triples (set T_2), or triangles (set T_3). A recent space efficient single pass algorithm proposed by Madhav *et al.* [56] uses *the birthday paradox*. Authors prove that, their algorithm requires $\mathcal{O}(\sqrt{n})$ space if the transitivity is constant and there is more edges than wedges. In another work, Madhav *et al.* [57] give an algorithm for estimating triangle count of multigraph stream of edges. Multigraph stream can have repeated appearance of edges in the stream.

None of the methods discussed in this section is applicable for restricted access networks (Section 2.2). In Chapter 6, we introduce MCMC sampling based methods for triangle count in restricted access networks.

[1] Let $\epsilon, \delta > 0$ and let Δ be the number of triangles. With probability at least $1 - \delta$, the algorithm outputs $\Delta^{'}$ such that $(1 - \epsilon)\Delta \leq \Delta^{'} \leq (1 + \epsilon)\Delta$.

3.2 Graphlet Analysis

The power of triple analysis is enhanced by generalizing analysis task for graphlets of size $k \in \{3, 4, 5\}$ ($k = 3$ for triple analysis). Introduced by Pržulj *et al.* [21], graphlets have been successfully used for a multitude of applications. Pržulj *et al.* [21] characterize biological graphs using graphlet frequency based distance metric. [13] characterizes local neighborhood structures of vertices in protein-protein interaction (PPI) network using graphlet frequency. Graphlet kernels proposed by [24] allow efficient network comparison mechanisms. Vacic *et al.* [25] proposes graphlet kernels for effective classification/clustering of nodes in protein structure networks.

3.2.1 Restricted Class of Graphlets

There exists a body of works performing graphlet analysis using only specific classes[2] of graphlets. Johnson [58] propose an algorithm to find all elementary/simple circuits of a directed network. Edges in a directed network has direction associated with them. A circuit is represented as an ordered list of nodes $c_u = (u = v_1, v_2, \ldots, v_k = u)$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i < k$. Note that first and last nodes of a circuit is same u. A circuit is elementary/simple if no node but the first and last appears twice. Work presented by Alon *et al.* [42] finds and counts simple cycles of given length for both directed and undirected networks. Akkoyunlu [59] presents maximal clique finding method on undirected networks. A clique is a completely connected subgraph. A clique is represented as a set of nodes $cl = \{v_1, v_2, \ldots, v_k\}$ such that $(u, v) \in E$ for $u, v \in cl$. Maximal clique is a clique which is not a subgraph of any other clique.

[2] For example circuits, cliques, max cliques etc.

3.2.2 $\{3, 4\}$-*Graphlets*

In this section we discuss graphlet analysis using graphlets of sizes 3 and 4 (3-*Graphlets*, 4-*Graphlets*). Lussier *et al.* [23] use graphlets with three and four vertices to characterize local structures of *information cascades* in large social networks. An *information cascade* is a phenomenon by which people influence others to acquire information or behaviors. They construct average graphlet frequency vector for each *information cascade*, using frequencies of graphlets of size three and four. Authors demonstrates the capability of such feature vector to capture evolution mechanism of *information cascades.*

Vacic *et al.* [25] proposed graphlet kernel based method for prediction of functional residues in protein structures. A protein structure can be represented as a network $G(V, E)$. Here, nodes in vertex-set V are *residues* (amino acid) and edges in edge-set E connect spatially neighboring *residues.* The *residues* can be either a functional residue or a not functional residue. Vacic *et al.* uses frequencies of graphlets of size two, three and four to construct the representation vector of local structure of a *residue* (node). A kernel function between two nodes is expressed as the inner product of their respective representation vectors and is used in a supervised learning framework to classify nodes (protein *residues*).

Efficient counting algorithms for 4-*Graphlets* is recently proposed by Marcus *et al.* [60]. Authors propose separate algorithms for counting induced graphlets 4-Clique (g_8), 4-Cycle with a chord (g_7), 4-Cycle (g_5), Tailed triangles (g_6), Claws (g_4) and Simple path of length three (g_3). Authors partitions the graphlet count by the node position in the graphlet. First they count non-induced graphlet using node position-aware graphlet counting. Then they calculate the count of induced graphlets using the non-induced count. A more recent work by Ahmed *et al.* [26], gives a parallel algorithm for counting 3-*Graphlets* and 4-*Graphlets*. The proposed method counts all *connect* and *disconnected* graphlets of size three and four. For each edge, authors count a selected set of small (mostly of size 3) graphlets. Graphlet counting task of an edge is dependent on only neighbors of the participating nodes. Which make is easy to parallelize the computation task. With these counts along with a number of proposed combinatorial arguments, authors obtain the exact counts of other graphlets in constant time.

3.2.3 5-*Graphlets*

Several recent works extend the graphlet analysis to incorporate graphlets of size five (5-*Graphlets*). Pržulj *et al.* [21] design graphlet frequency based fingerprint of a biological network. Authors use counts of all connected graphlets of size three, four and five ($\{3,4,5\}$-*Graphlets*) to give network comparison metric *relative graphlet frequency distance*, $D(G,G')$. The *relative graphlet frequency distance* $D(G,G')$, between two networks G and G' is defined as,

$$D(G,G') = \sum_{i=1}^{29} |F_i(G) - F_i(G')| \tag{3.1}$$

Here, $F_i(G) = -log(\frac{N_i(G)}{\sum_{i=1}^{29} N_i(G)})$, $N_i(G)$ represents the frequency of graphlet g_i in network G. Graphlet Frequency Distribution (GFD) (see Section 2.7) of a network is inspired by the definition of *relative graphlet frequency distance*.

In [13], the authors use the same set of 29 graphlets and introduce the concept of *automorphism orbits* to characterize local neighborhood structure of a node. Authors identify 73 different *automorphism orbits*, a node can touch. Thus, the signature vector of a node having 73 coordinates captures local network structure. Authors, demonstrate close relation between biological function of a node and its local network structure in a protein-protein interaction (PPI) network. Pržulj [3] defines Graphlet Degree Distribution (GDD) and similarity score *GDD Agreement*, for biological network comparison. Author generalizes the edge degree distribution which measures the number of nodes touching k edges, into distributions measuring the number of nodes touching k graphlets.

Shervashidze *et al.* [24] propose efficient graphlet kernels. The graphlet kernel $k_g(G, G')$ is defined as inner product of normalized graphlet count vector of two networks G and G',

$$k_g(G, G') = D(G)^T D(G') \tag{3.2}$$

Here, $D(G)$ is the normalized graphlet count vector of network G. This definition of graphlet kernel is different form the *relative graphlet frequency distance* defined by [21] (see Equation 3.1). [21] uses logarithm of the graphlet frequency to normalizes the frequencies of different graphlets, which can differ by several orders of magnitude. Shervashidze *et al.* [24] address the same issue by constructing graphlet kernels using graphlets of a constant size k. k can be either of sizes 3, 4 or 5 ($k \in \{3, 4, 5\}$). Also they consider both connected and disconnected graphlets while computing $k_g(G, G')$.

GraphCrunch-2 [27] is a graphlet based network analysis tool. Kuchaiev *et al.* [27] give efficient implementation of graphlet counting algorithms which parallelizes computationally intensive tasks to fully utilize the potential of modern multi-core CPUs.

3.3 Link Prediction

In this dissertation, we propose a novel use of graphlet analysis for performing link prediction task in dynamic networks. Since, its formal introduction to the data mining community by Liben-Nowell *et al.* [61] about a decade ago, link prediction problem has been studied extensively by many researchers from a diverse set of disciplines. Link prediction problems can be categorized into two classes, based on if the networks are static or dynamic. The topological structure of a static network remains constant across time, while the structure of a dynamic network changes with the progress of time.

3.3.1 Link Prediction in Static Networks

In this section, we discuss link prediction methods in static networks. First, we discuss topological feature based link prediction methods. Secondly, we present a nonparametric feature based method. Third, we shade light on a supervised link prediction method which maps the link prediction problem as a two class classification problem. Finally, we discuss a matrix factorization based link prediction method.

In their pioneering work [61] Liben-Nowell *et al.* introduce link prediction problem in static network setup. Authors develop a set of link prediction methods based on measures for analyzing the neighborhood of nodes in a network. The proposed link prediction methods utilize correlation between future link formation tendency of a node-pair and its topological proximity scores (like Common Neighbors, Jaccard's Coefficient, Adamic/Adar, Katz etc).

Miller *et al.* [62] propose a nonparametric latent feature model for link prediction in static networks. Authors assume that the directed network is observed as a $n \times n$ binary matrix A where, $A(i,j) = 1$ if an edge is observed from node i to node j and $A(i,j) = 0$ if absence of edge is observed. The position $A(i,j)$ is left unfilled if on observation is available. The proposed link prediction model learns a binary matrix $Z(n \times K)$ and a real-valued matrix $W(K \times K)$ to give a probabilistic link prediction model with likelihood, $Pr(A|Z,W)$. Here K is the number of features. An entry in binary matrix $Z(i,k) \in \{0,1\}$ indicates if node $i \in \{1 \dots n\}$ has feature $k \in \{1 \dots K\}$. Weight matrix W quantifies the effect of the features in link formation. The nonparametric method uses Indian Buffet Process (IBP) to choose appropriate number of features K.

Lichtenwalter *et al.* [63] propose a new perspective for link prediction problem. Authors categorize the topological proximity score based link prediction methods as unsupervised link prediction methods. They suggests that, if one accepts the basic premise that ground truth, whether a link forms or not, is available from prior incarnations of the network, there is no practical disadvantage to using a supervised framework. Moreover, using a supervised method (training classifier) in conjunction to a single unsupervised method can significantly improve the link prediction performance. For supervised link prediction method the data has to be in the format $(\overrightarrow{x}, y)$, such that $\overrightarrow{x}$ is a feature vector and $y \in \{0, 1\}$ is classification label. Authors assume that edges in the network have time information associated with them. Feature vector $\overrightarrow{x}$ is constructed using network structure before a time t_x and label y is constructed from network structure at time after t_x and before t_y. The method trains classification models using data available for edges observed in time window $[t_x, t_y]$ to predict links for unobserved node-pairs. Despite the availability of time information, I categorize this work as link prediction in static network, because both the training and prediction datasets are constructed from same time window.

A matrix factorization based method inspired by the success of collaborative filtering [64] is proposed by Menon *et al.* [65]. Analogous to [62], this work [65] assumes partial observation of the given network. Matrix factorization method effectively learns latent features using observed entries of the network. Using the latent features the method predicts links for unobserved node-pairs.

3.3.2 Link Prediction in Dynamic Networks

For many networks, additional temporal information, such as the time of link creation and deletion, is available over a time interval. For example, in an on-line social or a professional network, we may know the time when two persons have become friends; for collaboration events, such as, a group performance or a collaborative academic work, we can extract the time of the event from an event calendar. The networks built from such data can be represented by a *dynamic network*, which is a collection of temporal snapshots of the network. The link prediction task on such a network is defined as follows: *for a given pair of nodes, predict the link probability between the pair at time* $t+1$ *by training the model on the link information at times* $1, 2, \ldots, t$. We will refer this task as dynamic link prediction. Strictly speaking, this task should be called link forecasting as the learning model is not trained on partial observation of link instances at time $t+1$ (as was the case for link prediction proposed by Lichtenwalter *et al.* [63]); however, we refer it as link prediction due to the popular usages of this term in the data mining literature.

In the following paragraphs we discuss several link prediction methods. First, we discuss tensor factorization based link prediction methods. Secondly, we shade light on nonparametric link prediction methods. Third, we discuss deep learning based methods for dynamic link prediction. Finally, we discuss time series based link prediction methods.

Dunlavy *et al.* [66] propose tensor factorization based link prediction methods for dynamic networks. In this work, the dynamic network is represented as a three-dimensional tensor $\mathcal{Z}(n \times n \times t)$. Here, t is number of discrete time stamps available for the dynamic network. Proposed method uses CANDECOMP/PARAFAC (CP) [67] tensor decomposition method, which decomposes the given tensor into three component matrices. CP yields a highly interpretable factorization that includes a time dimension. Using factorized matrices, authors predicts links in future time stamp $t + 1$. The proposed method has the capability to predicts links in times beyond immediate $(t+1)$ future time stamp (i.e., $t+2$, $t+3$ etc). The method though originally deigned for bipartite networks, can easily be adopted for unipartite network setup.

A nonparametric link prediction method for dynamic network setup is proposed by Sarkar *et al.* [68]. The proposed model predicts links based on the features of its endpoints, as well as those of the local neighborhood around the endpoints. Given a node-pair, the proposed method predicts future link using Bernoulli distribution parameterized by local neighborhood of past snapshots of the network. For a node pair, authors use features *common neighbor* and *last appearance time of a link* to identify different types of neighborhood of the network. Once a neighborhood type is selected, statistical information collected from similar neighborhoods are used to compute the corresponding probabilistic function (Bernoulli distribution), which is then used for link prediction. The proposed method learns different models to predict links in different types of neighborhood.

Kevin *et al.* [69], use *stochastic blockmodel* for dynamic link prediction. Authors categorize nodes of a network into several groups (blocks) and generates edges with probabilities dependent on the group membership of participant nodes.

A deep learning based solution proposed by Li *et al.* [70] uses a collection of Restricted Boltzmann Machines with neighbor influence for link prediction in dynamic networks. Authors train a Conditional Temporal Restricted Boltzmann Machine (*ctRBM*) for each node in the network. Input to a *ctRBM* contains concatenation of adjacency lists of the corresponding node over a continuous window of past snapshots. This contains *temporal information* of past configuration of the network. Input also contains *neighbor feedback* from the expectation (output) of the local neighbors' *ctRBM*. The method needs to train n different *ctRBM*s, all of which gets input from *ctRBM*s of neighboring nodes.

Güneş *et al.* [71] propose time series based link prediction methods, which model history of similarity score of a node-pair as a time-series. Authors use variations of several popular topological feature based similarity scores like, Common Neighbors, Preferential Attachment, Adamin-Adar and Jaccard Coefficient. For a node-pair authors collect the scores for past time stamps. Then a powerful time series forecasting model ARIMA is used, in order to predict the future similarity scores. Link prediction is done using the predicted scores. Authors propose 16 different link prediction methods. The proposed methods trains separate ARIMA model for each node-pair, which makes the proposed methods embarrassingly parallel. Good surveys [12,72] on link prediction methods are available for interested readers.

4 GRAPHLET COUNTING ALGORITHMS

A fundamental task in network analysis is to count the frequency of various small subgraphs to discover network motifs—subgraphs that are significantly more frequent in a network relative to their occurrence in a randomized network of identical degree distribution [73]. Researchers have shown that network motifs are basic building blocks of different networks, including social networks, molecular interaction networks, and transportation networks [73–75]. To obtain effective algorithms for finding such motifs, researchers have developed a number of methods for counting the frequency of small subgraphs in a large networks [27, 76, 77]. Below, we discuss two problems, counting triangles and counting graphlets g_1-g_{29}.

Triangles

In recent years, researchers in the data mining community have shown an overwhelming interest in the problem of triangle counting [47–50, 52, 54, 55]. Although this is an old problem in graph theory, the renewed interest in this problem in mainly due to the fact that gigantic networks with millions of nodes and billions of edges are being available in recent years, on which the existing triangle counting methods perform poorly.

In the existing literature, the best practical algorithm for counting triangles exactly has a cost of $\mathcal{O}(m^{3/2})$, where m is the number of edges in the graph [41]. The algorithm iterates over the edges of the graph, and counts the number of triangles in which each of the edges contributes. Such an algorithm is known as EdgeIterator method. NodeIterator, a method which is dual to the EdgeIterator, iterates over the nodes and counts triangle in $\mathcal{O}(d_{max}^2 \cdot n)$ time, where n is the number of nodes, and d_{max} is the maximum degree of a node in the graph. However, in Chapter

5 we demonstrate the superiority of EdgeIterator method over NodeIterator method for triangle counting. Most of the existing methods belong to the EdgeIterator category.

The time complexity $\mathcal{O}(m^{3/2})$ of EdgeIterator method can be further improved by using matrix multiplication based methods. Alon et al. [42] gave such an algorithm that has a time complexity of $\mathcal{O}(m^{\frac{2\gamma}{\gamma+1}})$ where γ is the matrix multiplication cost; so the cost of this algorithm becomes $\mathcal{O}(m^{1.41})$, using the best known γ value which is 2.37 at present. In terms of time complexity, the fastest triangle counting methods are based on fast matrix multiplication. However, the methods that are based on matrix multiplication require large amount of memory, and hence they are not suitable for counting triangles in very large graphs. ■

Graphlets g_1-g_{29}

A natural generalization of triangle counting task is to consider subgraphs (graphlets) of larger size. In this chapter, we consider the task of counting graphlets g_1-g_{29} in a large network. The main motivation is building a fingerprint, called graphlet frequency distribution (GFD). To recap, GFD is a vector used to compare the frequencies of various graphlets for analyzing a large graph. Real-life networks are sparse, and in such networks the frequencies of larger-sized graphlets shrink in exponential proportion; hence, GFD uses logarithm scale for the frequency comparison so that the contribution of larger-sized graphlets are fairly accounted. Also, in constructing GFD, we limit the counting task for graphlets that have upto five vertices (shown in Figure 4.1). For justification of the restriction on graphlet size, we refer the reader to the Table 4.1; data from this table show that the number of possible (undirected) graphlets grows exponentially [1] with the number of vertices. The table also shows that there are 112 graphlets with six vertices in comparison to 21 graphlets with five vertices. With that many choices, the frequencies of most (except the line graphlet)

[1] growth is larger than the growth of a Fibonacci sequence

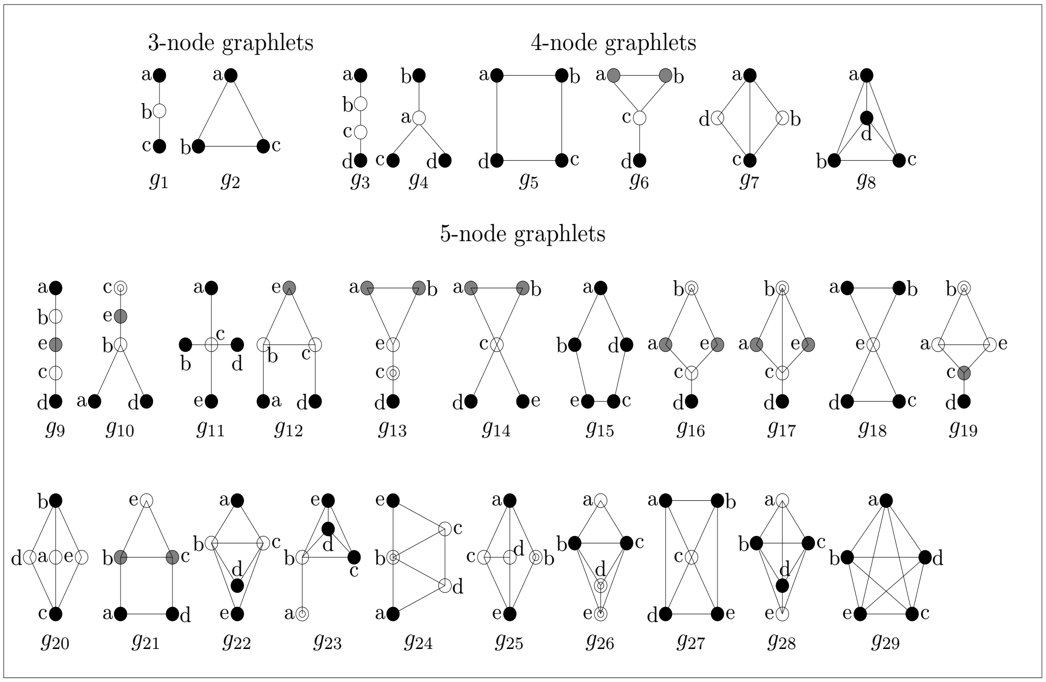


Figure 4.1.: All 3,4,5-node graphlets. Each vertex-orbit of a graphlet is represented by drawing the vertices of the orbit with same color (black, white and gray). Discussed in more details in Section 4.2.

Table 4.1.: Number of distinct graphlets (modulo isomorphism) with different number of vertices

Vertex Count	Graphlet Count
2	1
3	2
4	6
5	21
6	112
7	853

size-6 graphlets are, often, zero for a real-life graph and hence, are too un-reliable to be used for describing a phenomenon, or for obtaining a graph generation model. Needless to say that the cost of counting also rises in exponential proportion if we additionally consider the size 6 graphlets. Thus, the choice of constructing GFD using graphlets having upto five vertices is the best considering the cost-benefit tradeoff.

Computing graphlet frequencies of a large graph is a computationally expensive task. However, limiting the size of the graphlets makes the counting task polynomial; in fact, all induced embeddings of size-k graphlets in a graph $G(V, E)$ can be enumerated (and counted) in $\mathcal{O}(|V|^k)$ time by a brute-force search, which is prohibitively expensive for real-life networks having thousands of vertices, specifically for $k = 4$ or 5. However, in real-life large graphs are sparse, which helps designing graphlet counting algorithms that are significantly more efficient than a $\mathcal{O}(|V|^5)$ algorithm. The idea is to iterate over the vertices of a graph, and along that process enumerate the graphlet embeddings that are associated to each of the vertices. In recent year, a software named GraphCrunch [27] has been released, which follows this approach for exact counting of graphlets in biological networks. However, exact counting using EdgeIterator method though feasible for small graphs that arise in the domain of bioinformatics, such an approach may not be feasible for large graphs arising in the domains of social and information networks. For example, we ran GraphCrunch on Enronemail data set (38,692 vertices, 367,664 edges) and slashdot data set (77,357 vertices, 516,675 edges); neither of the counting processes finish after 5 days of running on a typical desktop computer.

Counting subgraphs from a large network is known as a computationally expensive task. Researchers attempted to develop algorithms for exact counting of specific classes of subgraphs, such as cycles [42, 58], cliques [59], and triangles [78]. Przulj *et al.* [21] describe how to count all induced subgraphs upto size 5 in PPI (Protein Protein Interaction) network using a nodeIterator method. They used the term *graphlet* to denote such subgraphs. Recently, [26, 60] present efficient counting algorithms for graphlets of size 4. Shervashidze *et al.* [24] provide methodology for counting all graphlets up-to size 5 in bounded degree graphs; main motivation of their work is to use graphlet counts to design efficient graph kernel. ■

Contributions

In this chapter, we propose two EdgeIterator algorithms. First method counts triangles of a large network. The proposed EdgeIterator method iterates over each edge of the given network. For each edge we count the number of triangles it participates in. The global triangle count of the network is obtained by aggregating the local triangle count of all edges. Second method follow a similar approach for computing the frequencies of graphlets of sizes $3-5$ in a large network. For each edge, the computational task is more complex and computationally intensive. The details is discussed in Section 4.3. Finally, we give a spark based distributed solution for graphlet counting. We explore the trade-offs between RDD generation and *Exploration* in order to get the best distributed solution.

Our work has the following contributions:

- We discuss an EdgeIterator based triangle counting method ExactTC [28] in Section 4.1.
- We propose an EdgeIterator based graphlet counting method ExactGC [29, 30] in Sections 4.2 and 4.3.
- We give a spark based distributed solution for graphlet counting in Section 4.4

4.1 Triangle Counting

Assume, $G(V, E)$ is a simple, connected, and undirected graph. We denote the adjacency list of a vertex $v \in V$ by $adj(v)$, which contains all the vertices that are adjacent to v. In our implementation, all the adjacent lists are sorted in the ascending order of the vertex-id. Since the graph is undirected, for an edge (u, v) u appears in v's adjacency list and vice-versa. A triangle is represented by a triple of (u, v, w), where $u, v, w \in V$ and there exists an edge between every pair of vertices in the triple.

An EdgeIterator algorithm iterates over each edge $e(v_i, v_j) \in E$ for counting the total number of triangles for which e is a participating edge. Let's call the number

Algorithm 1 ExactTC

Require: Large network $G(V, E)$

```
count = 0
for each edge $(v_i, v_j) \in E$ do
    $adj_1 = \{x | x \in adj(v_i), x > max(v_i, v_j)\}$
    $adj_2 = \{x | x \in adj(v_j), x > max(v_i, v_j)\}$
    $count_e = |intersection(adj_1, adj_2)|$
    $count+ = count_e$
end for
return count
```

of triangles incident to the edge e, the *partial triangle count* with respect to e, and represent it by $count_e$. Since a triangle is composed of 3 edges, a triangle will appear in exactly 3 of these partial counts. ExactTC can obtain the total count of triangles in G by simply adding the partial counts of all the edges followed by a division by 3. The triple counting of a triangle in the above method can be avoided by imposing a restriction that the third vertex of the triangle (say, v_k) has an id which is larger than the id's of both the vertices v_i and v_j of the edge e; this yields a more efficient version of ExactTC; a pseudo-code for which is shown in Algorithm 1. It computes the $count_e$ for an edge $e = (v_i, v_j)$ as follows: it takes the adjacency lists of the contributing vertices v_i ($adj(v_i)$) and v_j ($adj(v_j)$). Then, it finds the subsets adj_1 and adj_2 to ensure that the id of the possible third vertex (v_k) is strictly higher than the ids of both v_i and v_j, and then it finds the number of vertices that are common in both adj_1 and adj_2 i.e., $count_e = |intersection(adj_1, adj_2)|$. More than 50% of execution time can be saved using this method.

4.2 Graphlet Counting Preliminaries

In this section, we discuss some concepts and notations used for graphlet counting method. To recap, **graphlets** can be defined as small, connected, non-isomorphic, induced subgraphs of a large network. In this study, we work with all possible graphlets having k vertices; where, $k \in \{3, 4, 5\}$. We refer a graphlet with k vertices, as *k-Graphlet*; Note that, 1-*Graphlet* is simply a vertex, and 2-*Graphlet* is simply an edge. A *k-Graphlet* is called a **tree graphlet** if it is a tree, i.e., it has $k-1$ edges. A graphlet that is not a tree graphlet is called a **cyclic graphlet**.

Example: Figure 4.1 shows all the graphlets that have between 3 and 5 vertices; there are 29 graphlets in this set. They are referred as g_i, for i from 1 to 29. Among them, $g_1, g_3, g_4, g_9, g_{10}$, and g_{11} are tree graphlets, and the remaining are cyclic graphlets. For each graphlet, we identify each of its vertices by an English small letter, such as, a, b, c, etc. as shown in Figure 4.1. ■

The task of **Graphlets counting** over an input graph G is to find the counts of all distinct induced embedding of each of the graphlets having upto k vertices. To distinguish an embedding, we assign integer identifiers to each of the vertices in G, starting from 1 to $|V_G|$. Then an induced embedding of a *k-Graphlet* is denoted simply by a set of k vertex identifiers in the graph G, such that the subgraph induced by those vertices is isomorphic to that graphlet. Due to the “induced” constraint, at most one *k-Graphlet* is embedded in a given set of vertices of size k; same constrain also imposes that, if there exist two embeddings of the same graphlet in a graph, those embeddings should differ by at least one vertex. However, a subgraph of a size-k embedding may contain another graphlet of size smaller than k.

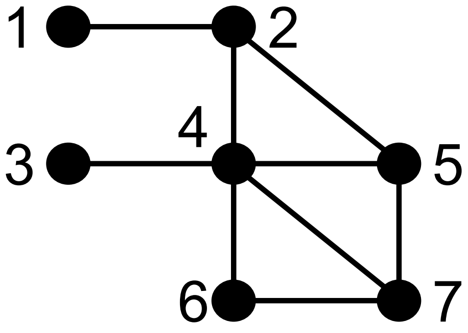


Figure 4.2.: Example network.

Example: In Figure 4.2, the induced-subgraph consisting with the nodes $\{1, 2, 4, 5\}$ of the example network is an induced embedding of g_6; no other graphlet of size 4 is embedded in the above set of vertices. Again, the vertex-set $\{2, 4, 5, 6, 7\}$ embeds only g_{24}; it does not embed g_{21} because of the induced restriction; in fact, the count of g_{21} is 0 in the example graph. Above example also confirms that a given set of k vertices embeds at most one *k-Graphlet*. ■

An isomorphism from a graph G to itself is called an **automorphism**. Thus, an automorphism π of a graph G is a structure-preserving permutation π_V on V_G along with a consistent permutation π_E on E_G. We may write $\pi = (\pi_V, \pi_E)$. For simple graphs, the permutation π_E is always consistent and is uniquely defined by π_V. The total number of automorphism of a graph is defined as $|\mathbf{Aut}(G)|$. Also, any permutation can be represented as a product of disjoint cycle, the vertices that belong to the same cycle under an automorphism form an equivalence class, which are called vertex-orbits. Similarly the equivalence classes of the edges are called edge-orbits.

Example: In Figure 4.1, each vertex-orbit of a graphlet is represented by drawing the vertices of the orbit by same color. For example, graphlet g_{14} has three vertex-orbits $(a, b), (c), (d, e)$ and three edge-orbits $(ab), (ac, bc), (cd, ce)$. Also, its automorphism count is 4. ■

4.3 Graphlet Counting

Proposed graphlet counting method ExactGC works as an EdgeIterator algorithm; where, the counting process iterates over the edges of the input graph, $G(V, E)$. For an edge $e \in E_G$, it finds the count of all induced embeddings of a graphlet g with the constraint that the edge e is part of the embeddings; we call this count a *partial count* of the graphlet with respect to the edge e. The partial count can be summed over all the edges to obtain a total count of the graphlet g in the input graph. However, in the above process, a distinct graphlet will be counted multiple times, by being accounted in different partial counts, so the above count needs to be corrected by dividing it with an appropriate normalization factor. Such a method yields an exact graphlet counting algorithm.

4.3.1 Partial Graphlet Count

We first discuss, how to obtain the partial count of a graphlet g that is associated with an edge e of the input graph. The first step for this task is to choose an specific edge e_g in the graphlet g, which will be aligned with the edge e in the large graph G. We will call the edge e_g the *first aligned edge* (FAE). Though, the choice of FAE can be arbitrary for exact counting, it is not the same for approximate counting; for the latter, a poor choice can drop the counting accuracy significantly. We discuss more on this in Chapter 5.

Once the FAE is chosen, the next task is to enumerate all the embeddings of a graphlet g with the constraint that in those embeddings, e_g is aligned with the edge e. The size of the set containing all the embeddings is the partial count of the graphlet g associated with the edge e. The enumeration process of the embeddings differs based on whether g is a tree graphlet or a cyclic graphlet.

Tree Graphlet Enumeration

Enumeration process is simpler for a tree graphlet. From Figure 4.1, there are one (g_1), two (g_3, g_4), and three (g_9, g_{10} and g_{11}) tree graphlets with 3, 4, and 5 vertices, respectively.

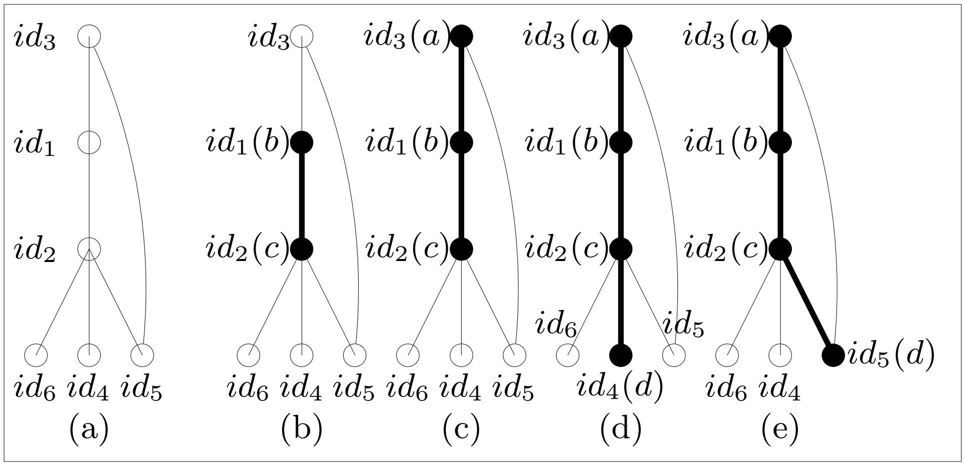


Figure 4.3.: Embedding tree graphlets g_3

We first explain the enumeration of g_3. Suppose, we are given the graph G shown in Figure 4.3(a) and we want to enumerate the embedding of graphlet g_3 in G. For this, ExactGC chooses the edge (b, c) of graphlet g_3 as FAE, and aligns it with the edge (id_1, id_2) of graph G (Figure 4.3(b)). Then step by step, it embeds the graphlet g_3 on the graph G over vertices $\{id_1, id_2, id_3$ and $id_4\}$ (From Figure 4.3(b) to 4.3(d)). At the end, we get an induced embedding of g_3 consisting with the vertices $\{id_1, id_2, id_3$ and $id_4\}$ (Figure 4.3(d)). By iterating over the adjacency lists of id_1 and id_2, ExactGC enumerates all possible embeddings of a and d. Note that, ExactGC only enumerates (counts) the induced embedding for g_3. For example, the embedding for g_3 consisting with the vertices $\{id_1, id_2, id_3$ and $id_5\}$ (Figure 4.3(e)) is not an induced embedding, and ExactGC will not enumerate it while counting the graphlet g_3.

Normalization: It is easy to see that $|\mathbf{Aut}(g_3)|$ is 2, as a chain (g_3 graphlet) can be embedded at most in two ways (forward and backward) over the same set of vertices in G. So, if there is no constraint on the embedding of g_3 we will count the same embedding of g_3 twice. To reduce duplication, ExactGC introduces restriction whenever possible. For example, edge (b, c) is mapped to edge (id_i, id_j), only if $id_i < id_j$. This constraint ensures that g_3 is mapped to an embedding only once; therefore, the $normalization\_factor$ for g_3 is 1.

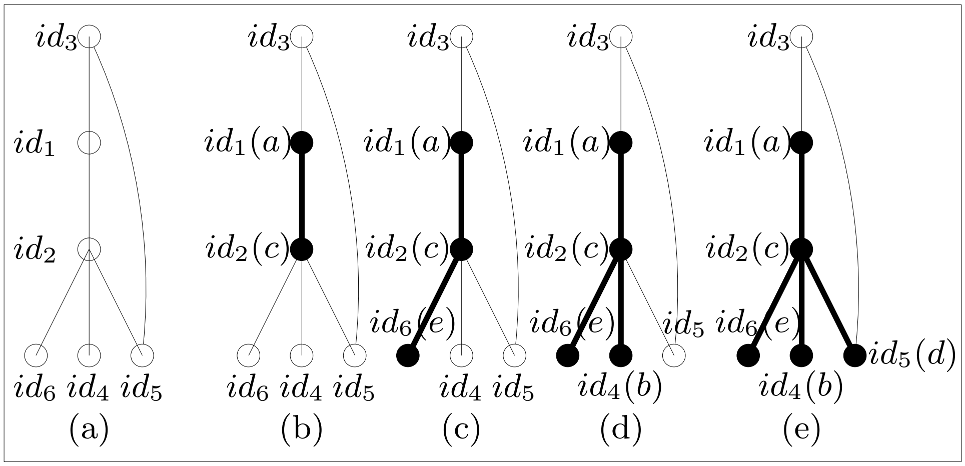


Figure 4.4.: Embedding tree graphlets g_{11}

Another example of tree graphlet enumeration, may be, the enumeration of g_{11}. Initially, ExactGC embeds the edge (a, c) of this graphlet with an edge (id_1, id_2) of the large graph (Figure 4.4). Then, it scans the adjacency list of vertex id_2 to find all possible mappings of vertices b, d and e of g_{11} to vertices id_4, id_5 and id_6 of the large graph. Thus, we get an embedding (not necessarily induced) of g_{11} in the large graph. Finally, ExactGC checks whether the embedding is induced or not. It only enumerates (counts) the induced embedding for g_{11}.

Normalization: The number of graph automorphism, $|\mathbf{Aut}(g_{11})|$, is $4! = 24$. So, for g_{11} we will have 24 repetitions of an embedding (if there is no constraint). If we apply the constraint that, mapping of vertices b, d and e of g_{11} to vertices id_4, id_5 and id_6 of the large graph is valid if and only if $id_4 < id_5 < id_6$, then we will have to deal with only 4 repetitions of an embedding. Therefore, the $normalization\_factor$ for g_{11} is 4.

Other tree graphlets (g_1, g_4, g_9 and g_{10}) can also be enumerated by following a similar mechanism.

Cyclic Graphlet Enumeration

For enumerating an embedding of a cyclic graphlet g, we can use one of the spanning trees of g; the specific spanning tree that is used for the generation is called a **generation tree graphlet**. A tree graphlet is it's own generation tree graphlet. Multiple graphlets can have the same generation tree graphlet (e.g., g_5 and g_6 both have g_3 as their generation tree graphlet). Also, for a cyclic graphlet, there can be multiple generation tree graphlets (e.g., g_{22} can have g_9, g_{10} and g_{11} as its generation tree graphlet).

To enumerate a cyclic graphlet, ExactGC initially embeds the generation tree graphlet (which is not induced). Then it checks explicitly whether the desired graphlet is induced in the embedding of it's generation tree graphlet.

For example, the generation tree graphlet of g_5 is g_3. So, in order to embed the graphlet g_5 in a large graph, g_3 must be embedded at the beginning. Figure 4.3(e) shows an embedding (not induced) of g_3 (consisting with the vertices $\{id_1, id_2, id_3$ and $id_5\}$), which gives us the induced embedding of g_5 since the edge (a, d) is induced by the existence of the edge (id_3, id_5), and no other edge (excluding the edges of tree graphlet) exists between a pair of vertices from the set $\{id_1, id_2, id_3, id_5\}$. The similar concept of normalization as tree graphlet enumeration applies here. If we apply the restriction to induce the edge (b, c) to edge (id_i, id_j) when the condition $id_i < idj$ satisfies, then we will have 4 duplications (using 4 left-rotations to align the edge (b, c) to different edges of the rectangle) of an embedding ($normalization\_factor$ is equal to 4).

Algorithm 2 ExactGC Algorithm

```
Require: Large network $G(V, E)$, Graphlet $g$
1: choose an specific edge $e_g$ in $g$
2: $count = 0$
3: for each edge $e \in E$ do
4:     align $e_g$ with $e$
5:     enumerate all induced embedding of $g$ in $G$, where $e$ and $e_g$ are aligned, $x$ is the total number of embedding found.
6:     $count = count + x$
7: end for
8: $count = count/normalization\_factor$
9: return $count$
```

4.3.2 Pseudo-Code

The outline of ExactGC is given in Algorithm 2. It accepts a graphlet g (it can be any of the graphlets shown in Figure 4.1) and a large graph G in which we like to count the occurrences of g.

ExactGC iterates over each of the selected edges sequentially (the **for** loop in Line 3). For an edge $e \in E$, it finds the count of all induced embedding of g in G with the constraint that those embeddings map one of the graphlet edge e_g to the edge,

e. (Line 4 and 5). As mentioned earlier, we call this the partial count. The partial count is accumulated sequentially as the method iterates over the edges in E (Line 6). The final count is then normalized appropriately to obtain the total count of the graphlet g in G (Line 8);

4.3.3 Joint Enumeration of Multiple Graphlets

As explained in Section 4.3.1, for counting a cyclic graphlet (say, g_x), ExactGC embeds the corresponding generation tree graphlet g_t followed by a validation to ensure that this embedding contributes to an induced embedding of g_x. If the validation step fails to find an induced embedding of g_x, then this specific embedding (not induced) of g_t does not contribute to the enumeration of the graphlet g_x. However, the embedding of g_t contributes to the induced embedding of some other graphlets, whose count should be incremented with this discovery. Therefore, if we count multiple graphlets (having the same generation tree graphlet) simultaneously, we will be able to share the workload of enumeration. Therefore, ExactGC's process of embedding graphlets having the same generation tree graphlet g_t is as below:

- Embed the generation tree graphlet g_t.
- Find the graphlet g_x whose induced embedding corresponds to this embedding of g_t.

The above optimization improves the execution time significantly, as every embedding of g_t contributes to the enumeration (count) of exactly one induced embedding of the graphlets.

4.4 Distributed Graphlet Counting

Recent advancement of distributed computational model has encouraged researchers to design scalable solutions for computationally demanding problems. Graphlet counting is no different. Suri and Vassilvitskii [43] adapt sequential triangle counting algorithms to MapReduce setting by partitioning the network into overlapping subsets. Tsourakakis *et al.* [50] propose a network sparsification based method Doulion which also uses distributed computational frame work MapReduce for triangle counting. A more recent algorithm proposed by Ahmed *et al.* [26] counts 3-*Graphlets* and 4-*Graphlets*. The proposed algorithm is claimed to be well-suited for distributed architecture.

In this section we propose a distributed solution of graphlet counting. The proposed solution counts graphlets of sizes $3, 4$ and 5 ($\{3, 4, 5\}$-*Graphlets*) and can easily be extended for graphlets beyond size 5. We use Spark distributed cluster computing system and resilient distributed dataset (RDD) provided by Spark to design and build an efficient distributed graphlet counting algorithm ExactSpark.

4.4.1 Spark Distributed Framework

Spark, a distributed framework technology developed by Apache foundation. Spark provides in-memory caching in distributed file system; this mechanism helps faster access to the data that resides in the file system. In addition to supporting four programming languages Scala, Python, Java and R; Spark also includes access to Machine leaning packages called MLib (Machine learning) and GraphX (for graph related problems). Spark supports the map-reduce model of programming and is built to support iterative and interactive computations. The notion of cashing and broadcasting in Spark enables data nodes to communicate efficiently during a computation.

Spark presents persistent Resilient Distributed Datasets (RDD), an efficient, expressive and fault tolerant abstraction for sharing the data in the cluster. RDD was built to support efficient iterative computation. Transformations like map, union, sample, join, groupByKey, flatMap and reduceByKey can be expressed in a few lines of code. Failure to a node can be easily recovered by re-executing the transformations of the failed RDD partitions.

4.4.2 Graphlet Counting by Enumeration

A popular method for graphlet counting is by enumerating the graphlet embeddings. After each enumeration, the statistics of the corresponding graphlet is updated. For ensuring the correctness and efficiency of the counting method, the enumeration process needs to have two properties. Firstly, every embedding has to be enumerated at least once. This ensures that all graphlet embeddings are accounted for; an algorithm with this property guaranties the completeness of the enumeration process. Second, every graphlet embedding has to be enumerated at most once. This property makes sure that the algorithm does not do any redundant work to gather graphlet statistics, thus enforcing better efficiency. An enumeration method with these two properties ensures that every graphlet embedding will be enumerated exactly once. The graphlet enumeration used by the graphlet counting method ExactGC presented in Algorithm 2, does not adhere to the second property. Most of the graphlets are enumerated multiple times, resulting an extra step to normalize the graphlet count (Line 8 of Algorithm 2). Both of these properties are preserved by the subgraph enumeration process proposed by Wernicke [77]. We adopt the method proposed by Wernicke, to give a sequential graphlet embedding enumeration algorithm SeqEnum(see Algorithm 3).

The algorithm starts by enumerating graphlets of size 1 (1-*Graphlets*) for all nodes $v \in V$. For each 1-*Graphlet* the algorithm computes the set of neighbor nodes, which can be added to enumerate graphlets of size 2 (2-*Graphlets*). The process is repeated to enumerate $(k+1)$-*Graphlets* from k-*Graphlets*. The neighbor nodes are computed in such a way that the desirable properties (a graphlet embedding is enumerated exactly once) of enumeration process is enforced. Details of the algorithm is discussed bellow:

The algorithm enumerates the graphlet in set *Graphlets* (line 2). The enumeration starts from each node of the network (lines 3 to 6). For each node the algorithm constructs an extension list $V_{Extension}$, using which further extension is possible. For

Algorithm 3 Sequential Graphlet Enumeration Algorithm

```
1: function SeqEnum(Large network $G(V, E)$)
2:     $Graphlets \leftarrow \phi$        ▷ $Graphlets$, set of graphlets of size $k$. Initially empty.
3:     for each vertex $v \in V$ do
4:         $V_{Extension} \leftarrow \{u \in N(\{v\}) : u > v\}$        ▷ $N(\{v\})$ is the set of neighbors of
   node-set $\{v\}$
5:         ExtendEmbedding($\{v\}, v, V_{Extension}, Graphlets$)
6:     end for
7:     return $Graphlets$
8: end function

9: function ExtendEmbedding($V_{Graphlet}$, $v$, $V_{Extension}$, $Graphlets$)        ▷
   $V_{Graphlet}$ is currently explored graphlet node-set, $v$ is the first node and $V_{Extension}$
   is Extension node-set
10:     if $|V_{Graphlet}| \in \{3, 4, 5\}$ then
11:         $Graphlets \leftarrow Graphlets \cup G[V_{Graphlet}]$
12:     end if
13:     if $|V_{Graphlet}| == 5$ then
14:         return
15:     end if
16:     while $V_{Extension} \neq \phi$ do
17:         Remove an arbitrarily chosen vertex $w$ from $V_{Extension}$
18:         $V'_{Extension} \leftarrow V_{Extension} \cup \{u \in N_{excl}(w, V_{Graphlet}) : u > v\}$        ▷ Here,
   $N_{excl}(w, V_{Graphlet}) = N(\{w\}) \setminus (V_{Graphlet} \cup N(V_{Graphlet}))$
19:         ExtendEmbedding($V_{Graphlet} \cup \{w\}, v, V'_{Extension}, k, Graphlets$)
20:     end while
21: end function
```

example, adding one node u from $V_{Extension}$ to the initial node v will give us a graphlet of size two $\{u, v\}$. A restriction $u > v$ imposed on the $V_{Extension}$ (Line 4) ensures the graphlet $\{u, v\}$ is constructed only once.

The sub function ExtendEmbedding is responsible to extent the current graphlet embedding $V_{Graphlet}$ by adding the nodes from $V_{Extension}$ (one at a time) (see Line 19). The function ExtendEmbedding adds the current embedding to $Graphlets$ if it is of sizes 3,4 or 5 (Lines 10-12). ExtendEmbedding returns if the current graphlet embedding is of size 5 (Line 13-15), thus terminating further extensions. Otherwise, the algorithm extends the current embedding $V_{Graphlet}$, by adding a new node w form the neighborhood $V_{Extensions}$ (Lines 16-20). Before recursively calling itself, the function ExtendEmbedding constructs the extension list $V'_{Extension}$ for the new extended graphlet embedding $V_{Graphlet} \cup \{w\}$. Note that the algorithm imposes restrictions on the construction of $V'_{Extension}$ (see Lines 17 and 18). These restrictions are enforcing the two properties; completeness and correctness.

4.4.3 Distributing The Enumeration

The sequential graphlet enumeration algorithm SeqEnum is a NodeIterator method. SeqEnum enumerates the graphlets in depth first search (DFS) order, i.e., if the initial vertex v is assigned nodeId v_a ($v = v_a$) in first iteration of for loop in Line 3, then all graphlets for which v_a is the smallest nodeId will be enumerated before any other graphlets in the network. On the other hand, from the perspective of a graphlet; a graphlet of size k (for $k > 1$) is expanded from a graphlet of size $k - 1$ by adding a neighboring node and its corresponding edges. This gives us a breadth first search (BFS) order exploration tree of graphlet enumeration (see Figure 4.5). In BFS order, all the graphlets of size k are enumerated first before any $k + 1$ graphlets.

While the algorithm SeqEnum is easily parallelizable, by forking new thread for each node in vertex set V (at Line 3). The workload distribution in each thread will not be uniform, as nodes with higher degree will be part of exponentially more graphlets than others. As a result some thread will work to enumerate a large number of graphlet embeddings while others will be left idle. The BFS exploration order of graphlet enumeration gives us an opportunity to repeatedly redistribute the workload of CPUs; by assigning equal number of *k-Graphlet* embeddings to each CPU. Once all the $(k + 1)$-*Graphlet* embeddings are enumerated by expanding *k-Graphlets*, we can again redistribute the workload by assigning equal number of $(k + 1)$-*Graphlet* embeddings to each CPU.

For distributing graphlet enumeration, we use Spark distributed computation framework. Initially we construct an RDD containing all 1-*Graphlet* embeddings.

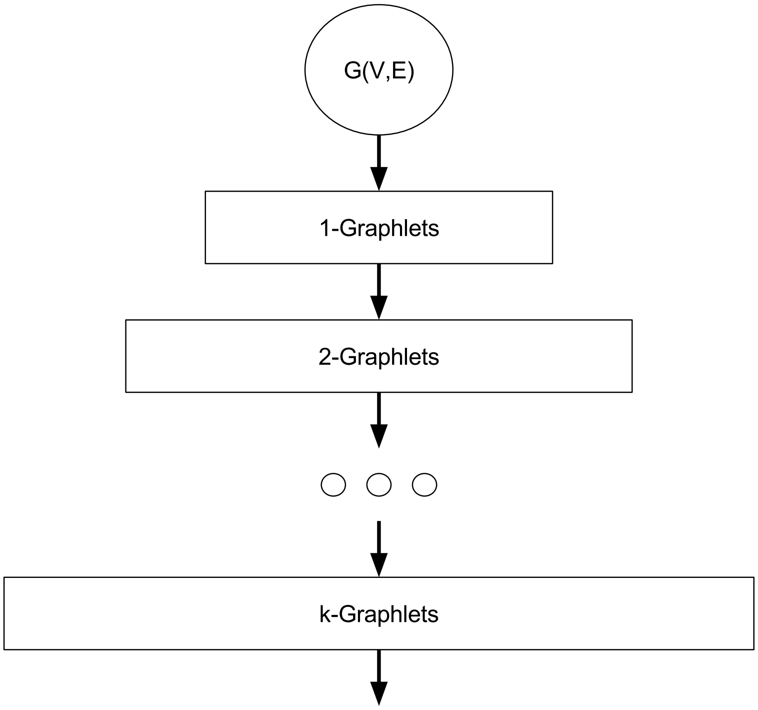


Figure 4.5.: BFS exploration of graphlet enumeration.

Each graphlet is a record in the RDD. To maintain the completeness and correctness of the enumeration method, we need to maintain the $V_{Extension}$ information for each graphlet embedding. We define graphlet record r_g, which contains all information necessary to identify a graphlet in embedding g and generate all valid extensions. r_g can be represented as tuple of size four, $r_g = (V, E, Neighbors, Extensions)$. Here, $r_g.V$ is the set of participating vertices of the graphlet, $r_g.E$ is the set of edges. The set of neighbors of all participating nodes is $r_g.Neighbors = N(r_g.V)$ and $r_g.Extensions$ is valid extensions (as $V_{Extension}$ used in algorithm SeqEnum).

Given a record for a graphlet of size k, the process of graphlet extension to get graphlet embeddings of size $k + 1$ can be divided into two stages: *extension* and *completion*.

Embedding Extension

For a record of k-*Graphlet* embedding r_g, the embedding extension stage enumerates all valid extensions to give records of $(k+1)$-*Graphlet* embeddings (see Algorithm 4). Initially the function EmbeddingExtension constructs an empty container $NewEmbeddings$ for gathering all graphlets extended from r_g (Line 2). Then in a loop (Lines 2-8) we iteratively extract neighboring nodes v from $r_g.Extensions$ to generate a valid extensions $r_g.V \cup \{v\}$ and collect them in $NewEmbeddings$. Finally, we return the list of all valid extensions in Line 9. An embedding can be uniquely identified by the corresponding set of vertices, but for graphlet type (g_1 to g_{29}) identification we need additional edge information. Records created in line 6, does not have all necessary information to do so. Because of the newly added node v the last three entries $(E, Neighbors, Extensions)$ of the record nr'_g needs reevaluation. The only additional information we need to do so is the adjacency list of last node v. The new records in $NewEmbeddings$ contains a tuple of size two (v, nr'_g).

Algorithm 4 Embedding Extension

```
 1: function EmbeddingExtension(Graphlet Embedding $r_g$)    ▷ $r_g$ = ($V$, $E$, $Neighbors$, $Extensions$)
 2:     $NewEmbeddings \leftarrow emptyList()$
 3:     while $r_g.Extensions.notEmpty()$ do
 4:         $v \leftarrow r_g.Extensions.top()$
 5:         $r_g.Extensions.remove(v)$
 6:         $nr'_g \leftarrow (r_g.V \cup \{v\}, r_g.E, r_g.Neighbors, r_g.Extensions)$
 7:         $NewEmbeddings.append(v, nr'_g)$
 8:     end while
 9:     return $NewEmbeddings$
10: end function
```

Embedding Completion

Embedding Completion stage is responsible for reevaluation of all the entries of newly created records (see Algorithm 5). At this stage all information necessary for graphlet type identification and further extensions of graphlet embedding are gathered using adjacency list $N(\{v\})$ of newly added node v. The function EmbeddingCompletion takes the tuple of form (v, nr'_g). First, we gather neighborhood information of vertex v (Line 2-3). Adj_v contains the complete adjacency list and $AdjFiltered_v$ is a restricted adjacency list containing all neighboring node of v whose id is larger than the smallest node in the embedding. $AdjFiltered_v$ is necessary to ensure that each graphlet is embedded only once. New edges of the embedding nr'_g is collected by intersecting neighborhood of v and all current nodes $nr'_g.V$ (Line 4). The reevaluated $Neighbors$ for the new embedding accounts for the neighborhood of v (Line 5). Finally the reevaluated $Extensions$ for the embedding is calculated by incorporating additional valid extensions possible from the last node v (Line 6). The function EmbeddingCompletion returns the completed graphlet embedding record (Line 7).

Algorithm 5 Embedding Completion

```
1: function EmbeddingCompletion($(v, nr'_g)$)
2:     $Adj_v \leftarrow N(\{v\})$
3:     $AdjFiltered_v \leftarrow \{x \in N(\{v\}) : x > nr'_g.V.min\}$
4:     $E \leftarrow nr'_g.E \cup \{e(v, v_2) : v_2 \in (Adj_v \cap nr'_g.V)\}$
5:     $Neighbors \leftarrow nr'_g.Neighbors \cup Adj_v$
6:     $Externsions \leftarrow nr'_g.Extensions \cup (adjFiltered_v - (nr'_g.Neighbors \cup nr'_g.V))$
7:     return $(nr'_g.V, E, Neighbors, Extensions)$
8: end function
```

4.4.4 RDD Generation vs Exploration

Exploring graphlet embeddings of size $k+1$ from graphlet embeddings of size k in breadth first search order, generates new RDDs for different size of graphlets. While this approach redistributes the computational burden equally to every CPU in the cluster; generation of multiple RDDs and their maintenance can be prohibitive for large networks. Memory requirement for multiple RDDs can be very high (specially as the number of embedding grows exponentially with the size of graphlets) and redistribution of graphlet embedding records requires a lot of communication across the cluster.

The sequential enumeration algorithm SeqEnum can be thought of as pure exploration method as no embedding is physically created, but explored in a sequential manner. The algorithm can finish the graphlet counting with out actually collecting the the graphlet embeddings. On the other extreme, the distributed solution proposed in Section 4.4.3 generated the embedding as records in RDD. This method makes the process distributed, but at the cost of higher memory requirement. In this section we propose a balance between the above two approaches. We generate RDDs for graphlets of sizes less than or equal to 3. And then we adopt the exploration approach from graphlet records of size 3, to obtain the counts of $\{4,5\}$-*Graphlets*. Since we are generating RDD for 3-*Graphlets*, the method retains good compatibility with distributed computational framework. Furthermore, the exploration approach triggered after size 3 graphlet embedding, allows graphlet counting for larger networks.

The exploration form graphlet embedding records of 3-*Graphlet* is detailed in Algorithm 6. The function HybridGraphletCounting takes a graphlet embedding record r_g and returns the record $glCount$ with counts of graphlet embeddings including and explored from r_g. First, $glCount$ is initialized to collect statistics of graphlets explored form the graphlet embedding r_g (Line 2). In Line 3, the index $glType(r_g)$ of $glCount$ is increased by one, thus counting the current graphlet embedding r_g. Function $glType(\cdot)$ returns the type of graphlet the graphlet embedding induces. Note that the information available in record r_g is sufficient for identifying graphlet types (g_1 to g_{29}). The while loop in Lines 4-21 expands current graphlet by adding nodes from the extension list $r_g.Extensions$. Extension done in Lines 5-7 is same as extension step in Algorithm 4. But, rather than creating a new RDD with this extended embeddings, we complete the records (Lines 8,9,13,14,15). If the extended graphlet is of size 5, which means the embedding will not be further extended; we increase its corresponding counter (Lines 10-11). Otherwise, we recursively call the

Algorithm 6 Hybrid Graphlet Counting

```
1: function HybridGraphletCounting($r_g$)
2:     $glCount \leftarrow map()$    ▷ $glCount$ contains the counts of different graphlet embeddings including and explored from $r_g$
3:     $glCount[glType(r_g)]++$
4:     while $r_g.Extensions.notEmpty()$ do
5:         $v \leftarrow r_g.Extensions.top()$
6:         $r_g.Extensions.remove(v)$
7:         $nr_g \leftarrow (r_g.V \cup \{v\}, r_g.E, r_g.Neighbors, r_g.Extensions)$
8:         $Adj_v \leftarrow N(\{v\})$
9:         $nr_g.E \leftarrow nr_g.E \cup \{e(v, v_2) : v_2 \in (Adj_v \cap nr_g.V)\}$
10:        if $|nr_g.V| == 5$ then
11:            $glCount[glType(nr_g)]++$
12:        else
13:            $AdjFiltered_v \leftarrow \{x \in N(\{v\}) : x > nr'_g.V.min\}$
14:            $nr_g.Neighbors \leftarrow nr_g.Neighbors \cup Adj_v$
15:            $nr_g.Externsions \leftarrow nr_g.Extensions \cup (adjFiltered_v - (nr_g.Neighbors \cup nr_g.V))$
16:            $glCountPartial \leftarrow$ HybridGraphletCounting($nr_g$)
17:            for $gl \leftarrow glCountPartial$ do
18:                $glCount[gl] \leftarrow glCount[gl] + glCountPartial[gl]$
19:            end for
20:        end if
21:    end while
22:    return $glCount$
23: end function
```

function HybridGraphletCounting with new embedding nr_g (Lines 12-20). The graphlet count statistics *glCount* is updated (Lines 17-19) to account for the partial graphlet count statistics *glCountPartial* returned by HybridGraphletCounting (Line 16). Finally, the function returns the record *glCount*.

The RDD generation for hybrid graphlet counting is illustrated in Figure 4.6. We generate RDDs for $\{1, 2, 3\}$-*Graphlets* following procedure in Section 4.4.3. Finally, an RDD is generated by executing HybridGraphletCounting for each records of 3-*Graphlets*. Later, the RDD with *glCount*s is aggregated to give final graphlet counts of the network. We call this distributed graphlet counting algorithm, ExactSpark.

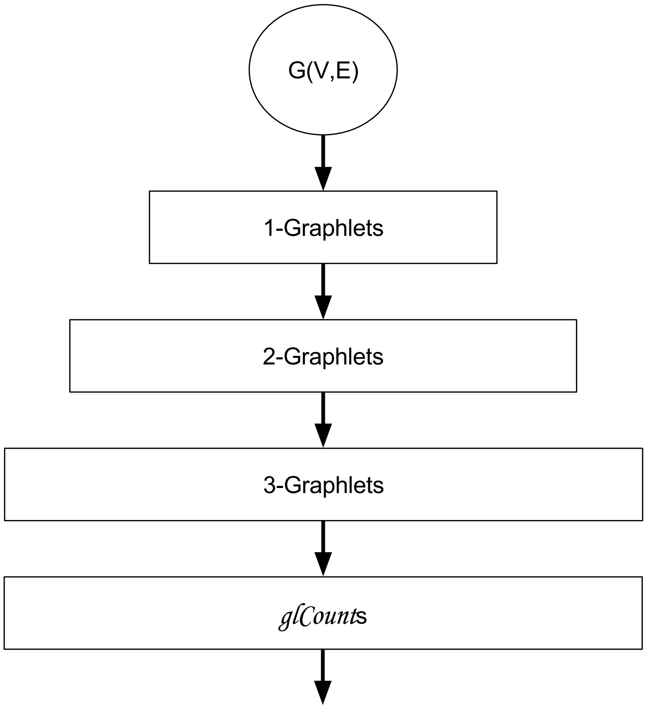


Figure 4.6.: RDD generation for hybrid graphlet counting using BFS exploration

4.5 Conclusion

In this chapter, we propose the counting methods for triangles and graphlets. We also introduce a novel distributed method of graphlet counting using Spark. All the methods discussed in this chapter perform exact computation of graphlet statistics. One of the most attractive properties shared by these methods is that, they can be easily adopted to give good estimation of graphlet statistics. In Chapter 5, we discuss the estimation methods of graphlet statistics. Additionally, we give comprehensive experimental evaluations and comparisons of exact and approximate graphlet counting algorithms in Chapter 5.

5 APPROXIMATE GRAPHLET COUNTING ALGORITHMS

For many applications, exact graphlet counting can be replaced by approximate graphlet counting for successful graphlet analysis. Approximate graphlet counting improves the scalability of graphlet based analysis in large networks. Several recent works [30, 33, 79] propose approximate graphlet counting algorithms.

To deal with the excessive computational cost of exact counting of graphical structures, researchers adopt approximate counting. Duke *et al.* [80] developed a method to approximate the frequencies of subgraphs in a given non-directed, labeled graph. Itzhack *et al.* [81] presented approximate counting algorithm for directed graphlets of size 3 and 4 based on node removal technique for network decomposition. Gonen *et al.* [82] presented an approximate counting algorithm for k-length paths, k-cycles and k-cycles with a chord graphlet. Another work [83] gave a sub-linear algorithm for approximating the count of non-induced star of any size.

For today's large network with millions of vertices and edges, all the exact methods for triangle counting can be deemed as expensive; so, the majority of the recent efforts of triangle counting either adopt a method for approximate counting, or design a parallel or distributed framework for solving the counting task effectively. For approximate counting, [50] proposes $DOULION$, which uses a probabilistic sparsification technique to obtain a sparser graph; then, it computes the exact triangle count on the sparse graph, from which it extrapolates an approximate triangle count of the original graph. For a user defined p, $DOULION$ iterates over each of the edges of the input graph and sparsify it by removing an edge with a probability of $1-p$; then it executes any exact triangle counting algorithm on the sparse graph and divide the result by p^3 to approximate the triangle count in the original graph. Though, $DOULION$ performs remarkably for counting triangles, our algorithm ApproxTC [28] gives a higher accuracy with the cost of higher execution time. Authors of $DOULION$ also offer a Hadoop (An implementation of MapReduce [51]) based solution for this work. Hadoop is also used in an exact triangle counting which is recently proposed by Suri and Vassilvitskii [43]. A linear algebraic method is also proposed for approximate triangle counting [47]. To the best of our knowledge, none of the works consider approximate counting of graphlets.

In this Chapter, we propose a variant of EdgeIterator method for approximate triangle counting. Our method has a surprisingly high accuracy, with a generous speedup. On large real-life graphs with millions of nodes and edges, the single processor version of our algorithm consistently achieve a 30-fold speedup (compared to the best exact method) with an accuracy that is around 99%. The most attractive feature of our method is that, both the speedup, and the accuracy of our method improve as the input graph becomes larger, so it is particularly suitable for very large graphs.

The simplicity of our triangle counting algorithm [28] also allows a simple multi-threaded implementation for executing on today's multi-core architecture—this improves the speedup even further without harming the counting accuracy. We indeed found that for all the real-life graphs that we encountered, the multi-core version of triangle counting algorithm that we propose is a better choice by a wide margin than all the Hadoop based solutions. For a specific example, on a Wikipedia graph with 1.63 millions vertices, and 18.5 millions of edges; using 32-threads our method obtains a whopping 837-fold speedup with an accuracy of 98.2%. None of the Hadoop based solution reports a speedup that is as high as this work.

Since, the cost of graphlet counting is much higher than the cost of triangle counting, an approximate counting algorithm for the former will be more useful from a practical standpoint. Also for many practical usages of graphlets, such as for the construction of GFD, approximate graphlet counting can be used in places of exact counting without any visible loss; although counting errors prevails by adopting an approximate counting, the effect of this error on GFD is negligible, because the latter compares the relative counts in a logarithm scale. Unfortunately, no algorithm exists that performs approximate counting of graphlets.

We propose an algorithm GRAFT, [1] [29,30] which performs approximate counting for $\{3, 4, 5\}$-*graphlets*. GRAFT samples a small number of edges uniformly and for each of the sampled edges it obtains a partial count for a graphlet such that the graphlet uses those sampled edges in its (induced) embedding; GRAFT then uses the partial count associated with each of the edges for approximating the total count of that graphlet. Experiments show that by sampling between 5% and 10% of the edges, we can easily achieve more than 95% of accuracy in counting the graphlet with a speedup factor between 20 and 10; for larger graphs, the sampling factor can be reduced to 1% (or less) to achieve similar accuracy and even higher speedup.

Finally, we propose a Spark based distributed graphlet counting algorithm. The proposed method uniformly samples from 3-*Graphlets* and for each of the sampled triple embeddings, it obtains a partial count of graphlets explored from and including the triple. The graphlet count is obtained by extrapolating the aggregated partial graphlet counts. Below we list our contributions.

- We propose a simple, yet powerful, method for approximate triangle counting in Section 5.1. It has surprisingly high accuracy and high speedup factor; both the metrics observably improve as the graph grows larger.

- We develop two variants of thread-based multiprogramming solutions of our approximate triangle counting algorithm in Section 5.1.1. The parallel implementation of these algorithms are simple, and effective.

- We compare the performance of our methods with those of the state-of-the-art approximate triangle counting methods that are available at present, on a collection of large real-life networks to validate the superiority of our methods (see Section 5.2).

[1] GRAFT is an anagram of the bold letters in **A**pproximate **GR**aphlet **F**requency coun**T**ing

- We propose Graft, a sampling based method for approximate counting of graphlets (Section 5.3). Our experiments show that this method provides significant speedup (in the range of 10 to 100 depending on graphs) for a counting accuracy of 95% on various real-life networks.

- Along with Graft,'s sampling, we propose various optimization schemes that improve the accuracy of graphlet counting (Section 5.3.2).

- We use Graft to obtain graphlet frequency distribution (GFD); our experiment with a time varying network shows that the GFD histogram preserves its shape across various temporal snapshots of the network; so GFD can be used as a signature for characterizing large networks. The comparison of GFD histograms between random graphs and power-law graphs also shows distinctive patterns to distinguish between these graph families (see Sections 5.4.5 to 5.4.7).

- We also propose AppSpark, a distributed method for approximate counting of graphlets (Section 5.5). We demonstrate the performance and scalability of the method using several large real-world networks (Section 5.6).

5.1 Triangle Counting

We give approximate triangle counting method ApproxTC by extending EdgeIterator based exact triangle counting method ExactTC discussed in Section 4.1. ApproxTC computes the partial count, $count_e$, only for a fraction of edges. For this, ApproxTC takes an additional parameter, a sample factor $p \in [0,1]$, which defines the fraction of edges in E for which we compute the partial count. A pseudo-code is shown in Algorithm 7. ApproxTC first chooses (Line 2) a set of p fraction of edges from the set E uniformly. Then, it computes the partial count of each of the chosen edges (Line 4 – 6). Finally, the sum of the partial count is divided by p to obtain an approximate count of triangles in the graph G (Line 9).

Algorithm 7 APPROXTC

```
Require: Large network $G(V, E)$, Sample factor $p$
 1: $count = 0$
 2: $E_p$ = sample $p * |E|$ edges from $E$ uniformly
 3: for each edge $(v_i, v_j) \in E_p$ do
 4:     $adj_1 = \{x | x \in adj(v_i), x > max(v_i, v_j)\}$
 5:     $adj_2 = \{x | x \in adj(v_j), x > max(v_i, v_j)\}$
 6:     $count_e = |intersection(adj_1, adj_2)|$
 7:     $count+ = count_e$
 8: end for
 9: $count = count/p$
10: return $count$
```

Table 5.1.: Execution time for EXACTTC. Average accuracy and speedup of APPROXTC and APPROXTC2. Here, $p = 0.1$. Speedup is with respect to EXACTTC . Statistics of the graphs are in Table 5.2.

Graph	EXACTTC	APPROXTC		APPROXTC2	
	time (sec)	Accuracy (%)	speedup	Accuracy (%)	speedup
Wiki-1	124	98.50	5.81	99.62	3.83
Wiki-2	273	99.57	5.12	99.86	3.33
Wiki-3	299	99.67	5.18	99.75	3.24
Wiki-4	361	99.75	4.66	99.93	3.12
Zewail	0.04	95.64	6.02	97.04	5.39
Flikr	38.92	99.57	9.54	99.76	6.02
EN	0.72	98.51	7.62	99.52	6.01
EAT RS	0.37	97.56	7.22	99.45	5.30

A second version of approximate counting algorithm can be obtained by lifting the constraint (imposed by Line $4-5$) that the id of the third vertex (v_k) of a triangle is larger than the ids of both v_i and v_j. As a consequence, we may end up counting each triangle at most thrice. So, the final count has to be normalized by $3 * p$ (at Line 9 of Algorithm 7). We will call this version, APPROXTC2. As expected, this version is slower than the earlier version of approximate counting algorithm as it performs more works when computing the partial counts by intersecting a pair of unfiltered adjacency lists. However, this version has better sampling performance, as the sampling space of this version is less restricted. So, ApproxTC2 typically achieves a better counting accuracy than ApproxTC. For a comparison, see Table 5.1.

5.1.1 Parallel Algorithm, ParApproxTC

In this section, we discuss the parallel version of the ApproxTC, which we call ParApproxTC; the idea for this is quite simple. Since, the ApproxTC algorithm only performs read operations on the graph data structures, multiple threads can access these data structures without requiring any exclusive access. So, ParApproxTC splits the $p * |E|$ edges, and assign each part to several threads so that each thread can compute the partial count of the edges in its part independently. Each thread th maintains it's own counter (say $count_{th}$) which contains the sum of all the $count_e$ processed by it. After every thread has done its share of computation, ParApproxTC sums the partial triangle counts from each thread ($count_{th}$) to get the $count_{partial}$. Then, it divides the $count_{partial}$ by appropriate normalization factor (p for ApproxTC, and $3*p$ for ApproxTC2) to get the approximate triangle count. The process is illustrated in Figure 5.1.

Optimization of ParApproxTC

As shown in Figure 5.1, ParApproxTC provides each of the threads approximately $|E| * p/tc$ edges from the set E_p, with the expectation that all the threads are assigned the same amount of work for parallel processing. But, most often it is not the case. Even though all the threads have to process the same number of edges, the computation associated with the edges can be significantly different based on the size of the adjacency lists of the vertices incident to an edge. As a result, when we execute ParApproxTC (as shown in Figure 5.1), most of the threads finish their computation within a short span of time; however, there exist some of the threads that take significantly longer time to finish their share of computation; thus, resulting a longer overall execution time (see, Figure 5.2). This problem is also discussed in

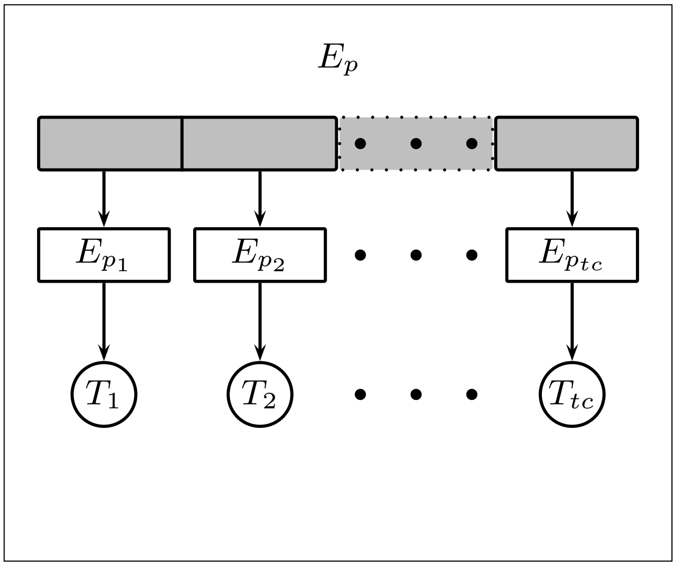


Figure 5.1.: Illustration of parallel workload distribution among tc number of threads. Here, T_i indicates thread i. E_p is the set of edges to be processed. $E_{p_i} \subset E_p$ is a disjoint set of edges assigned to the thread i.

earlier works with the phrase "the curse of last reducer" in the context of Hadoop based parallelization [43].

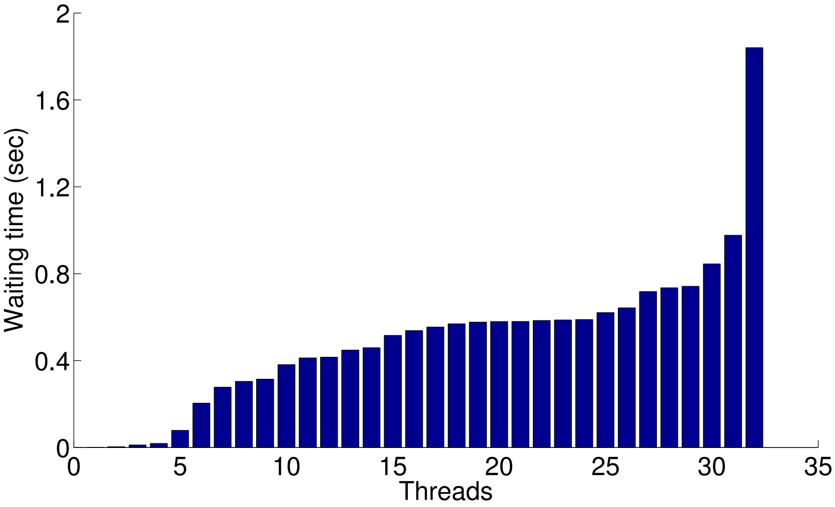


Figure 5.2.: Thread waiting time over 32 threads of ParApproxTC. Execution with $AtomicWorkLoad = (|E_p|/32)$ for graph "Wiki-4"

To work around the above limitation, we propose a different variant of the algorithm ParApproxTC. In this variant, each thread takes a small fraction of the total job at the beginning of the execution and upon finishing the execution, it takes additional fractions of job in subsequent iterations, until the entire job is finished. To ensure that different threads work on different fractions of the total job, a queue is used for storing the list of edges that are yet to be processed; this queue mediates the job allocation to different threads. At the beginning of every iteration of job allocation, each thread acquires an exclusive access of this queue to request new job. The process is explained in Figure 5.3.

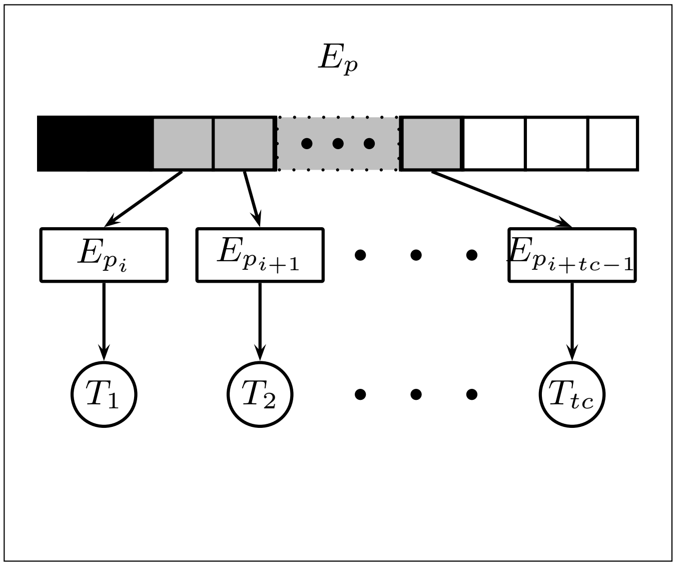


Figure 5.3.: Illustration of parallel workload distribution among tc number of threads using queue. Here, T_i indicates thread i. E_p is the set of edges to be processed, each small box in E_p is a packet of edges that is assigned to a thread at a given iteration. Dark packets of E_p are the edges that have already been processed by some thread. Gray packers are being processed, and finally the white packets will be assigned to the next available thread.

Though the above variant of ParApproxTC distributes the total work among different threads more evenly, it also suffers from a bottleneck caused due to the exclusive access to the job queue. So, there is a trade-off that is based on the size of job (edges) assigned in response to a job request. A large job assignment (the highest possible is, $AtomicWorkLoad = (|E_p|/tc)$ edges assigned to each thread) ensures that no time is spent for enforcing mutual exclusion but it yields larger waiting time for finishing the last thread. On the other hand, a smaller job assignment results smaller waiting times for finishing the last thread, but it suffers from a high waiting time in the semaphore queue due to the large number of exclusive accesses to the job allocation queue. In our experiments, we find that the differences in speed-up factors vary within a range of 5% to 10% based on the choice of packet size.

Table 5.2.: Graphs used in experiments.

Graph	Vertices	Edges
Wikipedia 2005-11-05 (Wiki-1)	$1,634k$	$18,540k$
Wikipedia 2006-09-25 (Wiki-2)	$2,983k$	$35,048k$
Wikipedia 2006-11-04 (Wiki-3)	$3,148k$	$37,043k$
Wikipedia 2007-02-06 (Wiki-4)	$3,566k$	$42,375k$
Zewail	$6k$	$54k$
Flikr	$820k$	$6,625k$
Epinions network (EN)	$75k$	$405k$
Edinburgh Associative Thesaurus (EAT RS)	$23k$	$305k$

5.2 Triangle Counting Experiments

We perform several experiments to observe the performance of approximate triangle counting. For this we use a collection of real-life networks available from `http://www.cise.ufl.edu/research/sparse/matrices/`. We choose the largest graphs used in [50]. The statistics of the graphs are shown in Table 5.2. To reflect the performance of an approximate counting algorithm, we use two metrics: speedup and accuracy (%). The speedup of a method M defines the ratio of the execution time between ExactTC and the corresponding algorithm, M and accuracy defines the counting accuracy of the algorithm M, in percentage. All experiments are executed on a 64 core 2.3GHz AMD machine.

Table 5.3.: Average and variance of execution time and accuracy of ParApproxTC (EI) and ParApproxNI (NI). ($tc = 16$,$p = 0.1$)

Graph	Execution Time				Accuracy			
	Average (s)		Variance		Average (%)		Variance	
	EI	NI	EI	NI	EI	NI	EI	NI
Wiki-1	3.13	11.99	0.01	79.78	96.17	96.39	0.19	8.87
Wiki-2	7.89	23.00	0.06	132.43	98.83	97.03	0.03	2.75
Wiki-3	8.60	45.90	0.03	2824.02	98.80	96.36	0.06	9.69
Wiki-4	10.68	47.98	0.08	3056.87	99.07	96.02	0.05	26.47
Zewail	0.00	0.00	0.00	0.00	99.65	95.47	0.02	12.56
Flikr	0.61	1.24	0.00	0.03	99.86	93.40	0.01	9.68
EN	0.02	0.03	0.00	0.00	99.35	92.97	0.28	60.87
EAT RS	0.01	0.01	0.00	0.00	99.39	93.52	0.54	13.28

5.2.1 EdgeIterator vs NodeIterator

In this experiment we compare the performance of ParApproxTC with that of ParApproxNI. ParApproxNI algorithm is a NodeIterator algorithm to approximate triangle count of a network using multiple threads. Both the ParApproxTC and ParApproxNI algorithms are given same set of parameters (a network $G(V, E)$, sampling factor p, thread count tc). A NodeIterator algorithm works by iterating over the nodes in V. For, each node $v \in V$ the ParApproxNI computes partial count of triangles incident on node v ($count_v$). Finally, the sum of all the partial counts gives the total triangle count $count$ ($count = \sum_{v \in V} count_v$). For approximate triangle count with sampling factor p, we sample a set V_p, where $|V_p| = p * |V|$ and each node has equal probability to be selected. Then, the approximate triangle count using ParApproxNI will be, $count = \frac{\sum_{v \in V_p} count_v}{p}$. The method is very similar with that of ParApproxTC as explained in Section 5.1.1. The most significant difference here is the process of computing partial triangle count $count_v$. To compute $count_v$ of node v we need to go over all the possible pairs of nodes from $adj(v)$ and check if the pair represents an edge ($count_v = |\{(x, y) : x \neq y \ and \ x, y \in adj(v) \ and \ (x, y) \in E\}|$).

For this experiment we consider $tc = 16$ and $p = 0.1$ for both the algorithms. Each approximation is executed 10 times. The average and variance of execution time and accuracy is reported in Table 5.3. From Table 5.3, we can see that, EdgeIterator algorithm is almost always better than NodeIterator in execution time and accuracy. NodeIterator algorithm is also shows higher variance on execution time and accuracy. The reason behind it is that, in many graphs node degree is exponentially distributed (Power Law model [4]). That is, number of nodes with low degree is very high compared to the number of nodes with high degree. In general, high degree nodes contribute higher in triangle count. Consequently, when we are performing uniform sampling from nodes in V, we are sampling from a very skewed distribution of partial-count $count_v$; as opposed to EdgeIterator which samples from a less skewed distribution of partial-count $count_e$. As can be observed from the Table 5.3, execution time of NodeIterator is also higher than that of EdgeIterator.

Table 5.4.: Average accuracy with respect to sample factors and speedups with respect to sample factors and total threads used.

Graph	Sample		Variance	ApproxTC	ParApproxTC Speedups			
	Factor	Accuracy	of	speedup	threads			
	p	(%)	accuracy		4	8	16	32
Wiki-1	0.10	99.21	0.40	4.49	24.23	42.84	68.57	91.68
	0.01	98.20	2.36	33.54	239.94	418.75	664.37	837.74
Wiki-2	0.10	99.56	0.10	4.14	20.10	35.60	55.43	63.42
	0.01	98.95	0.49	32.42	199.56	351.02	548.01	614.27
Wiki-3	0.10	99.61	0.12	4.21	19.87	35.54	54.22	60.96
	0.01	98.72	1.93	32.85	198.44	341.15	515.91	592.26
Wiki-4	0.10	99.60	0.09	4.33	19.54	34.95	52.55	56.84
	0.01	98.28	1.61	33.71	197.13	346.92	504.8	547.29
Zewail	0.10	98.45	0.08	4.29	11.35	12.46	10.00	6.19
	0.01	92.26	12.28	9.92	40.14	30.01	15.14	10.03
Flikr	0.10	99.74	0.07	5.18	28.92	51.46	77.67	96.67
	0.01	99.43	0.19	33.26	277.44	501.83	730.21	796.85
EN	0.10	99.03	0.62	4.93	22.69	33.10	40.60	38.71
	0.01	97.03	5.82	18.83	147.40	164.77	143.46	97.96
EAT	0.10	98.21	1.66	4.15	17.74	24.79	27.92	23.62
RS	0.01	96.64	3.88	13.62	101.70	111.67	86.82	56.30

5.2.2 Performance of ApproxTC and ParApproxTC

We show the performance of ApproxTC and ParApproxTC in Table 5.4. For both the methods we show results for both p equal to 0.1 and 0.01; for these two cases, the sampler samples 10% and 1% of edges of the original graph. For ParApproxTC we set $AtomicWorkLoad = 50 edges$, and thread-count $tc = 4, 8, 16$ and 32. For every parameter setting we execute the algorithm for 5 times and show the average of the counting accuracies and the speedups. As we can see, the speedup factor increases as we increase the number of threads for ParApproxTC . However, the speedup does not increase linearly (see Figure 5.4) with the number of threads, due to the bottleneck, as explained in Section 5.1.1. In ideal case, where there is no bottleneck for gaining mutual exclusion the speedup factor should be increased linearly with tc. For example, in case of “Wiki-1” graph and $p = 0.1$, speedup with $tc = 4$ is 24.23. So, for $tc = 8$ ideally the speedup should be close to 48.46. But, it is 42.84 for our experiment. Similarly, for $tc = 16$ speedup is 68.57 instead of 85.68. The more number of threads competing for mutual exclusion the higher the waiting time would be. As a result, increasing the thread count indefinitely does not ensures steady speedup. For example, the speedup of ParApproxTC decreases when we increase total threads from 16 to 32 for graph “EAT RS” for $p = 0.1$ (see Table 5.4). Figure 5.4 shows this relation graphically. The speedup increases with thread count up to certain point, but eventually increased number of thread damages the performance. Also, important to note that our approximate algorithm has shown better approximation accuracy for larger graphs.

5.2.3 Comparing The Performance of ParApproxTC with Changing Values of $MutexAccessCount$

For this experiment, we define an additional parameter $MutexAccessCount$. Assigning $MutexAccessCount = 1$ ensures that, every thread will try to obtain mutual exclusion (job assignment) only once. Where $MutexAccessCount = 10$ indicates that

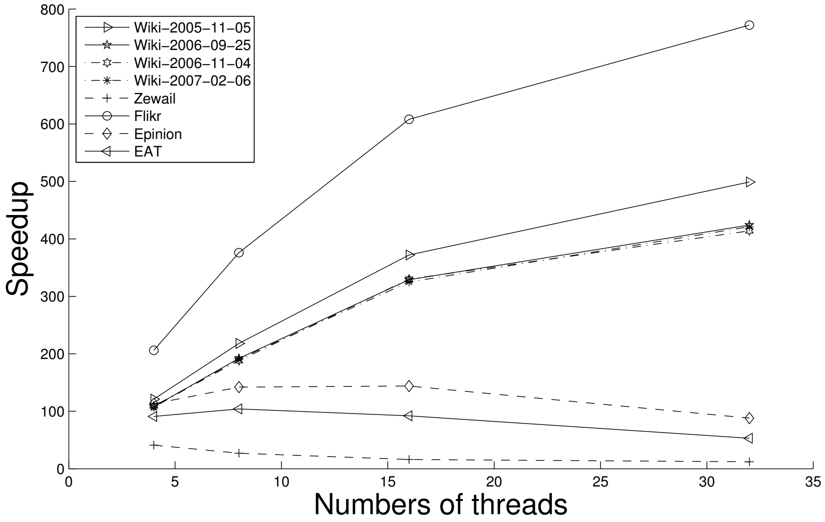


Figure 5.4.: Speedup vs Thread count tc for ParApproxTC

Table 5.5.: Speedup with respect to $MutexAccessCount$ for graph "Wiki-4" using 16 threads

$MutexAccessCount$	Speedup
1	32.50
2	31.71
4	32.40
8	33.14
16	33.53
32	33.96
64	33.99
128	34.34
256	34.20
512	34.64
1024	34.16
2048	34.20

on average every thread will try to obtain new job assignment 10 times. Increased value of $MutexAccessCount$ indicates smaller portion of job assigned to a thread at single access to mutually exclusive portion of program.

$$AtomicWorkLoad = \frac{|E_p|}{MutexAccessCount * tc}$$

In this experiment, we execute ParApproxTC (with $p = 1.0$) with different $MutexAccessCount$ for graph "Wiki-4". For a constant number of threads tc, as we increase the value of $MutexAccessCount$, the waiting time of all other threads for last thread to finish decrease but the waiting time to get exclusive access to job queue E_p increases. The result is presented in Table 5.5. As we can see, for a good span of possible value of $MutexAccessCount$ the speedup is approximately same.

5.2.4 ApproxTC vs *DOULION*

In this experiment we compare the performance of ApproxTC with that of $DOULION$ [50]. For that, we repeat the ApproxTC with $p = 0.01$ and $DOULION$ with $p = 0.1$ for all the graphs from Table 5.4. The speed up is with respect to ExactTC (see Algorithm 1). The implementation delivered by the authors of [50] performs worse than (shows less speed up) our implementation of $DOULION$. So in this result we report the speed up and accuracy of our implementation of $DOULION$. The result is shown in Table 5.6. Clearly, ApproxTC (single-threaded) is better than DOULION both in terms of speedup and accuracy for most of the graphs.

Table 5.6.: Average accuracy and speedup of our implementation of DOULION (p = 0.1) (not parallel) and ApproxTC. $p = 0.01$

Graph	DOULION		ApproxTC	
	Accuracy(%)	Speedup	Accuracy(%)	Speedup
Wiki-1	55.20	19.61	98.20	33.54
Wiki-2	88.52	20.54	98.95	32.42
Wiki-3	90.13	20.72	98.72	32.85
Wiki-4	94.31	21.64	98.28	33.71
Zewail	92.41	3.61	92.26	9.92
Flikr	99.04	17.69	99.43	33.26
EN	97.40	7.16	97.03	18.83
EAT RS	53.50	4.94	96.64	13.62

5.2.5 ParApproxTC vs *GraphPartition*

In this experiment we compare the performance of ParApproxTC with that of *GraphPartition*(GP) [43]. Since, GP is an exact counting method, We conduct the exact triangle counting using ParApproxTC with $p = 1$ (100% sampling), and $tc = 32$, and compare the execution time. The results are shown in Table 5.7. In this table, times under GP column are taken from corresponding paper [43], which is the time of running GP on a 1636-node Hadoop cluster. In all three cases for which we were able to collect graph from SNAP [2], our method significantly wins over GP using only 32 threads!

Table 5.7.: Average execution time of GraphPartition as stated by [43] and ParApproxTC with $p = 1$ and $tc = 32$

Graph	time(sec)	
	GP	Parallel Exact
web-Berk-Stan	102	12.04
as-Skitter	124.8	9.75
LiveJournal	654	26.46

5.3 Graphlet Counting

We give approximate graphlet counting algorithm Graft by extending EdgeIterator based exact graphlet counting method ExactGC discussed in Section 4.3. Similar to ExactGC, Graft works as an EdgeIterator algorithm; in such an algorithm, the counting process iterates over the edges of the input graph, $G(V, E)$. For an edge $e \in E_G$, it finds the count of all induced embeddings of a graphlet g with the constraint that the edge e is part of the embeddings; we call this count a *partial count* (see Section 4.3) of the graphlet with respect to the edge e. The partial count can be summed over all the edges to obtain a total count of the graphlet g in the input network. However, in the above process, a distinct graphlet will be counted

[2] http://snap.stanford.edu/

multiple times, by being accounted in different partial counts, so the above count needs to be corrected by diving it with an appropriate normalization factor. Such a method yields an exact graphlet counting algorithm. Graft obtains an approximate graphlet counting algorithm by iterating over a random subset of edges instead of all the edges of the input graph. The fraction of edges in the random subset with respect to all the edges is called the *sampling factor* of Graft. The lower the sampling factor, the faster Graft runs. On the other hand, the higher the sampling factor, the better the accuracy of Graft. For a sampling factor of 1, Graft returns an exact count.

5.3.1 Pseudo-Code

The outline of Graft is given in Algorithm 8. It accepts a graphlet g (it can be any of the graphlets shown in Figure 4.1), a large graph G in which we like to count the occurrences of g and a fraction p representing the sampling factor.

Graft first selects p fraction of edges (E_p) from the input graph G for which it will obtain the partial count of the graphlet g (Line 4). Then it iterates over each of the selected edges sequentially (the **for** loop in Line 5). For an edge $e \in E_p$, it finds the count of all induced embedding of g in G with the constraint that those embeddings map one of the graphlet edge e_g to the edge, e. (Lines 6 and 7). As mentioned earlier, we call this the partial count. The partial count is accumulated sequentially as the method iterates over the edges in E_p (Line 8). The final count is then normalized appropriately to obtain the total count of the graphlet g in G (Line 10); On line 11, Graft scales the count by dividing it with the sampling factor, p.

5.3.2 Optimization Schemes

Graft uses various optimization schemes which are crucial for its counting accuracy and running time. We discuss each of them in this section in the order of their significance.

Algorithm 8 GRAFT

```
function GRAFT(Large network $G(V, E)$, Graphlet $g$, sampling factor $p$)
    choose an specific edge $e_g$ in $g$
    $count \leftarrow 0$
    $Ep \leftarrow$ sampled $p$ fraction of edges from $E$ (without replacement)
    for each edge $e \in Ep$ do
        align $e_g$ with $e$
        enumerate all induced embedding of $g$ in $G$, where $e$ and $e_g$ are aligned, $x$ is the total number of embedding found
        $count \leftarrow count + x$
    end for
    $count \leftarrow count/normalization\_factor$
    $count \leftarrow count/p$
    return $count$
end function
```

Embedding Criteria and The Choice of FAE

From Figure 4.1, we can observe that some of the graphlets have multiple generation tree graphlets. For example, g_{22} has three generation tree graphlets: g_9, g_{10} and g_{11}. For the task of exact graphlet counting, we will obtain correct result, irrespective of the specific generation tree graphlet that we use for counting. But, complexity arises when GRAFT is used for approximate counting, which samples only a fraction of the edges.

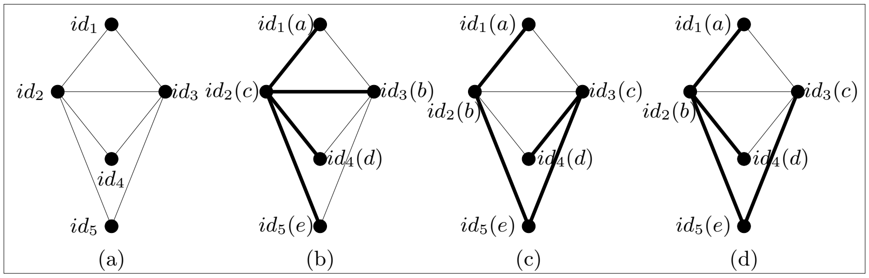


Figure 5.5.: Three ways for finding g_{22}, (b) using g_{11}, (c) using g_9 and (d) using g_{10}.

If we embed g_{22} using g_{11} (Figure 5.5(b)), the first step is to embed g_{11} by mapping the chosen FAE (b, c) to the edge (id_3, id_2); and, the second step is to check if this embedding contributes to an induced embedding of g_{22}. Here, the mapped edge (b, c)(of g_{22}) belongs to an edge-orbit of size 1. Therefore, for approximate graphlet counting, if we map edge (b, c) with edge (id_3, id_2), there is a high probability of missing an embedding of g_{22}.

Table 5.8.: Average percentage error of g_{22} graphlet counting ($p = 0.1$) using different generation tree graphlets.

Tree Graphlets →	g_9	g_{10}	g_{11}
Network ↓	Error (%)		
ca-GrQc	9.13	9.13	39.78
ca-HepTh	5.03	5.03	26.95
ca-CondMat	3.68	3.68	39.54

On the other hand, if we embed g_{22} using g_9 or g_{10} (Figure 5.5(c) and 5.5(d)), the edge (b, c) of g_{22} cannot be the FAE; rather, the FAE is the edge (b, e), which is mapped to the edge (id_2, id_5). The edge (b, e)(of g_{22}) belongs to an edge-orbit of size 6. Therefore, for approximate graphlet counting, if the edge (b, e) is used as FAE and is mapped to the edge (id_2, id_5) , the probability of missing an embedding of g_{22} is small. Hence, g_9 and g_{10} are more suitable generation tree graphlets for enumerating g_{22}. Table 5.8 shows the average error (in %) of approximate g_{22} counting using different tree graphlets with $p = 0.1$, for various real-life graphs. Therefore, the embedding criteria for optimizing the accuracy of approximate graphlet counting is choosing the generation tree graphlet for which the FAE belongs to the largest edge-orbit. For counting each graphlet, Graft chooses the best generation tree graphlet in such a way that the FAE of the generation tree graphlet belongs to a larger edge-orbit .

Stratified Sampling of Edges

For approximate counting, Graft chooses a subset of edges from an **iid** distribution, and then approximate the count of a graphlet in the input graph by a linear scaling (Line 11 in Algorithm 8). This would yield an exact count if the partial count associated with each of the edges were uniform. Real-life graphs have power-law distributions, that is the number of vertices having (relatively) high degree is exponentially smaller than number of vertices having low degree. Now, edges incident to high-degree vertices have comparably higher partial counts, specifically for the complex graphlets. On the other hand, many of the edges that are incident to low-degree nodes have a count of 0 for many complex graphlets. And due to same power-law distribution, the number of edges incident to high-degree vertices is exponentially smaller than number of edges incident to low-degree vertices. This high variance yields poor sampling accuracy for counting many of the complex graphlets; the same is also responsible for high variance of approximate count across different runs of Graft.

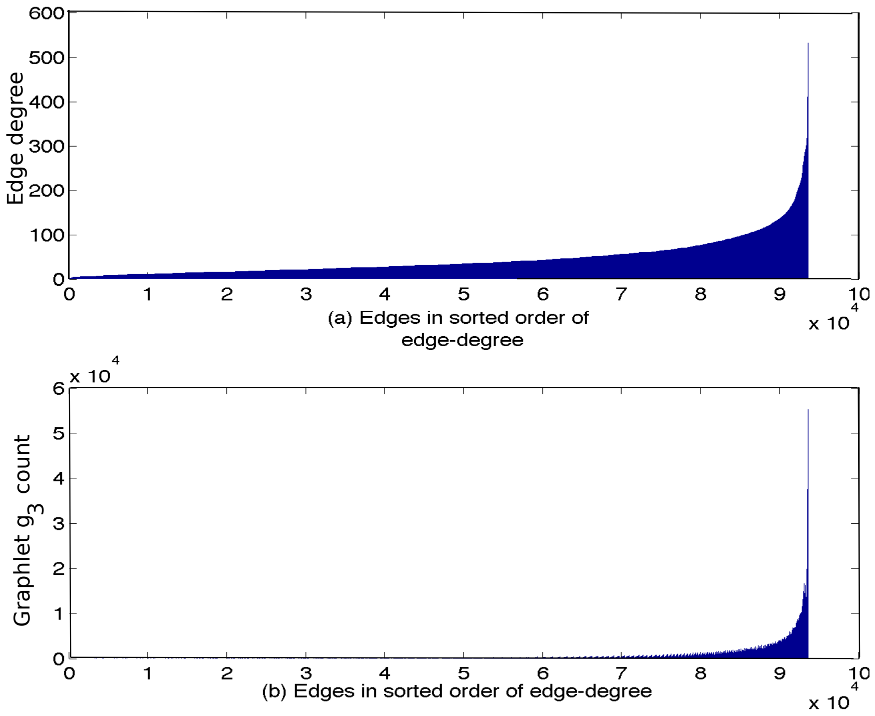


Figure 5.6.: Bar plot of (a) *edge* vs *edge-degree* in sorted order of *edge-degree*. (b) *edge* vs g_3 count in sorted order of *edge-degree*.

To improve the sampling quality, GRAFT adopts stratified sampling of edges, where each of the homogeneous sub-populations of edges have guaranteed representation in the set of sampled edges—here, a homogeneous sub-population of edges corresponds to a set of edges that have similar partial count for a given graphlet. To divide the edges into such a set of sub-populations, we do the following. We calculate the sum of degrees of two vertices of an edge and name it *edge-degree*; then we sort the edges in increasing order of *edge-degree*; a bar plot of the *edge-degree* is shown in Figure 5.6(a) for the graph *ca-CondMat*. We also plot the partial count of graphlet g_3 aligned with the corresponding edges in Figure 5.6(b). We observe that the partial graphlet count of different edges have high variance, but if we consider a set of edges that have similar *edge-degree*, the variance is reduced substantially, because of the high correlation between the partial count and the *edge-degree*. Guided by this observation, we split the edges into different and disjoint sub-population of edges based on a heuristic using the values of *edge-degree*. The heuristic is to set approximately equal intervals on the cumulative edge degree. And all the edges whose corresponding *edge-degree* participates in an interval belongs to the same sub-population. Consequently, at the tail part of the distribution where the variance of partial count is low, the interval width is relatively higher. Finally, the set of sampled edges is constructed by selecting p fraction of edges from each sub-population.

We provide an improved version of GRAFT that we call GRAFTStratified, which uses stratified sampling of edges that we discussed above. In such sampling, edges are sampled from each of these sub-population in such a way that the number of samples from each sub-population is proportional to the size of that sub-population. *t-test* confirms that GRAFTStratified performs better than GRAFT. GRAFTStratified reduces the sampling error by 25% with a negligible increment in the running time for most of the datasets.

In terms of implementation, GRAFTStratified differs from GRAFT only in Line 4 of Algorithm 8, which is replaced by the pseudo-code STRATIFIEDEDGESAMPLE shown in Algorithm 9. Lines 2 and 3 in this code computes the *edge-degree* and sort

Algorithm 9 Stratified Edge Sampling

```
 1: function StratifiedEdgeSample(Large network $G(V, E)$, sampling factor $p$,
    number of sub-populations (SP) $b$)
 2:     compute edge-degree for all the edges
 3:     sort the edges based on edge-degree
 4:     distribute the edges in $b$ different SPs such that, $e1 \in SP[i]$ and $e2 \in SP[j]$ and
    $i < j \implies$ edge-degree$(e1)$ <= edge-degree$(e2)$ and the cumulative edge-degree
    of the SPs are approximately equal
 5:     for $i = 1$ to $b$ do
 6:         $k = |SP[i]| * p$
 7:         sample $k$ edges from SP[$i$] and store in $Ep$
 8:     end for
 9:     return $Ep$
10: end function
```

edges in an increasing order of the *edge-degree*. Line 4 finds the subpopulations (SP) of the edges in such a way that edges with relatively similar *edge-degree* falls in the same SP and all SPs have approximately equal cumulative *edge-degree*. And finally, in lines 5 – 8 we sample the desired number of edges from each of the SPs.

5.3.3 Parameter Selection

Graft has only one parameter, the sampling factor p. The choice of p depends on the distribution of the partial counts associated with each of the edges in the given graph. For a degree-regular graph or for a random graph, the distribution of partial count is fairly uniform, so a very small value (in the range of 0.001 to 0.01) of p works well. But, real-life graphs have power-law degree distribution, so the partial count distribution of most graphlets on such graphs are skewed. As a result, for such graphs comparably more samples are needed. It is hard to know in advance what sampling factor will yield what kind of accuracy. However, we found from empirical experiments over various real-life graphs that for large power law graphs that have more than 10,000 vertices and 100,000 edges, a p value between 0.01 and 0.05 yields an accuracy which is around 95%. However, a simple heuristic to choose a sampling factor is to track the standard deviation of the partial counts along with the sampling process, and adjust the sample size dynamically.

To obtain a good sample size statistically, we tried to fit the distribution of partial count of different graphlets against known distributions. Since, most of the partial count distributions are exponential-like, we tried to fit with an exponential distribution with $\lambda = 1/\bar{x}$, where $\bar{x}$ is the average over the partial count of all the edges in the graph. We use Kolmogorov's test [84] to confirm the statistical validity of those fittings; the test shows strong evidence to support the hypothesis that the data are *not* from a exponential distribution. Beside exponential, we also tried to fit with power-law distributions (using some well-known exponent values); those did not fit either over a wide range of input graphs. Typically confidence interval is used to obtain an error bound for a given sample size, but such interval is known only for few well known distributions. Since the partial count does not fall in those distributions, it is hard to provide a statistical error bound for the Graft algorithm.

GraftStratified has an additional parameter (number of sub-population) besides p; however the performance does not vary much with the choice of b as long as b is chosen between 5 and 20. We choose a value of 5 for all our experiments.

5.3.4 Implementation

To implement Graft, we maintain two data structures of the input graph G; the first is the adjacency list representation of G, and the second is a hash-table that stores all the edges. The first is used for enumerating the generation tree graphlet, and the latter is used for fast checking whether an edge exists between a pair of vertices; the hash-table is used extensively when we want to find the induced graphlet that is embedded with a generation tree graphlet. The largest graph that we tried has about 1.7 millions vertices and 11 millions edges; for this graph, both the above data structures easily coexist in the main memory of a computer with a 4 GB of RAM.

5.4 Graphlet Counting Experiments

We perform several experiments to observe the performance of Graft. These experiments are performed on real-life graphs obtained from the following two web sites [3]. We choose graphs that are from various domains and have different sizes. The name and statistics of these graphs are available from Table 5.9. Note that, though we successfully execute Graft on graphs with millions of vertices, we show performance result on smaller graphs for which exact graphlet counting was possible.

The first set of experiments find the speedup factor and counting error(%) that we obtain by running Graft. Speedup factor is obtained by computing the ratio of execution times of Graft with some p less than 1 and Graft with $p = 1$; former is an approximate, and the latter is an exact graphlet counting. Apparently, Graft has a predictable value for the speedup factor. If Graft chooses p as its edge selection probability, the average speedup is close to $\frac{1}{p}$. This is so because we randomly select p fraction of total edges and then approximate the graphlet count using the partial count obtained from the selected edges. To obtain the counting error of Graft we first find the percentage of error in the count of each of the graphlets; then we average the error over all different graphlets. For counting error (%) computation, we use the following equation:

$$CountingError(\%) = \frac{|ExactCount - ApproxCount|}{ExactCount} * 100\%$$

The second set of experiments show the practical applications of graphlet counting for analyzing real-life graphs. For this, we obtain the graphlet frequency distribution (GFD) from the graphlet count and pictorially observe the GFD patterns. To obtain GFD, we first normalize the graphlet count vector (a vector of 29 integers which represents the count of each of the graphlets) so that the L_1-norm of the vector is 1; thus, the values in the vector represent a discrete probability distribution. Each of the entries of this vector is then replaced by its Logarithm (10 base), which yields

[3] http://snap.stanford.edu/data/index.html and
http://www-personal.umich.edu/~mejn/netdata

GFD vector of the given graph. Each of the entries in the GFD vector is called GFD value of that graphlet in the given graph. All GFD values are negative; the more negative the value, the rarer the graphlet. For visual comparison, we also plot the GFD vector; when analyzing such a plot, we should remember that the Y-axis of the plot is on logarithm scale (of normalized graphlet count).

5.4.1 Comparing The Counting Errors Among Different Graphlets

While counting with Graft, the counting error of various graphlets varies based on the complexity and the size of the graphlet. To compare the relative counting error among different graphlets, we use the *ca-CondMat* graph dataset with the edge selection probability of 0.1. We repeat the counting process for 10 times and report the average counting errors in Figure 5.7. Each column in this graph represents a distinct graphlet (which is shown as labels on the x-axis). The y-axis shows the percentage errors in counting. To show the variance of percentage error among different iterations, we show the results in a box-plot.

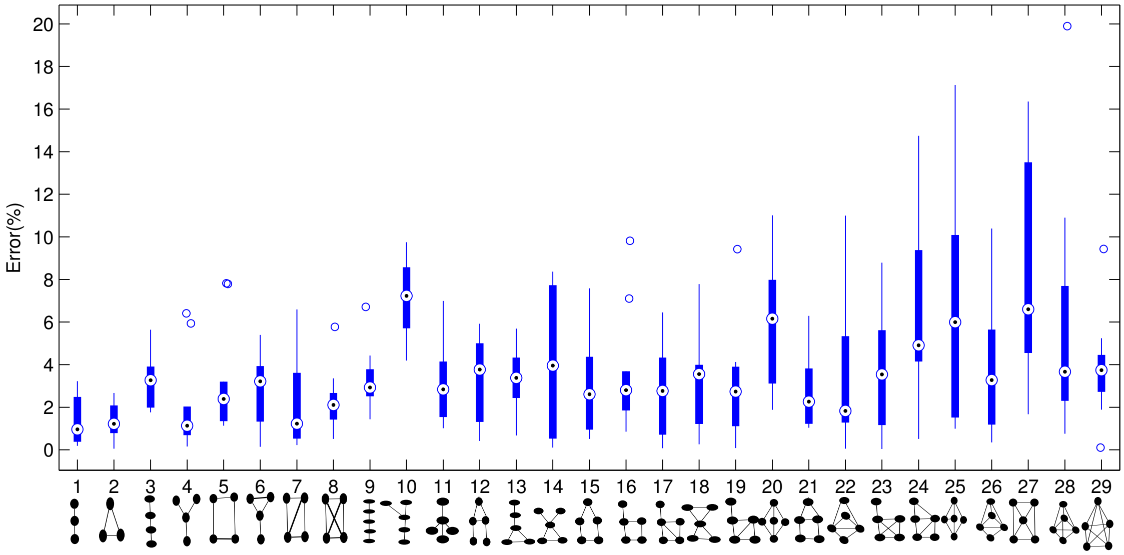


Figure 5.7.: Box-plots for approximation errors of different graphlet frequencies in network *ca-CondMat*.

The first observation from Figure 5.7 is that, the counting error increases with the number of vertices in the graphlets. The average counting error of 3-node (g_0 – g_1), 4-node(g_3 – g_8), and 5-node (g_9 – g_{29}) graphlets are 1.34%, 2.63% and 4.29%, respectively. Also, complex graphlets that have more cycles are more error-prone than tree graphlets. One reason for this increased error is due to the fact that the larger and/or complex graphlets have much lower exact count than other graphlets. For example, the count of g_{19} is approximately $23,400$ thousands, and the count for g_{20} is about 21 thousands. Though, both g_{19} and g_{20} have 5 vertices, the counting of rarer graphlet (g_{20}) exhibits more percentage error. The explanation for this is that the estimation by sampling is more difficult for rare objects. For the same reason, their counting also have larger variance as we can see in the box plot.

Further study reveals that, for a given graphlet, the distribution of its partial count associated with different edges of the input graph plays a major role in the accuracy of approximate counting of that graphlet. We can observe, that the average error of g_{10} is high (7.16%) because the normalized variance (62.6) of partial count distribution for this graphlet is high, although this graphlet is the most frequent with a count of ($10^{21.1}$). Also, counting error can be high if the exact count of the graphlet is low, despite the partial count distribution having a smaller normalized variance. g_{25} with a higher average error (6.55%) stands for an example for this case. In general, the accuracy of approximation using Graft depends on both the exact graphlet count and the variance of distribution of the partial count over the edges of the graph; however, it is more sensitive to the latter.

5.4.2 Sampling Factor vs Average Counting Error

In this experiment, we show how the counting error (averaged over all the graphlets) of Graft varies with the sampling factor. Figure 5.8 shows our findings. As the sampling factor increases, the error of our algorithm diminishes. We show these results for three real world collaboration networks of different sizes. An important observation

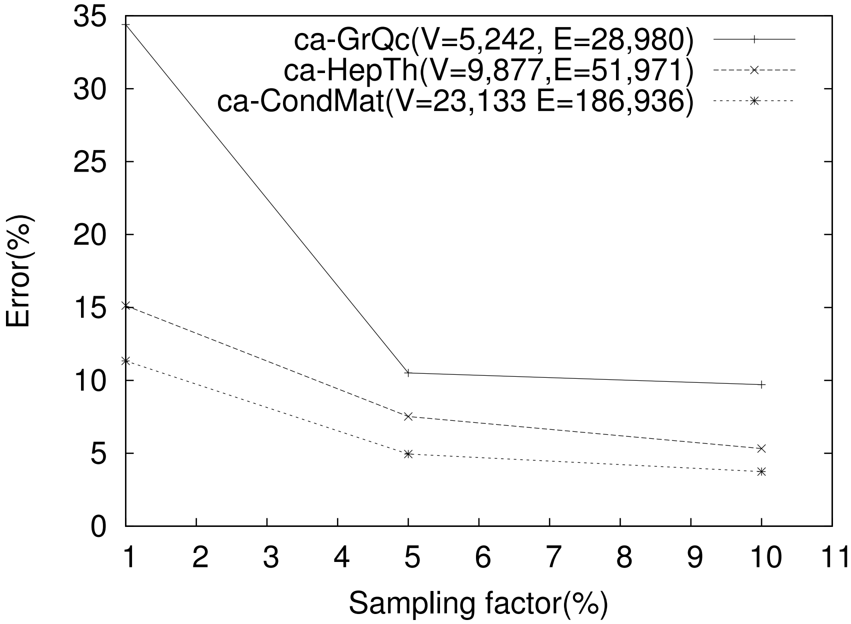


Figure 5.8.: Sampling factor vs average error measure for three real world collaboration networks.

is that for the same sampling factor, the larger graph have smaller percentage error. So, as the graph grows, we can afford to decrease the sampling factor (thus increase the speedup factor), while keeping the counting error at the same level. For example, 5% sampling factor obtains a 12% error on ca-GrQc graph, but almost identical error percentage is achieved with a sampling factor of 1% on ca-CondMat, as the second graph is much larger.

Table 5.9.: Speedup and error trade-off on seven real-life networks.($\star$ marked graph's approximate graphlet counting was done with p=0.01.)

Network	# Vertex	# Edge	$R_{e/v}$	Speed-up	Error(%) GRAFT/ GRAFTStratified
ca-HepTh	9,877	51,971	5.6	10.0	5.11 /4.26
OClinks	1,899	13,838	7.3	9.90	5.93 /4.28
ca-CondMat	23,133	186,936	8.1	9.80	3.62 /2.61
Polblogs	1,224	16,717	13.6	9.80	2.81 /1.98
ca-AstroPh$\star$	18,771	198,050	10.6	89.28	4.41 /3.10

5.4.3 Speed-Up and Percentage Error Comparison on Different Networks

In this experiment, we compare the speedup and percentage counting error of five real worlds networks from collaboration, blogs, and web domains that are shown in Table 5.9. For this we use $p = 0.1$, which gives us about $1/p = 10$ times speedup (except the case of ca-AstroPh, for which we use $p = 0.01$). The percentage error values are between 2.81% and 5.93%. Our method generally performs better as the graph becomes larger and denser. In case of network ca-AstroPh (which is much bigger than other networks), we use $p = 0.01$ which gives a much higher speed-up while maintaining the same level of accuracy. In this experiment, we also compare the performance between GRAFT and GRAFTStratified, the latter samples edges using stratified sampling with 5 sub-populations. The last column of the Table 5.9 shows for all the datasets, GRAFTStratified obtains at least 25% better accuracy than the GRAFT. The speed-up factor is almost the same, as the overhead of stratified sampling is negligible in comparison to the time of graphlet counting.

.

5.4.4 Comparison with Existing Method

In this experiment, we compare our algorithm with competing algorithms. To the best of our knowledge, no approximate graphlet counting algorithm is available. However, many algorithms are available that count triangle (graphlet g_2); we implement the best approximate triangle counting algorithm called $DOULION$ [50]. Note that, $DOULION$ is implemented on Hadoop, we implement $DOULION$ by following the algorithm in the author's paper using identical data structure as our algorithm. We compare the performance of GRAFT and $DOULION$ for counting triangle. For comparison, we choose "as-Skitter" dataset (with 1.7 million vertices and 11 million edges). We run GRAFT and $DOULION$ for the same sampling factor values, which are 0.01 and 0.005 respectively. GRAFT obtains much better performance in terms of counting accuracy with a higher execution time. For instance, with sampling factor

0.01, *DOULION* approximates triangle count with 10.27% error in 3.8 second, while Graft does that with 1.62% error in 13.4 second. Moreover, with sampling factor 0.005 *DOULION* approximates triangle count with 64.49% error in 3.2 second, while Graft does that with 2.84% error in 7.9 second. So, for lower sampling factor, *DOULION*'s performance falls sharply, while Graft's performance degrades only marginally.

We did not compare our algorithm with GraphCrunch because GraphCrunch is an exact counting algorithm. Further, it's implementation is targeted for parallel machines.

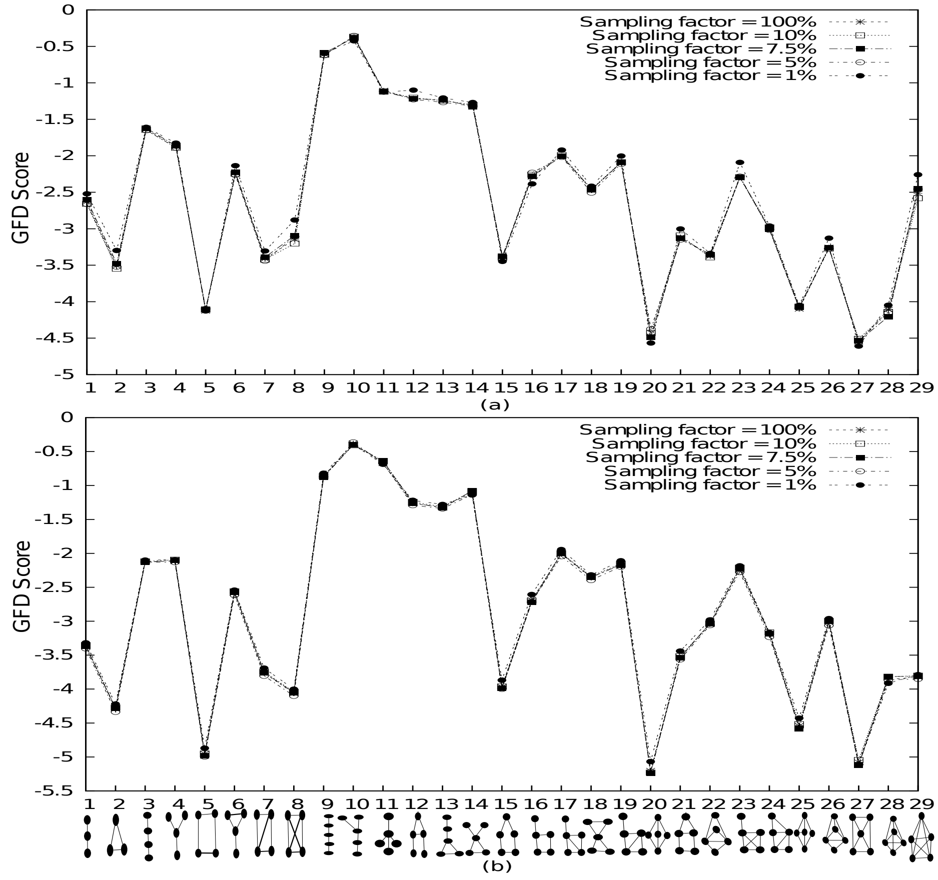


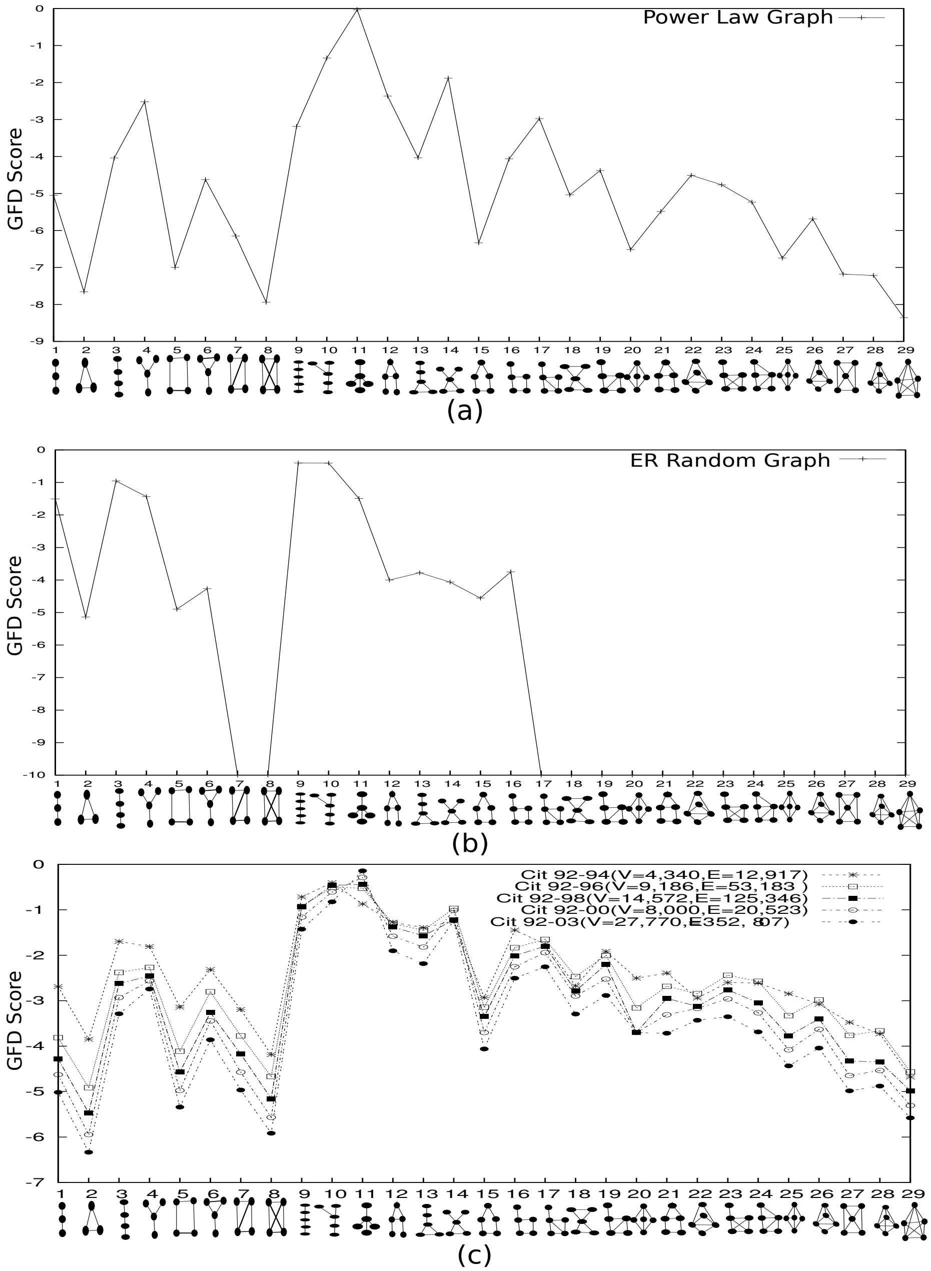
Figure 5.9.: GFD of 29 graphlets for different sampling factor on (a) ca-HepTh and (b) ca-CondMat networks.

5.4.5 Comparing GFD for Different Sampling Factors

In this experiment, we justify the utility of approximate graphlet counting algorithm(Graft). One of the main objectives for graphlet counting is to obtain the graphlet frequency distribution (GFD) for large graph analysis. We like to show that, although counting errors prevails by adopting sampling in Graft, the effect of this error on GFD is negligible, because the latter compares the relative counts in a logarithm scale (log scale is used for GFD because the frequency of different graphs varies in exponential proportion). For this, we obtain graphlet count by running Graft for different sampling factors $(10\%, 7.5\%, 5\%$ and $1\%)$ on various networks that we used in experiments discussed in Section 5.4.2. For these networks, We also obtain the exact graphlet count using Graft algorithm with p=1. We then find GFD from each of the results, and compare the GFD plots of a graph for different sampling factor. In Figure 5.9, we show this comparison for only two networks because the trend is similar for all the other networks. It is easy to see that the GFD histogram preserve its shape across different sampling factors, even for a sampling factor of 1% (which provides a 100-fold speedup).

5.4.6 GFD Over Time Variant Graphs

Does GFD signature holds for time varying networks? To answer this question, we use the citation network used for 2003 KDD cup. We collect citation data ranging from the year 1992 to 2003, and consider the citation graph incrementally for every two years, and plot the GFDs associated with each of these graphs in Figure 5.10(c). As we can see in this figure, the overall trend of the GFD remains almost the same as the graph evolves over the time. The dominant graphlet is g_{11}, the star network, for all temporal snapshots of the network. Star-centers in such graphs represent the heavily cited papers and they contribute to many star like graphlets. This experiment demonstrate that, GFD shows consistent behavior for a dynamic(time variant) graph over a period of time. That is GFD has a potential use in expressing the construction mechanism of a graph, instead an instance of it (the graph). Which gives us motivation to use GFD as clustering or classification criteria (see Section 8.1) for graphs generated following different mechanismsd.



(a)

(b)

(c)

Figure 5.10.: GFD of 29 graphlets for (a) synthetic Power-law network and (b) synthetic Erdos-Renyi network (c) time variant citation networks

5.4.7 GFD of Different Types of Graphs

This experiment demonstrates GFD's usability in analyzing large graphs. Here, we compare GFD plots of graphs from different families; Figure 5.10(a) and Figure 5.10(b) shows such a comparison. The graph corresponds to the plot-(a) is a Power-Law network and the graph corresponds to the plot-(b) is an Erdos-Renyi (ER) random graph—both are generated synthetically with identical counts of vertices and edges. As we can see, GFDs are very different for these two graphs; in power-law network the dominant graphlet is g_{11} (see Figure 4.1) which is a star of 5 nodes; on the other hand, in random network the dominant graphlet is g_9. The reason is fairly obvious; in power law network there exists a few vertices with extremely high degree. These vertices will be part of numerous star like graphlets, where they are at the center of the star. The same reason holds for the size 4 graphlets. On the other hand, in random network the dominant graphlet is g_9, a path of size 4; since we introduce edge randomly in random network, the likelihood of existence of a simple structure like a path of size k is higher than any structure with a complex formation. Another observation is that in power law network, there exists a significant number of cyclic graphlets, whereas in random network this count is almost zero—this observation is consistent with the known knowledge on random graphs [85]. While it may be easy to justify the above findings, however, our aim is to demonstrate that GFD distribution is a promising tool for building signature for characterizing graphs from different families.

5.5 Distributed Graphlet Counting

We give an approximate graphlet counting algorithm AppSpark by extending distributed graphlet counting algorithm ExactSpark discussed in Section 4.4.3. Similar to ExactSpark, AppSpark generates RDDs for $\{1, 2, 3\}$-*Graphlets*. Then for each record in 3-*Graphlet* RDD, it finds the counts of graphlet embeddings explored from and including the record. However, instead of computing partial graphlet count *glCount* for all records in 3-*Graphlet* RDD, AppSpark samples a fraction (p) of records for which it computes *glCount*s. Finally the aggregated (summation) result of *glCount* RDD is scaled by dividing the counts with the sampling factor p. The method is illustrated in Figure 5.11.

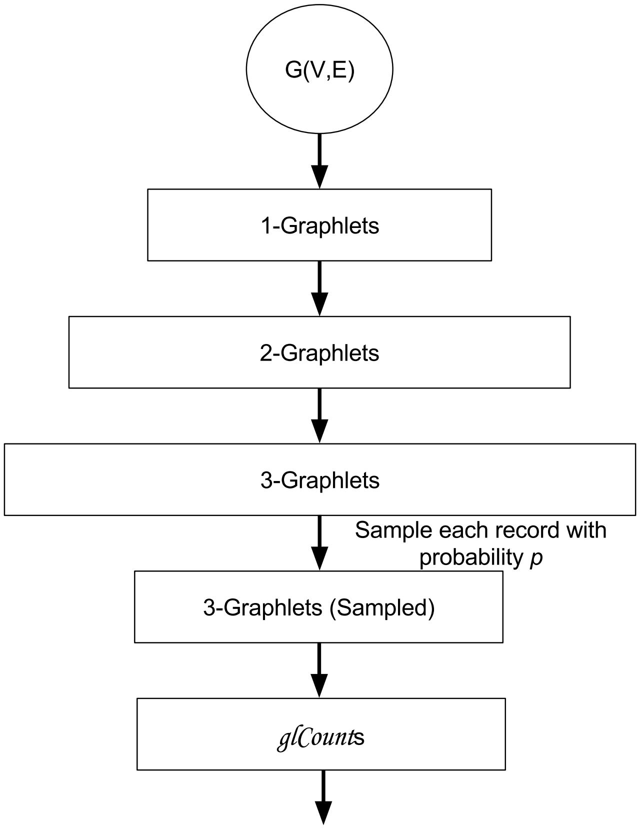


Figure 5.11.: RDD generation for approximate graphlet counting algorithm AppSpark using BFS exploration

5.6 Distributed Graphlet Counting Experiments

We perform several experiments to demonstrate the performance of AppSpark. These experiments are performed on real-life graphs obtained from publicly available source[4]. We choose networks that are from various domains and have different sizes

[4] http://networkrepository.com/

and densities. The name and statistics of these networks are available from Table 5.10.

The experiments compare the performance of AppSpark with Graft. For comparison, we use execution time and speedup factor. Speedup factor is obtained by computing the ratio of execution time of AppSpark with that of Graft (both with $p = 1$). We also compare the approximation performance of AppSpark with Graft using counting error(%) as defined in Section 5.4.

$$CountingError(\%) = \frac{|ExactCount - ApproxCount|}{ExactCount} * 100\%$$

Finally, we demonstrate the scalability of the proposed distributed solution by comparing the execution time with varying number of CPUs in Spark cluster.

5.6.1 Comparing Execution Time of AppSpark and Graft

In this experiment, we compare the execution time of AppSpark with that of Graft. While Graft is a sequential algorithm for graphlet counting, AppSpark utilizes a cluster of 10 nodes with 4 CPUs on each node (40 total CPUs). We demonstrate the performance of AppSpark with respect to Graft in Table 5.10. First six columns of Table 5.10 represents the statistics of the networks. Columns titled Nodes, Edges and Max degree represents the number of nodes, edges and maximum degree of any node in the network respectively. Fifth column in the table represents the number of 3-*Graphlets* in the network; this also represents the number of records in 3-*Graphlet* RDD. Note that we list the networks in increasing order of graphlet counts (Column six). Column six contains the total number of $\{3,4,5\}$-*Graphlets* in the network.

Table 5.10.: Execution time comparison of AppSpark with Graft. Exact graphlet count by using $p = 1$ for both the methods.

Network	Nodes	Edges	Max degree	# 3-*Graphlet* records	#graphlets	Graft ET (sec)	AppSpark ET (sec)	SpeedUp
bio-diseasome	516	1,188	50	6,758	$838K$	2	8	0.19
bio-yeast	1,458	1,948	56	11,524	$1.2M$	1	17	0.06
inf-roadNet-PA	1,087,562	1,541,514	9	3,256,097	$32M$	56	33	1.69
inf-italy-osm	6,686,493	7,013,978	9	8,243,351	$35M$	60	100	0.60
inf-roadNet-CA	1,957,027	2,760,388	12	5,747,736	$56M$	100	52	1.92
inf-road-usa	23,947,347	28,854,312	9	50,135,113	$384M$	958	880	1.09
bio-dmela	7,393	25,569	190	575,169	$954M$	996	145	6.87
ia-reality	6,809	7,680	261	497,715	$1.3B$	1,546	184	8.40
inf-openflights	2,939	15,677	242	712,328	$3.1B$	7,683	464	16.56
ca-dblp-2012	317,080	1,049,866	343	17,332,119	$34B$	124,527 ($34.5h$)	5,334 ($89m$)	23.35
ca-HepPh	11,204	117,619	491	8,560,145	$122B$	1,909,970 ($22d$)	22,028 ($6h$)	86.71
soc-brightkite	56,739	212,945	1,134	12,432,832	$419B$	682,846 ($7.9d$)	54,993 ($15.3h$)	12.42
socfb-CMU	6,621	249,959	840	32,788,398	$1.4T$	3,060,060 ($35.5d$)	228,347 ($2.6d$)	13.40

Notations: $K = 1000$, $M = 1000K$, $B = 1000M$, $T = 1000B$, $m \rightarrow minute$, $h \rightarrow hour$ and $d \rightarrow day$

We observe that the execution time of the algorithm generally increases with the increased number of graphlets. For example, the execution time of AppSpark (see Column 8) on *inf-road-usa* (880 seconds) is negligible compared to *socfb-CMU* (228,347 seconds), despite the fact that the first network is enormous (with $23M$ nodes and $28M$ edges) compared to the second network. This is because the network *socfb-CMU* is more densely connected network with exponentially more graphlet embeddings (approximately 1.4 trillion) to enumerate. We also observe that, for networks with more graphlets, AppSpark finishes much faster than Graft. We can attribute the improved performance of AppSpark, to the distribution of workload across 40 CPUs by the Spark distributed frameworks. AppSpark is outperformed by Graft for several small networks (see Columns 7 and 8); this is because the distribution overload (effort to distribute the task) for those networks is more than that of the computation workload (effort to actually perform the task) for the counting problem. But for lager networks with many graphlet embeddings AppSpark scales better than Graft. Column 9 gives the *speedup* = $\frac{Graft\ ET}{AppSpark\ ET}$.

5.6.2 Execution Time and Counting Error on Large Networks

In this experiment, we compare the execution times and counting errors of four real-world networks shown in Table 5.10. For this we use $p = 0.1$ for three networks and $p = 0.01$ for *socfb-CMU* (see Table 5.11). We run each method five times and report both the execution time and counting error in $mean \pm STD$ format. We observe that the execution time of AppSpark using 40 CPUs is better than sequential Graft method. More importantly, for all the networks the percentage counting error by AppSpark is less than by Graft. This is because the distributed method AppSpark samples records from 3-*Graphlets* and then performs partial graphlet counting for all sampled 3-*Graphlets*. On the other hand, Graft samples from the edges of the network and performs partial graphlet counting for all the sampled edges. For a network *socfb-CMU*, the total number of $\{3,4,5\}$-*Graphlets* is fixed (1.4

trillion). But for Graft the 1.4 trillion embedding is distributed as *partial graphlet count* among $249,959$ edges. On the other hand for AppSpark the same 1.4 trillion embedding is distributed as *partial graphlet count* among $32M$ 3-*graphlets*. Which means AppSpark approximates graphlet count by sampling from a larger and more uniform distribution of *partial graphlet count*. Hence, the better counting error.

Table 5.11.: Execution time and counting error comparison of AppSpark with Graft. The results are in $mean \pm STD$ format

Network	Sample	AppSpark	AppSpark	Graft	Graft
	Factor (p)	ET (sec)	Error(%)	ET (sec)	Error(%)
ca-dblp-2012	0.10	651±72	.59±.10	12,710±184	1.78±0.38
ca-HepPh	0.10	2,074±36	.40±.15	182,199 ±861	1.32±0.20
soc-brightkite	0.10	5,672±130	.32±.06	68,953±3,549	2.07±0.94
socfb-CMU	0.01	2,193±76	.60±.20	31,947±1,951	2.98±1.95

5.6.3 Performance of AppSpark with Different Number of CPUs

In this experiment, we demonstrate the performance of AppSpark with varying number of available CPUs in Spark cluster. We use real-world network *ca-dblp-2012* and $p = 1$ for exact graphlet counting (see Figure 5.12). Starting from 4 CPUs, we gradually increase the number of available CPUs by 4. Maximum number of CPUs available is 40. The X-axis represents the number of CPUs used, Y-axis represents the the execution time in seconds. The blue bars represents the execution time achieved by AppSpark, the green bars represents the execution times if the performance of AppSpark would change linearly with number of available CPUs (ideal case). For example, the execution time of AppSpark using 4 CPUs is $45,807$ seconds. In deal case the execution time with 8 CPUs should be $45,807/2 = 22,903.5$ seconds, with 12 CPUs $45,807/3 = 15269$ seconds, etc. As expected, the execution time decreases with increased number of available CPUs. We can also observe that, the performance of the proposed method AppSpark scales linearly (blue bars are very close to the green

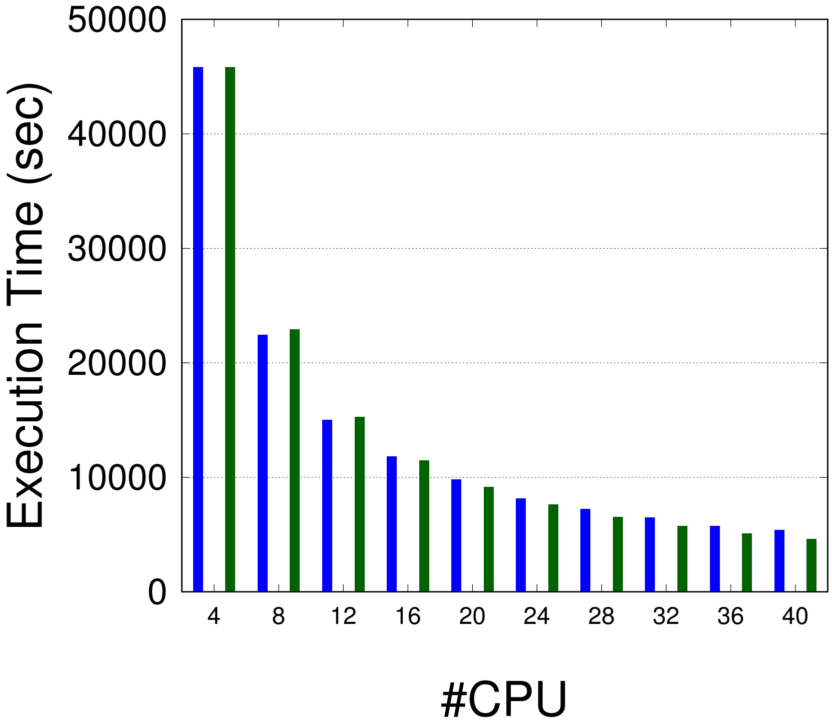


Figure 5.12.: Change in execution time with increased number of CPUs. Blue bars represent the execution times achieved by AppSpark. Green bars represent the execution times expected in ideal scenario. The network used in the experiment is *ca-dblp-2012*. $p = 1$ used for exact graphlet counting.

bars). This result proves that the proposed method is able to uniformly distribute the workload over a large number of processing units, thus making the proposed method perfect for distributed framework.

5.7 Conclusion

In this chapter we present an approximate triangle counting algorithm which is built on an EdgeIterator algorithm. Our method is simple, yet it achieves a surprisingly high accuracy and speedup. We also present a multi-threaded version of our algorithm which is suitable for multi-core machines. We show experimental results that validate that our approximate counting method is better compared to the state-of-the-art approximate triangle counting algorithm. Also, for exact triangle counting, our thread based implementation is significantly faster than an exact triangle counting method built on Hadoop cluster. Distributed computing paradigm is probably a better choice for graphs that are too large to fit in the main memory. However, we found that real-life graphs with as many as 6 millions nodes and 20 millions of edges easily fits in the memory of a typical desktop PC with 4GB of RAM, and for such graphs, our method is obviously a better alternative.

We also present Graft, an effective method for approximate graphlet counting for large graphs. The algorithm offers significant speedup with a negligible error in the count. For the same speedup factor, the counting accuracy of the algorithm improves with the size of the graph, so it is particularly suitable for counting graphlets in large real-life networks. We show that the graphlet frequency distribution is temporally invariant in real-life time varying networks, so it can be used to study graph evolution in a more holistic manner.

Finally, we present AppSpark, a distributed graphlet counting algorithm, using Spark distributed computation framework. This method harnesses the power of distributed computing paradigm to give a scalable graphlet counting algorithm. We compare the performance of AppSpark and Graft–for same sampling factor p, AppSpark can achieve better counting accuracy.

6 ESTIMATING TRIPLE BASED NETWORK METRICS

In large networks, the connected triples are found to be useful for solving various tasks including link prediction, community detection, and spam filtering. Existing works in this direction concern mostly with the exact or approximate counting of connected triples that are closed (aka, triangles). Evidently, the task of triple sampling has not been explored in depth, although sampling is a more fundamental task than counting, and the former is useful for solving various other tasks, including counting. In recent years, some works on triple sampling have been proposed that are based on direct sampling, solely for the purpose of triangle count approximation. They sample only from a uniform distribution, and are not effective for sampling triples from an arbitrary user-defined distribution. In this chapter we propose two indirect triple sampling methods that are based on Markov Chain Monte Carlo (MCMC) sampling strategy. Both of the above methods are highly efficient (several magnitudes faster on large datasets) compared to a direct sampling-based method, specifically for the task of sampling from a non-uniform probability distribution. Another significant advantage of the proposed methods is that they can sample triples from networks that have restricted access, on which a direct sampling based method is simply not applicable. The proposed methods are very effective for estimating triple based network Metrics (e.g., Transitivity) from large and restricted networks; where exact and approximate triple counting is computationally prohibitive.

6.1 Objective and Motivation

Our objective is to obtain an efficient method for sampling triples from an arbitrary (user defined) probability distribution (say, f) defined over the set of triples in a network. The distribution f need not be defined explicitly; for instance, one can only define a weighting function $w(\cdot)$ over the set of triples, and f is simply the probability vector obtained from the weights of each of the triples. Any locally computable weight function should be admissible. Such a function can be formed by the topological properties of the vertices in the triples, or in case, the graph contains vertex or edge labels, the weight function can be designed based on the label composition of the vertices or edges of the triples. In a social network, one may want to find triples in a dense community, where a triple may have many other neighboring triples that are triangles. In the same network, if the vertices are labeled with the set of skills of the corresponding person, one may want to sample triples such that the labels of its vertices are similar or complementary. In both the above cases, a suitable weight function can be defined over the triples that is computable from the information that is stored locally at a node.

To obtain a direct method for the sampling task that is defined in the above paragraph, we first need to compute f (probability mass function) from the weight function, and then obtain the *cmf* (cumulative mass function) of f. Obviously, this requires the knowledge of the entire sampling space (total number of vertices, edges, and triples). For a restricted graph, which can only be crawled by following the edges of the network, such information is not available, so direct sampling is infeasible for solving the above sampling task on a restricted network.

The motivation for considering a restricted network comes from real-life consideration. Say, an analyst is using a crawler for crawling a Web graph, and he does not have the resources to store the entire graph in memory/disk. Under this setting, he may want to sample a set of triples (from a uniform distribution) alongside crawling so that he can approximate the transitivity of the Web graph. Clearly, without storing the entire network, he has no knowledge of the number of vertices, or edges in this network, let alone the number of triples. Also, for a hidden network, a user may not have access to an arbitrary node in the network for security reason, rather the desired node can only be accessed from another node which is one-hop away from it; such scenarios are common in real-life and are considered in some of the recent works that compute various network properties by random walk over real-life networks, such as, Facebook [86].

Even if a network is not restricted, an indirect sampling method can be more desirable than a direct sampling method, both from viability and efficiency consideration. For example, if the network is highly dynamic (say, a Gnutella file sharing network), computing f (and the exact number of triples) is an infeasible task, because the vertices and the edges in such networks appear or disappear abruptly. So, indirect (MCMC) sampling is the only option for such networks. Finally, as we will show in this chapter, an MCMC based method can be significantly more efficient than a direct sampling method, for non-uniform (weighted) sampling where the probability distribution vector (f) is not readily available, because in such a case, computation of f and *cmf*(f) (cumulative mass function of f) are required for direct sampling (see Section 6.4.2 for details); this fixed cost can be very expensive, as the number of triples in large networks are, typically, in the order of billions.

In this chapter, we propose two methods for indirect triple sampling using Markov Chain Monte Carlo (MCMC) strategy. MCMC performs a random walk over the sample space such that the desired probability distribution (in this case, f) aligns with the stationary distribution of the random walk. Since MCMC computes the transition probability matrix of the random walk locally (on demand), it does not compute f explicitly; consequently, it does not need any information regarding the size of the sample space. As long as a state of the random walk can be visited from one of the neighboring states, an MCMC-based sampling works, which makes it an ideal candidate for sampling from a restricted network. Also, an MCMC-based method computes the transition probability matrix on-line, so it can accommodate addition or deletion of vertices (or edges) that happens in a dynamic network, even when the sampling process is running.

The sampling methods that we propose are called Vertex-MCMC, and Triple-MCMC: the former is more accurate and the latter is more versatile. Both the methods can sample from an arbitrary distribution, yet Vertex-MCMC is particularly suitable (both efficient and accurate) for sampling from a uniform distribution. So, we use it for approximating triangle count in a large network. In experiment section, we show that the performance of Vertex-MCMC is almost as good as a direct sampling based method. On the other hand, Triple-MCMC method is more suitable for sampling from non-uniform distribution; our experiments with one of the real-life graph show that it is 170 times faster than a direct sampling method with a better sampling quality.

6.2 Related Works

The popularity of sampling based techniques has grown in recent years for analyzing large graphs. For example, sampling has been used for finding high centrality nodes [87], interesting subgraphs [88], communities [89] representative subgraphs [90], and graphlet frequency distribution [34].

Triple sampling is considered in the context of approximate counting of triangles (or computing transitivity) in the following works [44, 45, 54]. Both [44] and [45] obtain uniform sampling of triples using a direct sampling method, which we will discuss in Section 6.4.2. Buriol et al. [54] propose a collection of streaming algorithms for triple sampling, also with the intention of triangle approximation. One of their methods, named, 3-pass-incident-stream, is actually similar to the direct sampling method of [44, 45]. Buriol et al. also consider another 3-pass method for arbitrary edge streaming; it samples triples by first sampling an edge, and then sampling a vertex, both uniformly. A triple that is obtained this way belongs to one of the following sets exclusively: disconnected triples (set T_1), connected open triples (set T_2), or triangles (set T_3). From the size of each of these sets, the authors find an approximation of the triangle count in a graph. To the best of our knowledge, no works exist that consider sampling of triples from a user-defined arbitrary sampling distribution.

Besides the above work on approximate triangle counting, there exist a few other works [50,53] that approximate the triangle counts in a network. DOULION [50] uses probabilistic graph stratification, [47] uses a linear algebraic method, and Kolountzakis et al. [53] uses sampling along with degree based vertex partitioning. DOULION is most likely the fastest among these methods; however, Seshadri et al. [45] have shown that the uniform triple sampling based method is cheaper in running time, and achieve as good or better accuracy than DOULION.

A set of recent works [86,91] considers the task of sampling from restricted networks that can only be crawled. The most notable among these is the work by Leskovec and Faloutsos [91] which used a collection of random walk methods, namely, BFS (breadth-first search), forest-fire, simple random walk (SRW), and snowball sampling for obtaining a representative sample of the restricted network. One can apply the above random walk methods for sampling a representative networks of appropriate size and return all the triples from that network as the sampled triples. However, such a sampling of triples does not guaranty uniform sampling of triples; furthermore, the user has no control over the probability distribution by which the triples would be sampled in the above approach. On the other hand, the MCMC method that we propose in this chapter can sample from any arbitrary user-defined distribution.

There are other recent works that adopt MCMC sampling strategy. Hubler et al. [90] use it for finding representative subgraphs, Bhuiyan et al. [33] use it for sampling graphlets, Maiya et al. [89] use it for sampling community structure, and Gjoka.Kurant.ea:10 [86] use it for finding the approximate degree distribution of the Facebook network. However, each of these works have a different objective and they sample from different population. Besides, none of these works consider sampling from from an arbitrary user-defined distribution.

6.3 Background

Newman, Watts and Strogatz [36] defined the transitivity of a graph G (say, $\gamma(G)$) as the fraction that represents the number of closed triples divided by the number of all the triples over the entire network.

$$\gamma(G) = \frac{|\Pi^{\triangle}|}{|\Pi|} = \frac{|\Pi^{\triangle}|}{|\Pi^{\angle}| + |\Pi^{\triangle}|} \tag{6.1}$$

Using Equation 2.1 and Equation 6.1, the triangle count ($\delta(G)$) of a network can be obtained from the transitivity of the network as below:

$$\delta(G) = \frac{1}{3} \cdot \gamma(G) \cdot |\Pi| \tag{6.2}$$

Following Equation 6.1, the transitivity of a graph, $\gamma(G)$, is the probability that an arbitrary triple in G is closed. To compute this probability exactly, we can simply count the closed triples ($\Pi^{\triangle}$) and the total number of triples (Π). There exist many algorithms for counting triangles exactly; however, for large networks they can be costly and one may be happy with a close approximation. If we compute the exact value of $|\Pi|$ (using Equation 6.2), we can approximate the triangle count with the identical approximation ratio of that of transitivity.

A uniform triple sampler can be used for approximating the transitivity of a graph G. For this, we sample a set of triples Ω ($\subset \Pi$) from G using a uniform distribution, and count the number of closed tripled in that set (say, $\Omega^{\triangle}$). Then, we define a random variable $\gamma_a(G) = \frac{|\Omega^{\triangle}|}{|\Omega|}$. The following lemma holds:

Lemma 2 $E[\gamma_a(G)] = \gamma(G)$

PROOF*: form the uniformity assumption,* $E\left[|\Omega^{\triangle}|\right] = \gamma(G) \cdot |\Omega|$*. Then,* $E[\gamma_a(G)] = E\left[\frac{|\Omega^{\triangle}|}{|\Omega|}\right] = \frac{E[|\Omega^{\triangle}|]}{|\Omega|} = \frac{\gamma(G)\cdot|\Omega|}{|\Omega|} = \gamma(G)$*.*

■

Thus, the expectation of the variable $\gamma_a(G)$ provides an unbiased estimate of the transitivity, which can subsequently be used in Equation 6.2 for finding an approximate triangle count in the graph G.

6.4 Methods

In this section, we discuss the proposed methods for MCMC based triple sampling.

6.4.1 Problem Formulation

Assume, Π is the set of triples in a large network G. Now, for a user defined non-negative weight function, $w : \Pi \rightarrow \mathbb{R}^+$, we can define a probability distribution over the set of triples (Π) by normalizing the weights, i.e, for a triple $t \in \Pi$, its probability is assigned as $\frac{w(t)}{\sum_{x \in \Pi} w(x)}$. The task of triple sampling is to sample triples from Π using the above probability distribution. We can represent the probability distribution using a probability mass function, f, which simply assigns a probability value to each of the triples in Π. If the weight function is a constant function, i.e, weights of all the triples are the same, then the above sampling becomes a uniform sampling of triples. For triple sampling, we also consider the scenario that the given network is restricted such that it is not explicitly visible, but can be crawled. More formally, in a restricted network, we can perform a random walk over the network, where at any given state of the walk, the currently visiting vertex, along with its adjacency list is visible to us.

In this chapter, we propose, explain and compare two MCMC based algorithms for solving the sampling problem that we define in the previous paragraph. The first among these two is Vertex-MCMC which we discuss in Section 6.4.3, and the second among these two is Triple-MCMC, which we discuss in Section 6.4.4. In the following we will discuss a direct sampling approach first to prove that for a restricted graph direct sampling is not feasible.

6.4.2 Direct Sampling

A direct sampling method for sampling a triple from Π first constructs the probability mass function (f) over the sample space (if not given) using the weight function, and from that it constructs the cumulative mass function (say, F) of f. Then it uses the inverse-transform method to sample an object from the sample space. More formally, if the sampled object is x, then $x = F^{-1}(U)$ where $U \sim Uni(0, 1)$. For computer implementation, we can simply store the function F in a vector of size $|\Pi|$ considering an arbitrary (but constant) ordering of triples, and then choose an index from the vector uniformly using binary search, and return the triple corresponding to that index. For a restricted network, construction of F is impossible, so direct sampling method is not applicable for such a network.

Authors of [44] and [45] use a slightly modified version of direct sampling for sampling triples. Theirs' is a two-step sampling process. The first step samples a vertex v from a multinomial distribution, ζ, which is constructed by summing $f(\cdot)$ of each of the triples at the vertex v. Mathematically, $\zeta(v) = \sum_{t \in \Pi_v} f(t)$. It is easy to see that $\sum_{v \in V} P_\zeta(v) = 1$. The second step samples a triple from the set of triples at vertex v (Π_v) using another multinomial distribution, τ_v. If $t \in \Pi_v$, then $P_{\tau_v}(t) = \frac{f(t)}{\zeta(v)}$. Thus, the probability of sampling a triple, $P(t) = P(t|v) \cdot P(v) = P_{\tau_v}(t) \cdot P_\zeta(v) = \frac{f(t)}{\zeta(v)} \cdot \zeta(v) = f(t)$, as desired. As there are $\mathcal{O}(n^2)$ triples in a graph, the cost of construction of cmfs of ζ and τ_v's is $\mathcal{O}(n^2)$, and the cost of sampling by inverse-transform is logarithm of the sample space (cost of binary search). Overall complexity of sampling k triples is $\mathcal{O}\left(n^2 + k(\lg n + \lg d_{max})\right)$, where n is the number of vertices, and d_{max} is the largest degree value for a vertex in the graph. Clearly, such a method is very inefficient.

However, note that the authors of [44] and [45] considered uniform distribution only. For this, $P_\zeta(v) = \frac{|\Pi_v|}{|\Pi|}$. Also, each of the τ_v's is trivially a uniform distribution. So, Z (cmf of ζ) can be computed in $\mathcal{O}(n)$ time using equation 2.2 considering that the degree of a vertex is available in $\mathcal{O}(1)$ time; we simply need to add the terms $\binom{d(v)}{2}$ in the above equation cumulatively for each of the vertices. Overall complexity of sampling k triples is then $\mathcal{O}(n + k \lg n)$. Thus, the direct sampling of triples is efficient for uniform sampling, but not for arbitrary sampling.

Example: For uniform triple sampling of the graph presented in Figure 2.1, we choose a vertex with the distribution ζ. Under this, the vertex 3 is selected with probability 3/8, as there are three triples for which vertex 3 is the center. If vertex 3 is selected, we randomly choose two vertices from the adjacency list of 3 $(2, 4, 5)$ and construct one of the three possible triples and return. Consequently, the probability of triple $(3, 4, 5)$ being selected is $(3/8 \times 1/3 = 1/8)$, which is equal to $1/|\Pi|$, as desired. ■

6.4.3 MCMC Walk Over Vertices for Triple Sampling

We have seen in previous section that sampling a triple from an arbitrary distribution requires the construction of the cmf of ζ. For a restricted graph this is an infeasible task due to the lack of information. Besides, this construction takes $\mathcal{O}(n^2)$ time. If we consider a graph which is unrestricted but dynamic, a direct sampling method can be used on this graph, but it will be very inefficient; every change (addition or deletion of vertices, or edges, or modification of weight values) in the network enforces the reconstruction of ζ (and also τ_v for some of the vertices) which has a quadratic complexity.

Our first indirect method to address the above limitation is to use an MCMC sampling method that does not construct ζ explicitly. We call it Vertex-MCMC; the justification of this name will be clear in short time. Vertex-MCMC sampling uses a similar approach as the two-step direct sampling, but unlike the latter, it replaces the first-step (sampling a vertex from ζ) with an indirect sampling via MCMC. The second step of Vertex-MCMC sampling remains unchanged from the two-step direct sampling method. The motivation of such an indirect method from dynamic graph point of view is that, in a dynamic graph in between two sampling iterations, only a few edges will be added or deleted, so we need to reconstruct the τ_v distributions only for a few vertices that are incident to the modified edges. Nevertheless, the construction of ζ is completely avoided. Also, construction of τ_v only requires $\mathcal{O}(d(v)^2)$ time, which is much better than constructing ζ which takes $\mathcal{O}(n^2)$ time. More details of Vertex-MCMC is given below.

For any MCMC algorithm, we need to define the states, the state transition process, the transition probability matrix, and the desired probability distribution. For Vertex-MCMC, the set of states are the vertex-set V and the transition over the states happens along the edges (E). So the MCMC process is simply a random walk on the graph G. However, we want the stationary distribution of this walk to be identical to the desired distribution, which is ζ—identical to the desired distribution of vertices for a two-step sampling. To achieve the desired sampling distribution we will use Metropolis-Hastings (MH) algorithm.

Assume that MCMC random walk of a Vertex-MCMC based triple sampler is visiting a vertex v. As was discussed in Section 2.11.1, MH algorithm uses a proposal distribution (q) to make a trial move; Vertex-MCMC chooses q to be uniform over the neighborhood of v, in other word, it chooses one of the vertices (say, u) from the adjacency list of v uniformly. Therefore, the proposal distribution $q(v,u) = 1/d(v)$; here $d(v)$ is the number of nodes adjacent to node v. $q(v,u)$ represents the probability of an adjacency node u to be selected from current node v. Similarly, $q(u,v) = 1/d(u)$. Now, using Equation 2.7, the acceptance probability of the proposal move is as shown in Equation 6.3.

$$\begin{aligned} \alpha(v,u) = \min\left\{1, \frac{P_\zeta(u)\cdot\frac{1}{d(u)})}{P_\zeta(v)\cdot\frac{1}{d(v)}}\right\} \\ = \min\left\{1, \frac{\sum\limits_{t\in\Pi_u} f(t)\cdot d(v)}{\sum\limits_{t\in\Pi_v} f(t)\cdot d(u)}\right\} = \min\left\{1, \frac{\sum\limits_{t\in\Pi_u} w(t)\cdot d(v)}{\sum\limits_{t\in\Pi_v} w(t)\cdot d(u)}\right\} \end{aligned} \tag{6.3}$$

Algorithm 10 illustrates the verex-MCMC algorithm. To sample a vertex from ζ, the algorithm calls the subroutine shown in Algorithm 11, which is simply an implementation of MH algorithm, where the sample space is the vertex set, and the

Algorithm 10 Triple sampling Vertex-MCMC

```
1: procedure tripleSampling2($G, k, \{w(i)\}_{i \in \Pi}$)
     ▷ Graph $G$ is given as vectors of adjacency vector, $k$ is number of triples to be sampled, $w(\cdot)$ is user-defined weights of the triples.
2:     $S \leftarrow \phi$
3:     $u$ = Uniform from the vertex-set $V$.
4:     while $|S| \neq s$ do
5:         $v \leftarrow SelectNodeMCMC(u, \{w(i)\}_{i \in \Pi})$          ▷ see Algorithm 11
6:         Select a triple $t \in \Pi_v$ using $t \sim \tau_v$
7:         $S.add(t)$
8:     end while
9:     return $S$                                        ▷ Return a set of $s$ triples.
10: end procedure
```

Algorithm 11 MCMC node sampling

```
1: procedure SelectNodeMCMC($current, \{w(i)\}_{i \in \Pi}$)          ▷ $current$ is the currently visiting node, $w(\cdot)$ is user-defined weights of the triples.
2:     $W_{current} = \sum_{x \in \Pi_{current}} w(x)$,   ▷ compute if not available from earlier iterations
3:     $next$ = Uniform from $adj(current)$                    ▷ Proposal step
4:     $W_{next} = \sum_{x \in \Pi_{next}} w(x)$        ▷ compute if not available from earlier iterations
5:     $acceptance \leftarrow \frac{W_{next} * d(current)}{W_{current} * d(next)}$                ▷ See Equation 6.3
6:     if $uniform(0,1) \leq acceptance$ then
7:         return $next$
8:     end if
9:     return $current$
10: end procedure
```

target distribution is ζ. Once a vertex is selected on Line 5, Vertex-MCMC computes the cmf of τ_v, and uses the direct sampling method to sample a triple using τ_v (Line 6). It is important to note that, in Line 2 and Line 4 of Algorithm 11, we compute the sum of weights associated to only the triples of u and v, which can be accomplished in a restricted graph. Thus this method works for a restricted graph. Besides it is efficient; the complexity of Vertex-MCMC for sampling k triples is $\mathcal{O}(kd_{max})$, as weight computations, and neighbor selection in Algorithm 11 can be performed in $\mathcal{O}(d_{max})$ time.

Lemma 3 *Algorithm 10 samples each triple t with a probability $\frac{w(t)}{W}$.*

Proof*: The first step of the algorithm uses MH algorithm for sampling a vertex from the ζ distribution. This will be successful if the Markov chain converges to the desired stationary distribution. To obtain a stationary distribution the random walk needs to be finite, irreducible and aperiodic [92]. The state space is finite with size $|V|$, because the number of vertices is finite. We also assume that the input graph G is connected, so in this random walk any state u is reachable from any state v with a positive probability and vice versa, so the random walk is irreducible. Finally the walk can be made aperiodic by allocating a self-loop probability at every node [1].*

Once we know that MCMC sampling chooses a vertex from the ζ distribution, the remaining part of the proof follows from the correctness of the two-step direct sampling method. ■

Uniform sampling using Vertex-MCMC:, $P_\zeta(v) = \binom{d(v)}{2} = \frac{d(v)(d(v)-1)}{2}$, The Equation 6.3 then changes as follows:

$$\begin{aligned} \alpha(v,u) &= \min\left\{\frac{d(u)(d(u)-1)/2 \cdot 1/d(u)}{d(v)(d(v)-1)/2 \cdot 1/d(v)}\right\} \\ &= \min\left\{\frac{d(u)-1}{d(v)-1}\right\} \end{aligned} \tag{6.4}$$

We do not show the pseudo-code of uniform triple sampling using Vertex-MCMC. But, it is easy to obtain by making minor changes in Algorithm 10 and Algorithm 11. In Line 2 and Line 4 of Algorithm 11, we can compute the vertex weights in $\mathcal{O}(1)$ time using the degree value of the corresponding vertex; the acceptance probability (Line 5) changes as shown in Equation 6.4. Finally, the Line 6 in Algorithm 10 requires to sample a triple from a uniform distribution instead of τ_v, which also takes $\mathcal{O}(1)$ time. Due to the above changes, Vertex-MCMC is faster in uniform sampling setting than

[1] This is required only from a theoretical standpoint; in our experiment we do not allocate any self-loop probability explicitly.

weighted sampling setting. However, the theoretical complexity of the uniform triple sampling is still $\mathcal{O}(kd_{max})$; although the weight computation cost is constant, we still need to find a neighbor of the currently visiting vertex, which in the worst case can take $\mathcal{O}(d_{max})$ time.

6.4.4 MCMC Walk Over Triples

Our second indirect sampling method is named Triple-MCMC, which performs MCMC walk over the triples. Triple-MCMC avoids computing *cmf* for both the distributions (ζ and $\{\tau_v\}_{v \in V}$). In fact, Triple-MCMC is completely oblivious about the total number of triples in the graph. The set of states for this sampling algorithm is the set of all the triples, Π. Thus the random walk proceeds over the set of triples along a neighborhood graph which is defined below.

The neighbor of a triple is another triple with two common vertices. Thus, the Triple-MCMC sampling obtains a sequence of dependent samples, where a sampled triple shares two vertices with the previous sampled triple. To compute the neighbor-set of a triple t, we need to find the other triples that can be obtained by replacing exactly one of the vertices of t.

Example: Suppose we are performing an MCMC walk on the graph shown in Figure 6.1 (a). Let $\langle v_2, v_3, v_8 \rangle$ be the currently visiting triple (triangle that is shown in bold line). In Figure 6.1 (b) we show the information of all its neighbors. The list labeled by v_2 contains the vertices that can be used to replace vertex v_2 to get a valid neighboring triple and similarly for the list labeled by v_3 and v_8. If the MCMC random walk chooses to go to the neighboring triple by replacing the vertex v_8 with v_1, the next sampled triple becomes a path $\langle v_1, v_2, v_3 \rangle$. On the other hand, if the vertex v_3 is replaced by v_7, we get the closed triple $\langle v_2, v_7, v_8 \rangle$ where the center of the triple is taken as v_7, which is the lastly added vertex of the triple. The transition between triples happens only between the neighboring triples. For example, the transition probability between $\langle v_2, v_3, v_8 \rangle$, and $\langle v_4, v_5, v_6 \rangle$ is zero, as they are not neighbors of each other according to our neighborhood definition. ■

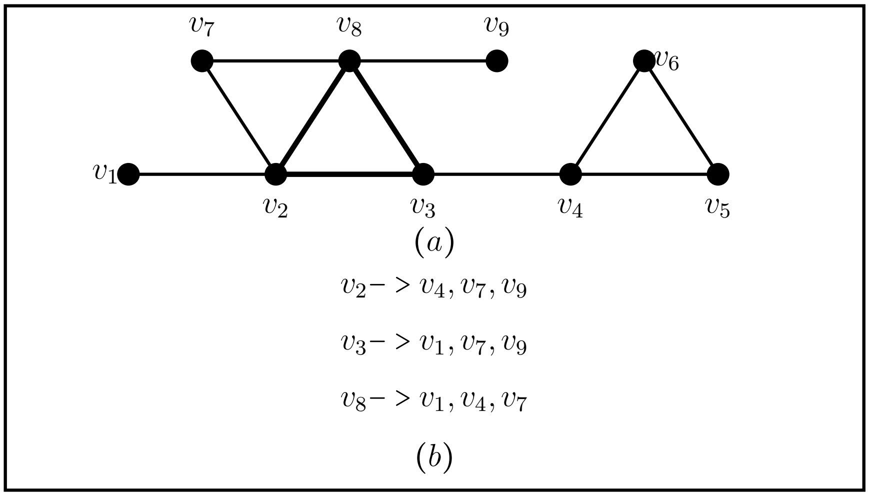


Figure 6.1.: Induced triple and neighbors

Let's assume that the random walk of Triple-MCMC is visiting a triple t. For proposal distribution (say q), we choose one of the triples from t's neighborhood (say, s) uniformly. So, $q(t, s) = \frac{1}{|\mathcal{N}(t)|}$. Here, $\mathcal{N}(t)$ is the set of neighbors of triple t. Using Equation 2.7, the acceptance probability of the proposal move is obtained as below:

$$\alpha(t, s) = \min\left\{1, \frac{f(s) \cdot \frac{1}{|\mathcal{N}|(s)|}}{f(t) \cdot \frac{1}{|\mathcal{N}(t)|}}\right\} = \min\left\{1, \frac{w(s) \cdot |\mathcal{N}(t)|}{w(t) \cdot |\mathcal{N}(s)|}\right\} \tag{6.5}$$

We show a pseudo-code in Algorithm 12. Since, the random walk is performed over the triple space, we initialize the walk with an arbitrary triple t_p, any path of length 2 suffices (Line 3). Now, for the currently visiting triple, t_p, we want to find the next triple using MH algorithm. For this, we first find all the neighboring triples of t_p (Line 5). Reader may review the Figure 6.1 to refresh the notion of the neighboring triples. This computation requires finding unions or intersections of the adjacency lists of the current triple's vertices, and its complexity is $\mathcal{O}(d_{max})$. Then a neighboring triple (say, t_q) is selected uniformly from all the neighbors, and accepted with the probability computed on Line 8. If the move is rejected, the currently sampled triple is sampled again. The process continues until k samples are obtained. The overall cost of obtaining k samples is $\mathcal{O}(kd_{max})$.

Algorithm 12 Triple sampling Triple-MCMC

```
procedure tripleSampling3($G, k, \{w(i)\}_{i \in \Pi}$)
    ▷ Graph $G$ is given as vectors of adjacency vector, $k$ is number of triples to be
    sampled, $w(\cdot)$ is user-defined weights of the triples.
    $S \leftarrow \phi$
    $t_p \leftarrow$ a random triple                    ▷ Any length 2 path is sufficient
    while $|S| \neq s$ do
        $t_q \leftarrow RandomNeighborTriple(t_p)$
        $n_p \leftarrow NeighborCount(t_p)$
        $n_q \leftarrow NeighborCount(t_q)$
        $acceptance \leftarrow \frac{w(t_q)*n_p}{w(t_p)*n_q}$
        if $uniform(0,1) \leq acceptance$ then
            $S.add(t_q)$
            $t_p \leftarrow t_q$
        else
            $S.add(t_p)$
        end if
    end while
    return $S$                                          ▷ Return a set of $s$ triples.
end procedure
```

Lemma 4 *Algorithm 12 samples each triple t with a probability proportional to $w(t)$*

PROOF: *Algorithm 12 performs a random walk over the triple space (Π) with a stationary distribution, which is proportional to w. So, we only need to show that the walk converges to a stationary distribution. The state space Π is finite, because the number of triples is finite. We also assume that the input graph G is connected, so in this random walk any triple y is reachable from another triple x with a positive probability and vice versa, so the random walk is irreducible. Finally the walk can be made aperiodic by allocating a self-loop probability at every node. Thus the random walk reaches a stationary distribution, which is proportional to w. Thus the lemma is proved.* ∎

Uniform Triple Sampling

For uniform sampling, the weight of all the triples are equal i.e, if we set $w(s) = w(t) = 1$ in Equation 6.5, we can make it a uniform sampler. The acceptance probability changes as below: $\alpha(s,t) = \min\{1, |\mathcal{N}(t)|/|\mathcal{N}(s)|\}$. MH algorithm guaranties that the Algorithm 12 using the above acceptance probability yields a uniform triple sampler.

6.4.5 Selection of Initial State

In our implementation we randomly select the initial state. Consequently, different execution of our algorithms will start from different initial states. However, for a rapidly mixing MCMC walk, the influence of initial state disappears after a few iterations; the number of iterations for mixing is called the mixing time or burn-in time of the MCMC walk, which can be computed using Geweke diagnostics [93] (see Section 6.5.5).

6.5 Experiments and Results

The triple sampling algorithms that we propose can sample triples from arbitrary distribution. In our experiments we first show the performance of uniform triple sampling. Then we show the performance of triple sampling from a nonuniform distribution. Finally, we demonstrate that, uniform sampling of triples can be applied for approximating triangle count, which provides an application driven method for measuring sampling effectiveness.

Table 6.1.: Small real-life networks used in sampling quality experiments.

Name	Nodes	Edges	Triple Count ($\|\Pi\|$)
ca-Hepth	8,638	24,806	297,397
ca-Grqc	4,158	13,422	227,919
ca-Cond	21,363	91,286	1,959,920

Table 6.2.: Large real-life networks used in approximate triangle count experiments.

Name	Nodes	Edges	Triangle Count
AS-Skitter	1,694,616	11,094,209	28,769,842
flickr	1,624,992	15,476,835	548,646,525
livejournal	5,189,809	48,688,097	310,784,143
orkut	3,072,441	117,185,083	627,584,181
Soc-LiveJournal	4,843,953	42,845,684	285,688,896
Wikipedia 2005/11	1,596,970	18,539,720	44,667,088
Wikipedia 2006/9	2,935,762	35,046,792	84,018,181
Wikipedia 2006/11	3,099,074	37,042,065	88,823,813
Wikipedia 2007/2	3,512,462	42,374,383	102,434,914

6.5.1 Datasets

All the graphs [2] listed in Table 6.1 and 6.2, are undirected, unweighted, simple and connected. We preprocess them to ensure these properties. The specification of the graphs (vertex count and edge count) may not match with the source, as in source, for some networks an undirected edge is represented by two directed edges in opposite directions; in our representation, for such edges we discard one edge of the edge-pairs. Additionally, we ensure that the graph is connected as MCMC algorithms perform a random walk over the graph. However, for all the experimental graphs, their connected part retain more than 90% of the edges.

[2] obtained from http://snap.stanford.edu, http://socialnetworks.mpi-sws.org/, http://www.cise.ufl.edu/research/sparse

6.5.2 Uniform Sampling Performance

It can be easily demonstrated that for uniform sampling Vertex-MCMC is more efficient than Triple-MCMC. Vertex-MCMC sampling requires weight of each vertex to construct *cmf*(ζ); however, the weight computation cost for each node is constant. On the other hand, Triple-MCMC needs to generate the list of neighbors of the current triple t, represented as $\mathcal{N}(t)$; which it uses to select the next triple s. For a newly sampled triple, construction of $\mathcal{N}(t)$ requires three set operations (union or intersection) over the adjacency list of t's vertices, whose cost is linear with respect to the length of the adjacency lists of t. As a result, Vertex-MCMC sampling is better than Triple-MCMC method for uniform triple sampling.

Our first experiment shows the performance of Vertex-MCMC for sampling triples uniformly. For this we compare Vertex-MCMC with the direct sampling method discussed in Section 6.4.2. We use ca-Hepth, ca-Grqc, and ca-Cond as input graphs; their statistics are shown in Table 6.1. For each of these graphs, we run the sampler for $|\Pi|\times 50$ iterations, where Π is the set of distinct triples in that graph (see, Column 4 in Table 6.1). For example, in "ca-Hepth", there are in total $297,397$ distinct triples. Hence, using each of the sampling algorithms, we sample a total of $297,397\times 50$ triples. As the sampling proceeds, we keep track of the sample count for each of the triples. Now, for a perfect uniform sampler, sample count of each triple will be 50, which is impossible for a random process. To analyze the distribution of sample count, we create the frequency histogram of sample counts for the ca-Hepth network (shown in Figure 6.2); in this plot, x-axis shows different sample count values, and y-axis represents the number of distinct triples that achieves that value for its sample count. The shape of the histogram is a perfect normal graph, which is expected (explained below) from an ideal iid distribution.

We also show the statistics of sample counts in Table 6.3 indicating variance for both of the sampling algorithms. In the same table, we also report the value of variance for the ideal case. For a graph with $|\Pi|$ triples, we perform a uniform sampling

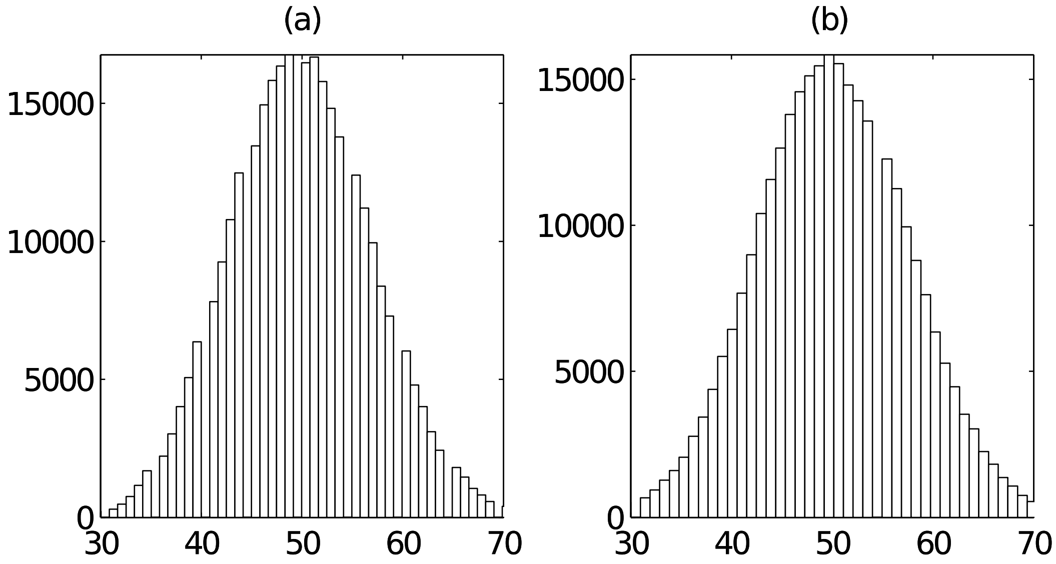


Figure 6.2.: Frequency histogram of the visit counts on ca-Hepth network using (a) Direct triple sampling (b) Vertex-MCMC triple sampling.

Table 6.3.: Comparison of variances among Direct sampling, Vertex-MCMC sampling, and ideal sampling on different graphs (Median is 50 for all the cases).

Graph	Ideal	Direct	Vertex -MCMC
ca-Hepth	49.99	49.93	59.37
ca-Grqc	49.99	49.96	58.15
ca-Cond	49.99	50.09	52.05

for $|\Pi|.i$ times. The random number that denotes the count by which a triple is sampled (sample count) can be described by a binomial distribution $\mathcal{B}(k, m, p)$, where $m = |\Pi|.i$ and $p = \frac{1}{|\Pi|}$. For this distribution, the median sample count will be identical to the mean, which is $m.p = |\Pi|.i.\frac{1}{|\Pi|} = i$ and the variance is $m * p(1-p) = \frac{i(|\Pi|-1)}{|\Pi|}$. If we set the m value sufficiently large, this binomial distribution would resemble a normal distribution (as is shown in Figure 6.2). For ca-Hepth graph, $x = 297,397$, by setting $i = 50$, we expect that the median of sample count will be 50 with the variance $\frac{50*(297,397-1)}{297,397} = 49.99$. Using Vertex-MCMC uniform sampler, the median of sample count is 50; the variance is 59.37 (variance is 49.93 for Direct triple sampler) as shown in Table 6.3. Direct method's performance is almost identical to an ideal sampler; for Vertex-MCMC method, variance of sample count is slightly bigger. We also show similar results for "ca-Grqc", and "ca-Cond" graphs in Table 6.3; for these graphs the histograms are not shown, as they are almost identical to the one for the "ca-Hepth" graph. Note that, We do not perform this experiment for large graphs, because for this experiment we need to store the visit count of all the triples in the memory, and the number of such triples is in the order of billions for large graphs.

6.5.3 Verification of Nonuniform Sampling

In this experiment we verify the quality of sampling when the triples to be sampled follow a nonuniform distribution. We compare Direct sampling with both Vertex-MCMC and Triple-MCMC algorithms. In this experiment we use the dataset listed in Table 6.1 for the reason discussed in Section 6.5.2. Here, our objective is to sample triple t in proportion to $w(t)$. For this experiment, we consider $w(t) = |\mathcal{N}(t)|$, i.e., a triple is sampled with the probability proportional to the size of its neighborhood. One motivation of choosing such a sampling distribution can be to sample triples from a community or a dense region of a graph; in such a neighborhood, a triple will be surrounded by many triples, so $|\mathcal{N}(t)|$ will be high for a triple t in a dense neighborhood. For the above choice of target distribution, the acceptance probability of Vertex-MCMC is,

$$\alpha(v,u) = \min\left\{1, \frac{\sum_{t\in\Pi_u} \mathcal{N}(t)\cdot d(v)}{\sum_{t\in\Pi_v} \mathcal{N}(t)\cdot d(u)}\right\}$$

and the acceptance probability of Triple-MCMC is $\alpha(t,s) = 1$.

For a network with $|\Pi|$ triples, the desired distribution over Π can be expressed as a vector f of size $|\Pi|$, here, $f(t) = \frac{w(t)}{W}$. For each of the graphs, we run each sampler for $|\Pi|\cdot i$ times (we choose $i = 10$ for our experiment). The distribution f can be approximated by the sample frequency of each of the triples, Therefore, $\hat{f}(t) = \frac{count(t)}{|\Pi|\cdot i}$; here, $count(t)$ is the number of times the triple t was sampled and $\hat{f}$ is the approximation

Table 6.4.: Correlation between target distribution and achieved distribution by different sampling algorithms.

Graph	Direct	MCMC	
		vertex	triple
ca-Hepth	0.86	0.86	0.87
ca-Grqc	0.87	0.87	0.92
ca-Cond	0.93	0.93	0.94

of f that is obtained by the sampling algorithm. The performance of a sampling method can be measured by the correlation between f and $\hat{f}$. Table 6.4 shows that all the three methods achieves excellent value for the correlation (more than 0.85). Interestingly, direct method sometimes perform worse than the other methods, our investigation shows that this is because of the precision issue of the floating-point while handling very small probabilities. More precisely, Ca-Hepth network has $227,919$ triples, and the cumulative mass probablities (which sums to 1) is stored in a vector of that size; in this vector the difference between successive cells are somethimes as small as 10^{-8}, and to perform well a uniform random number generator's precision has to be good for that many decimal points, which apparently is not true for existing random number generators. In all our experiments we use Boost random number generator library that has much better performance that those available in standard C++ library. In Figure 6.3, we compare f vs $\hat{f}$ distributions for all the three methods using scatter plots for one of the graphs (ca-Grqc). The superiority of Triple-MCMC over other sampling methods is easily visible in this figure.

Table 6.5 shows the execution time for sampling $1k$ and $10k$ triples from the networks using different sampling algorithms. As the table shows, Triple-MCMC is much better than Vertex-MCMC and the direct method. For example, Triple-MCMC takes only 0.08 second to sample $1k$ triples from ca-Cond network, whereas Direct method and Vertex-MCMC takes 145.5 and 82.65 seconds respectively. This is because, Triple-MCMC does not need to compute the *cmf* ζ and τ_v explicitly. Computing ζ is a fixed cost for the direct sampling method. Vertex-MCMC distribute

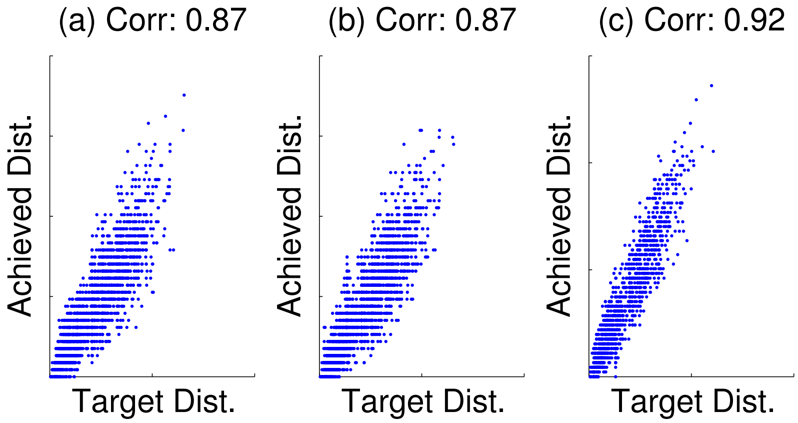


Figure 6.3.: Target distribution vs achieved distribution plot for ca-Grqc network (a) Direct triple sampling (Corr. 0.87) (b) Vertex-MCMC triple sampling (Corr. 0.87) (c) Triple-MCMC triple sampling (Corr. 0.92).

this fixed cost over the iterations because it computes the ζ of each vertex only on demand. On the other hand, Triple-MCMC computes $w(t)$ of a specific triple only on demand. If we take more samples, the difference between the direct sampling and Vertex-MCMC slowly diminishes, as with many iterations, both the methods can amortize the fixed cost over those iterations. Here, it should be noted that, *cmf* τ_v is not explicitly stored in memory. Storing τ_v will require memory in the order of $\mathcal{O}(|\Pi|)$, which is same as enumerating the whole set of triples. And if enumeration is possible, then we do not need to sample triples in the first place.

Table 6.5.: Execution times of the algorithms for sampling $1k$ triples and $10k$ triples.

Graph	Direct		Vertex-MCMC		Triple-MCMC	
	Time(s)		Time(s)		Time(s)	
	$/1k$	$/10k$	$/1k$	$/10k$	$/1k$	$/10k$
ca-Hepth	5.07	27.46	3.87	27.66	0.03	0.33
ca-Grqc	10	78.83	8.83	80.51	0.05	0.47
ca-Cond	145.5	1225.05	82.65	1075.79	0.08	0.81

6.5.4 Approximate Triangle Counting

In this experiment, we compare the performance of Vertex-MCMC algorithm with direct triple sampling [44, 45], and one of the sampling method by Buriol et al. [54] [3]. Note that, recent works [45] have shown that the direct sampling method is the best among the existing methods for approximate triangle counting; so we do not include other triangle counting methods such as DOULION [50] in this experiment. However, we do include Buriol et al.'s method in this comparison, because it is also a triple sampling method like Vertex-MCMC. We exclude Triple-MCMC in this comparison as when performing uniform sampling its execution time is not competitive with these methods (although it is the best choice for weighted sampling). The task assigned to each sampling method is to approximate the triangle count by sampling triples from uniform distribution. Here, we use the sampled set of triples to approximate triangle count using the idea that was explained in Section 2.3. For our experiment, we use 9 large real-life networks; name and statistics of these networks are shown in Table 6.2.

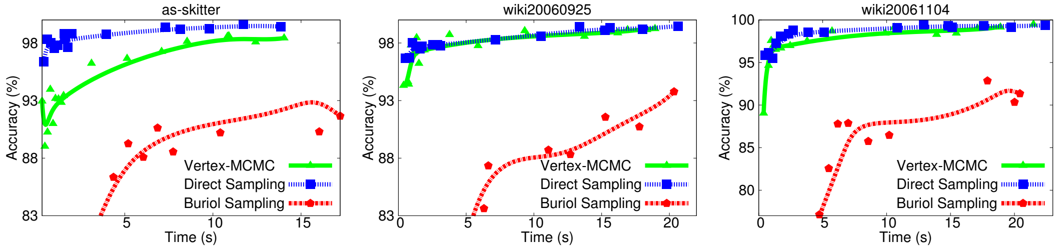


Figure 6.4.: Comparison (of running time and approximation accuracy of transitivity) among the sampling algorithms (a) as-skitter (b) wikipedia 2006/09 (c) wikipedia 2006/11. Exact triangle counting times are 3.8s, 66.6s and 72.57s respectively. Results on other graphs are similar, hence not shown.

The performance of approximate triangle counting is measured by two metrics: execution time and accuracy. Typically, the method that wins in accuracy loses in running time. So, to make the comparison easy, we compare the accuracy (plotted on y axis) of different methods against different running time (plotted on x-axis).

[3] this method is actually proposed for arbitrary edge stream setting, but for fair comparison we implement it as a non-stream in memory method.

However, do note that for a given value for time, the number of triples that the methods sample differ. We show the results in Figure 6.4(a-c). We fitted the data with Bezier curve to show the trend of the algorithms. All the accuracy and execution times are average value that are computed from 10 runs of the algorithms. We can see that the direct sampling method performs the best, even for a small number of samples, and its performance improvement remains almost flat as the sample count increases; on the other hand, Vertex-MCMC improves sharply as the number of samples increases, and for some graphs its performance even surpasses the performance of direct sampling. So, Vertex-MCMC is particularly suitable for large graphs, where a sampling method can afford to take many sample, and yet can be competitive with an exact algorithm. For instance, in wiki20060925 graph, both direct sampling and Vertex-MCMC obtain 90% counting accuracy for a 6 seconds execution time, but the exact method that uses an efficient edge iterator algorithm takes 67 seconds to execute. The charts in this figure also confirm that Buriol et el.'s method is not competitive with either of these methods.

The accuracy of a direct sampling-based method is better than that of a Vertex-MCMC based method. This is because the latter performs indirect sampling in which a pair of consecutive samples are dependent. So, it's result has high variances and it requires more samples in order to ensure uniformity of sampled set of triples. However, MCMC based methods can work perfectly on restricted or dynamic networks, whereas direct sampling based methods are not applicable to those.

6.5.5 Convergence Analysis

Convergence analysis is important for any MCMC sampling because through such analysis we can estimate the mixing (or burn-in) time (number of walks to overcome the influence of the starting state) of a Markov chain. Mixing time depends on (i) the neighborhood structure of the space on which the walk is performed, and (ii) the desired target distribution. In this experiment, we study the mixing time of the

Vertex-MCMC method while sampling from a uniform target distribution. The result of Triple-MCMC is similar, hence is not included. This experiment is not shown for user-defined distribution, as depending on the choice of distribution, the conclusion can be very different.

For this experiment we consider X to be a single sequence of samples. Also let, $X_i = 1$ if ith sample of the sequence is a triangle and $X_i = 0$ otherwise. Consequently, our metric of interest for this experiment is transitivity $\gamma(G)$ (See Equation 6.1), which can be estimated from uniform sampling of triples. In fact, the expected value of the random variables in the sequence X gives as unbiased estimate of transitivity, i.e., $E(X) = \gamma(G)$. Now, we compute burn-in using Geweke diagnostics [93].

Geweke considers two subsequences of samples, X_a form the beginning part of X (typically first 10%) and X_b from the last part of X (typically last 50%). From these two subsequence he computes z-statistic: $z = \frac{E(X_a)-E(X_b)}{\sqrt{Var(X_a)+Var(X_b)}}$. X_a ans X_b goes further apart as the number of samples is increased. Consequently, the correlation between the subsequences decreases. After convergence, there is no correlation between X_a and X_b, and z becomes normally distributed with mean 0 ($E[X_a] = E[X_b] = \gamma(G)$, so, $E[X_a] - E[X_b] = 0$) and variance 1. The number of iterations that it takes for the z-score to fall between [-1, 1] is called the burn-in time. However, one should run the experiment for at least a few distinct walks, and declare convergence when z-scores from all the walks fall within the $[-1, 1]$ range.

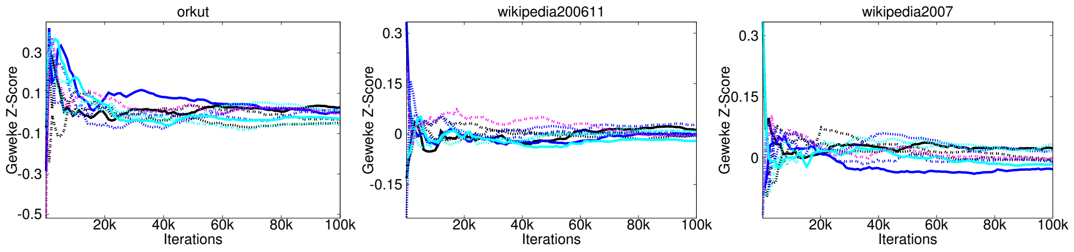


Figure 6.5.: Geweke z score of transitivity for iterations $100 - 100k$ (a) orkut (b) wikipedia 2006/11 (c) wikipedia 2007/2.

For each dataset, We use 10 distinct walks. For each walk, we compute z-score starting from 100 upto $100k$. We declare convergence when all 10 z-scores fall in the range $[-1, 1]$. The convergence of z-score is shown in Figure 6.5 for 3 graphs; the others are similar, hence not shown.

6.6 Conclusion

In this chapter, we propose two MCMC based algorithms for sampling triples form a large network. We show experimental results that demonstrate that both the algorithms achieve excellent performance while sampling triples from a large network using a given distribution. Direct sampling method's performance is almost identical to an ideal sampler, but it is costly, specifically while sampling from a weighted distribution. On the other hand, the MCMC sampling methods that we propose is faster as it does not compute the *cmf* of the desired distribution. More importantly, MCMC sampling methods can sample triples from networks that are restricted or dynamic, for which direct sampling methods fail.

7 ESTIMATING GRAPHLET BASED NETWORK METRICS

In this chapter we propose a graphlet sampling algorithm GUISE. For building the GFD fingerprint of a network the relative frequencies among various graphlets suffices. Also note, if a frequency distribution among various graphlets is available, we can easily reconstruct the count of all the graphlets from the count of only one of the graphlets. For instance, if a GFD considers all three, four, and five size graphlets, from the information of triangle count [1], the count of all other graphlets can be recovered very easily. In reality, exact count is only a reflection of the size of the input graph, so an analysis should only use relative frequencies among various graphlets, so that the metric can be used across different graphs. Earlier works [21] construct GFD after computing the frequencies of each of the graphlets; in this chapter, we show a sampling-based method that approximates (almost identically) the GFD of a network with millions of nodes in a minute, for which the exact counting of graphlet frequencies takes more than three days!

7.1 Related Works

The problem of designing a fast approximation algorithm for computing the frequencies of sub graphs in a given graph was targeted by R. Duke et al. [80] back in 1993 where given a labelled graph G of n vertices and a list of all labelled graphs on k vertices, they attempted to provide for each graph H of this list an approximation to the number of induced copies of H in G keeping the approximation error small. The main tool in designing their algorithm is a variant of the regularity lemma of Szemeredi.

[1] A triangle is a size-3 graphlet which is shown as g_2 in Figure 1.1; scalable algorithms are available for counting triangles in graphs with millions of vertices and edges.

No work exists that sample graphlets from a large graph. The closest to our work is probably the work by N. Kashtan et al. [79], who use sampling algorithm for estimating subgraph concentrations to detect network motifs. They also build a software tool, named MFinder which is highly popular. However, this work finds motifs that have unusual high concentrations, on the other hand GFD construction requires the relative frequencies of all the graphlets (both frequent and infrequent). FANMOD [94] uses the technique of node-sampling to detect network motifs, their search is also targeted for finding frequent motifs. More recently, the work proposed in this chapter [33,34] has inspired several improvements and generalizations [95,96], for efficient estimation of motif statistics in large networks.

Among other available tools for network motif detection in a biological network MAVisto [97], NeMoFinder [98] and Kavosh [99] are based on network centric algorithm depending on pattern growth tree. All these three algorithms does exact census of the sub graphs in a network. Apart from these, there are motif centric algorithms like Grochow [100] and MODA [101] specially for PPI networks. Unfortunately performances of all these algorithms are heavily domain specific and strictly defined by the network size and the size of the motif. Other than these, a work of N. Przulj and her team [102] have described two heuristics: Targeted Node Processing (TNP) and Neighborhood Local Search (NLS) for graphlet frequency estimation that work well for high-confidence PPI and geometric random networks. But neither of them work well for ER-DD and SF networks in terms of the error estimates and running times.

7.2 Research Contribution

We propose GUISE [2], an algorithm for constructing graphlet frequency distribution (GFD) that samples graphlets without enumerating the occurrences of each of the graphlets. GUISE uses a Markov Chain Monte Carlo (MCMC) sampling strategy to sample the embedding occurrences of each of the graphlets, such that each such

[2] GUISE is an anagram of the bold letters in **U**n**I**form **S**ampling of **G**aphl**E**ts

embeddings [3] is sampled with a uniform probability—this enables GUISE to build a frequency histogram of various graphlets by accumulating the respective counts of each of the graphlets from the obtained samples. We prove both theoretically and empirically that GUISE can build the GFD of a network which is all but identical with the one that we can achieve by enumerating (thus counting) the occurrences of all the graphlets. The key difference is that on a graph with millions of vertices and edges, GUISE takes minutes, whereas a counting based approach may takes months, or even years.

The contributions of this work are summarized as below:

- We propose GUISE, a sampling algorithm that samples embedding occurrences of various graphlets in a large network from a uniform distribution; GUISE uses an MCMC sampling algorithm that is theoretically sound.
- We show that GUISE can be used for building GFD, a fingerprint for analyzing large networks using local topological templates. Without counting the frequencies of various graphlets, GUISE builds an approximate GFD in minutes, for which the counting based method may take several months.
- We show experimental results on large real-life networks to validate sampling effectiveness, convergence, and efficiency of GUISE for constructing GFD.
- We provide spectral-gap analysis and variation distance analysis to show the effectiveness of MCMC walk in sampling graphlets.

7.3 Method

As explained earlier, a naive approach to generate GFD is to count the frequencies of each graphlet, which requires enumeration of all distinct induced embeddings. This task becomes infeasible when the input graph is large. We propose an efficient method

[3] We sometimes use graphlet to mean a specific embedding of a graphlet, if it is clear from the context of the discussion.

that utilizes uniform sampling to approximate the GFD. Below, we discuss the method in details.

7.3.1 Uniform Sampling of Graphlets for GFD Construction

Given a graph G, assume the set $\mathcal{S}$ contains all the (induced) embeddings of all the graphlets in the graph G. Then, $|\mathcal{S}| = \sum_{i=1}^{29} f(i)$, where $f(i)$ is the frequency of graphlet i in the graph G. Then, the task of uniform sampling of graphlets is to sample one of the graphlet embeddings in $\mathcal{S}$ uniformly at random; i.e., the selection probability of each of the graphlet embeddings is exactly $1/|\mathcal{S}|$. The task is no harder than the enumeration of all the graphlets in $\mathcal{S}$. In fact, after enumerating all the graphlet embeddings, we only need a random number generator to sample one of those embeddings from an iid distribution. But, enumerating all the graphlets is not practical, so we want to sample a graphlet uniformly without explicitly enumerating all the embeddings of all the graphlets, which is a challenging task. Fortunately, the problems of above characteristics have been efficiently managed by Monte Carlo Markov Chain (MCMC) algorithms for years. MCMC algorithms perform a random walk on the sample space with a locally computable transition probability matrix in such a manner that the stationary distribution of the random walk aligns with the desired probability distribution. Once the random walk mixes, any object that the walk visits in the sample space is considered to be a sample taken using the desired probability distribution. For our task, the sample space is the set $\mathcal{S}$, and the desired probability distribution is the iid distribution.

Before we discuss the details of the MCMC method for iid sampling of a graphlet embedding, we discuss, given a uniform sampler how GUISE constructs the graphlet frequency distribution (GFD) effectively. The process is quite simple, for this GUISE keeps one counter for each of the graphlets, a total of 29 counters all initialized to 1 [4] Then GUISE calls the sampler repeatedly for a large number of iterations. For each

[4] In GFD, graphlet counts are compared in a logarithm scale; since, $\log 0$ is undefined, we initialize the graphlet count with 1.

iteration, if the sampled embeddings is an embedding of graphlet i, the algorithm increments the counter for i. GUISE constructs GFD by normalizing the values of each of the counters, and taking the logarithm of those values in a vector in the correct order. The following Lemma holds.

Lemma 5 *When the size of the sample set, $\mathcal{C}$ approaches to infinity,* GUISE *returns the correct GFD for a graph.*

PROOF*: Since, each sample returns one of the 29 graphlets using a uniform distribution, the random variable (say, X) that defines the type of graphlet returned in an iteration follows a categorical distribution, with $Pr(X = g_i) = p_i = \frac{f(i)}{\sum_{i=1}^{29} f(i)}$, where $f(i)$ is the frequency of g_i in G. Also note that the i'th entry of GFD is $\log p_i$.*

In set $\mathcal{C}$, the expected count for graphlet g_i is $|\mathcal{C}| \cdot p_i$. Now, if $c(i)$ is the actual count of g_i in $\mathcal{C}$, using strong law of large numbers $Pr\left(\lim_{|\mathcal{C}|\to\infty} c(i) = |\mathcal{C}| \cdot p_i\right) = 1$

So, As $|\mathcal{C}|$ approaches to infinity, the i'th entry of GFD using GUISE *is equal to $\log \frac{c(i)}{|\mathcal{C}|} = \log \frac{|\mathcal{C}| \cdot p_i}{|\mathcal{C}|} = \log p_i$ Therefore, for the limiting case,* GUISE *returns the correct GFD for a graph.* ■

7.3.2 MCMC Algorithm for Uniform Sampling of a Graphlet

For any MCMC algorithm, we need to define the sample space, the state transition process, the transition probability matrix, and the desired probability distribution. As mentioned earlier, the set of states are the embeddings of any of the 3-, 4-, or 5-*Graphlet*s, on which GUISE performs the random walk. Let's call this $\mathcal{S}$. At any time of the random walk, the GUISE visits a specific object in $\mathcal{S}$. It then walks to one of the neighboring states with the probability that is defined by an appropriate state transition probability matrix, T.

Neighboring graphlets

For a *k-Graphlet*, all the $(k-1)$-*graphlets*, *k-graphlets*, and $(k+1)$-*graphlets* that have $k-1$, $k-1$ and k nodes in common respectively, are its neighboring graphlets. In our case, $k+1$ cannot be higher than 5 and $k-1$ cannot be lower than $max(3, k-1)$, which means a 3-*Graphlet* can have a 3-*Graphlet* or a 4-*Graphlet* as one of its neighbors; a 4-*Graphlet* can have a 3-*Graphlet* or a 4-*Graphlet* or a 5-*Graphlet* as one of its neighbors, and a 5-*Graphlet* can have a 4-*Graphlet* or a 5-*Graphlet* as one of its neighbors. To obtain a same-size neighboring graphlet of a graphlet embedding, e, Guise simply replaces one of the existing vertex of e with another vertex which is not part of e, after ensuring the connectedness of the new embedding. For obtaining a *k-Graphlet* from a $(k-1)$-*Graphlet* Guise adds one embedding vertex, and for the reverse it deletes one embedding vertex, again ensuring the connectivity of the embeddings for both the actions. Note that, during the random walk process, Guise populates the neighborhood of currently visiting embedding on-line by using the adjacency list information of the constituting vertices.

Example: Suppose Guise is performing an MCMC walk on the graph shown in Figure 7.1(a). Let $\langle 1,2,3,4 \rangle$ is the currently visiting graphlet (which is a g_5 graphlet) of size 4. One of it's neighboring graphlet is $\langle 1,2,3,8 \rangle$, which is a graphlet g_3 that can be obtained by replacing the vertex 4 by the vertex 8 (see Figure 7.2(a)). Another neighbor can be $\langle 1,2,3,4,8 \rangle$ (g_{21}, a size-5 graphlet), which can be obtained by simply adding the vertex 8. In Figure 7.1(b) we show the information of all the neighbors of graphlet $\langle 1,2,3,4 \rangle$. The box labeled by 0 contains the vertices that can be deleted to get all valid 3-*Graphlet* neighbors. Box labeled by 1 contains all the vertices that can be used as a replacement of vertex 1 to get valid neighboring 4-*Graphlet*s. Same is true for the box labeled by 2,3 and 4. The box labeled by 5 contains all the vertices that can be added with the current graphlet to get all valid neighboring 5-*Graphlet*s. Note that, by adding the number of elements in each box labeled from

0 to 5 we can obtain the neighbor-count of currently visiting graphlet; for instance, the neighbor-count of $\langle 1, 2, 3, 4\rangle$ is 31. ■

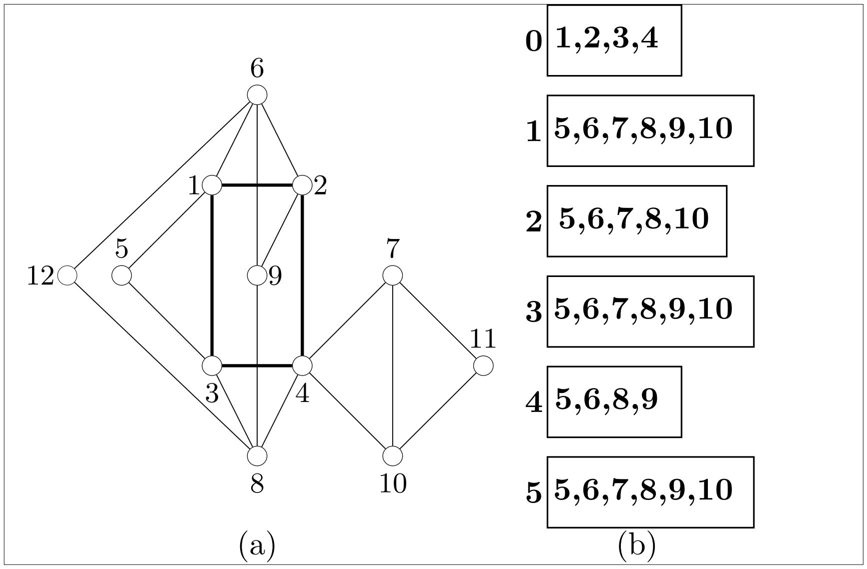


Figure 7.1.: (a) A toy graph (b) neighborhood population of currently visiting graphlet (1,2,3,4)

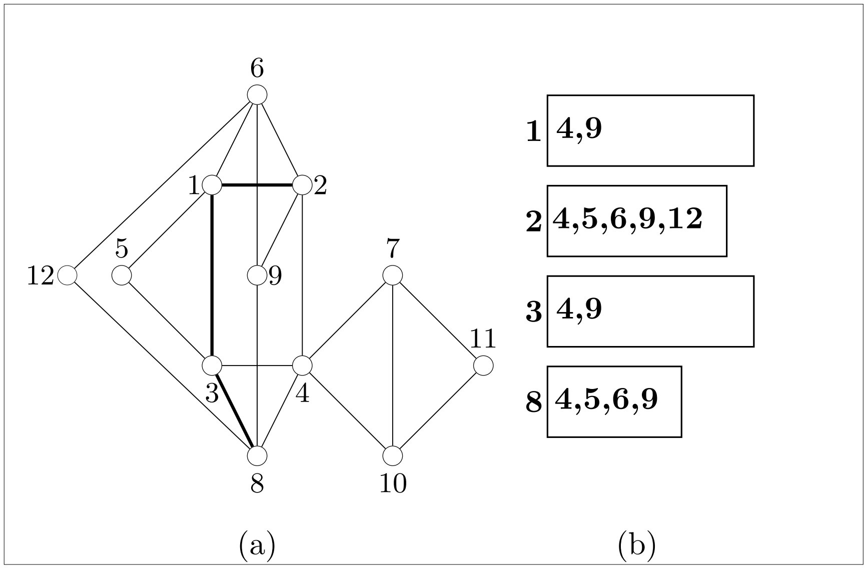


Figure 7.2.: (a) Toy graph (b) neighborhood population of graphlet (1,2,3,8)

Transition Probability Matrix

The transition probability matrix, T defines the state transition probability between a pair of neighboring graphlets p and q. The transition probability between two graphlets that are not neighbors of each other is zero. For example, in the graph in Figure 7.1, the transition probability between $\langle 1,2,3,4 \rangle$, and $\langle 1,3,5,6 \rangle$ is zero, as they are not neighbors of each other according to the neighborhood definition in the previous paragraph.

Note that if the random walk achieves a stationary distribution π, then the Equation 2.5 holds. For our case, the stationary distribution is $\left\langle \frac{1}{m}, \frac{1}{m}, \cdots, \frac{1}{m} \right\rangle$, where $m = |\mathcal{S}|$, which is a uniform vector of size m. One way to ensure the uniformity in π is to design a symmetric Markov chain, i.e. the probability of moving from the state i to the state j and the probability of moving from the state j to the state i are equal. GUISE adopts this strategy, i.e., it uses a symmetric transition probability matrix T, i.e. $T = T^{\mathrm{T}}$.

For a graphlet i, its degree is defined as $d(i)$, which is the number of its total neighbors in the random walk space. The usage of the term *degree* has an intuitive meaning. If we consider each of the graphlet embeddings as a vertex of a graph, and represent the neighbor relationship between two graphlet embeddings as an edge, then the degree of that embedding is exactly equal to its degree in the above graph. In that case, the random walk can be viewed as a walk along the edges of the graph. Now, consider two neighboring graphlets, p and q; By setting $T(p,q)$ and $T(q,p)$ equal to $min(\frac{1}{d(p)}, \frac{1}{d(q)})$ makes T a symmetric matrix (note if, if p and q are not neighbors, $T(p,q) = T(q,p) = 0$, thus maintains the symmetry). By definition, the row entries of T requires to sum to 1; the above probability setting ensures that the sum of row entries of all the rows are equal or less than 1. In case, it is less than 1, we allocate the remaining probability as a self-loop to the resident state. This symmetric transition probability matrix is also called doubly stochastic, as the sum of both the rows and the columns of such matrix equal to 1. Now the following Lemma (Exercise

6.9 of [92]) is sufficient to prove that the above Markov chain achieves a uniform stationary distribution.

Lemma 6 *An ergodic random walk achieves a uniform stationary distribution if and only if its transition probability matrix is doubly stochastic.*

PROOF:*According to the Equation 2.5 we have*

$$\pi \;=\; \pi T \tag{7.1}$$

Here, π a row vector of size m defines the uniform probability distribution of a random walk with m states. T is the transition probability matrix. The above equation can be written as:

$$(\pi_1, \ldots \pi_m) = (\pi_1, \ldots \pi_m) \begin{pmatrix} T(1,1) & \cdots & T(1,m) \\ T(2,1) & \cdots & T(2,m) \\ \vdots & \ddots & \vdots \\ T(m,1) & \cdots & T(m,m) \end{pmatrix} \tag{7.2}$$

For any state i,

$$\pi_i = \pi_i * \sum_{j=1}^{m} T(j,i)$$

Since each state has equal probability in the stationary distribution, $\pi_i = \frac{1}{m}$. so,

$$\frac{1}{m} = \frac{1}{m} * \sum_{j=1}^{m} T(j,i) \Rightarrow \sum_{j=1}^{m} T(j,i) = 1$$

Which means the sum of the column vectors of T is equal to 1 for each column of the matrix, i.e. T is column stochastic. Moreover, T is a transition probability matrix and is row stochastic, thus, T is doubly stochastic. Since an ergodic random walk has an exclusive stationary distribution, both the necessary and sufficient condition for the proof is met. ■

Nevertheless, we still need to prove the following:

Lemma 7 *The random walk that* GUISE *uses is ergodic.*

PROOF: *A Markov Chain is ergodic if it converges to a stationary distribution. To obtain a stationary distribution the random walk needs to be finite, irreducible and aperiodic [92]. The state space $\mathcal{S}$ is finite with size m, because the number of graphlets is finite. We also assume that the input graph G is connected, so in this random walk any state y can be reachable from any state x with a positive probability and vice versa, so the random walk is irreducible. Finally the walk can be made aperiodic by allocating a self-loop probability at every node [5]. Thus the lemma is proved.* ■

Example: Continuing from the example in Figure 7.1(a), where $\langle 1, 2, 3, 4 \rangle$ is the current visiting graphlet of the MCMC walk. To determine the next jump site (graphlet) in the space, GUISE remove node 4 from the current graphlet and randomly select a node from a set build by merging nodes from box – labeled 1, 2 and 3 (figure 7.1(b)). Suppose GUISE pick node 8 and form a size 4 graphlet $\langle 1, 2, 3, 8 \rangle$. Figure 7.2(a) and (b) illustrate the newly formed graphlet and its neighboring graphlets information. MCMC walk will make this transition with probability $min(\frac{|d_{\langle 1,2,3,4 \rangle}|}{|d_{\langle 1,2,3,8 \rangle}|}, 1)$, which is $min(\frac{31}{20}, 1) = 1$. Here 31 and 20 are the neighbor-count (degree) of the node representing graphlet $\langle 1, 2, 3, 4 \rangle$ and $\langle 1, 2, 3, 8 \rangle$ respectively in the graphlet space. ■

Mixing Rate of Random Walk

One important aspect of any MCMC algorithm (including MH, which is essentially a special kind of MCMC algorithm) is the rate at which the initial distribution converges to the desired distribution. The mixing rate of a random walk has been studied extensively in spectral graph theory [103], since it plays an important role in obtaining efficient MCMC algorithms. A Markov chain is called *rapidly mixing* if it is close to stationary after only a polynomial number of simulation steps, i. e., after

[5] This is required only from a theoretical standpoint; in our experiment we do not allocate any self-loop probability, unless needed.

$poly(\lg m)$, where m is the number of states in the Markov chain. Note that, m can be exponentially large with respect to the input size of the algorithm. An algorithm that is *rapidly mixing* is considered efficient.

A method to measure the mixing rate is to find the *spectral gap* of the transition probability matrix T. T has m real eigenvalues $1 = \lambda_0 > \lambda_1 \geq \lambda_2 \geq ... \geq \lambda_{m-1} \geq -1$. Then, the *spectral gap* is defined as $\lambda = 1 - max\{\lambda_1, |\lambda_{m-1}|\}$. Since the absolute values of all the eigenvalues are less than one with the largest eigenvalue λ_0 be exactly one, the spectral gap is always between 0 and 1. The higher the spectral gap, the faster the convergence [104]. For this task, the entire T is not available to us, so it is generally difficult to measure the spectral gap. However in experiment section, we will show results involving spectral gap of smaller datasets for which computing T is feasible.

Mixing time is also studied by analyzing the convergence of *total variation distance.* Let x be the state at time $t = 0$ and denoted by $P^t(x,.)$ the distribution of the state at time t. The *total variation distance* at time t with initial state x can be denoted as,

$$\Delta_x(t) = \frac{1}{2}\sum_{y\in\Omega}|P^t(x,y) - \pi(y)| \tag{7.3}$$

Here $\pi(.)$ is the target distribution. Using this, if the initial state of a Markov chain is x, and $p_x^t = Pr[X_t = y|X_0 = x]$, then the mixing time of a Markov chain is defined as below:

$$\tau_{mix} = \min_t\{\|p_x^t - \pi\|_{tv} \leq \epsilon, \forall x\} \tag{7.4}$$

In other words, it is the minimum time needed for the chain to come to within ϵ of π in total variation distance, considering over all possible initial states. Typically, $\frac{1}{2e}$ or $\frac{1}{4}$ is used for ϵ values to claim that a Markov chain mixes *rapidly.* For GUISE, Ω is the set of all 3-, 4- and 5-*Graphlets*, and π is equal to $\frac{1}{|\Omega|}\mathbf{1}$. Finding an analytical bound on total variation distance is not possible, unless we consider very specialized graph. However, we will show in the experiment section that the mixing time of GUISE is less than 100 walks for some of the graphs that we experiment with. We believe that the result should be similar for other graphs as well, because the quality of sampling on larger graphs are also very close to the results for the ideal cases (see Table 7.2). In real-life applications, we run the sampler for millions of iterations; we can simply ignore a few hundreds initial steps in our computation to ensure that the random walk mixes before the sampling starts.

Algorithm 13 Uniform graphlet sampling, GUISE

```
procedure GUISE($G, SCount, STrial$)
    $g_x$ = get_a_initial_graphlet($G$)
    $d_{g_x}$ = populate_neighborhood($g_x$)
    $sampled = 0$
    while $true$ do
        choose a neighbor $g_y$, uniformly from, all possible neighbors
        $d_{g_y}$ = populate_neighborhood($g_y$)
        $acceptance\_probability$ = min $(\frac{|d_{g_x}|}{|d_{g_y}|}, 1)$
        if $uniform(0,1) \leq acceptance\_probability$ then
            $g_x = g_y$
            $d_{g_x} = d_{g_y}$
        end if
        if $sampled < STrial$ then
            $sampled+ = 1$
        else
            $sampled+ = 1$
            get_graphlet_type($g_x$)+=1
        end if
        if $sampled > SCount + STrial$ then
            return
        end if
    end while
end procedure
procedure POPULATE_NEIGHBORHOOD($g_x$)
    $neighbor\_list$ = generate all potential neighboring graphlets
    return $neighbor\_list$
end procedure
```

7.3.3 Pseudo-Code

The pseudo-code of GUISE is given in Algorithm 13. It takes three parameters, an input graph $\mathcal{G}$, the total number of samples ($SCount$) and mixing period ($STrial$).

GUISE starts by picking (Line 2) any initial graphlet (g_x). In Line 3 it populates the neighborhood of g_x according to the technique discussed in subsection 7.4.1 and save the set of neighbors in a graphlet data structure (d_{g_x}). Then in an iterative way, it chooses a graphlet g_y from g_x's neighbors with uniform distribution. To compute whether the move to the neighbor g_y is accepted or not, it also computes the neighbor-count of g_y (Line 7). After computing the acceptance probability on Line 8, if the move is accepted, GUISE replaces the current graphlet g_x by g_y (line 10 and 11), otherwise g_x is kept unchanged. It also increments the number of embedding sampled (*sampled*) and visit count of the current graphlet type by one (line 16 and 17) after the end of mixing period of the walk; in this way the sampling statistics of the mixing period is ignored. GUISE terminates when the *sampled* count exceeds $SCount + STrial$.

In the above pseudo-code, Line 6 chooses a neighbor g_y of g_x with probability $1/|d_{g_x}|$ and Line 9 accepts that choice with a probability $min(|d_{g_x}|/|d_{g_y}|, 1)$. So, the overall transition probability is $T(g_x, g_y) = min(1/|d_{g_x}|, 1/|d_{g_y}|)$, which is desired.

7.4 Implementation Details

GUISE accepts a connected graph G for which it computes the GFD. It starts from a random graphlet embedding (say g_t) —it can simply be an embedding of a g_1 graphlet which is easy to get. Then it computes the transitional probability matrix T locally, which requires the knowledge of degree of g_t; it is also important to know the graphlet-type of g_t, so that the correct counter can be incremented.

7.4.1 Populating The Neighborhood of a Graphlet

Populating the neighborhood of a graphlet is the most time-consuming task. In the following we will explain how GUISE populates the neighborhood of a 4-*Graphlet* g_t.

To obtain a 3-*Graphlet*, GUISE first deletes one of the vertices from g_t and checks whether the remaining 3 vertices are still connected in the input graph. If yes, a 3-*Graphlet* neighbor of g_t is obtained. Note that, g_t can have (at most) 4 such neighboring graphlets. To obtain all neighboring 4-*Graphlet* of $g_t\langle x, y, zw\rangle$, GUISE first removes one of the vertices from g_t (say x) and checks whether the remaining 3 vertices (y, z, and w) are still connected. If the check succeeds, it finds the union of the adjacent vertices of these 3 vertices. If they are not connected it considers the subset of the union-set of adjacent vertices

Each vertex of the resultant set of vertices along with the 3 undeleted vertices (of g_t) represents a neighbor of g_t. The process of removing and combining is repeated for all the vertices of g_t. Finally, to get all neighboring 5-*Graphlet*s of g_t, GUISE takes the union of adjacency lists of all 4 of its vertices and pick a vertex from the union set and combine with g_t.

Following the above techniques, GUISE can populates neighborhood of size 3, 4 and 5 graphlets.

7.4.2 Identifying Graphlet Type

To identify the type of a graphlet g_t, GUISE first treats g_t as a graph $g(v, e)$ where cardinality of v and e is between 3 to 5 and 2 to 10, respectively. First of all, graphlets can be categorized based on the cardinality of v. To distinguish graphlets in each category we introduce a signature of each graphlet based on the degree count of each vertex in $g(v, e)$. We denote this signature as *degree-signature.* We first compute the degree of each vertices of graph $g(v, e)$ and save it in a vector of size $|v|$. Then we sort the vector and use this sorted vector as a signature of each graphlet. GUISE finds this signature and based on the signature it identifies which type of graphlet g_t is. It is possible that two distinct type of graphlets have the same *degree-signature*; in that case GUISE checks additional criteria to make them distinguishable. Please note that, above scenario occur only for two pairs of graphlets, (g_{13}, g_{16}) and (g_{20}, g_{21}).

Example: Lets take three graphlets, g_{12}, g_{22} and g_{26}. These are all size 5 graphlets. For g_{12} we can easily compute the degree of all vertices, save it in a vector and after sorting the vector we get degree-signature of g_{12} which is $(1,1,2,3,3)$. In a similar fashion we can get degree-signature of g_{22} and g_{26}, which are $(2,2,2,4,4)$ and $(2,3,3,4,4)$, respectively. As we can see, degree-signatures are unique for all three graphlets. Similar trend holds for other graphlets except two pairs of graphlets we mention above. For g_{13} and g_{16}, both of them have $(1,2,2,2,3)$ as their degree-signature. We can easily make them distinguishable if we hop in to their structural level. Note that, for g_{13} minimum degree count node has only neighboring node with degree count 2 but for g_{16} it has neighboring node with degree count 3. Using the similar trick, we can make g_{20} and g_{21} distinguishable.

7.4.3 Complexity Analysis

Most expensive part of Guise is neighborhood computation. For neighborhood computation we need to perform union operations on adjacency lists of current graphlet. Our assumption that the adjacency lists are stored in sorted order allows us to perform union operation in linear time with respect to the length of the participating sets (adjacency list). For example, the cost for performing union over two adjacency lists of size m and n is $\mathcal{O}(m+n)$.

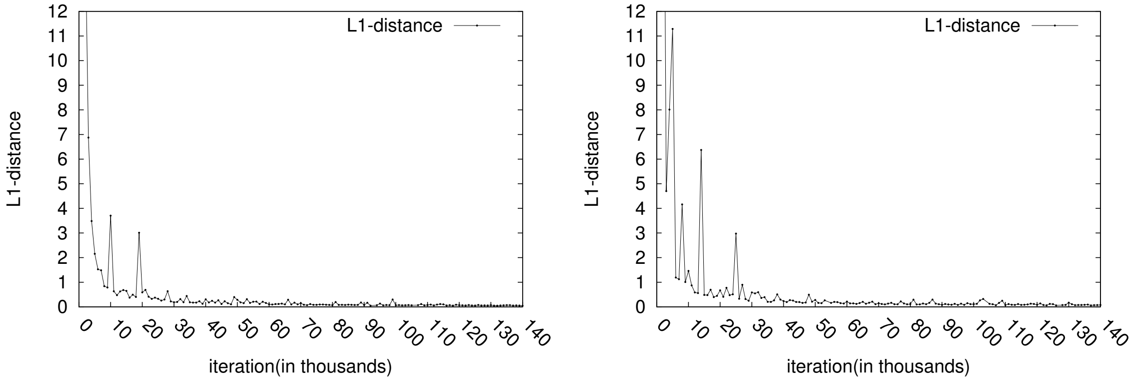
The worst case time for finding the neighbors of a 3-*Graphlet* is $\mathcal{O}(9p)$ $(3*\mathcal{O}(2p)+\mathcal{O}(3p))$; where, p is the average length of adjacency lists. Similarly, for 4-*Graphlet* and 5-*Graphlet* the time complexities are $\mathcal{O}(24p)$ $(4*\mathcal{O}(2p)+4*\mathcal{O}(3p)+2*\mathcal{O}(2p))$ and $\mathcal{O}(30p)$ $(5*\mathcal{O}(2p)+5*\mathcal{O}(4p))$ respectively. The total execution time for all iterations is $(\mathcal{O}(9p)*|3\text{-}Graphlets|+\mathcal{O}(24p)*|4\text{-}Graphlets|+\mathcal{O}(30p)*|5\text{-}Graphlets|))$. Where, $|k\text{-}Graphlets|$ represents the number of k-*Graphlet* embeddings sampled.

7.4.4 Choice of Parameters

Apart from the input graph, GUISE has two additional parameter, which are $SCount$, and $STrial$; the former defines the number of iterations that GUISE should run and the latter defines the required number of iterations for the mixing of the random walk.

Since, the frequency of various graphlets varies in several order of magnitude, the $SCount$ value should be at least as high as few millions for a moderately large graph; otherwise the frequency of the least frequent graphlets may turn out to be simply zero. Since, the iterations of GUISE are cheap, it can be run for as many iterations as is desired. From the law of large numbers in Statistics, the more sample we takes, the more GUISE converges to the true GFD. As we discussed earlier that the mixing time of the random walk of GUISE is small, so a few hundred walks should be sufficient for the $STrial$ value.

We also attempted to find methods to automatically set the value of $SCount$ by adopting some heuristics. The first heuristic is that, we choose $SCount$ in such a way that the visit count of each type of graphlets is higher than a threshold c where $c > 0$. This heuristic works fine for all real-life graphs, as in real-life graphs, all the graphlets have a non-zero count. Nevertheless, It is possible that an input graph does not have any distinct embedding for a graphlet at all. In that case, we will never find a suitable $SCount$ for which the count of all types of graphlets is higher than c. To combat such a scenario we design another heuristic. According to this, We construct approximated-GFD along with sampling process and compute the difference between current approximated-GFD with a previously computed approximated-GFD using L_1-distance measure. If the difference falls below a pre-defined threshold $diff$, and does not change much for specified number of steps then GUISE will stop sampling. Since we are computing L_1-distance within the sampling task, this heuristic results in slightly higher running time of GUISE.



(a)

(b)

Figure 7.3.: Choosing parameter *SCount* in (a)Ca-GrQc (b) Ca-Hepth graph

In Figure 7.3(a) and (b) we show that the second heuristic works fine in choosing *SCount* in two real-life graphs. In this experiment, on every 1000th iteration we measure the L_1-distance between the current approximated-GFD with the previously computed one. As we can see for both the cases, L_1-distance measure does not change much after 30K iterations. Note that, for both of these datasets, we choose *SCount* to be 40K, and the same *SCount* value is also used to generate empirical results on these datasets in Figure 7.6(a)(b) and Table 7.3.

7.5 Experiments and Results

In this section, we evaluate the run-time performance and sampling performance of GUISE. First we experimentally show how the sampling distribution of GUISE matches with the desired uniform distribution. Then we present empirical results on the convergence of the sampler. Finally we exhibit the timing performance of GUISE with the brute-force approach that computes GFD after counting all the graphlets. We also show some results on the spectral-gap, and variational distance that analyze the mixing time of GUISE's random walk. For the first two experiments, we choose input graphs for which we know the exact frequency count of all 3, 4, and 5 node Graphlets. If the total number of such graphlets is m, we take sufficiently large number of samples, say $i \times m$, so that the sampling distribution can be constructed. For comparison purpose, we get the exact frequency count using an existing graphlet counting algorithm GraphCrunch2 [27], and use them to assess the sampling quality that we obtain using GUISE.

Table 7.1.: Datasets details (we could not get exact frequency count of the graphlets for the graphs marked with ⋆)

Graph	Vertex	Edge	Total Number of 3,4,5-Node Graphlets
Football	115	613	3,74,023
Dolphin	62	159	24,879
ca-GrQc	5,241	14,484	38,778,801
ca-HepTh	9,877	25,998	91,773,466
Yeast	2,361	7,182	52,496,900
Jazz	198	2,742	55,062,209
ca-AstroPh	18,771	198,110	1.0978E11
soc-sign-Slashdot081106⋆	77,357	468,554	-
roadNet-PA⋆	1,088,092	1,541,898	-
amazon0302⋆	262,111	899,792	-
Email-enron ⋆	36,692	183,831	-
cit-Patents⋆	3774768	16,518,948	-

7.5.1 Datasets

We use graphs of different sizes from different domains. Table 7.1 lists the graphs/networks we use for our experiments, with vertex and edge counts. The graphs were collected from the following two web sites [6]. All input graphs used in our experiments are made undirected (if necessary) during the preprocessing step. In the same table, we also report the total count of all 3-,4-, and 5-*Graphlet*s in the third column as was obtained from GraphCrunch2; for the first four graphs the process finishes within a reasonable amount of time. For ca-AstroPh dataset, GraphCrunch2

[6] http://snap.stanford.edu/data/index.html and
http://www-personal.umich.edu/~mejn/netdata

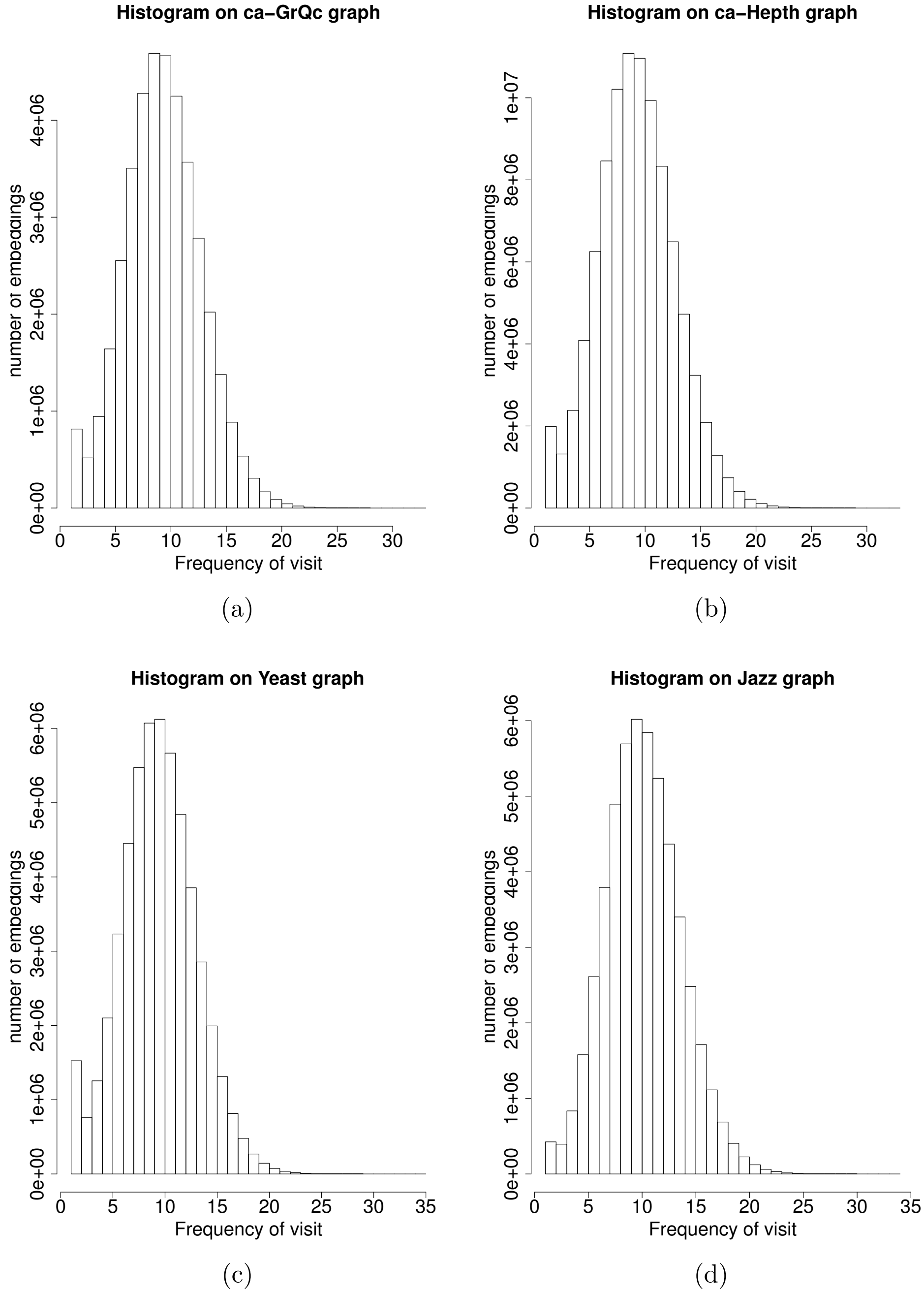
returns the exact count after 3 whole days of running. For the remaining graphs (marked with $\star$ in Table 7.1), GraphCrunch2 fails to return the graphlet frequencies even after 3 days of running.

7.5.2 Uniform Sampling of Graphlets

In this experiment we empirically show GUISE's performance on sampling graphlets uniformly. For this, given a graph with m graphlet embeddings, we perform a uniform sampling for $m \cdot i$ times (i is a positive integer). The random variable that represents the number of times a specific embedding is sampled then follows a binomial distribution $\mathcal{B}(m \cdot i, p)$, where $p = \frac{1}{m}$. For this distribution, the median visit count will be identical to the mean, which is $m.i.\frac{1}{m} = i$ and the variance is $m \cdot i \cdot p(1-p) = \frac{i(m-1)}{m}$. This binomial distribution resembles a normal distribution, because the success probability is very small.

We show results for four input graphs, ca-GrQc, ca-Hepth, Jazz and Yeast. For example, in "ca-GrQc", there are in total 38,778,801 distinct graphlets embeddings. We run GUISE for a total 387,788,010 samples ($i = 10$). For an ideal uniform case, each of the graphlet embeddings will be visited 10 times. Throughout the GUISE's running, we keep track of the visit counts of each embedding of graphlets. In Figure 7.4(a) we show the frequency histogram of visit counts, where the x-axis shows various frequency values, and the y-axis represents the number of distinct embeddings that are visited with that frequency. The shape of the histogram is very close to a normal graph, as we have expected. Normal graphs are also obtained for the other three input graphs as can be seen in Figure 7.4(b-d).

We also show the statistics of visit counts in Table 7.2 indicating maximum, minimum, mean, median and variance and compare those with the values for the ideal cases. For ca-GrQc graph, $m = 38,778,801$, by setting $i = 10$, we expect that the median of visit count will be 10 with the variance $\frac{10*(38,778,801-1)}{38,778,801} = 9.9$. For GUISE, the median of visit count is 10 with variance 11.96; the median is exactly equal, and



(a)

(b)

(c)

(d)

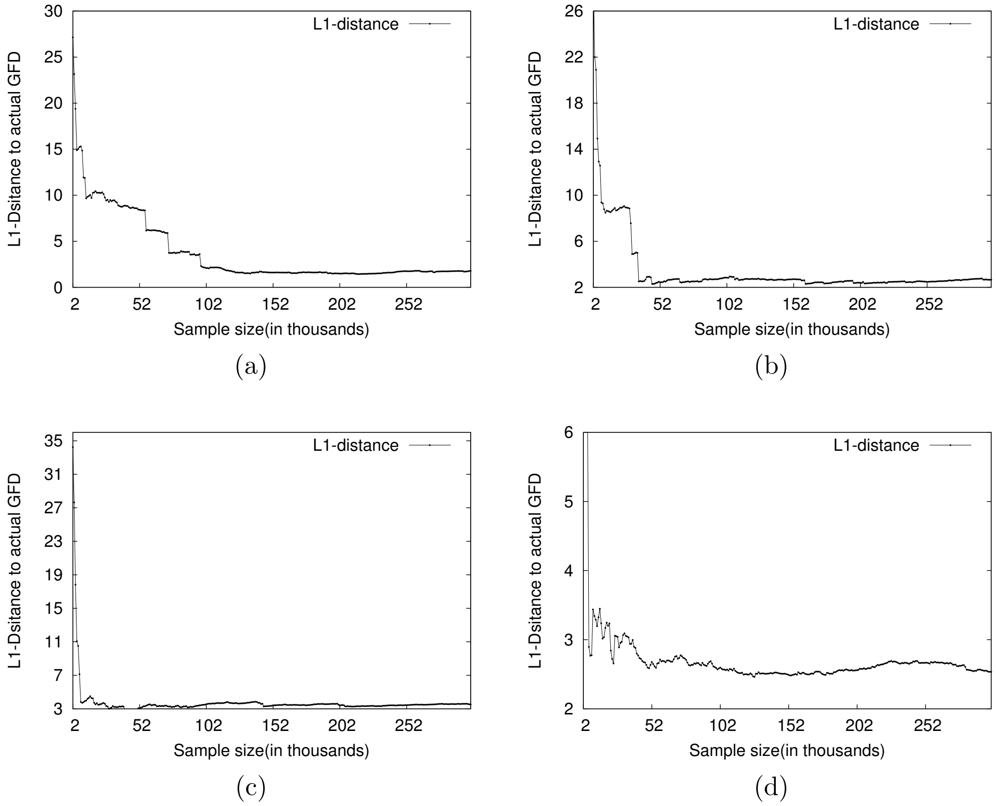
Figure 7.4.: Frequency histogram of the visit counts on (a) ca-GrQc (b) ca-Hepth (c) Yeast (d) Jazz graph

the variance is very close to its ideal value. We also obtain similar results for the other three graphs (see, Table 7.2). We do not perform this experiment for very large

Table 7.2.: Statistics of uniform sampling on ca-GrQc, ca-Hepth, Yeast and Jazz graphs and comparison with ideal case.

Graph	Uniform-Sampling				Ideal	
	Max	Min	Median	Variance	Median	Variance
ca-Grqc	33	1	10	11.96	10	9.9
ca-Hepth	33	1	10	12.13	10	9.9
Yeast	35	1	10	13.03	10	9.9
Jazz	34	1	10	12.22	10	9.9

graphs, because for this experiment we need to store the visit count of all the graphlet embeddings in the main memory, and the number of such embeddings is more than billions for those large graphs.



(a)

(b)

(c)

(d)

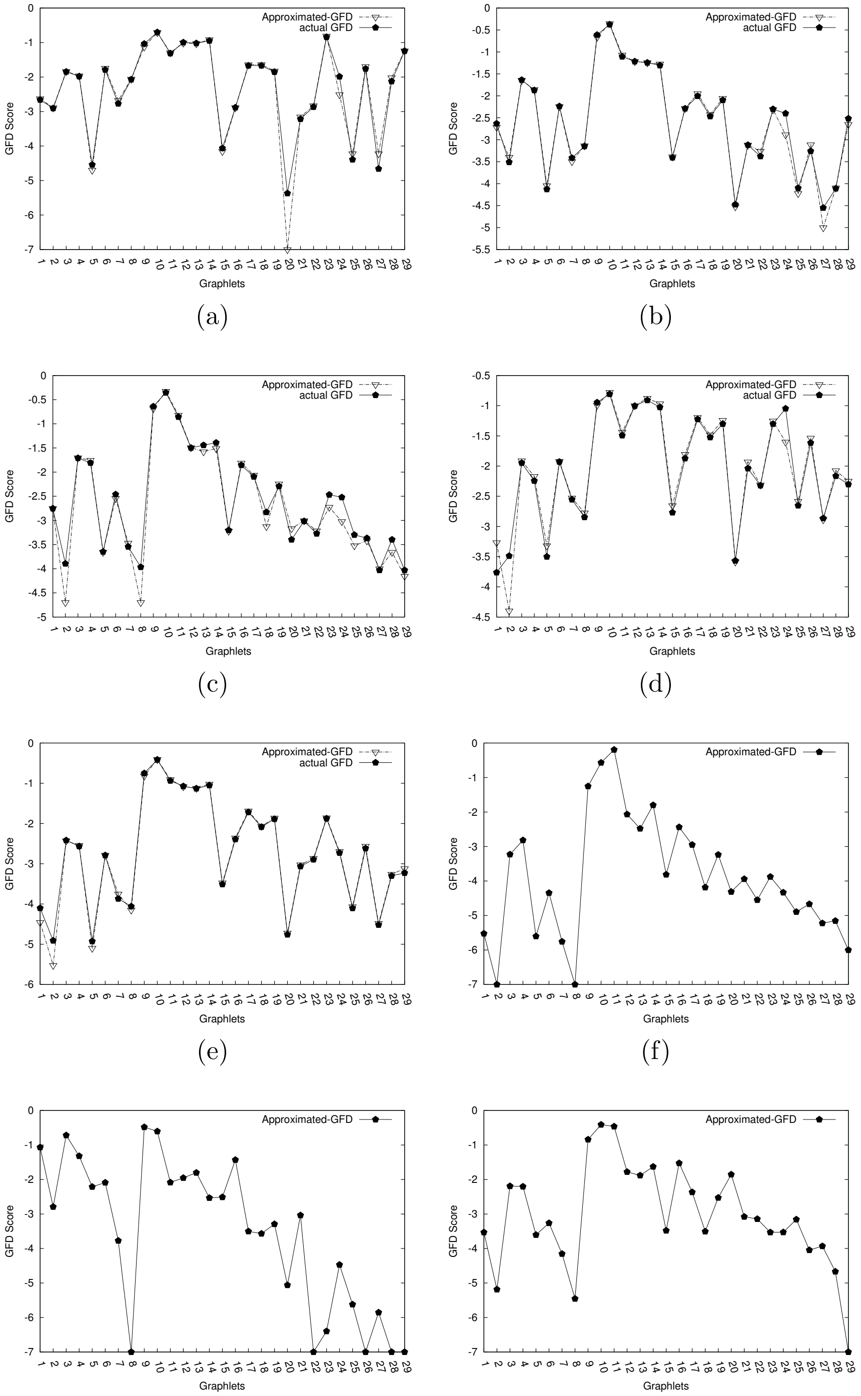
Figure 7.5.: Convergence of uniform sampler on (a) ca-GrQc (b) ca-Hepth (c) Yeast (d) Jazz graph

7.5.3 Convergence to Uniform Distribution

In this experiment, we show that through uniform sampling, we can obtain a GFD, that converges to the true GFD. To show this convergence we use L_1-norm distance measure. We use GraphCrunch2 [27] to construct the true GFD and run GUISE for 300 thousands iterations. Next we compute the L_1-Norm distance between the actual GFD vector and the GUISE GFD vector and plot the measure against the iterations count (*SCount*). Figure 7.5 exhibits the results for four of the datasets. We can see that for each of the datasets, after 200K iterations L_1-distance remains unchanged, which is the indication of the convergence of GUISE. We also exhibit the visual similarity (GFD using line plot) between the actual GFD and the approximate-GFD for all these four graphs and also for an additional graph (ca-AstroPh) in Figure 7.6(a-e). The *SCount* values that we use to get the GUISE GFDs are mentioned in the second column of Table 7.3 within parentheses. Note that, in Figure 7.6(f-h) we show the GFD that we obtain from GUISE for an additional three large graphs. For these graphs, actual GFD was not available so no comparison is shown. But, the theoretical and empirical proof of GUISE makes us confident to claim that these are the actual GFD signature of those graphs.

Table 7.3.: Timing performance of GUISE and comparison with naive algorithm

Graph	GUISE (*SCount*)	brute force
ca-GrQc	3.8 sec(100,000)	1.3 min
ca-HepTh	3.6 sec(100,000)	34 sec
Yeast	3.4 sec (100,000)	17 sec
Jazz	1.84 sec (50,000)	48 sec
ca-AstroPh	78.6 sec(1,000,000)	3 day
soc-sign-Slashdot081106	527.62 sec (4,000,000)	> 3day
roadNet-PA	313.2 sec(10,000,000)	> 3day
amazon0302	46.6 sec (1000000)	> 1day
Email-Enron	184.2 sec (2000000)	> 3day
cit-Patents	125.28 sec (2000000)	> 1day



(a) (b)

(c) (d)

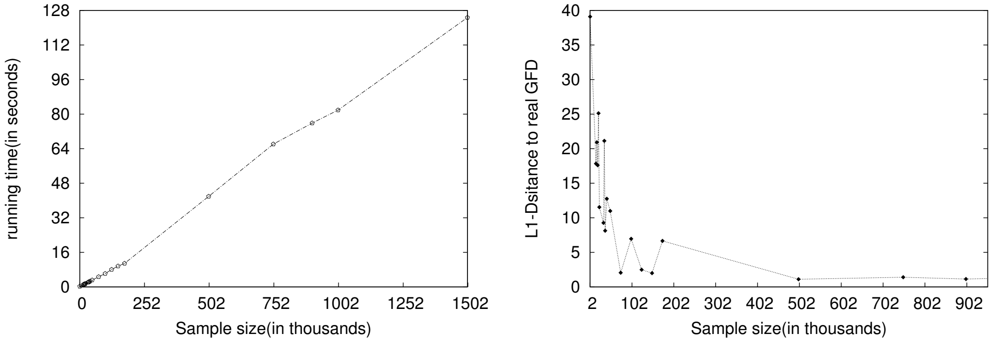
(e) (f)

(g) (h)

Figure 7.6.: Comparison with actual and approximated GFD for (a) ca-GrQc (b) ca-hepth (c) yeast (d) Jazz (e) ca-AstroPh and approximated GFD for (f) Slashdot (g)roadNet-PA (h)cit-Patents graphs

7.5.4 Timing Analysis

In Table 7.3, we show GUISE's running time for sufficiently large number of iterations. We also show the time (when available) that GraphCrunch2 takes for obtaining the frequencies of each of these graphlets. For large graph, GraphCrunch2 is clearly not usable due to its large computation time, but GUISE returns the GFD in few minutes. For one large graph (ca-AstroPh), for which GraphCrunch2 was able to finish in 3 days, GUISE returns the GFD in 1 minutes and 20 seconds. If we compare the GFD of ca-AstroPh from GUISE with that of the actual one in Figure 7.6(e), they are simply indistinguishable.



(a)

(b)

Figure 7.7.: (a) Relation between running time and $SCount$ (b) how L_1-Norm changes with $SCount$ for ca-AstroPh graph

We also perform an experiment where we show how the running time of GUISE increases with $SCount$ values. We use "ca-AstroPh" in this experiment. we also show how the L_1-norm distance changes. Running time increases linearly with $SCount$ and L_1-norm gets better with the increment of $SCount$. In Figure 7.7 we show our findings.

7.5.5 Spectral-Gap Analysis

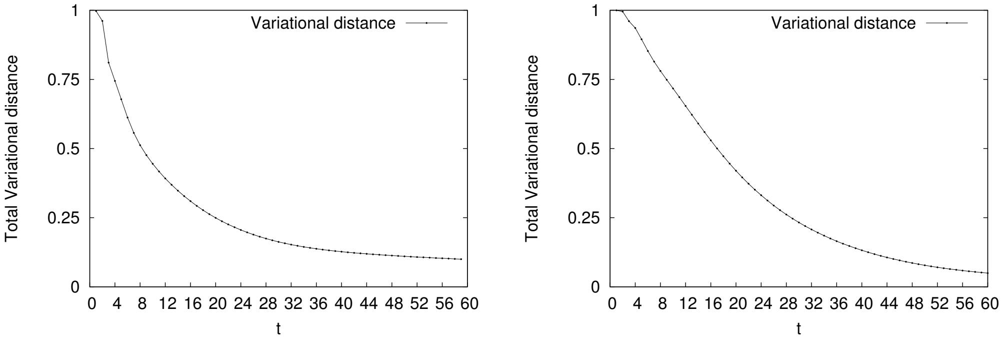
For spectral gap analysis we pick "Dolphin" dataset where total number of graphlets of size 3 to 5 reported is 24,879. In order to compute spectral gap, we generate $24,879X24,879$ symmetric transition probability matrix that contains $1,206,644$ non zero entries in the upper triangle of the matrix (excluding diagonals entry). Table 7.4 shows that the spectral gap value for this dataset is 0.0117. Note that we could not use larger dataset for this experiment, because they have too many graphlets, and finding a square matrix of that size is simply not feasible.

As we mentioned earlier that the spectral gap of a random walk can be between 0 and 1, and the larger the spectral gap, the better the mixing time of a random walk. Considering a range between 0 and 1, a spectral gap value of 0.0117 may seems to be poor. However, this value is in-fact much better compared to the spectral gap of an identical random walk on a power-law graph. To show this, we build two synthetic graphs, of which one is random graph and the other is a power-law graph. The number of nodes and edges in both of these graphs are $24,879$ (number of graphlets in dolphin dataset) and $1,206,644$ (number of transitions in dolphin dataset). We then compute the spectral gap of the two random walks on these graphs to obtain uniform sampling of vertices. The spectral gap values are 0.6414, and 0.0003 respectively. For the random graph the spectral gap is high because it is easy to perform a uniform sampling on such a graph due to the uniformity in its degree distribution. On the other hand, the spectral gap on the power law graph is very small because the degree distribution of such a graph is highly non-uniform. In a real-life network, the graphlet space is more like a power-law graph, where the neighbor-count distribution of various graphlets are highly non-uniform, so a low spectral gap, such as, 0.0117, is expected; nevertheless, it does not affect the performance at all, as the mixing time is still less than a few hundred walks. This can be shown from the fact that the inverse of spectral gap of a reversible Markov chain captures the mixing time of that walk [105], As we

Table 7.4.: Comparative Spectral-gap analysis for MCMC-walk of Guise

Graph	Spectral gap
Dolphin	0.0117
Random Graph	0.6414
Power Law Graph	0.0003

compute the spectral gap of Guise's random walk over dolphin dataset is 0.0117, thus the mixing time of this walk is 1/0.0117=85.47 unit.



(a) (b)

Figure 7.8.: Total variational distance (a) dolphin (b) football network

7.5.6 Variation Distance Analysis

Mixing time of a random walk can also be analyzed by variation distance analysis, which is our objective in this experiment. For this, we use equation 7.3. For this experiment also, we choose small datasets, such as, dolphin and football because of their reasonable size to carry out computation that includes handling very large sparse matrix (Transition probability matrix). From the total variational distance, we also compute the mixing time using equation 7.4. As we discussed in section 7.3.2, we set the value of ϵ to $\frac{1}{4}$. In Figure 7.8(a) and (b) we plot the value of total variational distance with respect to t. As we can see that for both dolphin and football dataset, random walk converges very fast, i.e. variational distance for these datasets fall bellow

ϵ after $t = 20$ and $t = 29$ steps respectively. The mixing time bound through total variation distance is even sharper than the one that we found by using spectral gap.

7.6 Conclusion

In this chapter, we present GUISE, an efficient method for approximating the graphlet frequency distribution (GFD) in a principled manner that offers significant speed up in comparison with the existing time consuming ways of brute force counting. Our experiments with many real-life networks show that GFD that GUISE obtains is all but identical to the actual GFD that one can obtain from the exhaustive counting of graphlets.

8 APPLICATIONS

In this chapter we discuss the application of graphlet analysis. Graphlets, are increasingly being used for large-scale graph analysis. For example, frequencies of various graphlets are used for classifying networks from various domains [26, 29]. They are also used for designing effective graph kernels [24]. In biological domain, graphlet frequencies are used for comparing structures of different biological networks [3]. In all these works, graphlets are used as a topological building block of a static network.

Contrary to static networks; a dynamic network's structure changes with time. In a dynamic network the vertices and/or edges are added or removed at different time stamps. In this dissertation we consider dynamic network with constant vertex-set and evolving edge-set (see Section 2.8). In this chapter, we focus on applications of graphlet based network analysis in both static (Section 8.1) and dynamic network setups (Sections 8.2- 8.4).

8.1 Clustering Static Networks Using GFD

In this section we demonstrate the usability of GFD in clustering static-networks. Networks can be generated from many sources e.g., citation networks, collaboration networks etc. Our objective is to do graph clustering on some of the graphs using GFD as a 29 dimensional metric. We use agglomerative hierarchical clustering with Euclidean distance as the distance metric.

For this experiment, we consider several sources of graphs (Table 8.1). Three graphs from road networks of California, nine graphs from Internet peer-to-peer file sharing networks, five collaboration networks and five graphs constructed from citation network for 2003 KDD cup (Same data as Figure 5.10(c)).

Table 8.1.: Graphs used for agglomerative hierarchical graph-clustering

Network	Type	# Vertex	# Edge	$R_{e/v}$	p	time (sec)
Gnutella04		10,876	39,994	3.68	1	310
Gnutella05		8,846	31,839	3.60	1	449
Gnutella06		8,717	31,525	3.62	1	373
Gnutella08		6,301	20,777	3.30	1	594
Gnutella09	p2p	8,114	26,013	3.21	1	680
Gnutella24		26,518	65,369	2.47	1	1,498
Gnutella25		22,687	54,705	2.41	1	136
Gnutella30		36,682	88,328	2.41	1	335
Gnutella31		62,586	147,892	2.36	1	615
HepTh		9,875	25,973	2.63	0.1	22.8
GrQ		5,241	14,484	2.76	0.1	47.9
CondMat	collaboration	23,133	93,439	4.04	0.1	755
AstroPh		18,771	198,050	10.55	0.01	3,000
HepPh		12,006	118,489	9.87	0.01	18,496
TX		1,379,917	1,921,660	1.39	1	54
PA	RoadNet	1,088,092	1,541,898	1.42	1	46
CA		1,965,206	2,766,607	1.41	1	82
kdd92-94		4,244	12,371	2.91	.001	0.16
kdd92-96		9,131	52,768	5.78	.001	22.8
kdd92-98	Citation	14,484	124,632	8.60	.001	452
kdd92-00		20,334	216,748	10.66	.001	1,663
kdd92-03		27,769	352,285	12.69	.001	5,442

We first calculate GFD of all the graphs using Graft. For smaller graphs we did exact graphlet counting and for larger graphs approximate graphlet counting . This can be done easily by changing the value of parameter p in Graft (See Column 6 of Table 8.1). GFD of the graphs are presented in Figure 8.1. We can see from Figure 8.1, that graphs from same source has correlated GFDs while GFDs of graphs from different sources are relatively different. Then we use the values of GFD to compute the Euclidean distance between two clusters. Initially, we assume that each graph is it's own cluster. Now, we compute the distance between all possible pairs of clusters and merge the pair of cluster with smallest distance. GFD of a cluster is the mean GFD of member graphs in the cluster. We repeat the process until the number of cluster equals to a user specified value C (assuming we know the number of clusters)

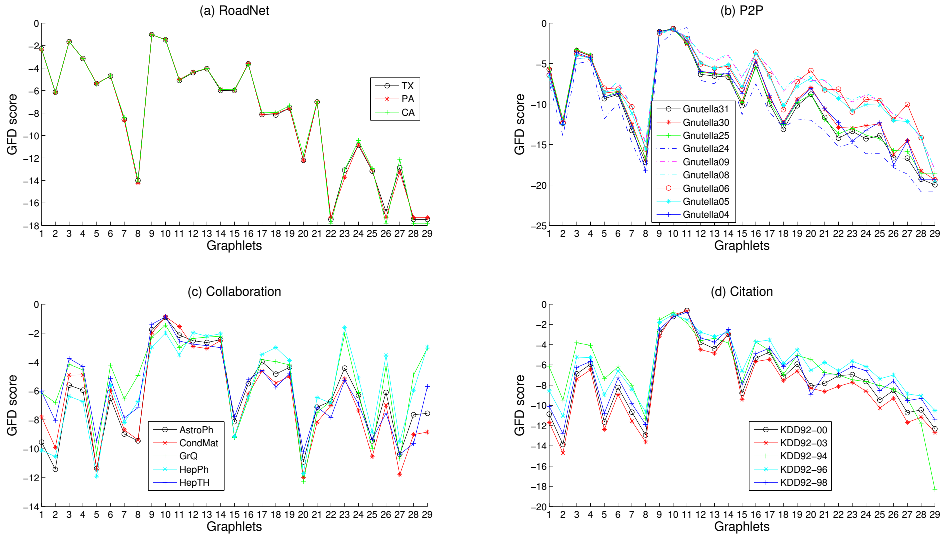


Figure 8.1.: GFD of 29 graphlets for (a) road networks and (b) P2P networks (c) collaboration networks (d) time variant citation networks

Table 8.2.: Result of agglomerative hierarchical graph-clustering (citation graphs excluded). *Purity* = $\frac{5+4+5+3}{22}$ = 0.77

		Real Graph-Groups			
		P2P	Collaboration	RoadNet	Citation
Graph cluster	P2P	5	0	0	0
	Collaboration	0	5	0	4
	RoadNet	0	0	3	0
	P2P	4	0	0	1

Table 8.3.: Result of agglomerative hierarchical graph-clustering (citation graphs excluded). *Purity* = $\frac{9+5+3}{9+5+3}$ = 1

		Real Graph-Groups		
		P2P	Collaboration	RoadNet
Graph cluster	P2P	9	0	0
	Collaboration	0	5	0
	RoadNet	0	0	3

We repeated this experiment twice; once with all the graphs in Table 8.1 and $C = 4$ and once with all graphs in Table 8.1 excluding five citation graphs and $C = 3$.

A graph cluster is assigned to the class which is most frequent in the cluster. *Purity* (accuracy) of the clustering is measured by dividing the total number of correctly assigned graphs by total number of graphs. As we can see in Table 8.2, inclusion of citation graphs results in bad clustering ($Purity = 0.77$); where as exclusion of citation graphs results in perfect ($Purity = 1$) clustering in this experiment (see Table 8.3). The reason behind this may be that, GFDs of citation graphs fails to represent a distinct cluster and moreover it shrinks cluster boundaries of other cluster who were sufficiently well separated otherwise. These results shows that, GFD is a good clustering metric.

8.2 Link Prediction in Dynamic Networks Using Graphlet Transitions

Understanding the dynamics of an evolving network is an important research problem with numerous applications in various fields, including social network analysis, information retrieval, recommendation systems, epidemiology, security, and bioinformatics. A key task towards this understanding is to predict the likelihood of a future association between a pair of nodes, given the existing state of the network. This task is commonly known as the *link prediction* problem. Since, its formal introduction to the data mining community by Liben-Nowell et al. [106] about a decade ago, this problem has been studied extensively by many researchers from a diverse set of disciplines. Comprehensive surveys [12, 72] on link prediction methods are available for interested readers.

The majority of the existing works on link prediction consider a static snapshot of the given network, which is the state of the network at a given time [62, 63, 65, 106]. Nevertheless, for many networks, additional temporal information, such as the time of link creation and deletion, is available over a time interval. For example, in an on-line social or a professional network, we may know the time when two persons have become friends; for collaboration events, such as, a group performance or a collaborative academic work, we can extract the time of the event from an event calendar. The networks built from such data can be represented by a *dynamic network*, which is a collection of temporal snapshots of the network. The link prediction task on such a network is defined as follows: *for a given pair of nodes, predict the link probability between the pair at time $t+1$ by training the model on the link information at times* $1, 2, \cdots, t$. We will refer this task as dynamic link prediction[1].

A key challenge of dynamic link prediction is finding a suitable feature representation of the node-pair instances which are used for training the prediction model. For the static setting, various topological metrics (common neighbors, Adamic-Adar, Jaccard's coefficient, etc) are used as features, but they cannot be extended easily for the dynamic setting having multiple snapshots of the network. In fact, when multiple (say t) temporal snapshots of a network are provided, each of these scalar features becomes a t-size sequence (see Figure 8.2). Flattening the sequence into a t-size vector distorts the inherent temporal order of the features. Authors of [71] overcome this issue by modeling a collection of time series, each for one of the topological features; but such a model fails to capture signals from the neighborhood topology of the edges. There exist a few other works on dynamic link prediction, which use probabilistic (nonparametric) and matrix factorization based models. These works consider a feature representation of the nodes and assume that having a link from one node to another is determined by the combined effect of all pairwise node feature interactions [66, 68, 69]. While this is a reasonable assumption, the accuracy of such models are highly dependent on the quality of the node features, as well as the validity of the above assumption.

Graphlets, which are collection of small induced subgraphs, are increasingly being used for large-scale graph analysis in static network setup. In all these works, graphlets are used as a topological building block of a static network. Nevertheless, as new edges are added or existing edges are removed from the given dynamic network, the graphlets which are aligned with the affected edge transition to different graphlets.

[1] Strictly speaking, this task should be called link forecasting as the learning model is not trained on partial observation of link instances at time $t+1$; however, we refer it as link prediction due to the popular usages of this term in the data mining literature.

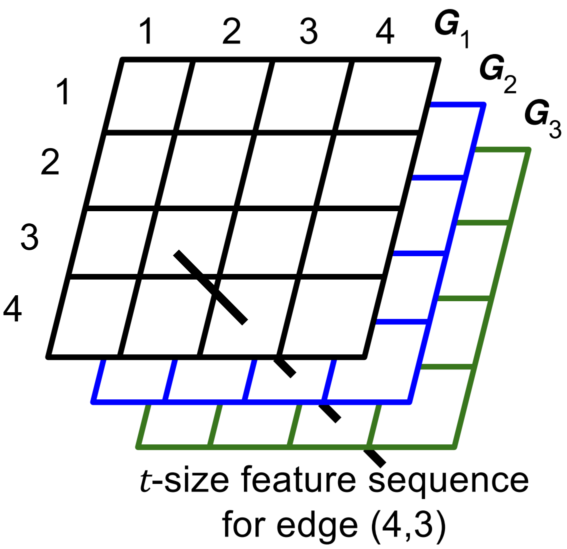


Figure 8.2.: Example of t-size feature sequence for a dynamic network with 3 time stamps.

For illustration, let us consider a dynamic network with two temporal snapshots G_1 and G_2 (Figure 8.3). In this example, G_2 has one more edge $(2, 3)$ than G_1. We observe three different types of transition events, where a type of graphlet is changed into another type in the subsequent snapshot (see the table in Figure 8.3). Here, all the events are triggered by the edge $(2, 3)$. In this work, we use the frequency of graphlet transition events associated with a node-pair for predicting link between the node-pairs in a future snapshot of the dynamic network.

A key challenge of using graphlet transition event for dynamic link prediction is to obtain a good feature representation for this task. This is necessary because graphlet transition event matrix is sparse, and on such dataset, low dimensional feature representation effectively captures the latent dependency among different dimensions of the data. There exist a growing list of recent works which use unsupervised methodologies for finding features from raw data representations of various complex objects, including images [107] and audio [108]. For graph data, we are aware of only one such work, namely DeepWalk [109], which obtains the feature representation of nodes for

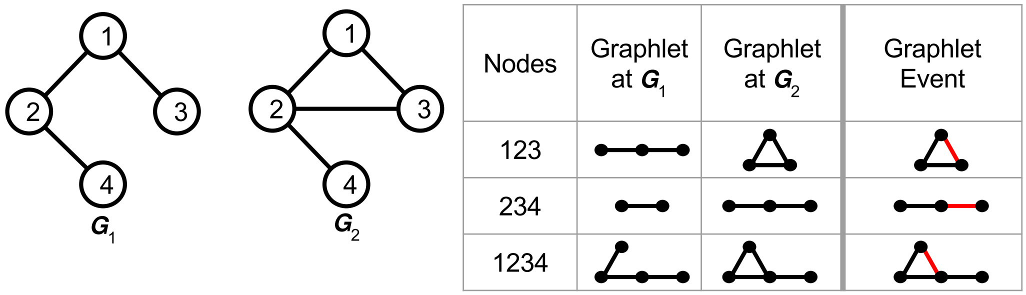


Figure 8.3.: A toy dynamic network. G_1 and G_2 are two snapshots of the network. Three different types of graphlet events are observed.

solving a node classification task. However, no such work exists for finding feature representation of node-pair instances for the purpose of link prediction in a dynamic network.

In this work, we propose a novel learning method GraTFEL (**Gra**phlet **T**ransition and **F**eature **E**xtraction for **L**ink Prediction) for obtaining feature representation of node-pair instances from graphlet transition events in the observed snapshots of the given network. GraTFEL considers the feature learning task as an optimal coding problem such that the optimal code of a node-pair is the desired feature representation. The learning can be considered as a two-step process (compression and reconstruction), where the first step compresses the input representation of a node-pair into a code by a non-linear transformation, and the second step reconstructs the input representation from the code by a reverse process and the optimal code is the one which yields the least amount of reconstruction error. The input representation of a node-pair is given as a vector of graphlet transition events (GTEs) associated with the corresponding node-pair. After obtaining an appropriate feature representation of the node-pairs, a traditional supervised learning technique is used (we use SVM and AdaBoost) for predicting link states at future times in the given dynamic network. Below we summarize our contributions in this work:

- We use graphlet transition events (GTEs) for preforming link prediction in a dynamic network. To the best of our knowledge we are the first to use GTEs for solving a prediction task over a dynamic network.

- We propose a learning model (GraTFEL) for unsupervised feature extraction of node-pairs for the purpose of link prediction over a dynamic network.

- We compare the performance of GraTFEL with multiple state-of-the-art methods on three real-life dynamic networks. This comparison results show that our method is significantly superior than all the competing methods.

8.3 Problem Definition and Methods

To recap, link prediction task on a dynamic network predicts whether two vertices u and v will have a link in snapshot G_{t+1}, given a sequence of snapshots $\mathbb{G} = \{G_1, G_2, \ldots, G_t\}$ of a network. Note that, we assume no link information regarding the snapshot G_{t+1} is available, except the fact that G_{t+1} contains the identical set of vertices. We use graphlet transition event as features for the link prediction task on a dynamic network.

A key challenge for dynamic link prediction is choosing an effective feature set for this task. Earlier works choose features by adapting topological features for static link prediction or by considering the feature values of different snapshots as a time series. GraTFEL uses graphlet transition events (GTEs) as features for link prediction. For a given node-pair, the value of a specific GTE feature is a normalized count of the observed GTE involving those node-pairs over the training data. The strength of GTEs as feature for dynamic link prediction comes from the fact that for a given node-pair, GTEs involving those nodes capture both the local topology and their transition over the temporal snapshots. We consider graphlets up to five vertices, so GTEs also involve only those graphlets. This restriction is inspired by the fact that more than 95% new links in a dynamic network happen between vertices that are at most 3 distances apart in all three real-life dynamic networks that we use in this work. So, for a given node, GTE of a five vertex graphlet in the neighborhood of that node covers a prospective link formation event as a graphlet transition event. Another reason for limiting the graphlet size is the consideration of computation burden, which increases exponentially with the size of graphlets. There are 30 different graphlets of size up to 5 and the number of possible transition event (GTE) is $\mathcal{O}(30^2)$. Increasing the size of graphlets to 6 increases the number of GTE to $\mathcal{O}(142^2)$! Finally, all the existing works on graphlets also limit their analysis for graphlets up to size five [21,29].

Feature representation for a node-pair in a dynamic network is constructed by concatenating GTE features from a continuous set of graph snapshots. Concatenation, being the simplest form of feature aggregation across a set of graph snapshots is not essentially the best feature representation to capture temporal characteristics of a node-pair. So, GraTFEL uses Unsupervised Feature Learning (UFL) to get optimal feature representation from GTE features. The motivation for using Unsupervised Feature Learning (UFL) comes from the benefit of Representation Learning (RL), as the proposed UFL is an example of RL for dynamic link prediction. The motivation of RL is discussed in a review article by Bengio et al. [110]. UFL provides better feature representation by discovering dependency among different data dimensions, which cannot be achieved by simple aggregation. It also reduces the data dimension and overcomes the sparsity issue in GTE features. Finally, GraTFEL uses the learned optimal features to solve the link prediction task using a supervised classification model.

The discussion of the proposed method GraTFEL can be divided into three steps: (1) graphlet transition event based feature extraction (Section 8.3.1), (2) unsupervised (optimal) feature learning (Section 8.3.2), and (3) supervised learning for obtaining the link prediction model (Section 8.3.3).

8.3.1 Graphlet Transition Based Feature Extraction

Say, we are given a dynamic network $\mathbb{G} = \{G_1, G_2, \dots, G_t\}$, and we are computing the feature vector for a node-pair (u, v), which constitute a row in our training data matrix. We use each of $G_i : 1 \le i \le t-1$ (time window $[1, t-1]$) for computing the feature vector and G_t for computing the target value (1 if edge exist between u and v, 0 otherwise). We use $\boldsymbol{e}^{uv}_{[1,t-1]}$ to represent this vector. It has two components: graphlet transition event (GTE) and link history (LH).

The first component, Graphlet Transition Event (GTE), $\boldsymbol{g}^{uv}_{[1,t-1]}$ is constructed by concatenating GTE feature-set of (u, v) for each time stamp. i.e., $\boldsymbol{g}^{uv}_{[1,t-1]} = \boldsymbol{g}^{uv}_1 \parallel \boldsymbol{g}^{uv}_2 \parallel \dots \parallel \boldsymbol{g}^{uv}_{t-1}$. Here, the symbol $\parallel$ represents concatenation of two horizontal vectors (e.g., 0 1 0 $\parallel$ 0.5 0 1 = 0 1 0 0.5 0 1) and $\boldsymbol{g}^{uv}_i$ represents (u, v)'s GTE feature-set for time stamp i, and it captures the impact of edge (u, v) at its neighborhood structure at time stamp i. We construct $\boldsymbol{g}^{uv}_i$ by enumerating all graphlet based dynamic events, that are triggered when edge (u, v) is added with G_i.

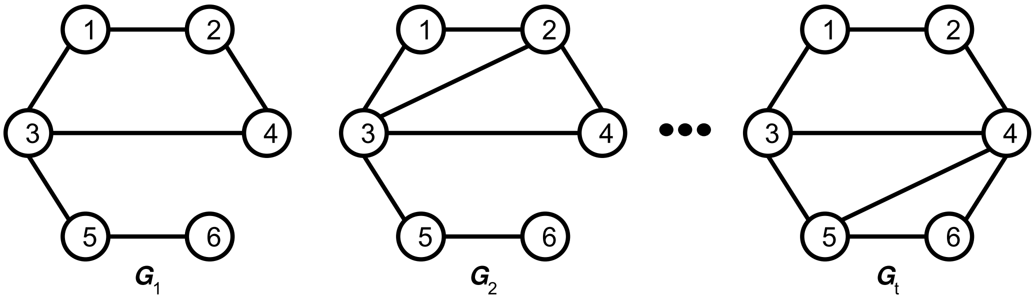


Figure 8.4.: A toy dynamic network with t snapshots. First two and last snapshots are given in this figure.

For example, consider the toy dynamic network in Figure 8.4. We want to construct the GTE feature vector $\boldsymbol{g}^{36}_1$, which is the GTE feature representation of node-pair $(3, 6)$ at G_1. We illustrate the construction process in Figure 8.5. In this figure, we show all the graphlet transitions triggered by edge $(3, 6)$ when it is added in G_1. These transition events are listed in center table of Figure 8.5. Column titled *Nodes*

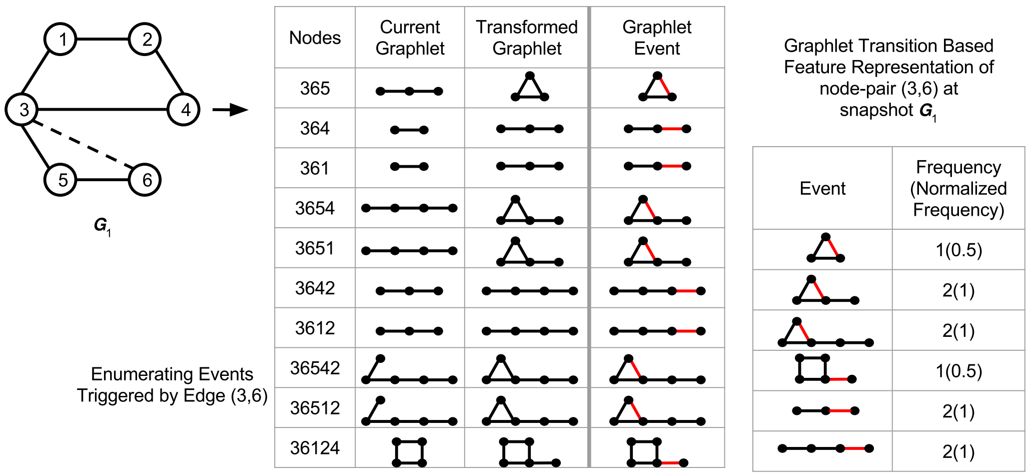

1
2
3
4
5
6
G_1
Enumerating Events Triggered by Edge (3,6)
Nodes
Current Graphlet
Transformed Graphlet
Graphlet Event
365
364
361
3654
3651
3642
3612
36542
36512
36124
Graphlet Transition Based Feature Representation of node-pair (3,6) at snapshot G_1
Event
Frequency (Normalized Frequency)
1(0.5)
2(1)
2(1)
1(0.5)
2(1)
2(1)

Figure 8.5.: Construction of graphlet transition based feature representation $\boldsymbol{g}_1^{36}$ of node-pair $(3,6)$ at 1^{st} snapshot of the toy network.

lists the sets of nodes where the graphlet transitions are observed and Column *Current Graphlet* shows the current graphlet structure induced by these nodes. Column *Transformed Graphlet* shows the graphlet structure after $(3,6)$ is added. The last column *Graphlet Event* is a visual representation of the transition events, where the transition is reflected by the red edges. Once all the transition events are enumerated, we count the frequencies of these events (Table on the right side of Figure 8.5). Graphlet transition frequencies can be substantially different for different edges, so the GTE vector is normalized by the largest value of graphlet transition frequencies associated with this edge. Also note that, all possible graphlet transition events are not observed for a given edge. So, among all the possible types of GTE, those that are observed in at least one node-pair in the training dataset are considered in GTE feature-set.

The second component of node-pair feature vector is Link History (LH) of node-pair, which is not captured by GTE feature-set, $\boldsymbol{g}_{[1,t-1]}^{uv}$. Link History, $\mathbf{lh}_{[1,t-1]}^{uv}$ of a node-pair (u,v) is a vector of size $t-1$, denoting the edge occurrences between the participating nodes over the time window $[1,t-1]$. It is defined as, $\mathbf{lh}_{[1,t-1]}^{uv} = \mathbf{G}_1(u,v) \parallel \mathbf{G}_2(u,v) \parallel \ldots \parallel \mathbf{G}_{t-1}(u,v)$. Here, $\mathbf{G}_i(u,v)$ is 1 if edge (u,v) is present in snapshot G_i and 0 otherwise. An appearance of an edge in recent time indicates a higher chance of the edge to reappear in near future. So, we Consider weighted link history, $\mathbf{wlh}^{uv}_{[1,t-1]} = w_1 \cdot \mathbf{G}_1(u,v) \parallel w_2 \cdot \mathbf{G}_2(u,v) \parallel \ldots \parallel w_{t-1} \cdot \mathbf{G}_{t-1}(u,v)$. here, $w_i = i/(t-1)$ (a time decay function) represents the weight of link history for time stamp i. Finally, a frequent appearance of an edge over time indicates a strong tendency of the edge to reincarnate in the future. This motivates us to reward such events by considering cumulative sum. We define Weighted Cumulative Link History, $\mathbf{wclh}^{uv}_{[1,t-1]} = CumSum(\mathbf{wlh}^{uv}_{[1,t-1]})$

Finally, the feature vector of a node-pair (u,v), $\mathbf{e}^{uv}_{[1,t-1]}$, is the concatenation of GTE feature-set ($\boldsymbol{g}^{uv}_{[1,t-1]}$) and LH feature-set ($\mathbf{wclh}^{uv}_{[1,t-1]}$); i.e., $\mathbf{e}^{uv}_{[1,t-1]} = \boldsymbol{g}^{uv}_{[1,t-1]} \parallel \mathbf{wclh}^{uv}_{[1,t-1]}$. For predicting dynamic links in time stamp $t+1$, we right-shift the time window by one. In other words, we construct graphlet feature representation $\mathbf{e}^{uv}_{[2,t]}$ by using snapshots from time window $[2,t]$. Final feature representation for all node-pairs,

$$\begin{aligned} \hat{\mathbf{E}} &= \{\mathbf{e}^{uv}_{[1,t-1]}\}_{u,v \in V} \\ \bar{\mathbf{E}} &= \{\mathbf{e}^{uv}_{[2,t]}\}_{u,v \in V} \end{aligned} \tag{8.1}$$

Here, $\hat{\mathbf{E}}$ is the training dataset and $\bar{\mathbf{E}}$ is the prediction dataset. Both $\hat{\mathbf{E}}$ and $\bar{\mathbf{E}}$ can be represented as matrices of dimensions $(m \times k)$. $k = |\mathbf{e}^{uv}_{[1,t-1]}| = c * (t-1) + t - 1$ is the feature size, where c is the total number of distinct GTE that we consider as feature.

GTE enumeration

We compute GTEs by using a local growth algorithm. For computing $\boldsymbol{g}^{uv}_i$, we first enumerate all graphlets of G_i having both the nodes, u and v. Starting from node-pair (edge graphlet) $gl = \{u,v\}$, in each iteration of growth we add a new vertex w from the immediate neighborhood of the graphlet gl to obtain a larger graphlet $gl = gl \cup \{w\}$. Growth is terminated when $|gl| = 5$. The enumeration process is identical to the work of Rahman et al. [29]. After enumeration, GTE is easily obtained from graphlet embedding by marking the edge (u,v) as the transition trigger (see Figure

8.5). The computation of GTEs of different node-pairs are not dependent on each other, this makes GTE enumeration task embarrassingly parallel.

8.3.2 Unsupervised Feature Learning

GraTFEL performs the task of unsupervised feature extraction as learning an optimal coding function h. Lets consider, $\boldsymbol{e}$ is a feature vector from either $\hat{\mathbf{E}}$ or $\bar{\mathbf{E}}$ ($\boldsymbol{e} \in \hat{\mathbf{E}} \cup \bar{\mathbf{E}}$). Now, the coding function h compresses $\boldsymbol{e}$ to a code vector $\boldsymbol{\alpha}$ of dimension l, such that $l < k$. Here l is a user-defined parameter which represents the code length and k is the size of feature vector. Many different coding functions exist in the dimensionality reduction literature, but GraTFEL chooses the coding function which incurs the minimum loss in the sense that from the code $\boldsymbol{\alpha}$ we can reconstruct $\boldsymbol{e}$ with the minimum error over all possible $\boldsymbol{e} \in \hat{\mathbf{E}} \cup \bar{\mathbf{E}}$. We frame the learning of h as an optimization problem, which we discuss below through two operations: Compression and Reconstruction.

Compression: It obtains $\boldsymbol{\alpha}$ from $\boldsymbol{e}$. This transformation can be expressed as a nonlinear function of linear weighted sum of the graphlet transition features.

$$\boldsymbol{\alpha} = f(\boldsymbol{W}^{(c)}\boldsymbol{e} + \boldsymbol{b}^{(c)}) \tag{8.2}$$

$\boldsymbol{W}^{(c)}$ is a $(k \times l)$ dimensional matrix. It represents the weight matrix for compression and $\boldsymbol{b}^{(c)}$ represents biases. $f(\cdot)$ is the Sigmoid function, $f(x) = \frac{1}{1+e^{-x}}$.

Reconstruction: It performs the reverse operation of compression, i.e., it obtains the graphlet transition features $\mathbf{e}$ from $\boldsymbol{\alpha}$ which was constructed during the compression operation.

$$\boldsymbol{\beta} = f(\boldsymbol{W}^{(r)}\boldsymbol{\alpha} + \boldsymbol{b}^{(r)}) \tag{8.3}$$

$\boldsymbol{W}^{(r)}$ is a matrix of dimensions $(l \times k)$ representing the weight matrix for reconstruction, and $\boldsymbol{b}^{(r)}$ represents biases.

The optimal coding function h constituted by the compression and reconstruction operations is defined by the parameters $(\boldsymbol{W}, \boldsymbol{b}) = (\boldsymbol{W}^{(c)}, \boldsymbol{b}^{(c)}, \boldsymbol{W}^{(r)}, \boldsymbol{b}^{(r)})$. The objective is to minimize the reconstruction error. Reconstruction error for a graphlet transition feature vector (e) is defined as, $J(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{e}) = \frac{1}{2} \parallel \boldsymbol{\beta} - \boldsymbol{e} \parallel^2$. Over all possible feature vectors, the average reconstruction error augmented with a regularization term yields the final objective function $J(\boldsymbol{W}, \boldsymbol{b})$:

$$\begin{aligned} J(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{2m} \sum_{\boldsymbol{e} \in \hat{\mathbf{E}} \cup \bar{\mathbf{E}}} (\frac{1}{2} \parallel \boldsymbol{\beta} - \boldsymbol{e} \parallel^2) \\ + \frac{\lambda}{2} (\parallel \boldsymbol{W}^{(c)} \parallel_F^2 + \parallel \boldsymbol{W}^{(r)} \parallel_F^2) \end{aligned} \tag{8.4}$$

Here, λ is a user assigned regularization parameter, responsible for preventing over-fitting. $\parallel \cdot \parallel_F$ represents the Frobenius norm of a matrix. In this work we use $\lambda = 0.1$.

Optimization:

The training of optimal coding defined by parameters $(\boldsymbol{W}, \boldsymbol{b})$ begins with random initialization of the parameters. Since the cost function $J(\boldsymbol{W}, \boldsymbol{b})$ defined in Equation (8.4) is non-convex in nature, we obtain a local optimal solution using the gradient descent approach. Such approach usually provides practically useful results (as shown in the Section 8.4). The parameter updates of the gradient descent are similar to the parameter updates for optimizing Auto-encoder in machine learning. One iteration of gradient descent updates the parameters using following equations:

$$
\begin{aligned}
W_{ij}^{(c)} &= W_{ij}^{(c)} - \sigma \frac{\partial}{\partial W_{ij}^{(c)}} J(W,b) \\
W_{ij}^{(r)} &= W_{ij}^{(r)} - \sigma \frac{\partial}{\partial W_{ij}^{(r)}} J(W,b) \\
b_{i}^{(c)} &= b_{i}^{(c)} - \sigma \frac{\partial}{\partial b_{i}^{(c)}} J(W,b) \\
b_{i}^{(r)} &= b_{i}^{(r)} - \sigma \frac{\partial}{\partial b_{i}^{(r)}} J(W,b)
\end{aligned}
\tag{8.5}
$$

Now, from Equation (8.4), the partial derivative terms in equations (8.5) can be written as,

$$
\begin{aligned}
\frac{\partial}{\partial W_{ij}^{(c)}} J(W,b) &= \frac{1}{2m} \sum_{\boldsymbol{e} \epsilon \hat{\mathbf{E}}} \frac{\partial}{\partial W_{ij}^{(c)}} J(\mathbf{W}, \mathbf{b}, \mathbf{e}) + \lambda W_{ij}^{(c)} \\
\frac{\partial}{\partial W_{ij}^{(r)}} J(W,b) &= \frac{1}{2m} \sum_{\boldsymbol{e} \epsilon \hat{\mathbf{E}}} \frac{\partial}{\partial W_{ij}^{(r)}} J(\mathbf{W}, \mathbf{b}, \mathbf{e}) + \lambda W_{ij}^{(r)} \\
\frac{\partial}{\partial b_{i}^{(c)}} J(W,b) &= \frac{1}{2m} \sum_{\boldsymbol{e} \epsilon \hat{\mathbf{E}}} \frac{\partial}{\partial b_{i}^{(c)}} J(\mathbf{W}, \mathbf{b}, \mathbf{e}) \\
\frac{\partial}{\partial b_{i}^{(r)}} J(W,b) &= \frac{1}{2m} \sum_{\boldsymbol{e} \epsilon \hat{\mathbf{E}}} \frac{\partial}{\partial b_{i}^{(r)}} J(\mathbf{W}, \mathbf{b}, \mathbf{e})
\end{aligned}
\tag{8.6}
$$

The optimization problem is solved by computing partial derivative of cost function $J(\mathbf{W}, \mathbf{b}, \mathbf{e})$ using the back propagation approach [111]. Once the optimization is done, the unsupervised feature representation of any node-pair (u, v) can be obtained by taking the outputs of compression stage (Equation (8.2)) of the trained optimal coding $(\boldsymbol{W}, \boldsymbol{b})$.

$$
\begin{aligned}
\boldsymbol{\alpha}_{[1,t-1]}^{uv} &= f(\boldsymbol{W}^{(c)} \boldsymbol{e}_{[1,t-1]}^{uv} + \boldsymbol{b}^{(c)}) = h(\boldsymbol{e}_{[1,t-1]}^{uv}) \\
\boldsymbol{\alpha}_{[2,t]}^{uv} &= f(\boldsymbol{W}^{(c)} \boldsymbol{e}_{[2,t]}^{uv} + \boldsymbol{b}^{(c)}) = h(\boldsymbol{e}_{[2,t]}^{uv})
\end{aligned}
\tag{8.7}
$$

Computational Cost

We use Matlab implementation of optimization algorithm L-BFGS (Limited-memory Broyden-Fletcher-Goldfarb-Shanno) for learning optimal coding. Non-convex nature of cost function allows us to converge to local optimum. We execute the algorithm for a limited number of iterations to obtain unsupervised features within a reasonable period of time. Each iteration of L-BFGS executes two tasks for each node-pair: back-propagation to compute partial differentiation of cost function, change the parameters $\boldsymbol{(W, b)}$. For each node-pair the time complexity is $\mathcal{O}(kl)$; here, k is the length of Graphlet Transition Event based feature representation, l is length of unsupervised feature representation. Therefore, the time complexity of one iteration is $\mathcal{O}(n^2kl)$, as the number of all node-pairs is $\mathcal{O}(n^2)$.

8.3.3 Supervised Link Prediction Model

Training dataset, $\hat{\mathbf{E}}$ is feature representation for time snapshots $[1, t-1]$, The ground truth ($\hat{\boldsymbol{y}}$) is constructed from G_t. After training the supervised classification model using $\hat{\boldsymbol{\alpha}}$=$h(\hat{\mathbf{E}})$ and $\hat{\boldsymbol{y}}$, prediction dataset $\bar{\mathbf{E}}$ is used to predict links at G_{t+1}. For this supervised prediction task, we experiment with several classification algorithms. Among them SVM (support vector machine) and AdaBoost perform the best.

Algorithm 14 GraTFEL

```
function GraTFEL($\mathbb{G}$: Dynamic Network, $t$: Time steps)    ▷ Output: $\bar{\mathbf{y}}$: Forecasted links at time step $t+1$
    $\hat{\mathbf{E}}$=GraphletTransitionFeature($\mathbb{G}$,1,$t-1$)
    $\hat{\mathbf{y}}$=Connectivity($G_t$)
    $\bar{\mathbf{E}}$=GraphletTransitionFeature($\mathbb{G}$,2,$t$)
    $h$=LearningOptimalCoding($\hat{\mathbf{E}} \cup \bar{\mathbf{E}}$)
    $\hat{\boldsymbol{\alpha}}$=$h(\hat{\mathbf{E}})$
    $\bar{\boldsymbol{\alpha}}$=$h(\bar{\mathbf{E}})$
    $C$=TrainClassifier($\hat{\boldsymbol{\alpha}}, \hat{\mathbf{y}}$)
    $\bar{\mathbf{y}}$=LinkForecasting($C, \bar{\boldsymbol{\alpha}}$)
    return $\bar{\mathbf{y}}$
end function
```

The pseudo-code of GraTFEL is given in Algorithm 14. For training link prediction model, we split the available network snapshots into two overlapping time windows, $[1, t-1]$ and $[2, t]$. GTE features $\hat{\boldsymbol{E}}$ and $\bar{\boldsymbol{E}}$ are constructed in Lines 2 and 4, respectively. Then we learn optimal coding for node-pairs using graphlet transition features $(\hat{\mathbf{E}} \cup \bar{\mathbf{E}})$ in Line 5. Unsupervised feature representations are constructed using learned optimal coding (Lines 6 and 7) using output of compression stage (Equation 8.7). Finally, a classification model C is learned (Line 8), which is used for predicting link in G_{t+1}(Line 9).

8.4 Experimental Results

In this section we demonstrate the performance of GraTFEL using three real world dynamic network datasets: **Enron**, **Collaboration** and **Facebook**. We also show performance comparison between GraTFEL, and existing state-of-the-art dynamic link prediction methodologies. Experimental results also discuss the contribution of Unsupervised Feature Learning on GraTFEL's performance. Bellow, we discuss the datasets, evaluation metrics, competing methods, implementation details and results.

Table 8.4.: Basic statistics of the datasets used.

Characteristics	Enron	Collaboration	Facebook1
# Snapshots	11	10	9
# Nodes (n)	184	315	663
Avg.# Edge (ae)	217	255	1,299
$\#Node\text{-}Pairs(m)$	$16,836$	$49,455$	$219,453$
Avg. Graph Density (ae/m)	.013	.005	.006

8.4.1 Dataset Descriptions

Here we discuss the construction and characteristics of the datasets used for experiments. Detail statistics of the datasets is presented in Table 8.4. Although the number of vertices in these networks are in hundreds, these are large datasets considering the possible node-pairs and multiple temporal snapshots.

Enron email corpus [112] consists of email exchanges between 184 Enron employees. The Enron dataset has 11 time stamps; the task is to use first 10 snapshots for predicting links in the 11^{th} snapshot. Following [69], we aggregate data into time steps of 1 week. We use the data from weeks 147 to 157 of the data trace for the experiments. The reason for choosing that window is that the snapshot of the graph at week 157 has the highest number of edges.

Collaboration dataset has 10 time stamps with collaboration between 315 authors. The Collaboration dataset is constructed from citation data containing 1.4 million papers [113]. We process the data to construct a network of authors with edges between them if they co-author a paper. Considering each year as a time stamp, the data of years 2000-2009 (10 time stamps) is used for this experiment, where the data from the first nine time stamps is used for training and the last for prediction. Since this data is very sparse, we pre-process the data to retain only the active authors, who have last published papers on or after year 2010; moreover, the selected authors participate in at least two edges in seven or more time stamps.

Facebook is a network of Facebook wall posts [114]. Each vertex is a Facebook user account and an edge represents the event that one user posts a message on the wall of another user. Facebook has 9 time stamps and 663 nodes. For pre-processing Facebook we follow the same setup as is discussed in [115]; wall posts of 90 days are aggregated in one time step. We filter out all people who are active for less than 6 of the 9 time steps, along with the people who have degree less than 30, leaving 663 remaining people (nodes).

8.4.2 Evaluation Metrics

For evaluating the proposed method we use two metrics, namely, area under Precision-Recall (PR) curve (PRAUC) [116] and an information retrieval metric, Normalized Discounted Cumulative Gain (NDCG).

PRAUC is best suited for evaluating two class classification performance when class membership is skewed towards one of the classes. This is exactly the case for link prediction; the number of edges ($|E|$) is very small compared to the number of possible node-pairs m (see average graph density in Table 8.4). In such scenarios, area under the Precision-Recall curve (PRAUC) gives a more informative assessment of the algorithm's performance than other metrics such as, accuracy. The reason why PRAUC is more suitable for the skewed problem is that it does not factor in the count of true negatives in its calculation. In skewed data where the number of negative examples is huge compared to the number of positive examples, true negatives are not that meaningful.

We also use NDCG, an information retrieval metric (widely used by the recommender systems community) to evaluate the proposed method. NDCG measures the performance of link prediction system based on the graded relevance of the recommended links. $NDCG_p$ varies from 0.0 to 1.0, with 1.0 representing ideal ranking of edges. Here, p is a parameter chosen by user representing the number of links ranked by the method. We use $p = 50$ in our experiments. $NDCG_p$ is defined as below:

$$DCG_p = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{log_2(i+1)}$$
$$NDCG_p = \frac{DCG_p}{IDCG_p}$$

Here, rel_i is 1 if the ranked link is an edge at future time stamp $t+1$ or 0 otherwise. $IDCG_p$ is the maximum possible value of DCG_p; its appearance in the denominator of $NDCG_p$ definition normalizes the value between 0 and 1.

Some of the earlier works on link prediction have used area under the ROC curve (AUC) to evaluate link prediction works [71,117]. But recent works [118] have demonstrated the limitations of AUC and argued in favor of PRAUC over AUC for evaluation of link prediction. So we have not used AUC in this work.

8.4.3 Competing Methods for Comparison

We compare the performance of GraTFEL with link prediction methods from four categories: (1) topological feature based methods, (2) feature time series based methods [71], (3) a tensor factorization based method CANDECOMP/PARAFAC (CP) [66], and (4) node representation based methods DeepWalk [109] and Node2Vec [119].

Besides these four works, there are two other existing works for link prediction in dynamic network setting; one is based on deep Learning [70] and the other is based on a nonparametric method [68]. We cannot compare with these models as we were unsuccessful in obtaining the implementations of these methods from their corresponding authors. Besides, both the methods have numerous parameters and their implementation is complex.

For **topological feature based methods**, we consider four prominent topological features: Common Neighbors (CN), Adamic-Adar (AA), Jaccard's Coefficient (J) and Katz measure ($Katz$). However, in existing works, these features are defined for static networks only; so we adapt these features for the dynamic network setting by computing the feature values over the collapsed[2] dynamic network. We also combine the above four features to construct a combined feature vector of length four (**J**accard's Coefficient, **A**damic-Adar, **C**ommon Neighbors and **K**atz), which we call $JACK$ and use it with a classifier to build a supervised link prediction method, and include this model in our comparison.

[2] Collapsed network is constructed by superimposing all network snapshots.

Second, we compare GraTFEL with **time-series based** neighborhood similarity scores proposed in [71]. In this work, the authors consider several neighborhood-based node similarity scores combined with connectivity information (historical edge information). Authors use time-series of similarities to model the change of node similarities over time. Among 16 proposed methods, we consider 4 that are relevant to the link prediction task on unweighted networks and also have the best performance. TS-CN-*Adj* represents time-series on normalized score of Common Neighbors and connectivity values at time stamps $[1, t]$. Similarly, we get time-series based scores for Adamic-Adar (TS-AA-*Adj*), Jaccard's Coefficient (TS-J-*Adj*) and Preferential Attachment (TS-PA-*Adj*).

Third, we compare GraTFEL with a tensor factorization based method, called **CANDECOMP/PARAFAC (CP)** [66]. In this method, the dynamic network is represented as a three-dimensional tensor $\mathcal{Z}(n \times n \times t)$. Using CP decomposition $\mathcal{Z}$ is factorized into three factor matrices. The link prediction score is computed by using the factor matrices. We adapted the CP link prediction method for unipartite networks; which has originally been developed for bipartite networks.

Finally, we compare GraTFEL with latent node representation based methods **DeepWalk** [109] and **Node2Vec** [119]. We use DeepWalk to construct latent representation of nodes from the collapsed dynamic network. Then we construct latent representation of node-pairs by computing cross product of latent representation of the participating nodes. For example, if the node representations in a network are vectors of size l, then the representation of a node-pair (u, v) will be of size l^2, constructed from the cross product of u and v's representation. The DeepWalk based node-pair representation is then used with a classifier to build a supervised link prediction method. We choose node representation size $l = 2, 4, 6, 8, 10$ and report the best performance. Similarly, we use Node2Vec to construct latent representation of nodes for the collapsed dynamic network. Node-pair features are constructed by using binary operators suggested by Grover et al. [119]. Authors suggest four operators, namely *Average*, *Hadamard*, *Weighted-L1* and *Weighted-L2*; all of which give us node-pair representations of size l (contrary to the cross product which gives node-pair representations of size l^2). The parameters are chosen as suggested by the authors, specially for *return parameter* p and *in-out parameter* q (which controls the walk strategy for Node2Vec) we perform grid search over $p, q \in \{0.25, 0.50, 1, 2, 4\}$ and report the best result.

8.4.4 Implementation Details

For implementation of GraTFEL we use a combination of Python and Matlab version $R2014b$. Graphlet transition is enumerated using a python implementation. Feature vector construction and unsupervised feature learning are done using Matlab. The unsupervised feature learning method runs for a maximum of 50 iterations or until it converges to a local optimal solution. We use coding size $l = 200$ for all three datasets [3]. For supervised link prediction step we use several Matlab provided classification algorithms, namely, AdaBoostM1, RobustBoost, and Support Vector Machine (SVM). We use Matlab for computing the feature values (CN, AA, J, Katz) that we use in other competing methods. Time-series methods are implemented using Python. We use the ARIMA (autoregressive integrated moving average) time series model implemented in Python module **statsmodels**. Tensor factorization based method CP was implemented using Matlab Tensor Toolbox. The DeepWalk implementation is provided by the authors of [109]. We use it to extract node features and extend it for link prediction (using Matlab). Similarly, the Node2Vec implementation is made available by the authors [119]. We use the extracted features form Node2Vec to construct node-pair representations for link prediction (using Matlab). The Source-Code of GraTFEL is available at `https://github.com/DMGroup-IUPUI/GraTFEL-Source`.

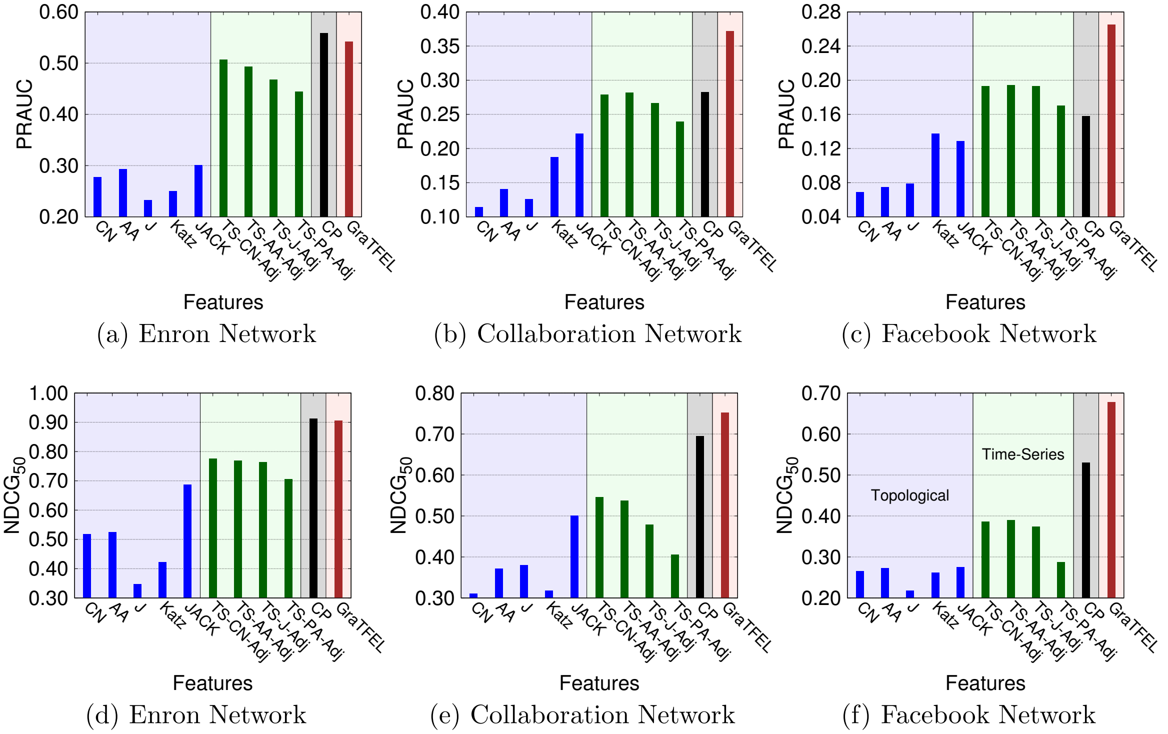
[3] We experiment with different coding sizes ranging from 100 to 800. The change in link prediction performance is not very sensitive to the coding size. At most 2.9% change in PRAUC was observed for different coding sizes.

8.4.5 Performance Comparison Results with Competing Methods

In Figure 8.6 we present the performance comparison results of GraTFEL with the three kinds of competing methods that we have discussed earlier. The figure have six bar charts. The bar charts from the left to the right columns display the results for Enron, Collaboration and Facebook datasets, respectively. The bar charts in a column show comparison results using PRAUC (top), and $NDCG_{50}$ (bottom) metrics. Each chart has eleven bars, each representing a link prediction method, where the height of a bar is indicative of the performance metric value of the corresponding method. In each chart, from left to right, the first five bars (blue) correspond to the topological feature based methods, the next four (green) represent time series based methods, the tenth bar (black) represents tensor factorization based method CP, and the final bar (brown) represents the proposed method GraTFEL.

GraTFEL vs. Topological

We first analyze the performance comparison between GraTFEL and topological feature based methods (first five bars). The best of the topological feature based methods have a PRAUC value of 0.30, 0.22 and 0.137 in Enron, Collaboration, and Facebook dataset (see Figures 8.6(a), 8.6(b) and 8.6(c)), whereas the corresponding PRAUC values for GraTFEL are 0.54, 0.37 and 0.265, which translates to 80%, 68% and 93.4% improvement of PRAUC by GraTFEL for these datasets. Superiority of GraTFEL over all the topological feature based baseline methods can be attributed to the capability of graphlet transition based feature representation to capture temporal characteristics of local neighborhood. Similar trend is observed using $NDCG_{50}$ metric, see Figures 8.6(d), 8.6(e) and 8.6(f).



(a) Enron Network

(b) Collaboration Network

(c) Facebook Network

(d) Enron Network

(e) Collaboration Network

(f) Facebook Network

Figure 8.6.: Comparison with competing link prediction methods. Each bar represents a method and the height of the bar represents the value of the performance metric. Results for Enron network are presented in charts (a,d), results of Collaboration data are presented in charts (b,e), and results of Facebook data are presented in charts (c,f). The group of bars in a chart are distinguished by color, so the figure is best viewed on a computer screen or color print.

GraTFEL vs. Time-Series

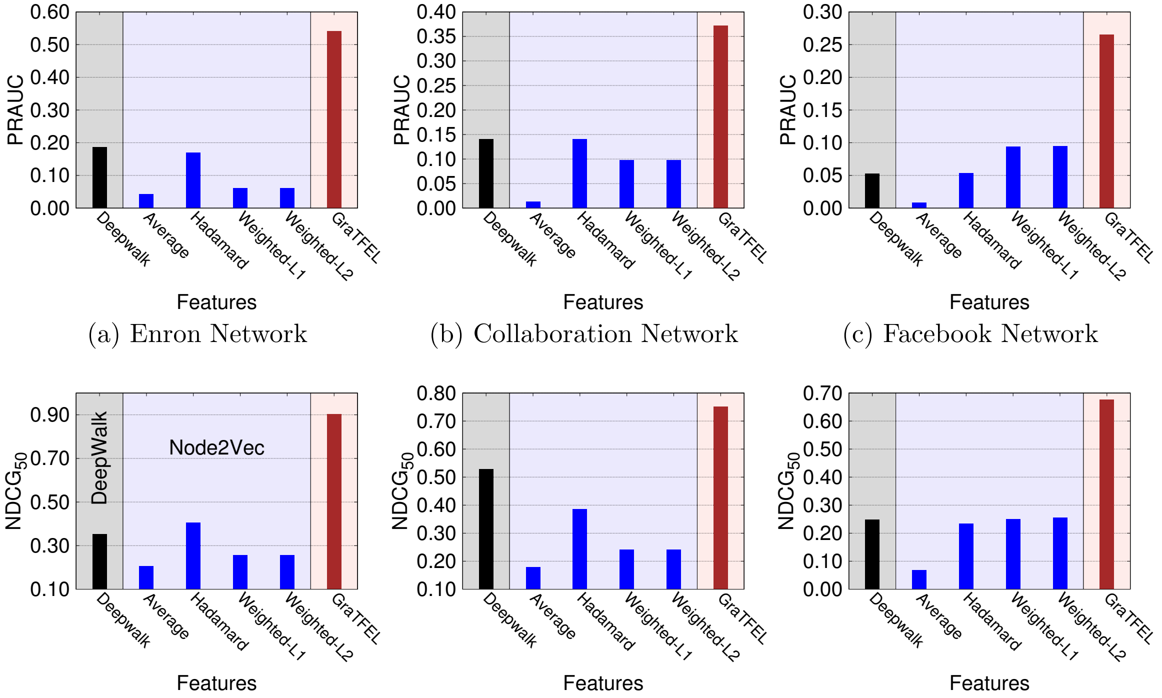
The performance of time-series based method (four green bars) is generally better than the topological feature based methods. The best of the time-series based method has a PRAUC value of 0.503, 0.28 and 0.19 on these datasets, and GraTFEL's PRAUC values are better than these values by 7.3%, 32.1% and 39.5% respectively. Time-series based methods, though model the temporal behavior well, probably fail to capture signals from the neighborhood topology of the node-pairs. Superiority of GraTFEL over Time-Series methods is also similarly indicated by information retrieval metric $NDCG_{50}$.

GraTFEL vs. CANDECOMP/PARAFAC (CP)

Finally, the tensor factorization based method CP performs marginally better (by around 3.2% in PRAUC, and 0.84% in $NDCG_{50}$) than GraTFEL in small and simple networks, such as Enron (see Figure 8.6(a, d)). But its performance degrades on comparatively large and complex networks, such as Collaboration and Facebook. On Facebook network, the performance of CP is even worse than the time-series based methods (see Figure 8.6(c)). GraTFEL comfortably outperforms CP on larger graphs, see Figures 8.6(b, c, e, f). In terms of PRAUC, GraTFEL's performance is 31.6% better in Collaboration and 67.6% better in Facebook than CP. This demonstrates the superiority of GraTFEL over one of the best state-of-the-art dynamic link prediction. A reason for CP's bad performance on large graphs can be its inability to capture network structure and dynamics using high-dimensional tensors representation.

8.4.6 Comparison with Node Representation Methods

In Figure 8.7 we present the performance comparison results of GraTFEL with the two node representation methods. The figure have six bar charts. The bar charts from the left to the right columns display the results for Enron, Collaboration and Facebook datasets, respectively. The bar charts in a column show comparison results using PRAUC (top), and $NDCG_{50}$ (bottom) metrics. Each chart has six bars, each representing a link prediction method, where the height of a bar is indicative of the performance metric value of the corresponding method. In each chart, from left to right, the first bar (black) is for DeepWalk, the next four bars (blue) correspond to the Node2Vec feature representation of node-pairs, and the final bar (brown) represents the proposed method GraTFEL.



(a) Enron Network (b) Collaboration Network (c) Facebook Network

(d) Enron Network (e) Collaboration Network (f) Facebook Network

Figure 8.7.: Comparison with node representation based link prediction methods. Each bar represents a method and the height of the bar represents the value of the performance metric. Results for Enron, Collaboration and Facebook networks are presented in charts (a,d), (b,e) and (c,f) respectively. The group of bars in a chart are distinguished by color, so the figure is best viewed on a computer screen or color print.

GraTFEL vs. DeepWalk

The DeepWalk based method (black bars in Figure 8.7) performs much poorly in terms of both PRAUC and $NDCG_{50}$—even poorer than the topological based method in all four datasets. Possible reason could be the following: the latent encoding of nodes by DeepWalk is good for node classification, but the cross-product of those codes fails to encode the information needed for effective link prediction.

GraTFEL vs. Node2Vec

The Node2Vec node-pair representations (blue bars in Figure 8.7) also performs poorly in terms of both PRAUC and $NDCG_{50}$. Feature representation of node-pairs are constructed using Node2Vec representations of participating nodes using binary operations (*Average*, *Hadamard*, *Weighted-L1* and *Weighted-L2*) proposed by [119]. Each blue bar represents a binary operation used. Link prediction performance is poor for all Node2Vec based methods. This demonstrates, the absence of a suitable mechanism for constructing effective node-pair representations from node representations. Contrary to DeepWalk and Node2Vec, the proposed method GraTFEL gives effective node-pair representations using Graphlet Transition Events.

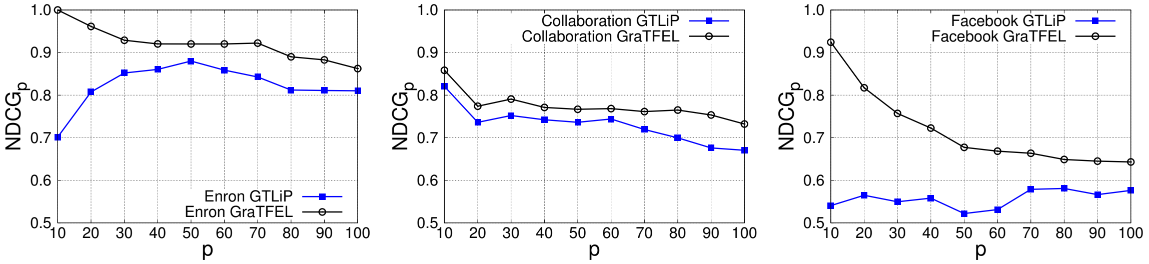
Performance Across Datasets

When we compare the performance of all the methods across different datasets, we observe varying performance. For example, for both the metrics, the performance of dynamic link prediction on Facebook graph is lower than the performance on Collaboration graph, which, subsequently, is lower than the performance on Enron graph, indicating that link prediction in Facebook data is a harder problem to solve. In these harder networks, GraTFEL perform substantially better than all the other competing methods that we consider in this experiment.

8.4.7 Contribution of Unsupervised Feature Learning

GraTFEL has two novel aspects: first, utilization of Graphlet Transition Events (GTEs) as features, and the second is Unsupervised Feature Learning (UFL) by optimal coding. In this section, we compare the relative contribution of these two aspects in the performance of GraTFEL. For this comparison, we build a version of GraTFEL which we call GTLiP. GTLiP uses GTEs just like GraTFEL, but it does not use optimal coding, rather it uses the GTEs directly as features. In

Figure 8.8, we show the comparison between GraTFEL, and GTLiP using $NDCG_p$ for different p values for all the datasets. The superiority of GraTFEL over GTLiP for all the datasets over a range of p values is clearly evident from the three charts. UFL improves the classifier's performance by learning optimal feature encoding.



(a) Enron Network (b) Collaboration Network (c) Facebook Network

Figure 8.8.: Performance comparison between link prediction methods with (GraTFEL) and without (GTLiP) unsupervised feature learning. Y-axis represents the $NDCG_p$ score and X-axis represents the value of p.

8.5 Conclusion

In this Chapter, we present two applications of graphlet analysis. Firstly, categorization/classification of networks using GFD characterization. We show that, graphlet frequency distributions (GFDs) are potent structural characterization of static networks, by successfully clustering networks of various domains. Second, we present GraTFEL a supervised approach for link prediction in dynamic networks, which uses unsupervised coding of graphlet transition events (GTEs). GraTFEL uses an unsupervised feature extraction method for learning effective feature vector for a node-pair over multiple temporal snapshots of a given network. Then it uses supervised machine learning methodologies for predicting future links. The proposed method outperforms several existing methods that are based on topological features, time series, tensor analysis and representation learning. In the future, this work can be extended using several other unsupervised feature learning models where each one builds models using node-pair representations of similar topological neighborhood or past linkage history.

9 FUTURE WORK AND CONCLUSION

Graphlet based network analysis, uses counts of graphlets to capture local network structures to characterize a network. Generally it's computationally expensive to gather graphlet based statistics for real life networks. Multitude of techniques have been developed for achieving better scalability for graphlet analysis. Restricted set of graphlets (cliques, paths, cycles etc) have been considered, which enables researcher to develop efficient algorithms by taking advantages provided by the imposed restrictions. Parallel and distributed computational paradigms are also explored by many researchers. Several recent works use sampling based techniques for efficient graphlet analysis. Triple analysis methods designed for streaming and semi-streaming access models, allow efficient triple analysis for large networks using limited memory. In this dissertation, we propose efficient sampling based methods for both triple and graphlet analysis. This dissertation also coins a category of algorithms (MCMC based sampling) which are able to estimate graphlet based network metrics without observing the whole network.

Graphlet analysis has been demonstrated to be useful in many applications. For example, triangle counting and triple analysis techniques are used in social network characterization. Furthermore, biological network fingerprints are developed using frequencies of graphlets. Graphlet frequencies are also used to characterize structural similarities of nodes in biological networks and to design effective graph kernels. Despite being a very successful tool for capturing network characteristics, graphlets have not been very popular in solving many exciting network based problems (like link prediction, link forecasting etc.). This dissertation is a forerunner in this endeavor and it has tremendous opportunity to grow in terms of scope, methods, algorithms and applications.

The first direction of future works is to make Graphlet Transition Event (GTE) more efficient, so that the link prediction method GraTFEL is scalable for large dynamic networks. As discussed in Chapter 8, computation of GTEs of node-pairs are inherently parallel. In future works, we intend to look into parallel and distributed computational paradigm to design efficient GTE analysis techniques.

The second direction of future works is to increase the scope of GTE based analysis. The scope of GTE can be extended by defining GTE with respect to a node instead of a node-pair. GTE corresponding to a node has the potential to capture the dynamics of the node and its local structure. This will make the analysis techniques more scalable, as the number of node is $\mathcal{O}(n)$. The number of node-pairs used for GraTFEL is $\mathcal{O}(n^2)$.

In conclusion, we like to reiterate the main contributions of this dissertation. We propose sample based triangle and graphlet counting algorithms. These algorithms reduce the required computational effort by sampling edges of the given network and counting triangles/graphlets incident on the sampled edges only. We also propose MCMC walk based sampling methods; which walk on graphlet space and are oblivious of the network structure. For triple sampling, we propose two MCMC based algorithms. MCMC sampling methods can sample triples from networks that are restricted or dynamic, for which direct sampling methods fail. Similarly for larger graphlets, we present Guise, an efficient method for approximating the graphlet frequency distribution (GFD) in a principled manner that offers significant speed up in comparison with the existing time consuming ways of brute force counting. Finally, we propose a network analysis paradigm named Graphlet Transition Event (GTE), which uses the transition of graphlets in a dynamic network to capture temporal dynamics of local network structure. We present GraTFEL a supervised approach for link prediction in dynamic networks, which uses unsupervised coding of graphlet transition events (GTEs).

REFERENCES

REFERENCES

[1] Stephen P. Borgatti, Ajay Mehra, Daniel J. Brass, and Giuseppe Labianca. Network analysis in the social sciences. *Science*, 323:892–895, 2009.

[2] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 251–262, 1999.

[3] Natasa Pržulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 23(2):e177–e183, 2007.

[4] Albert Laszlo Barabàsi and Reka Albert. Emergence of scaling in random networks. *Science*, 286:509–512, October 1999.

[5] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of "small-world" networks. *Nature*, 393:440–442, 1998.

[6] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 177–187, 2005.

[7] Ellen W. Zegura, Kenneth L. Calvert, and Michael J. Donahoo. A quantitative comparison of graph-based models for internet topology. *IEEE/ACM Transactions on Networking (TON)*, 5(6):770–783, 1997.

[8] P. Erdős and A. Rényi. On random graphs. I. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

[9] P. Erdős and A. Rényi. On the evolution of random graphs. In *Publication of the Mathematical Institute of The Hungarian Academy of Sciences*, pages 17–61, 1960.

[10] William Eberle and Lawrence Holder. Graph-based approaches to insider threat detection. In *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*, pages 44:1–44:4, 2009.

[11] J. Baumes, M. Goldberg, M. Magdon-Ismail, and W. Wallace. Discovering hidden groups in communication networks. In *Proceedings of the 2nd NSF/NIJ Symposium on Intelligence and Security Informatics*, pages 378–389, 2004.

[12] Mohammad Al Hasan and Mohammed J. Zaki. A survey of link prediction in social networks. In Charu C. Aggarwal, editor, *Social Network Data Analytics*, pages 243–275. Springer US, 2011.

[13] Tijana Milenkoviæ and Natasa Pržulj. Uncovering biological network function via graphlet degree signatures. *Cancer Informatics*, 6:257–273, 2008.

[14] Tyson J. J. and Novak B. Functional motifs in biochemical reaction networks. *Annual Review of Physical Chemistry*, 61:219–40, 2010.

[15] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient algorithms for large-scale local triangle counting. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 4(3):13:1–13:28, 2010.

[16] S. Wasserman and K. Faust. *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.

[17] James S. Coleman. Social capital in the creation of human capital. *American Journal of Sociology*, 94:S95–S120, 1988.

[18] Brooke Foucault Welles, Anne Van Devender, and Noshir Contractor. Is a friend a friend?: Investigating the structure of friendship networks in virtual worlds. In *Extended Abstracts on Human Factors in Computing Systems (CHI EA)*, pages 4027–4032, 2010.

[19] Jean-Pierre Eckmann and Elisha Moses. Curvature of co-links uncovers hidden thematic layers in the World Wide Web. *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, 99(9):5825–5829, 2002.

[20] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 16–24, 2008.

[21] Natasa Pržulj, D. G. Corneil, and I. Jurisica. Modeling interactome: Scale-free or geometric? *Bioinformatics*, 20(18):3508–3515, 2004.

[22] H. Azari Soufiani and E. M Airoldi. Graphlet decomposition of a weighted network. *ArXiv e-prints*, 2012.

[23] J. Lussier and J. Bank. Final report: Local structure and evolution for cascade prediction. 2011.

[24] N. Shervashidzea, S. Vishwanathan, T. Petri, K. Mehlhorn, and K.Borgwardt. Efficient graphlet kernels for large graph comparison. In *Proceedings of 12th International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 488–495, 2009.

[25] Vladimir Vacic, Lilia M. Iakoucheva, Stefano Lonardi, and Predrag Radivojac. Graphlet kernels for prediction of functional residues in protein structures. *Journal of Computational Biology*, pages 55–72, 2010.

[26] Nesreen K Ahmed, Jennifer Neville, Ryan A Rossi, and Nick Duffield. Efficient graphlet counting for large networks. In *Proceedings of the 15th IEEE International Conference on Data Mining (ICDM)*, pages 1–10, 2015.

[27] Oleksii Kuchaiev, Aleksandar Stevanović, Wayne Hayes, and Nataša Pržulj. Graphcrunch 2: Software tool for network modeling, alignment and clustering. *BMC Bioinformatics*, 12(1):1–13, 2011.

[28] Mahmudur Rahman and Mohammad Al Hasan. Approximate triangle counting algorithms on multi-cores. In *Proceedings of the 2013 IEEE International Conference on Big Data*, pages 127–133, 2013.

[29] M. Rahman, M. A. Bhuiyan, and M. Al Hasan. GRAFT: An efficient graphlet counting method for large graph analysis. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 26(10):2466–2478, 2014.

[30] Mahmudur Rahman, Mansurul Bhuiyan, and Mohammad Al Hasan. GRAFT: an approximate graphlet counting algorithm for large graph analysis. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM)*, pages 1467–1471, 2012.

[31] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, pages 10–10, 2010.

[32] Mahmudur Rahman and Mohammad Al Hasan. Sampling triples from restricted networks using MCMC strategy. In *Proceedings of the 23rd ACM International Conference on Information and Knowledge Management (CIKM)*, pages 1519–1528, 2014.

[33] M.A. Bhuiyan, M. Rahman, M. Rahman, and M. Al Hasan. GUISE: Uniform sampling of graphlets for large graph analysis. In *Proceedings of the 12th IEEE International Conference on Data Mining (ICDM)*, pages 91–100, 2012.

[34] Mahmudur Rahman, Mansurul Alam Bhuiyan, Mahmuda Rahman, and Mohammad Al Hasan. GUISE: a uniform sampler for constructing frequency histogram of graphlets. *Knowledge and Information Systems*, 38(3):511–536, 2014.

[35] Mahmudur Rahman and Mohammad Al Hasan. Link prediction in dynamic networks using graphlet. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, pages 394–409, 2016.

[36] M. E. J. Newman, D. J. Watts, and S. H. Strogatz. Random graph models of social networks. *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, 99(Suppl 1):2566–2572, 2002.

[37] Christophe Andrieu, Nando de Freitas, Arnaud Doucet, and Michael I. Jordan. An introduction to MCMC for machine learning. *Machine Learning*, 50(1):5–43, 2003.

[38] P. W. Holland and S. Leinhardt. Transitivity in structural models of small groups. *Small Group Research*, 2:107–124, 1971.

[39] Ziv Bar-Yossef, Ravi Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 623–632, 2002.

[40] Nurcan Durak, Ali Pinar, Tamara G. Kolda, and C. Seshadhri. Degree relations of triangles in real-world networks and graph models. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM)*, pages 1712–1716, 2012.

[41] T. Schank. *Algorithmic Aspects of Triangle-Based Network Analysis*. PhD thesis, Department of Computer Science, University of Karlsruhe, 2007.

[42] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17:209–223, 1997.

[43] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web*, pages 607–614, 2011.

[44] T. Schank and D. Wagner. Approximating clustering-coefficient and transitivity. *Journal of Graph Algorithms and Applications*, 9(2):265–275, 2005.

[45] Tamara G. Kolda, Ali Pinar, and C. Seshadhri. Triadic measures on graphs: The power of wedge sampling. In *Proceedings of the 13th SIAM International Conference on Data Mining (SDM)*, pages 10–18, 2013.

[46] C. Seshadhri, Ali Pinar, and Tamara G. Kolda. Wedge sampling for computing clustering coefficients and triangle counts on large graphs. *Statistical Analysis and Data Mining*, 7(4):294–307, 2014.

[47] Charalampos E. Tsourakakis. Fast counting of triangles in large real networks without counting: Algorithms and laws. In *Proceedings of the 8th IEEE International Conference on Data Mining (ICDM)*, pages 608–617, 2008.

[48] Charalampos E. Tsourakakis. Counting triangles in real-world networks using projections. *Knowledge and Information Systems*, 26(3):501–520, 2011.

[49] Haim Avron. Counting triangles in large graphs using randomized matrix trace estimation. In *Large-scale Data Mining: Theory and Applications (KDD Workshop)*, volume 10, 2010.

[50] Charalampos E. Tsourakakis, U Kang, Gary L. Miller, and Christos 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*, pages 837–846, 2009.

[51] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, pages 107–113, January 2008.

[52] Rasmus Pagh and Charalampos E. Tsourakakis. Colorful triangle counting and a mapreduce implementation. *Information Processing Letters*, 112(7):277–281, 2012.

[53] M.N. Kolountzakis, G.L. Miller, R. Peng, and C.E. Tsourakakis. Efficient triangle counting in large graphs via degree-based vertex partitioning. *Internet Mathematics*, 8(1-2):161–185, 2012.

[54] Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. Counting triangles in data streams. In *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 253–262, 2006.

[55] Hossein Jowhari and Mohammad Ghodsi. New streaming algorithms for counting triangles in graphs. In *Proceedings of the 11th Annual International Conference on Computing and Combinatorics (COCOON)*, volume 3595, pages 710–716. 2005.

[56] Madhav Jha, C. Seshadhri, and Ali Pinar. A space-efficient streaming algorithm for estimating transitivity and triangle counts using the birthday paradox. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 9(3):15:1–15:21, 2015.

[57] M. Jha, A. Pinar, and C. Seshadhri. Counting triangles in real-world graph streams: Dealing with repeated edges and time windows. In *Proceedings of the 49th Asilomar Conference on Signals, Systems and Computers*, pages 1507–1514, November 2015.

[58] D.B Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[59] E. Akkoyunlu. The enumeration of maximal cliques of large graphs. *SIAM Journal on Computing*, 2:1–6, 1973.

[60] D. Marcus and Y. Shavitt. RAGE - A rapid graphlet enumerator for large networks. *SIAM Journal on Discrete Mathematics*, 56(2):810–819, 2012.

[61] David Liben-Nowell and Jon Kleinberg. The link prediction problem for social networks. In *Proceedings of the 12th International Conference on Information and Knowledge Management (CIKM)*, pages 556–559, 2003.

[62] Kurt Miller, Michael I. Jordan, and Thomas L. Griffiths. Nonparametric latent feature models for link prediction. In *Proceedings of the 22nd International Conference on Neural Information Processing Systems (NIPS)*, pages 1276–1284. 2009.

[63] Ryan N. Lichtenwalter, Jake T. Lussier, and Nitesh V. Chawla. New perspectives and methods in link prediction. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 243–252, 2010.

[64] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, August 2009.

[65] Aditya Krishna Menon and Charles Elkan. Link prediction via matrix factorization. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, pages 437–452, 2011.

[66] Daniel M. Dunlavy, Tamara G. Kolda, and Evrim Acar. Temporal link prediction using matrix and tensor factorizations. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 5(2):10:1–10:27, February 2011.

[67] J. Douglas Carroll and Jih-Jie Chang. Analysis of individual differences in multidimensional scaling via an n-way generalization of "eckart-young" decomposition. *Psychometrika*, 35(3):283–319, 1970.

[68] Purnamrita Sarkar, Deepayan Chakrabarti, and Michael I. Jordan. Nonparametric link prediction in dynamic networks. In *Proceedings of the 29th International Conference on Machine Learning (ICML-12)*, pages 1687–1694, 2012.

[69] Kevin S Xu and Alfred O Hero III. Dynamic stochastic blockmodels for time-evolving social networks. *IEEE Journal of Selected Topics in Signal Processing*, 8(4):552–562, 2014.

[70] Xiaoyi Li, Nan Du, Hui Li, Kang Li, Jing Gao, and Aidong Zhang. A deep learning approach to link prediction in dynamic networks. In *Proceedings of the 2014 SIAM International Conference on Data Mining*, pages 289–297, 2014.

[71] İsmail Güneş, Şule Gündüz-Öğüdücü, and Zehra Çataltepe. Link prediction using time series of neighborhood-based node similarity scores. *Data Mining and Knowledge Discovery*, 30(1):147–180, 2015.

[72] Peng Wang, BaoWen Xu, YuRong Wu, and XiaoYu Zhou. Link prediction in social networks: The state-of-the-art. *Science China Information Sciences*, 58(1):1–38, 2015.

[73] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298:824–827, 2002.

[74] Alex Arenas, Alberto Fernandez, Santo Fortunato, and Sergio Gómez. Motif-based communities in complex networks. *Journal of Physics A: Mathematical and Theoretical*, 41(22):224001, 2008.

[75] S. Itzkovitz and U. Alon. Subgraphs and network motifs in geometric networks. *Physical Review E, Statistical, Nonlinear, and Soft Matter Physics*, 71(2 Pt 2), 2005.

[76] Pedro Ribeiro and Fernando Silva. Querying subgraph sets with g-tries. In *Proceedings of the 2nd ACM SIGMOD Workshop on Databases and Social Networks (DBSocial)*, pages 25–30, 2012.

[77] Sebastian Wernicke. Efficient detection of network motifs. *EEE/ACM Transactions on Computational Biology and Bioinformatics Proceedings*, 3(4):347–359, October 2006.

[78] Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science*, 407:458–473, 2008.

[79] N. Kashtan, S. Itzkovitz, R. Milo, and U. Alon. Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs. *Bioinformatics*, 20(11):1746–1758, 2004.

[80] R. Duke, H. Lefmann, and V. Rdl. A fast approximation algorithm for computing the frequencies of subgraphs in a given graph. *SIAM Journal on Computing*, 24(3):598–620, 1995.

[81] Royi Itzhack, Yelena Mogilevski, and Yoram Louzoun. An optimal algorithm for counting network motifs. *Physica A: Statistical Mechanics and its Applications*, 381:482 – 490, 2007.

[82] M. Gonen and Y. Shavitt. Approximating the number of network motifs. In *Proceedings of the 6th International Workshop on Algorithms and Models for the Web-Graph*, pages 13–24, 2009.

[83] M. Gonen, D. Ron, and Y. Shavitt. Counting stars and other small subgraphs in sublinear time. *Computer Networks: The International Journal of Computer and Telecommunications*, 25(2):1365–1411, 2011.

[84] J. D. Gibbons and S. Chakraborti. *Nonparametric Statistical Inference, 5th Edition*. Chapman and Hall/CRC, 2010.

[85] Béla Bollobás and Paul Erdös. Cliques in random graphs. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 80, pages 419–427, 1976.

[86] M. Gjoka, M. Kurant, C.T. Butts, and A. Markopoulou. Walking in facebook: A case study of unbiased sampling of OSNs. In *Proceedings of the 29th Conference on Information Communications (INFOCOM)*, pages 2498–2506, 2010.

[87] Nicolas Kourtellis, Tharaka Alahakoon, Ramanuja Simha, Adriana Iamnitchi, and Rahul Tripathi. Identifying high betweenness centrality nodes in large social networks. *Social Network Analysis and Mining*, 3(4):899–914, 2013.

[88] Mohammad Al Hasan and Mohammed J. Zaki. Output space sampling for graph patterns. *Proceedings of the VLDB Endowment*, 2(1):730–741.

[89] Arun S. Maiya and Tanya Y. Berger-Wolf. Sampling community structure. In *Proceedings of the 19th International Conference on World Wide Web*, pages 701–710, 2010.

[90] Christian Hübler, Hans-Peter Kriegel, Karsten Borgwardt, and Zoubin Ghahramani. Metropolis algorithms for representative subgraph sampling. In *Proceedings of the 8th IEEE International Conference on Data Mining (ICDM)*, pages 283–292, 2008.

[91] Jure Leskovec and Christos Faloutsos. Sampling from large graphs. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 631–636, 2006.

[92] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[93] John Geweke. Evaluating the accuracy of sampling-based approaches to the calculation of posterior moments. In *Bayesian Statistics*, pages 169–193, 1992.

[94] Sebastian Wernicke and Florian Rasche. FANMOD: A tool for fast network motif detection. *Bioinformatics*, 22(9):1152–1153, 2006.

[95] Pinghui Wang, John C. S. Lui, Bruno Ribeiro, Don Towsley, Junzhou Zhao, and Xiaohong Guan. Efficiently estimating motif statistics of large networks. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 9(2):8:1–8:27, September 2014.

[96] Tanay Kumar Saha and Mohammad Al Hasan. Finding network motifs using MCMC sampling. In *Proceedings of the 6th Workshop on Complex Networks (CompleNet)*, pages 13–24, 2015.

[97] Falk Schreiber and Henning Schwobbermeyer. Frequency concepts and pattern detection for the analysis of motifs in networks. *Transactions on Computational Systems Biology*, 3:89–104, 2005.

[98] Jin Chen, Wynne Hsu, Mong Li Lee, and See-Kiong Ng. NeMoFinder: Dissecting genome-wide protein-protein interactions with meso-scale network motifs. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 106–115, 2006.

[99] Zahra Razaghi Moghadam Kashani, Hayedeh Ahrabian, Elahe Elahi, Abbas Nowzari-Dalini, Elnaz Saberi Ansari, Sahar Asadi, Shahin Mohammadi, Falk Schreiber, and Ali Masoudi-Nejad. Kavosh: a new algorithm for finding network motifs. *BMC Bioinformatics*, 10(1):1–12, 2009.

[100] Joshua A. Grochow and Manolis Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. In *Proceedings of the 11th International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 92–106, 2007.

[101] Saeed Omidi, Falk Schreiber, and Ali Masoudi-Nejad. MODA: An efficient algorithm for network motif discovery in biological networks. *Genes & Genetic Systems*, 84(5):385–395, 2009.

[102] N. Pržulj, D. G. Corneil, and I. Jurisica. Efficient estimation of graphlet frequency distributions in protein–protein interaction networks. *Bioinformatics*, 22(8):974–980, April 2006.

[103] Fan RK Chung. *Spectral Graph Theory*, volume 92. American Mathematical Society, 1997.

[104] Venkatesan Guruswami. Rapidly mixing markov chains: a comparison of techniques (a survey). *arXiv preprint arXiv:1603.01512*, 2016.

[105] Ravi Montenegro and Prasad Tetali. Mathematical aspects of mixing times in markov chains. *Foundations and Trends in Theoretical Computer Science*, 1(3):237–354, May 2006.

[106] David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *Journal of the American Society for Information Science and Technology*, 58(7):1019–1031, May 2007.

[107] Joe Ng, Fan Yang, and Larry Davis. Exploiting local features from deep networks for image retrieval. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 53–61, 2015.

[108] Honglak Lee, Peter Pham, Yan Largman, and Andrew Y Ng. Unsupervised feature learning for audio classification using convolutional deep belief networks. In *Advances in Neural Information Processing Systems*, pages 1096–1104, 2009.

[109] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. DeepWalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 701–710, 2014.

[110] Yoshua Bengio, Aaron C. Courville, and Pascal Vincent. Unsupervised feature learning and deep learning: A review and new perspectives. *CoRR*, abs/1206.5538, 2012.

[111] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chapter Learning Internal Representations by Error Propagation, pages 318–362. MIT Press, Cambridge, MA, USA, 1986.

[112] Carey E Priebe, John M Conroy, David J Marchette, and Youngser Park. Scan statistics on enron graphs. *Computational & Mathematical Organization Theory*, 11(3):229–247, 2005.

[113] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. Arnetminer: Extraction and mining of academic social networks. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 990–998, 2008.

[114] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P Gummadi. On the evolution of user interaction in Facebook. In *Proceedings of the 2nd ACM Workshop on Online Social Networks (WOSN)*, pages 37–42, 2009.

[115] Kevin S. Xu. Stochastic block transition models for dynamic networks. In *Proceedings of the 18th International Conference on Artificial Intelligence and Statistics*, pages 1079–1087, 2015.

[116] Jesse Davis and Mark Goadrich. The relationship between precision-recall and ROC curves. In *Proceedings of the 23rd International Conference on Machine Learning (ICML)*, pages 233–240, 2006.

[117] Chao Wang, Venu Satuluri, and Srinivasan Parthasarathy. Local probabilistic models for link prediction. In *Proceedings of the 7th IEEE International Conference on Data Mining (ICDM)*, pages 322–331, 2007.

[118] Yang Yang, Ryan N. Lichtenwalter, and Nitesh V. Chawla. Evaluating link prediction methods. *Knowledge and Information Systems*, 45(3):751–782, 2015.

[119] Aditya Grover and Jure Leskovec. Node2Vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 855–864, 2016.

VITA

VITA

Md Mahmudur Rahman

Home Page: http://cs.iupui.edu/~mmrahman/

Emails: mmrahman@iupui.edu

prime059@gmail.com

Education:

Ph.D. Computer Science (Specialization in Data Mining), 2011-2016

Purdue University, West Lafayette, Indiana, USA

M.Sc. Computer & Information Science, 2011-2013

Indiana University-Purdue University Indianapolis (IUPUI), Indianapolis Indiana, USA

B.S. Computer Science & Engineering, 2004-2009

Bangladesh University of Engineering and Technology (BUET), Dhaka, Bangladesh

Research Interests:

Data Mining, Graph Mining, Data Science, Big Data, Graph Analytics.

Experience:

- *Software Development Engineer*, Amazon Corporate LLC, Seattle, WA, USA, 2016-
- *Research Assistant*, IUPUI, Indianapolis, IN, USA, 2011-2016

- *Graphic Analytics Intern*, The Bosch Research and Technology Center North America, Palo Alto, CA, USA, Summer of 2014
- *Graph Analytics/Graph Mining Intern,* CareerBuilder LLC, Norcross, GA, USA, Summer of 2015

Related Publications:

1 **Mahmudur Rahman**, Mohammad Al Hasan. Link prediction in dynamic networks using graphlet. In Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD, Proceedings, Part I, pages 394-409, 2016.

2 **Mahmudur Rahman**, Mansurul Alam Bhuiyan, Mohammad Al Hasan: GRAFT: An Efficient Graphlet Counting Method for Large Graph Analysis. IEEE Trans. Knowl. Data Eng. 26(10): 2466-2478 (2014)

3 **Mahmudur Rahman**, Mansurul Bhuiyan, Mohammad Al Hasan: GRAFT: an approximate graphlet counting algorithm for large graph analysis. CIKM 2012: 1467-1471

4 **Mahmudur Rahman**, Mansurul Alam Bhuiyan, Mahmuda Rahman, Mohammad Al Hasan: GUISE: a uniform sampler for constructing frequency histogram of graphlets. Knowl. Inf. Syst. 38(3): 511-536 (2014)

5 Mansurul Bhuiyan, **Mahmudur Rahman**, Mahmuda Rahman, Mohammad Al Hasan: GUISE: Uniform Sampling of Graphlets for Large Graph Analysis. ICDM 2012: 91-100

6 **Mahmudur Rahman**, Mohammad Al Hasan. Sampling triples from restricted networks using MCMC strategy. In Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014: 1519-1528.

7 **Mahmudur Rahman**, Mohammad Al Hasan. Approximate triangle counting algorithms on Multi-cores. IEEE Big Data 2013. Workshop on Scalable Machine Learning.

8 **Mahmudur Rahman**, Mohammad Al Hasan. Graphlet counting in distributed computational system. (In Preparation)