SOLVING TIME PREDICTION PROBLEMS IN NETWORKS USING GRAPHLETS AND EMBEDDING BASED LOCAL FEATURES

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Vachik S. Dave

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2019

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Mohammad Al Hasan, Co-Chair

Department of Computer Science, Purdue University Indianapolis

Dr. David Gleich, Co-Chair

Department of Computer Science, Purdue University West Lafayette

Dr. Christopher W. Clifton

Department of Computer Science, Purdue University West Lafayette

Dr. Snehasis Mukhopadhyay

Department of Computer Science, Purdue University Indianapolis

Approved by:

Dr. Voicu Popescu

Head of the Graduate Program

To my parents for their unconditional love, and patience.

ACKNOWLEDGMENTS

I would like to take this opportunity to thank my Ph.D. advisor Professor Mohammad Al Hasan for his continuous support during my doctoral study. He introduced me to advanced research in information retrieval, graph analysis, and machine learning. He guided me to develop critical and independent thinking for any research problem. He taught me, how to design experiments to validate the research ideas, and how to write high-quality papers to present research contributions. He always gave me the freedom to investigate various research problems of my interest and skill-set. I also learned a lot by taking his graduate level data mining and algorithm courses, which were beneficial to my academic research, as well as, to my professional career.

Next, I would like to pay gratitude to all professors who served as my thesis committee members, namely Professor David Gleich, Professor Christopher Clifton and Professor Snehasis Mukhopadhyay for their guidance, encouragement, and suggestions to improve my thesis work. Additionally, I like to thank Professor Chandan K. Reddy of Virginia tech. for his valuable guidance during my learning of Survival Analysis models and its application in the graph analysis domain. I also like to thank Dr. Nesreen K. Ahmed (Intel Labs) for her important feedback and guidance in the graphlet counting project of my thesis.

Lastly, I am grateful to my parents and my sister for their long-term support and love. Their love and companionship have been my greatest motivation to complete my Ph.D. thesis. I would like to thank them for patience and continuous encouragement during these stressful times. Also, I would like to thank my friends and lab-mates for their constant support during my Ph.D. journey.

TABLE OF CONTENTS

Page

Page

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABBREVIATIONS

RLTP	Reciprocal Link Time Prediction
TCTP	Triangle Completion Time Prediction
FFNN	Feed Forward Neural Network
SVR	Support Vector Regression
BFS	Breadth First Search
SVM	Support Vector Machine
LDA	Latent Dirichlet Allocation
GLM	Generalized Linear Model
SCC	Strongly Connected Component
DAG	Directed Acyclic Graph
AFT model	Accelerated Failure Time model
MLE	Maximum Likelihood Estimation
AUC	Area Under Curve
TD-AUC	Time-Dependent AUC
PR-AUC	Precision-Recall AUC
LGFD	Local Graphlet Frequency Distribution
HR	Hit Rate
NDCG	Normalized Discounted Cumulative Gain
BPR	Bayesian Personalized Ranking
NNMF	Non-Negative Matrix Factorization
NMI	Normalized Mutual Information
NGF	Normalized Graphlet Frequency
MAE	Mean Absolute Error

ABSTRACT

Vachik S. Dave Ph.D., Purdue University, August 2019. Solving Time Prediction Problems in Networks using Graphlets and Embedding based Local Features. Major Professor: Mohammad Al Hasan.

Real-world networks are inherently dynamic; vertices and edges of these networks appear or disappear in a systematic pattern over time. Hence, the exact time at which the network elements, such as, the vertices or the edges appear or disappear is important information for modeling such networks. Additionally, as we predict a future event (say, link generation) on the network, it is also important to predict the exact time of that future event, because the availability of the event time makes the event prediction more valuable in terms of real-life utility of that prediction. Unfortunately, existing works on dynamic networks do not consider the time value; neither do they use the time information for modeling the network nor do they predict the time of a future event.

In this thesis, I have solved the event time prediction variant of multiple well-known problems in social networks. For instance, link prediction is a well known and possibly the most-studied problem in network analysis. But, the existing solutions to this task only predict whether a future link will appear or not—on some occasions, with a probability value. Unlike the existing works, in my thesis, I have proposed machine learning solutions that answer the question, "when will a link appear?", instead of answering whether a link will appear. I have also developed methods to use the time value of a link for the network modeling task, specifically, for learning features of a network element (a vertex, or an edge), which can subsequently be used for predicting the time value of a future event in the network. As I solve the time prediction problem in the network, I target to predict the link creation time in

both directed and undirected networks. To put it succinctly, this thesis opens up a new dimension in the network analysis task, where the time of the event has been considered explicitly both in the modeling and also in the prediction. The specific time prediction problems that I have designed are discussed below.

The first problem is Reciprocal Link Time Prediction (*RLTP*) problem, which is designed to predict the creation times of reciprocal links. *RLTP* is a versatile tool which can be applied to many real-world applications. For example, *RLTP* can be used to predict the elapsed time between receiving an email and sending its reply, or it can be used to determine the follow-back time or friend request acceptance time in online social networks. The second problem is Triangle Completion Time Prediction (*TCTP*) problem, which is designed to predict the creation time of a link that completes one or more triangles. A triangle is a prominent and basic building block of social networks and it is shown by researchers that the majority of new links created in social networks complete a triangle(s) in the network. Hence, a good solution of this problem can effectively improve the performance of various network analysis problems such as the link prediction problem, network expansion study, network generation models, community structure generation. Also, a solution of this problem provides an ordering of the future links based on their creation time, which is very useful to rank the user recommendations in different domains. Lastly, time prediction is the main theme of my research, but the machine learning solutions to such prediction problems require effective and efficient feature generation schemes, which can scale to large, and complex networks. So, some of my published works, which are also part of this thesis, focused on building effective features for network time prediction.

1. INTRODUCTION

In network analysis domain, recently dynamic networks are becoming increasingly popular due to the fact that many real-world networks are dynamic in nature, for example, email networks, road networks, biological networks, and social networks such as Twitter, Facebook, LinkedIn, etc. Hence, researchers started studying the growing nature of these dynamic networks [1–4], and more recently they proposed various models for network evolution [5–7]. Note that, the growth of these networks is mainly contributed by creation of new nodes and edges over a period of time. Hence, the creation time, time-stamp of a node/edge creation event, is a crucial component of the dynamic networks. However, the predominant network analysis problems avoid the creation time from their prediction and modeling tasks [8,9]. For instance, one of the prominent problems, the link prediction problem [10], aims to predict whether a link will be created in the future or not [9,11], but it completely ignores the prediction of the creation times of future links.

The creation times of the future links contain valuable insights of a dynamic network that can significantly improve various real-world applications of network analysis problems. The creation times provide comprehensive information on the future links and hence substantially enhance the performance of many applications of the link prediction problem. For instance, when an online social network platform wants to recommend a friend, it is much better for them if they can recommend a friend who is likely to accept the friend request (a link creation event) in a day (the creation time) than recommending a user who may accept the friend request after a week. Likewise, if we are looking for an endorsement from a friend on our LinkedIn profile (another link creation event), we would not only prefer to ask a person who will provide the endorsement, but also would prefer that the person provides the endorsement immediately (the creation time). The utility of the creation

time prediction goes beyond better link prediction for social media platforms, rather considering the creation time in social network prediction opens a completely new dimension for network evolution study and helps researchers to design more realistic network growth models. For example, closing an open triple is a phenomenon used by several network models [12–14], but knowing the time, *when an open triangle will close?*, can help us to obtain a dynamic network model which can generate a large number of snapshots of a network at a continuous temporal axis. Hence, the time prediction problem, which predicts the creation times of the future links, helps to improve the performance of many real-world applications and is also useful in different network analysis studies. But unfortunately, the time prediction problem has never been studied by researchers for either directed or undirected networks, which is the main focus of this thesis.

The time prediction problem is a highly challenging problem, mainly because, it is a regression problem and so inherently it is a more difficult task than corresponding classification problems. For instance, the popular link prediction problem is a binary classification problem [10], where the task is to find a decision boundary that discriminates node-pairs based on whether a link will appear between a node-pair in the future or not. On the other hand, to solve the time prediction problem, we need to model the creation times of the past links such that the prediction model can produce creation times of all future links. Another challenge in solving the time prediction problem is the unavailability of adequate features. As creation time is ignored from the objective of major network analysis problems, researchers also neglected the temporal aspect of dynamic networks while designing network features. Hence, none of the traditional network features capture the creation time patterns [9]. Therefore, designing and generating effective features to solve the time prediction problem is an important part of this thesis.

Generally, the link creation event in a real-world network is not completely arbitrary, but follows a fixed set of patterns and builds specific structures [15–17]. These link creation patterns have been studied by social scientists and to explain the most

frequent creation patterns they proposed different social theories such as reciprocal altruism [18], directed altruism [19] and social balance theory [20]. In [18], R. Trivers explains how the reciprocal altruism drives users to create more reciprocal links in a directed network. This reciprocal altruism phenomenon suggests that the creation of reciprocal links is more likely in the future, but it does not explain the patterns of the creation times of the reciprocal links. To fill this gap, I design, study and solve the reciprocal link time prediction (*RLTP*) problem in a directed network, which asks the question "when will a reciprocal link appear?". Similarly, T. Antal et al. [20] and other researchers [21,22] observe prevalence of the social balance theory in real-world networks; the theory states that an open triple is an imbalanced structure and it tends to convert into a triangle, which is a balanced structure. Based on this theory, researchers have designed and solved a triad closure problem, which predicts whether an open triple will be closed in the future or not [13,22,23]. However, conveniently, all these works also ignore the creation time of the triangle i.e. creation time of the third closing link. To predict the creation time of the third link of a triangle, I design the triangle completion time prediction (*TCTP*) problem in an undirected network. In my dissertation research, I study both *RLTP* and *TCTP* problems and provide novel methods with effective features to solve both (*RLTP* and *TCTP*) time prediction problems.

1.1 Contributions

In this thesis, I provide novel and efficient solutions for time prediction problems in both directed and undirected networks using meaningful and useful features designing. In directed networks, I solve the reciprocal link time prediction (*RLTP*) problem, For that, first I calculate useful topological features based on directed altruism [19] and social stratification [24] theories. During these calculations, I find that existing methods for calculating the shortest distance feature are highly inefficient [25, 26]. Therefore, I develop an efficient and novel indexing method named

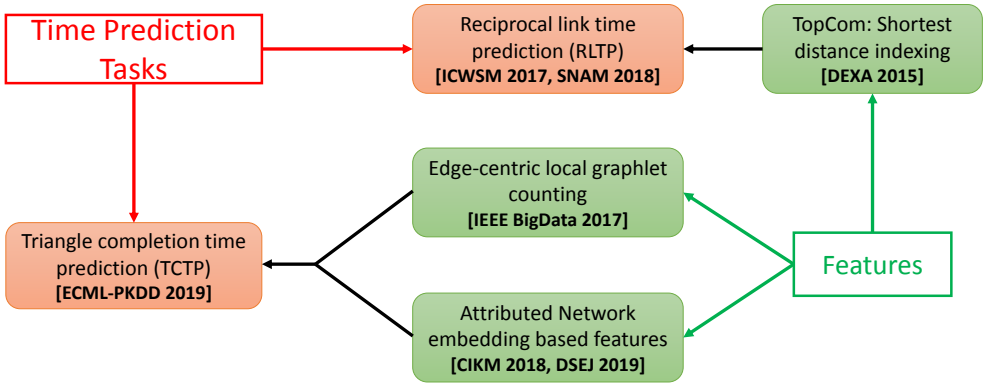


Fig. 1.1.: Major contributions of the thesis.

TopCom, to quickly answer the shortest distance query in a directed network. With these topological features and survival analysis methods, I am able to solve the *RLTP* problem accurately. While solving the *RLTP* problem I observe that, similar to any other difficult task, for time prediction problem finding a suitable set of features is extremely important and traditional topological features cannot capture comprehensive information local to a given node-pair (edge). Hence, I design edge features using graphlet frequencies local to an edge and provide a highly efficient algorithm to count these local graphlets. After that, I propose a novel representation learning base method to incorporate this graphlet based information in network embedding. Lastly, I design an efficient framework to solve the triangle completion time prediction (*TCTP*) problem in an undirected network. In the following subsections, I provide a brief discussion on each of the major contributions of this thesis mentioned in Figure 1.1.

1.1.1 Time prediction in a directed network

The majority of directed social networks, such as Twitter, Flickr, and Google+ follow a social psychology phenomenon called *reciprocal altruism* during edge creation. This reciprocal altruism drives a source vertex to create a reciprocal link with a target

vertex which has created a directed link towards the source in an earlier time. Based on this phenomenon, scientists have designed and solved a problem of predicting the possibility of creation of reciprocal links—a task known as "reciprocal link prediction".

Reciprocal link prediction is a binary classification task; as we discussed before, predicting the creation times of the reciprocal links is a more important problem. Additionally, this problem is interesting and useful in the real world; for example, after sending an email, a user is always interested to know the reply time; after endorsement on LinkedIn, a user is interested to know when the endorsement receiver will reciprocate the behavior; after following a person on Twitter a user would like to know when the other person will follow back; and so on. To answer all these questions, I design and solve a novel problem named *Reciprocal Link Time Prediction (RLTP)* [27].

Surprisingly, no existing work has considered solving the *RLTP* problem, partly because it is an extremely challenging problem. There are two major challenges in solving this problem: First, the presence of *ever-waiting* links in the graph (i.e., parasocial links for which a reciprocal link is not formed within the observation period), which makes the traditional supervised regression methods unsuitable for such graph data; Second, there is a lack of effective features, since well-known link prediction features are designed for undirected networks and for binary classification tasks, hence they do not work well for the time prediction task.

To solve the *RLTP* problem, we need to understand the behavior of the creation times of reciprocal links. For that, I study three real-world datasets and design topological features inspired from two social psychology phenomena: 1) Directed Altruism and 2) Social Stratification. I thoroughly study the relation of these features with reciprocal link creation time and show the usefulness of these features [28]. After that, I discuss why solving this problem using traditional regression models is not efficient. Instead, I map this problem into a survival analysis framework and solve it using different survival analysis models. Through experiments on real-world datasets, I prove that survival models consistently perform better than many known regression

models such as Ridge/Lasso regression, Feed-Forward Neural Networks (FFNN) and Support Vector Regression (SVR).

1.1.2 Time prediction in an undirected network

It is a known fact that the prevalence of triangles in social networks is much higher than their prevalence in a random network. It is caused predominantly by the social phenomenon that friends of friends are typically friends themselves. A large number of triangles in social networks is also due to the "small-world network" property [15], which suggests that in an evolving social network, new links are formed between nodes that have short distance between themselves. Leskovec et al. [29] have found that depending on the kinds of networks, 30 to 60 percent of new links in a network are between vertices that are only two-hops apart, i.e., each of these links is the third edge of a new triangle in the network. High prevalence of triangles is also observed in directed networks, such as, trust networks, and follow-follower networks—social balance theory [20] can be attributed for such observations.

The knowledge of triangle completion time is practically useful. For instance, given that the majority of new links in a network complete a triangle, the knowledge—whether a link will complete a triangle in a short time—can be used to improve the performance of a link prediction model [9]. Specifically, by utilizing this knowledge, a link prediction model can assign a different prior probability of link formation when such links would complete a triangle in the near future. Besides, link creation time is more informative than a value denoting the chance of link formation. Say, an online social network platform wants to recommend a friend; it is much better for the platform if it recommends a member who is likely to accept the friend request in a day or two than recommending another who may accept the friend request after a week or few weeks. In the e-commerce domain, a common product recommendation criterion is recommending an associated item (say, $item_2$) of an item (say, $item_1$) that a user u has already purchased. Considering a user-item network, in which $item_1 - item_2$

is a triangle's first edge, $item_1 - u$ is the triangle's second edge, the $TCTP$ task can be used to determine the best time interval for recommending the user u the $item_2$, whose purchase will complete the $u - item_1 - item_2$ triangle. Given the high prevalence of triangle in real-life networks, the knowledge of triangle completion time can also improve the solution of various other network tasks that use triangles, such as, community structure generation [14], designing network generation models [29], and generating link recommendation [30].

To solve the $TCTP$ problem, I provide an effective framework that uses two representation learning based network embedding models. The main objective of the embedding approaches is to embed edges with similar triangle completion time in close proximity in the latent space, which is a completely different objective than the majority of existing network embedding models [31–33], which captures the structural similarity and proximity in the representation vector for a node. This model also incorporates graphlet based structural information into the embedding vectors. I show using thorough experiments that the proposed framework is an accurate and effective approach to solve the $TCTP$ problem.

1.1.3 Edge feature design using local graphlet frequency

In recent years, graphlet counting has emerged as an important task in topological network analysis because of its ability to capture key graphical patterns of a network. It is known that global graphlet frequency is a very useful feature for graph classification in a variety of domains [34–36], also it is used for network modeling [37] and other network analysis studies. However, all these applications use global graphlet frequencies to get features of a whole network but global graphlets cannot provide features for a node or an edge.

On the other hand, the graphlets counted locally for a given edge (node-pair) can be used as edge features, which can also be used in many network analysis tasks such as link prediction, edge classification, etc. Though there are plenty of graphlet

counting works in the literature [34, 38, 39], very few works obtain local graphlet frequencies for a given node or edge. Especially, there is not a single work that provides graphlet frequencies up to size-5 for a given edge. Therefore, I develop a novel scalable method called *E-CLoG* (Edge Centric Local Graphlet) [40], which calculates local graphlet counts for a given edge (node-pair). This method can provide counts of local graphlets up to size-5.

E-CLoG is highly efficient in counting local graphlets and these graphlet frequencies are effective edge features for solving the link prediction problem. For comparison, there exists very few recent works that can handle size-5 graphlet counting, but all the existing works are for global graphlet counting. Out of which, I found one recent method (GRAFT) which counts graphlets edge-wise i.e. GRAFT counts graphlets for each edge in the graph and later combines the counts to provide global graphlet counts. I compare the performance of *E-CLoG* with GRAFT and prove that our method is highly efficient and scalable compared to the state of art edge based counting work. I utilize the normalized graphlet frequency vector as edge features for link prediction task and show significant improvement in the link prediction performance compared to traditional topological features such as common neighbors, adamic-adar, preferential Attachment, Katz measure, etc. for real world datasets.

1.1.4 Attributed network embedding for features

In the past few years, we have witnessed a surge in research on embedding the vertices of a network into a low-dimensional, dense vector space. The embedded vector representation of the vertices in such a vector space enables effortless invocation of off-the-shelf machine learning algorithms, thereby facilitating several downstream network mining tasks, including node classification [41], link prediction [31], community detection [42], job recommendation [43], and entity disambiguation [44]. Most existing network embedding methods, including DeepWalk [32], LINE [33], Node2Vec [31], and SDNE [45], utilize the topological information of a network with the rationale

that nodes with similar topological roles should be distributed closely in the learned low-dimensional vector space. While this suffices for node embedding of a bare-bone network, it is inadequate for most of today's network datasets which include useful information beyond link connectivity. Specifically, for most of the social and communication networks, a rich set of nodal attributes is typically available, and more importantly, the similarity between a pair of nodes is dictated significantly by the similarity of their attribute values. Yet, the existing embedding models do not provide a principled approach for incorporating nodal attributes into network embedding and thus fail to achieve the performance boost that may be obtained through modeling attribute based nodal similarity. Intuitively, a joint network embedding that considers both attributional and relational information could entail complementary information and further enrich the learned vector representations.

We provide a few examples from real-life networks to highlight the importance of vertex attributes for understanding the role of the vertices and to predict their interactions. For example, users on social websites contain biographical profiles like age, gender, and textual comments, which dictate who they befriend with, and what are their common interests. In a citation network, each scientific paper is associated with a title, an abstract, and a publication venue, which largely dictates its future citation patterns. In fact, nodal attributes are specifically important when the network topology fails to capture the similarity between a pair of nodes. For example, in the academic domain, two researchers who write scientific papers related to "machine learning" and "information retrieval" are not considered to be similar by existing embedding methods (say, DeepWalk or LINE) unless they are co-authors or they share common collaborators. In such a scenario, node attributes of the researchers (e.g., research keywords) are crucial for compensating for the lack of topological similarity between the researchers. In summary, by jointly considering the attribute homophily and the network topology, more informative node representations can be expected. Hence, we provide an effective and robust network embedding approach called *NeuralBrane* that incorporates attribute information into structural information.

1.1.5 Index for shortest distance query in a directed network

Finding the shortest distance between two nodes in a graph (*distance query*) is one of the most useful operations in graph analysis. Besides the application that stands for its literal meaning, i.e. finding the shortest distance between two places in a road network, this operation is useful in many other applications in social and information networks. For instance, in social networks, the shortest path distance is used in the calculation of different centrality metrics, including closeness centrality and betweenness centrality [46, 47]. It is also used as a criterion for finding highly influential nodes [48], and for detecting communities in a network [1]. Scientists have also used shortest path distance to generate features for predicting future links in a network [9]. In information networks, shortest path distance is used for keyword search [49], and also for relevance ranking [50].

Due to the importance of the shortest path distance problem, researchers have been studying this problem from the ancient time, and several classical algorithms (Dijkstra, Bellman-Ford, Floyd-Warshall) exist for this problem, which run in polynomial time over the number of vertices and the number of edges of the network. However, as real-life graphs grow in the order of thousands or millions of vertices, classical algorithms deem inefficient for providing real-time answers for a large number of distance queries on such graphs. For example, for a graph of a few thousand vertices, a contemporary desktop computer takes an order of seconds to answer a single query, so thousands of queries take tens of minutes, which is not acceptable for many real-time applications. Hence, there is a growing interest for the discovery of more efficient methods for solving this task. We propose *TopCom*, an indexing based method for obtaining an exact solution of a distance query in an arbitrary directed graph.

1.2 Organization of the Thesis

This thesis includes several research articles that I published over a period of six years of my doctoral study. The time prediction problem in a directed network named reciprocal time prediction (*RLTP*) problem is published in two steps, first the primary study is published in AAAI International Conference on Web and Social Mining (ICWSM), 2017 and the elaborated study with prediction methods is published in Springer journal Social Network Analysis and Mining (SNAM), 2018. This research is presented in Chapter 3. In Chapter 4, I discuss the triangle completion time prediction (*TCTP*) problem and explain a novel framework to solve the *TCTP* problem. This work is currently under review at an international conference.

In the remaining chapters, I discuss efficient methods for generating features that help to solve the time prediction problems accurately. To generate graphlet based edge features, I have developed a scalable and efficient algorithm called *E-CLoG*, which calculates edge-centric graphlet frequencies. This *E-CLoG* algorithm is thoroughly explained in Chapter 5 and it is published in IEEE International Conference on Big Data (IEEE BigData), 2017. I have also designed representation learning based features using two novel attributed network embedding methods. The first method assumes that the node attributes are inter-related and it learns representation vectors for both nodes and attributes into the same latent space. This representation learning method is published in ACM International Conference on Information and Knowledge Discovery (CIKM), 2018 and it is discussed in Chapter 6. The second method is described in Chapter 7, where the network embedding approach assumes that the attributes are sparse and independent. The article, which discusses the second embedding method, is accepted for publication in SpringerOpen Journal Data Science and Engineering (DSE), 2019. I have also developed an indexing method to efficiently answer the shortest distance query in a directed network. This indexing method is discussed in Chapter 8, which is published in the International Conference on Database and Expert Systems Applications (DEXA), 2015. Lastly, in Chapter 9, I

briefly conclude the thesis and discuss future directions for solving the time prediction problems.

1.3 Preliminaries

In this section, I provide background information on the topics related to this thesis.

1.3.1 Notations.

Throughout this thesis, scalars are denoted by lowercase letters (e.g., n). Vectors are represented by boldface lowercase letters (e.g., $\mathbf{x}$). Bold uppercase letters (e.g., $\mathbf{X}$) denote matrices, the i^{th} row of a matrix $\mathbf{X}$ is denoted as $\mathbf{x}_i$ and j^{th} element of the vector $\mathbf{x}_i$ is represented as x_i^j. The transpose of the vector $\mathbf{x}$ is denoted by $\mathbf{x}^T$. The dot product of two vectors is denoted by $\langle \mathbf{a}, \mathbf{b} \rangle$. $\|\mathbf{X}\|_F$ is the Frobenius norm of matrix $\mathbf{X}$. Calligraphic uppercase letter (e.g., $\mathcal{X}$) is used to denote a set and $|\mathcal{X}|$ is used to denote the cardinality of the set $\mathcal{X}$.

1.3.2 Graphlets

In the simplest language, a graphlet is a set of a small number of connected nodes that creates specific structures. For example, there are two types of graphlets with 3 nodes (size-3 graphlets) i.e. an open triple and a closed triangle. Similarly, there are a total of 6 different graphlets of size-4 as shown in Figure 1.2. The occurrences or frequencies of these graphlets in a network, are good representatives of the overall structural properties of the network [34]. While calculating frequency, we count only induced graphlets, where an induced graphlet of a network is a subgraph of the network such that the

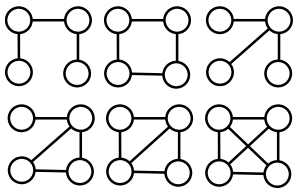


Fig. 1.2.: size-4 graphlets

selected subgraph includes all edges from the original network between nodes of the subgraph. Formally,

Definition 1.3.1 (Induced graphlet) *For a given undirected network $G(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of edges; an undirected subgraph $g(\mathcal{V}', \mathcal{E}')$ is an induced graphlet of $G(\mathcal{V}, \mathcal{E})$, if $\mathcal{V}' \subseteq \mathcal{V}$ and $\mathcal{E}'$ consists of all of the edges in $\mathcal{E}$ that have both endpoints in $\mathcal{V}'$.* ■

If $|\mathcal{V}'| = k$, we call it a k-size induced graphlet, or in-short, a k-graphlet in G. The frequencies of these induced graphlets are used as graph features for a graph classification problem [34]. Similarly, we can use graphlet frequencies as node features if we count it for node-centric local graphlets, where the targeted node is always a part of the graphlets. For example, in [51], the authors use frequencies of node-centric local graphlets to solve the node classification task. Similarly, we can count the frequencies of graphlets local to an edge (edge-centric local graphlets) to get edge features and use them to solve the link prediction problem [40]. The edge-centric local graphlet is an induced graphlet that includes the targeted edge. Below, I formally define the edge-centric local graphlets:

Definition 1.3.2 (Edge centric local graphlet) *Say, $g(\mathcal{V}', \mathcal{E}')$ is a k-graphlet of a graph $G(\mathcal{V}, \mathcal{E})$. Now, given an edge $(u, v) \in \mathcal{E}$, g is called an edge centric local graphlet, if $(u, v) \in \mathcal{E}'$ and $w \in \mathcal{V}' \setminus \{u, v\} \Rightarrow w \in \Gamma(u) \cup \Gamma(v)$.* ■

Where, $\Gamma(u)$ is a set of neighbors of node $u \in \mathcal{V}$. I use edge-centric local graphlet frequencies to solve the *TCTP* problem.

1.3.3 Network Embedding

A network embedding is a lower dimensional representation of network elements such that the representation vector preserves important information from the network such as topological proximity, structural and conceptual similarity, and community

structure. To find such network embeddings, researchers have proposed various representation learning approaches that capture topological proximity and network structural information [33, 45], where majority of these representation learning methods find a low dimensional vector for each node in a network [31, 41]. Formally,

Definition 1.3.3 (Representation learning Method) *For a given undirected network $G(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of edges, the objective is to learn a mapping function $f : u \rightarrow \mathbf{m}_u$, where $u \in \mathcal{V}$ and $\mathbf{m}_u \in \mathbb{R}^k$ is a k dimensional representation vector for node u. This learned mapping function f is called representation learning method and corresponding matrix of vectors $\mathbf{M} \in \mathbb{R}^{|\mathcal{V}| \times k}$ is called network embedding matrix.* ■

Note that, this learned embedding matrix $\mathbf{M}$ must preserve useful network information for various downstream tasks. Also, the dimensionality of the representation vector (k) should be far lesser than the number of nodes i.e. $k \ll |\mathcal{V}|$. These lower dimensional vectors for each node can be used as informative node features and proved to be very useful for various network analysis tasks such as node classification [41], link prediction [31], and community detection [42].

To learn the suitable representation vectors for each node, we can use traditional statistical methods such as principal component analysis (PCA), where an adjacency matrix $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$ can be used to find an informative lower dimensional ($|\mathcal{V}| \times k$) matrix. However, many recent network embedding approaches learn the latent representation for each node of a network using more specialized methods such as random walk based method [32], matrix factorization based methods [52, 53] or deep neural network based methods [45, 54].

There are mainly two broad categories of embedding approaches, 1) transductive methods and 2) inductive methods. The majority of embedding approaches assume that the node set $\mathcal{V}$ is fixed and hence their embedding methods cannot produce a representation vector for a new node $z \notin \mathcal{V}$, these methods are called transductive methods [31, 33, 55, 56]. On the other hand, there are few recent methods that use in-

ductive learning approaches which can produce representation vectors for an unknown new "out-of-sample" nodes [57–59].

1.3.4 Survival analysis models

Survival analysis is widely used in the medical domain to predict survival time or time to a specific event (such as death) for patient datasets [60], [61]. In the survival analysis setup, for a set of instances under observation, events happen over a time period, from which a survival model learns the temporal patterns of these events and predicts the survival time. Broadly, all types of survival analysis models try to predict the survival time of an instance in the data by modeling three functions: 1) Survival function, 2) Hazard function and 3) Event density function. Definitions and relationship between these three functions are described below:

Survival function $S(t)$: Survival models provides a principled approach for interval time prediction by modeling a survival function, which is defined as the probability value that the reciprocal edge creation does not happen for a given parasocial link before a specified time t.

$$S(t) = Pr(\boldsymbol{T} \geq t)$$

Here, $\boldsymbol{T}$ is a random variable representing the time of the reciprocal edge creation event.

Hazard function $\lambda(t)$: It is the reciprocal event rate at time t conditional on the fact that the reciprocal event has not occurred until that time t,

$$\lambda(t) = \frac{f(t)}{S(t)} \tag{1.1}$$

where $f(t)$ is the *reciprocal event density function*, which is given as follows:

$$f(t) = \frac{d}{dt}(1 - S(t)) = -\frac{d}{dt}S(t)$$

We can observe that both the *survival function* and the *hazard function* are interrelated and we can model either function for the interval time prediction. There are two types of widely used survival models: 1) semi-parametric models and 2) parametric models.

Semi-parametric Cox regression model

Cox regression model [62] is the most widely used semi-parametric model for predicting the (interval) time taken for a reciprocal event to occur. The basic Cox model follows the proportional hazard assumption, for which the hazard function $\lambda(t \mid \mathbf{x}_i)$ takes the following form:

$$\begin{aligned} \lambda(t \mid \mathbf{x}_i) &= \lambda_0(t) \times \exp(\beta_1 x_{i1} + \beta_2 x_{i2} + ... + \beta_d x_{id}) \\ &= \lambda_0(t) \times \exp(\mathbf{x}_i^T \boldsymbol{\beta}) \end{aligned} \tag{1.2}$$

where, $\mathbf{x}_i$ is the (topological) feature vector of a parasocial link represented as i^{th} data instance in the training data and d is the dimensionality of the features. Here, $\lambda_0(t)$ is called baseline hazard function, and $\boldsymbol{\beta}$ is the model parameter which Cox regression model learns. The Cox regression is called semi-parametric because the baseline hazard function $\lambda_0(t)$ can be any non-negative function of time. The probability of occurrence of reciprocal event for the i^{th} parasocial link (data instance) at time t can be represented as ratio $\frac{\lambda(t|\mathbf{x}_i)}{\sum_{j \in \mathcal{R}_t} \lambda(t|\mathbf{x}_j)}$, where $\mathcal{R}_t$ is the set of all instances for which the

reciprocal event did not happen until t. The product of these probabilities gives the partial likelihood function:

$$L(\boldsymbol{\beta}) = \prod_{i=1}^{n_p} \left[\frac{\exp(\mathbf{x}_i^T \boldsymbol{\beta})}{\sum_{j \in R_t} \exp(\mathbf{x}_j^T \boldsymbol{\beta})} \right]^{C_i} \tag{1.3}$$

Here, n_p is the total number of parasocial links that appeared during the training period and C_i is an event indicator value, i.e., if the reciprocal link for the i^{th} parasocial link appears during the training period then $C_i = 1$ otherwise $C_i = 0$. The model parameter $\boldsymbol{\beta}$ is learnt by minimizing the negative log likelihood function. If $\widehat{\boldsymbol{\beta}}$ is the optimal model parameter, we have:

$$\widehat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \quad \frac{1}{n_p} \sum_{i=1}^{n_p} \left[-C_i(\mathbf{x}_i^T \boldsymbol{\beta}) + C_i \log \left(\sum_{j \in R_t} \exp(\mathbf{x}_j^T \boldsymbol{\beta}) \right) \right] \tag{1.4}$$

To find this $\widehat{\boldsymbol{\beta}}$, Cox proposed a partial likelihood based optimization method. [62].

Parametric Models

The main idea behind a parametric model is that it assumes that the interval time follows a specific statistical distribution. There are two ways to relate interval time and a statistical distribution: first, assume that the actual interval time for all parasocial links follows a distribution; and second, assume that the logarithm of the interval time follows a distribution. The models under the first assumption are referred to as linear regression models, and the models under later assumption are called accelerated failure time (AFT) models.

Generally, parametric models use a maximum likelihood estimation (MLE) approach to learn model parameters. Let's assume that all the parameters of a model are represented by $\boldsymbol{\beta} = (\beta_1, \beta_2, ...)^{\mathcal{T}}$. For a given parasocial link (say i^{th} link in the training data), if it is an *ever-waiting* link, then the corresponding survival function

$S(t, \boldsymbol{\beta})$ at time t (in fact, any t value during a training period) should be near 1, and if it is not an *ever-waiting* link then the reciprocal event density function $f(t_i, \boldsymbol{\beta})$ at time t_i (time of reciprocal event for the i^{th} parasocial link) should be high (near to 1) for that link. Hence, the likelihood of all parasocial links of a training period is the product of their reciprocal event density functions or survival functions based on their state (whether the link is *ever-waiting* or not), i.e.

$$L(\boldsymbol{\beta}) = \prod_{C_i=1} f(t_i, \boldsymbol{\beta}) \cdot \prod_{C_i=0} S(t_i, \boldsymbol{\beta}) \tag{1.5}$$

Linear regression model: The statistical linear regression with least squares estimation is widely used for a variety of regression tasks. However, the issue with this model is that it cannot use information from *ever-waiting* links. For interval time prediction this issue can be handled by using a specific survival model such as the Buckley-James model (BJ model). The BJ model first estimates the interval time of training *ever-waiting* links using the Kaplan-Meier (KM) [63] estimation method and then by using all parasocial links from training period to train a linear model. This linear model can be trained using the MLE method as described above.

Accelerated Failure Time (AFT) model: An AFT model assumes that the logarithm of the interval times $\log(\mathcal{T}_{Int})$ follows a statistical distribution and it is linearly related to the (topological) feature vectors. The general form for AFT regression model is

$$log(\mathcal{T}_{Int}) = \boldsymbol{X} \cdot \boldsymbol{\beta} + \sigma \cdot \epsilon \tag{1.6}$$

where $\boldsymbol{X}$ is the covariate matrix of size $n_p \times d$ where i^{th} row of $\boldsymbol{X}$ is $\mathbf{x}_i$, $\boldsymbol{\beta}$ is a d dimensional coefficient vector (model parameters), σ ($\sigma > 0$) is an unknown scale parameter, and ϵ is an error variable which follows a similar distribution to $\log(\mathcal{T}_{Int})$.

2. RELATED WORKS

In network analysis, there are many prediction problems on nodes and edges such as node classification [64], edge classification [65], community detection [66] and link prediction [67]. Researchers have proposed various solutions for these problems such as network propagation based methods [68,69], topological features based methods [9, 70], matrix factorization based methods [71, 72] and more recently embedding based methods [32, 73].

Though none of the above works solve time prediction problems, there are a few research articles, which are directly or indirectly related to the time prediction problems discussed in this thesis. Hence, in this chapter, first, I discuss the related works for both time prediction problems i.e. Reciprocal link time prediction ($RLTP$) and triangle completion time prediction ($TCTP$). Also, I observe that adequate feature designing is an internal part of accurate prediction approach, therefore, in this thesis, I design effective features and provide efficient methods to build these features. So next in this chapter, I discuss previous related works for various feature generation methods. For that first, I review related works on *local graphlet counting methods*; next, I discuss previous *network embedding approaches* and finally, I review existing *indexing methods for shortest distance query* in a directed network.

2.1 Reciprocal link time prediction

The traditional binary classification task of link prediction has received enormous attention over the years since the inception of this problem by Liben-Nowell and Kleinberg in 2003 [10]. Over the years researchers have solved the link prediction problem for a variety of graphs - for example link prediction in homogeneous networks [67, 74, 75], link prediction in heterogeneous information networks [30, 76], and link

prediction for knowledge graphs [77, 78]. Other related problems, such as link/sign prediction and ranking in signed social network [79,80], and a recommendation system using link prediction techniques [81] have also been studied.

Reciprocal link prediction is a variant of link prediction which works on directed networks. Even though the majority of social and communication graphs are directed, only a few works exist which consider predicting reciprocal links. In one of the earliest works, J. Hopcroft et al. [24] predicted reciprocal edges in a Twitter network. However, many of the features that they proposed are too specific to the Twitter dataset and do not apply to a generic directed network. N. Gong et al. [82] compared reciprocal and parasocial link creation in Google+ and Flickr datasets and solved the reciprocal link prediction problem as an outlier detection task using one-class SVM. Authors of [83] compared structural differences of reciprocal links and parasocial links and they also studied a Twitter dataset and corresponding node features to predict reciprocal links. In another work [84], the authors reported that the majority of reciprocating links are created within a very short time after the creation of corresponding parasocial links. B. Dumba et al. [4] studied the structural properties of a reciprocal network and discussed user behavior patterns.

A closely related problem to reciprocal link prediction is online dating recommendation. There exist a few works that solve this problem, mainly by using traditional recommendation methods with novel feature extraction processes. For example, in [85] the authors modified the classical collaborative filtering method for the dating recommendation task. P. Xia et al. [86, 87] proposed different reciprocal score matrices and used them with collaborative filtering for a recommendation. The authors in [88] proposed an LDA (Latent Dirichlet Allocation) based approach to learn latent preferences of users with two side matching based recommendations. Recently, X. Zang at al. [89] proposed a method that extracts profile-based features, topological features, and preference features from a dating social network for a recommendation. All the existing works discussed so far target the binary classification problem, which predicts whether the reciprocal link will be created or not. On the other hand, in this

thesis, I target to predict the creation time of the reciprocal links, which is a more difficult problem than the binary classification problem.

2.2 Predicting triangle completion time

There exist many works which study triangle statistic and their distribution in social networks. The majority of these works are focused on triangle counting in a network [90–93]. While a few other works investigate how different network models perform in generation synthetic networks whose clustering coefficient matches with real-life social networks [94]. However, none of these works target to predict the triangle completing edges. There are only a few works that study the prediction problem for the triangle completion, which I discuss below.

Romero and Kleinberg [95] initially studied the directed closure (feed-forward triangle completion in a directed network) process and its relation with preferential attachment. Another study is done by T. Lou et al. [22], where they investigate how the reciprocal links and triangle completing links are developed using social theories and also provide a solution based on a graphical model to predict these links in a directed network. After that, H. Huang et al. [13, 96, 97] thoroughly study a microblogging social network and observe various user demographics based and social theory based factors in the formation of triangles. They proposed an extended version of the graphical model proposed by T. Lou et al. to predict the triangle completion problem in online social networks. Note that, these works study triangle completion in directed networks, but there are few other works that study triangle completion in an undirected network. For example, M. Zignani et al. [98] provides metrics to understand dynamic properties of a social network by studying link delay and triangle completion delay. Lastly, Estrada and Arrigo [23] proposed a communicability distance based objective to predict the creation of triangle completing edges.

None of the above works study and solve the time prediction problem, which is one of the main objectives of this thesis. To the best of our knowledge, there are only

two works that target the time prediction problem; the first one is by Y. Sun et al. [99] and the second by M. Li et al. [100]. In both of these works, the authors have extracted unique features for a DBLP-like (author-paper) heterogeneous network. Y. Sun et al. proposed the meta path based topological features and used a generalized linear model (GLM) for the prediction task. Similarly, M. Li et al. proposed a novel time difference labeled path (TDLP) based method for the knowledge graph. Both methods are designed specifically for DBLP like networks, hence they are difficult to apply to other general networks. On the other hand, in this thesis, I study and solve time prediction problem in general directed and undirected networks without any assumptions on networks.

2.3 Local graphlet counting methods

There are various methods proposed to count graphlets in a given network, where initial works count only size 3 graphlets i.e. open and close triangles in the network [93]. Recently, there are multiple works for graphlet counting, which gives global graphlet counts for larger graphlets of size 4 and 5 [34, 39, 101, 102]. As the counting cost increases with larger networks , there are few works that give good estimation of the global graphlet counts [103–105].

However, all these works are giving global graphlet counts and there are very few works address local graphlet counting, for example, Y. Lim et al. [106] and L. De Stefani et al. [107] have proposed estimation method to count local triangles from streaming data. Similarly, C. Seshadhri et al. [108] proposed a sampling based method that estimates the local clustering coefficient. But, all these local graphlet counting works give counts local to a node, on the other hand our method is providing an edge attribute i.e. our method counts local graphlets for a given edge. Best of our knowledge, there is only one work that counts local graphlets for an edge by N. Ahmed et al. [109]. However, N. Ahmed et al. are counting up to 4 size graphlets and they do not count all graphlets obtained by different edge orbit. I believe, a given edge

in different orbits represents different characteristics of the edge, hence finding the count of graphlets with all possible edge orbits is very important. Hence, I provided an algorithm to count all $3, 4, 5$ size local graphlets including all possible different orbits of the given edge.

2.4 Neural network based network embedding

Recently, there are many works on representation learning on graphs (a.k.a. network embedding). For example, approaches such as DeepWalk [32] and Node2Vec [31] learn the node embeddings based on either uniform or biased random walks by extending the Skip-Gram language model. LINE [33] utilizes first and second order proximities and trains the embedding by negative sampling. GraRep [55] is a factorization based approach that leverages both local and global structural information. Furthermore, a few neural network based approaches are proposed for network embedding, such as SDNE [45], DNGR [73] and HNE [110]. The detailed network embedding survey is available in the article [111].

Most of the aforementioned works only investigate the topological structure for network embedding, which is in fact only a partial view of an attributed network. To bridge this gap, a few attributed network embedding based approaches [52–54, 112–115] are proposed. The general philosophy of such works is to integrate nodal features, such as text information, into a topology-oriented network embedding model to enhance the performance of downstream network mining tasks. For example, TADW [52] performs low-rank matrix factorization considering graph structure and text features. AANE [53] also uses a generalized matrix factorization approach to incorporate textual information associated with nodes. However, such matrix factorization based methods have a huge limitation as they are not scalable. EP-B [112] is an embedding propagation based approach, which is a modification of locally linear embedding (LLE) [116]. Furthermore, TriDNR [54] adopts a two-layer neural networks to jointly learn the network representations by leveraging inter-node, node-

word, and label-word relationships. Similarly, if we can create heterogeneous graphs using node and its attributes, we can use PTE [113] to merge attributes information in embedding. However, the training procedure of three (or more) components is difficult to weight and adjust; additionally, TriDNR uses DeepWalk to access textual information associated with node which can handle only a continuous ordered text (sentences) as node attributes. UPP-SNE [114] is another DeepWalk based approach with non-linear function for merging attributes into embedding. Notice that, the majority of these methods only consider sparse textual features associated with each node, and fail to handle the node attributes with rich types in general. The objective of our embedding approach is to design a general embedding method that can handle both sparse textual attributes and dense graphlet based attributes.

2.5 Index for shortest distance query in a directed network

Shortest distance on a directed graph has many interesting works. In this section I discuss the most important works among these under two categories: (1) Online shortest distance calculation, (2) Offline (Index based) shortest distance calculation. The first, online calculation mainly includes Breadth First Search (BFS), Dijkstra's algorithm and Bidirectional Dijkstra's algorithm. Here, I mainly discuss index based related works. Mainly because for large graphs, online methods are slower than an indexing based method, so most of the recent research efforts are concentrated towards indexing based methods. The literature for shortest distance indexing is quite vast, so, I review few of the works that have published in the recent years. For a detailed review, I refer the readers to read [117, 118]. Here, I discuss few of the recent and relevant indexing based works.

Many of the existing works for shortest distance computation is specifically designed for the **road networks** [119–126]. Such networks show hierarchical structures with the presence of junctions, hubs, and highways; the shortest distance computation methods for these networks exploit the hierarchical structure for compressing

distance matrix or for building distance indices [120,124]. Finding exact shortest distance in a large graph is a costly task, hence few researchers have proposed methods for computing **estimated shortest distance** [127–130]. The most common among the estimated shortest distance based methods is the landmark based method, which selects a set of landmark nodes based on some criteria and finds shortest paths that must go through those landmark nodes. The main task here is to decide the set of vertices that are optimal choice as landmarks. However, it has been shown that this optimization problem is $\mathcal{NP}$-Hard [130], so researchers adopt various heuristics based approaches for choosing those landmarks.

There are some other works for finding shortest distance in large graphs which are proposed very recently; examples include [25, 26, 131–138]. Many of these have unique ideas, so it is difficult to categorize them under a generic shortest path method. For example, Gao et al. [136] use a relational approach and propose an index called SegTable which stores local segments of a shortest distance. Zhu at al. [131] propose a method to answer single source shortest distance query for a huge graph on disk. Akiba et al. [135] propose a unique pruning method based on degree of a vertex, which can efficiently reduce the search space of BFS. Highway centric label (HCL) [132] is one of the fastest recent methods that is proposed for a shortest distance query on both directed and undirected graphs. In a follow-up work, Xiang proposes TreeMap [26], a tree decomposition based approach for solving distance query exactly; the author compares TreeMap's solution with those of HCL to show that the former has better performance. Another recent method is called IS-Label which is proposed by Fu et al. [25]. They have also shown that that IS-Label has superior performance than HCL. I compare proposed method *TopCom* with both IS-Label and TreeMap, which are among the best of the existing index based methods.

3. PREDICTING RECIPROCAL LINK CREATION TIME

3.1 Introduction

Reciprocity is a phenomenon in social psychology which mandates that people should repay voluntarily what another person has provided for them. It is different from *altruism* [139] in the way that reciprocity follows from others' initial action, while altruism is a spontaneous action of gift-giving without the hope or expectation of future positive responses. There also exists another social psychology, named *reciprocal altruism*, which is a behavior whereby one performs an act of gift-giving with the expectation that the receiving person will act in a similar manner at a later time [18]. People's day-to-day activities on online social networks are filled with many examples of reciprocal altruism: we follow a friend's Twitter feed with the hope that he will follow back our feed; we like a friend's Facebook posts or her Flicker images with the expectation that she will do the same; we endorse our friends for their technical skill in LinkedIn hoping that they will return the favor in a similar manner.

However reciprocity usually is in conflict with another social phenomenon called *social stratification*, which favors hierarchical arrangement of people in a society based on various factors such as power, wealth, and reputation [24]. This phenomenon is prevalent in online social networks as well, but in a different manner. Apparently, for such networks, the social hierarchy is reflected in various prestige metrics which rank vertices based on their topological bearings, such as pagerank, and in-degree. Given this hierarchical arrangement in an online social network, people who are higher up in the hierarchy are sometimes reluctant to perform a reciprocal act for an individual who is lower in the hierarchy; they instead defer the reciprocal action to a later time, or sometimes indefinitely.

For reciprocal link creation, understanding the criteria which control the interval time and building learning models which predict the interval time are important. From a research standpoint, such studies help scientists to understand the interaction between reciprocity and social stratification phenomena. From the perspective of real-life applications in social network analysis, such prediction models enable better link suggestions, where the interval time is also factored in within the suggestion. Reciprocity, along with the interval time for reciprocal link creation, is particularly important for recommendation in online dating systems [86].

The majority of existing works on link prediction assume an undirected network [9, 140], in which the concept of reciprocal edges does not exist. A few works consider reciprocal link prediction [24,82] in a directed network where the prediction is binary, yielding a yes/no answer to the question of whether a reciprocal link will be created within a fixed observation window. Several other works utilize reciprocity as a tool for network compression [141] and information propagation in social networks [142]. Reciprocal links also influence the degree correlations in complex networks, hence they play an important part in modeling the growth of directed social networks [143]. However, none of the existing works consider predicting the interval time for the creation of a reciprocal edge.

Extending a model which solves a binary class reciprocal link prediction problem to a model which predicts the interval time of reciprocal links is non-trivial. The major challenge for interval time prediction is that typical link prediction features for an undirected network, such as common neighbors, Jaccard's similarity, and Adamic-Adar do not have a well-defined counterpart for directed networks, which makes interval prediction a difficult task. Additionally, for generating the training data for building a prediction model, a network is observed for a finite time window, and the absence of a reciprocal link within that time window does not necessarily mean the absence of that reciprocal edge, because a reciprocal edge might have formed outside (after) the observation time window. This yields numerous right censored data instances, for which the target variable, i.e., the reciprocal link formation time

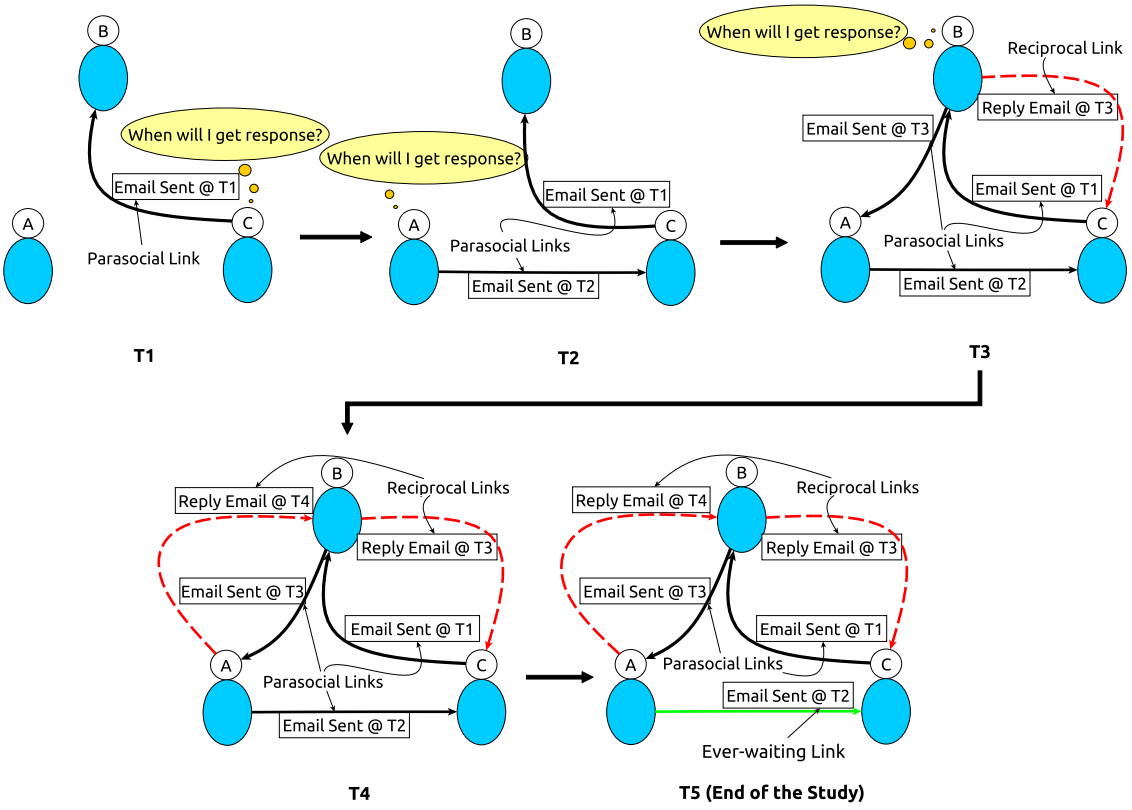


Fig. 3.1.: An illustration of reciprocal link time prediction *RLTP* problem.

is not available. Traditional supervised regression models cannot include censored data instances in the training data and hence perform poorly in predicting reciprocal link creation time.

I explain the cases of right-censored data instances in reciprocal interval time prediction task using a toy example shown in Figure 3.1. In this figure I show a small part of an email communication network consisting of only three vertices representing three persons, A, B, and C. The observation period of this network has five timestamps, $T1$ to $T5$. At $T1$, C sends an email to B, thus creating the first of the directed links (such links are called parasocial links). At T_2, the parasocial link from A to B is created. At $T3$, the reciprocal link from B to C is created; thus the interval time of this edge is $T3 - T1$. At $T3$, another parasocial link ($B \rightarrow C$) is created. More links are created in subsequent time intervals $T4$ and $T5$. At $T5$, I reach the end of

the observation period, but the reciprocal link from C to A is yet to be created. The potential reciprocal link $C \rightarrow A$ is an instance of right-censored data for which we only know that the interval time is higher than $T5 - T1$; this value, as well, can be infinity in the case that the link is never created. Either way, the exact value of the target variable for this reciprocal edge is unknown. Unfortunately, for any reasonable observation time window, a significantly large number of potential reciprocal links are censored data instances, which is the main challenge for the task of reciprocal link creation time prediction.

In this chapter, I present a supervised learning model for predicting the interval time for the creation of a reciprocal edge between a pair of vertices in an online social network, given that a parasocial edge already exists between the vertex-pair. I study real-life networks and validate a collection of topological features that may influence the reciprocal edge creation time. Then, we design the prediction task as a survival analysis problem and choose five censored regression models. My experimental results show that Cox regression performs better than traditional supervised learning models for reciprocal link prediction.

3.2 Problem Formulation

Definition 3.2.1 ***Directed time-stamped network.*** *Consider a network $G(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of directed edges. $\mathcal{T}$ is a set of time values, and τ is a mapping function, which maps an edge to one of the time values in the set $\mathcal{T}$, i.e., $\tau : \mathcal{E} \rightarrow \mathcal{T}$. For an edge $e \in \mathcal{E}$, $t_e \in \mathcal{T}$ denotes the creation time of the edge e. Collectively, G, $\mathcal{T}$, and τ are called a directed time-stamped network.*■

For vertices $u, v \in \mathcal{V}$ and link $e = (u, v) \in \mathcal{E}$ the corresponding time-stamp t_e can be represented as t_{uv}. If an edge e is created multiple times, I keep only the oldest (earliest) creation time and assign that to t_e. For a vertex $u \in \mathcal{V}$, $\Gamma_{in}(u)$ and $\Gamma_{out}(u)$ are the set of in-neighbors and the set of out-neighbors of u, and $d(u, v)$ is the directed shortest path distance from u to v.

Definition 3.2.2 ***Reciprocal/Parasocial Link.*** *For a pair of vertices, u, and v, the edge $(u, v) \in \mathcal{E}$ is called a parasocial link if the edge $(v, u) \notin \mathcal{E}$. On the other hand, if $(v, u) \in \mathcal{E}$ and $(u, v) \in \mathcal{E}$, and $t_{vu} < t_{uv}$ then (u, v) is called a reciprocal link.* ■

The objective of the *RLTP* problem is to predict the time of a reciprocal link for the given parasocial link with time. The **interval time** for a reciprocal link (u, v) is defined as $Int(u, v) = t_{uv} - t_{vu}$. My model for the *RLTP* problem actually predicts $Int(u, v)$, instead of predicting t_{uv} (the reciprocal link creation time). Nevertheless, the reciprocal link creation time t_{uv} can be obtained from the model by using the expression $t_{vu} + Int(u, v)$. The advantage of predicting $Int(u, v)$ instead of predicting t_{uv} is that for predicting $Int(u, v)$ we do not need to use the parasocial link creation time t_{vu} as part of input of the model, which makes the model independent of temporal bias. Thus the supervised model of my proposed *RLTP* task uses only the topological features of an edge (u, v) as its covariates and the interval time $Int(u, v)$ as its target variable, making the model simple.

3.3 Dataset Study

In this section, I discuss the datasets that I use in this study. I also provide some statistical analysis of the datasets; specifically, for each of these datasets, I provide the empirical distribution of observed interval time and its goodness of fit with known statistical distributions. For the *Enron* dataset, the persons (along with their rank in the company) associated to a vertex is known, so in this dataset I have also performed a qualitative study by checking for the evidences of social stratification phenomenon, which I present at the end of this section.

I used three real-world directed network datasets for this study. We selected datasets where reciprocal link creation is an important (meaningful) event; another selection criterion is that the datasets should have a sufficient number of reciprocal links to train and test the models [144]. The first dataset, *Epinion* is a trust network

Table 3.1.: Basic statistics of the datasets used in the *RLTP* study.

Dataset	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	$\lvert\mathcal{T}\rvert$	*Recipro*
Epinion	131,828	841,373	938	0.3083
MC-Email	167	5,783	237	0.876
Enron	182	3,007	944	0.6053

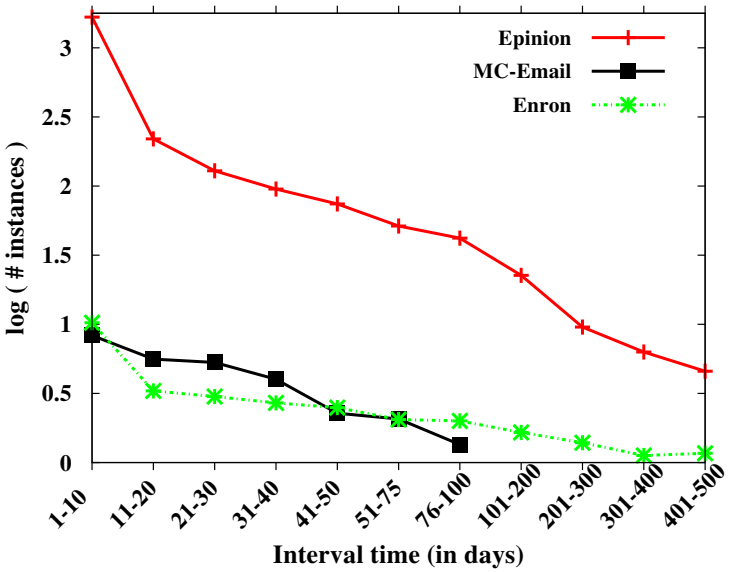


Fig. 3.2.: Histogram of interval time of reciprocal link.

where a directed link from one vertex to another represents the fact that the former trusts the latter. The *RLTP* task for this dataset is to find the time at which a trusted person acknowledges that (s)he also holds a similar sentiment towards the other person. The dataset was collected from KONECT web page.[1] I have also collected two email datasets: *MC-Email*[2] and *Enron*. Both of these datasets are email communication networks from two distinct enterprises, and for these datasets the *RLTP* task is to predict the response time for an email. More information on these datasets is provided in Table 3.1, where $\lvert\mathcal{V}\rvert$, $\lvert\mathcal{E}\rvert$, $\lvert\mathcal{T}\rvert$, and *Recipro* are the

[1] `http://konect.uni-koblenz.de/networks/`
[2] This is Manufacturing Company email dataset available from R. Michalski's website, `https://www.ii.pwr.edu.pl/~michalski`

number of vertices, the number of edges, the number of timestamps (in days), and the reciprocity of the dataset within the observation window, respectively.

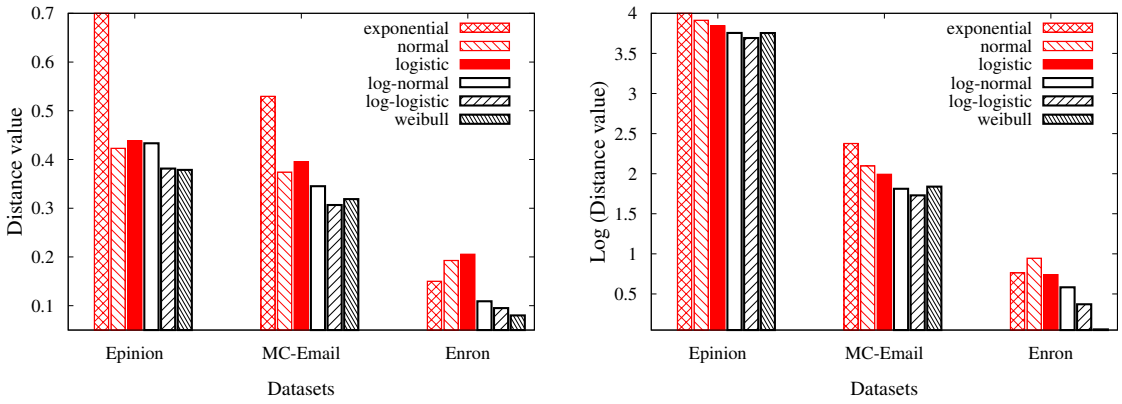
For these three datasets, I plot the histogram of the interval time for reciprocal links in log scale (Figure 3.2). I observed that the majority of the responses are received within a short period of time (within 10 days or less). However, there also exist a few late responders whose reply time is much larger than the average reply time.

3.3.1 Modeling interval time using Parametric Distribution

From the distribution plots in Figure 3.2 I observe that the number of reciprocal link instances reduces exponentially with the increment of the interval time (note that, y-axis is in log-scale). Hence, I fit different exponential family distributions to model the time interval of reciprocal link for all three datasets. Specifically, I fit exponential distribution, normal distribution, logistic distribution, log-normal distribution, log-logistic distribution and Weibull distribution. To evaluate the goodness of fit I use the following four metrics: Kolmogorov-Smirnov (KS) statistic, Cramer-von Mises (CM) statistic, Aikake's Information Criterion (AIC) [145] and Bayesian Information Criterion (BIC) [146]. In Figure 3.3, I show the quality of fitting results. The results of BIC are very similar to AIC for all three datasets, so I did not show the results of BIC. As depicted in Figure 3.3, exponential, normal, and logistic distributions (shown in red) have relatively high distance from empirical distribution compared to log-normal, log-logistic and Weibull distributions (shown in black). For the *Enron* dataset, Weibull distribution performs the best over all metrics. Similarly, for the *Epinion* and the *MC-Email* datasets log-logistic distribution fits the best. Results of log-normal distribution are very similar to both Weibull and log-logistic distributions. Hence, I use log-normal, log-logistic and Weibull distributions for parametric survival models, which are discussed later in Section 3.6.

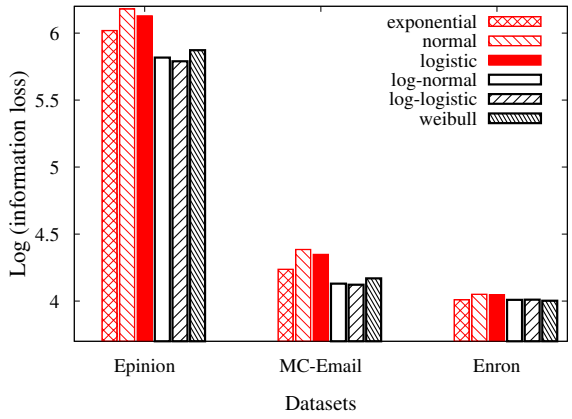
3.3.2 Social Stratification in Enron

One of the influencing factors for late responses to a specific user is social stratification - particularly in corporations, people tend to give quicker replies to their superior as compared to their colleagues and other juniors. I study the *Enron* dataset, for which the employee details are available with email communications. In the dataset, "Louise Kitchen" is a president; I observed that her email replying practice follows social stratification phenomenon. She generally takes more than $2-3$ days to reply to people with lower ranking positions such as vice-president (VP), employees, etc. For



(a) Kolmogorov-Smirnov statistic

(b) Cramer-von Mises statistic



(c) Aikake's Information Criterion

Fig. 3.3.: Goodness of fit comparisons for different distributions.

example, she replied to VPs "Kevin Presto", "James Steffes" and "Fletcher Sturm" in 3, 6 and 19 days respectively. She replied to "Sally Beck" (Chief Operating Officer) in 5 days. On the other hand she replied to "David Delainey" (Chief Executive Officer (CEO)) on the same (0) day. Another example is "Philip Allen", who is a manager; he replied within a day to higher ranking officers such as "David Delainey" (CEO), "Barry Tycholiz" (VP), "Hunter Shively" (VP) and "Richard Shapiro" (VP). On the other hand, he took 2 to 3 days to reply to "Michael Grigsby" (manager), "Jay Reitmeyer" (employee) and "Matthew Lenhart" (employee).

3.4 Topological Feature Study

In online social networks, user behavior based features are useful for solving different problems, such as, link prediction [140], personality prediction [147], user attribute prediction [148], link sign prediction [149], prediction of positive and negative users in Twitter [150], etc. Hence, I believe social (behavioral) phenomena based topological features can contribute substantially to solve the *RLTP* problem. Though there are works that study and design user behavior features such as topic-specific modeling [151], a behavioral model for Facebook wall posts [152], etc., I assume to have only topological information. Topological features that I use comes from two different social phenomena: directed altruism and social stratification. Below I discuss them in two different sections.

3.4.1 Directed Altruism Based Features

Directed altruism in social networks is described in [19], where the authors have argued that people are more generous to friends and friends of friends than to a complete stranger. This phenomenon also reflects in people's reciprocal link creation behavior. Below, I define some topological features which quantify the directed altruism phenomena for reciprocal link prediction.

Shortest directed distance: In this problem, one directional link (v, u) already exists, and I am predicting the creation time for the reverse link (u, v). Generally people are more generous to indirect friends than complete strangers. Hence u is more likely to respond quickly to v for small value of the directed distance from u to v i.e.

$$DirectedDist(u, v) = d(u, v)$$

Common in/out neighbors count: The number of common neighbors is a frequently used topological feature for the link prediction task in undirected networks; however, for directed graphs, I have two separate features: common in-neighbors and common out-neighbors. Both of these topological features capture the idea that if a user has more common neighbors with another user, then she is more likely to reply fast. Also, more common friends increase the network flow, which is an important factor for building trust [19] and with higher trust people tend to reply faster.

$$Common_{in}(u, v) = |\Gamma_{in}(u) \cap \Gamma_{in}(v)|$$

$$Common_{out}(u, v) = |\Gamma_{out}(u) \cap \Gamma_{out}(v)|$$

Jaccard coefficient (In/Out): The Jaccard coefficient is another widely used topological feature for undirected networks. It is the normalized version of common neighbors counts. Similar to the common neighbor count feature, this feature also split into two features due to the directed-ness of the edges. Jaccard coefficients help to predict the trust level between two nodes. Since, higher trust leads to faster response, this is a good feature for the *RLTP* task.

$$Jaccard_{in} = \frac{|\Gamma_{in}(u) \cap \Gamma_{in}(v)|}{|\Gamma_{in}(u) \cup \Gamma_{in}(v)|}$$

$$Jaccard_{out} = \frac{|\Gamma_{out}(u) \cap \Gamma_{out}(v)|}{|\Gamma_{out}(u) \cup \Gamma_{out}(v)|}$$

Local Reciprocity. In [82], the authors studied two local reciprocity features and they showed relative influence of both features on linking back probability. The first is Acceptance Local Reciprocity (ALR), which is defined as:

$$ALR(v) = \frac{|\Gamma_{in}(v) \cap \Gamma_{out}(v)|}{|\Gamma_{in}(v)|}$$

I compute ALR for the head node (v) of the reciprocal link (u, v). This feature captures the tendency of node v to accept a link. The second feature is Request Local Reciprocity (RLR), defined as:

$$RLR(u) = \frac{|\Gamma_{in}(u) \cap \Gamma_{out}(u)|}{|\Gamma_{out}(u)|}$$

I compute RLR for the tail node (u) of the reciprocating link (u, v). RLR represents the response behavior of the node u and captures its tendency to initiate a reciprocal link.

3.4.2 Social Stratification Based Features

It is observed that in online social networks people behave according to their status in the network [24]. A similar behavior is observed in many real world applications, such as the one described in Section 3.3 or in online dating [153]. I have also shown evidence of social stratification in *Enron* dataset, specifically in connection to the *RLTP* task. The following topological features quantify the extent of social stratification that is practiced by the node u or v.

Preferential Attachment: This feature computes a value which reflects the social stratification induced rank order of a given node. The basic idea of preferential attachment is to give more weight to the higher degree nodes. Traditionally, preferential attachment has been computed for undirected networks, so I change the formula to adapt it for a directed networks. For undirected graph, it is simply the product of the degrees of the node u and v. For directed graph, I take the product of the

out-degree of the tail node (u) and the in-degree of the head node (v) of a prospective reciprocal link (u, v). The formula is given below:

$$PrefAtt(u, v) = |\Gamma_{out}(u)| \times |\Gamma_{in}(v)|$$

Preferential Jaccard: *PrefJacc* is inspired by both Preferential Attachment and Jaccard Coefficient. It is a trade-off between two concepts—first, high degree nodes are prone to create more edges, and second, nodes prefer to connect with similar nodes (social stratification). Both these phenomena can influence reciprocal edge creation. I calculated *PrefJacc* by using the following equation:

$$PrefJacc(u, v) = \frac{|\Gamma_{out}(u) \cap \Gamma_{in}(v)|}{|\Gamma_{out}(u) \cup \Gamma_{in}(v)|}$$

In/Out Ratio: A node in the upper hierarchy has a tendency to a create reciprocal edges with other nodes at the same hierarchy level than to nodes which are at a lower hierarchy level [24]. To reflect this knowledge in this model, I need to find an efficient way for comparing the hierarchy of a pair of nodes, which I compute by the ratio of their in-degrees and the ratio of their out-degrees. Higher *InRatio* is indicative of higher tendency of the numerator node to attract links compared to the denominator node; similarly, higher *OutRatio* represents a higher tendency of the numerator node to create links compared to the denominator node. In this way, these two features capture the relative patterns of link creation and link acceptance by the pair of the vertices. For reciprocating link (u, v), I calculate *InRatio* and *OutRatio* by using the following equations:

$$InRatio = \frac{|\Gamma_{in}(u)|}{|\Gamma_{in}(v)|}$$

$$OutRatio = \frac{|\Gamma_{out}(u)|}{|\Gamma_{out}(v)|}$$

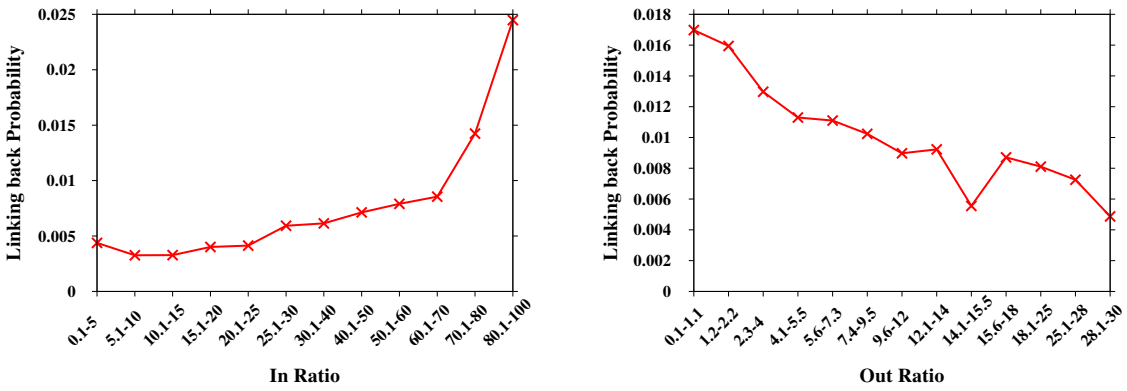


Fig. 3.4.: Relation of *In/OutRatio* and linking back probability in *Epinion* dataset

PageRank: *PageRank* represents the prestige of the node in the network. I use both, pagerank of u and pagerank of v as features. If *PageRank(u)* is lower than *PageRank(v)*, then the node u is highly likely to respond faster to the node v.

Table 3.2.: Correlation of features with Interval time

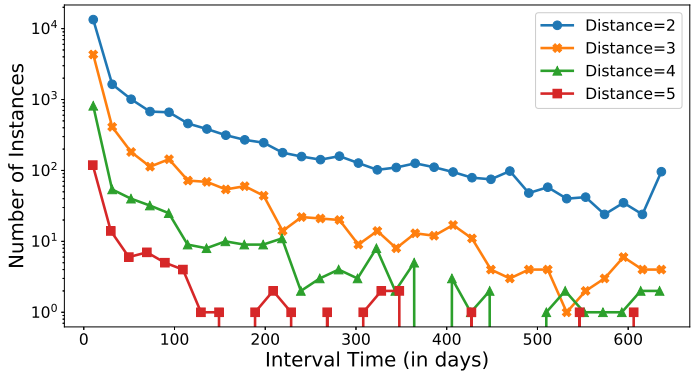
Features/ Datasets	*Epinion*	*MC-Emails*	*Enron*
DirectedDist	-0.04127	-0.03792	-0.13336
$Common_{In}$	0.38109	0.33447	0.44398
$Common_{Out}$	0.27254	0.31194	0.27534
$Jaccard_{In}$	0.17161	0.22101	0.24831
$Jaccard_{Out}$	0.11015	0.18925	0.20195
RLR(u)	-0.00290	0.05820	0.16053
ALR(v)	-0.06093	0.15383	0.19256
PrefAtt	0.19289	0.23930	0.25443
PrefJacc	0.09136	0.20054	0.25502
InRatio	-0.03165	-0.07053	-0.14302
OutRatio	-0.01132	0.04269	0.13108
PageRank(u)	0.24783	-0.07523	-0.07609
PageRank(v)	0.14300	0.00211	0.02049

3.4.3 Feature analysis

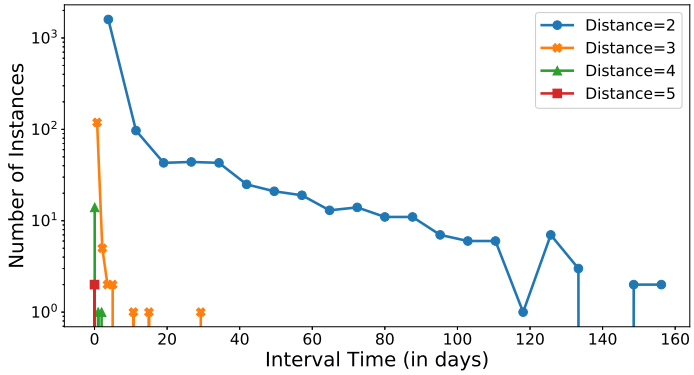
To validate the strength of these features (13 in total) for predicting the interval time of reciprocal edges, I compute the Pearson's correlation of the above topological features with the interval time value for three real-life graph datasets (Table 3.1) and show the correlation values in Table 3.2. As we can see, for the *MC-Emails* dataset most of the features (mainly $Common_{in}$, $Common_{out}$, $Jaccard_{In}$, $Jaccard_{Out}$, *PrefAtt* and *PrefJacc*) have good correlation value (between 0.2 to 0.5). Similarly, for the *Enron* dataset the same set of features is highly related to interval time. But, for *Epinion* dataset the correlation values for most of the features are poor except for $Common_{in}$, $Common_{out}$, and *PageRank(u)*; the worst features are *InRatio*, *OutRatio* and *RLR(u)*. To check the influence of these features on reciprocal link creation, I also check the average linking back probability over different range of values for different features. I discuss observation in the following paragraphs.

In Figure 3.4, I plot the observation for two of the features: *InRatio* and *OutRatio*. Here, for each bin of *InRatio*, the linking back probability is calculated as a fraction of reciprocal links over all the links in that bin. Figure 3.4 clearly shows high linking back probability for higher *InRatio* and lower *OutRatio*, which is expected behavior for these features. In [82], the authors provided a thorough study of some features, such as, *RLR(u)* and *ALR(v)*, and proved their significant influence on reciprocal link creation.

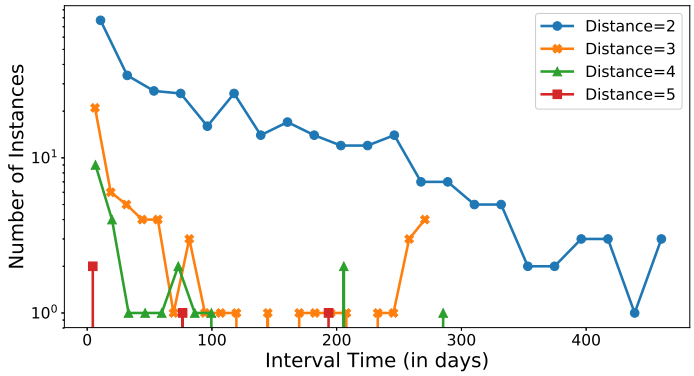
in Figure 3.5, I show three plots (one for each dataset) of *DirectedDist* vs. interval time. Within each plot I have several graphs, each representing the directed distance value between the vertices. Along the x-axis is the interval time and along the y-axis is the number of reciprocal link instances that have the corresponding interval time. For all dataset, I observe that links with small directed distance value (such as, 2 or 3) can have high interval time, i.e. the reciprocal link may appear after many days; but as distance increases there are few or almost no instances of reciprocal links with high interval time. This observation may appear counterintuitive as I expect



(a) *Epinion* Dataset



(b) *MC-Email* Dataset



(c) *Enron* Dataset

Fig. 3.5.: *DirectedDist* vs. Interval Time

short distance to influence a short interval time. However, This observation can be explained as follows: people tend to trust other people who are within their circles, and they will ultimately create a reciprocal links with them, even if they do not do it immediately. On the other hand, for people who are outside someone's circle (having a high directed distance value, such as 4 or 5), reciprocal links will be created either in a short interval time or will not be created at all. The short interval time can be the cases when two strangers meet in-person in a social event and then mutually agree to be connected online (or trust each other). On the other hand, the negative case happens, when a stranger trusts (or sends an invite to) someone, and the second person just ignore that forever. Due to this complex relation, the correlation between directed distance and interval time is poor, yet I consider *DirectedDist* to be a useful feature.

Correlation with Low and High Interval time.

There are a variety of different social behaviors that influence the interval time, hence some social based features impact the interval time differently over a period. To understand the impact of different features over a period, I split the target variable (interval time) into lower and higher range and calculate feature correlations with lower and higher interval times separately. For this study, I calculate average interval time for each dataset and if the interval time is less or equal to average interval time I call it low interval time, otherwise, we call it high interval time. For each dataset and each feature I calculate the correlation value between the feature and low and high interval times; these correlation values are shown in Table 3.3.

In Table 3.3, I observe that features like $Common_{in}$, $Common_{out}$, $Jaccard_{In}$, $Jaccard_{Out}$, *PrefAtt* and *PrefJacc* have high correlation with higher interval time. For the *Enron* dataset, some of these features ($Common_{in}$, $Jaccard_{In}$ and *PrefJacc*) are highly correlated to lower interval time as well. For the *MC-Email* dataset, *DirectedDist*, *ALR(v)*, *Out Ratio* and *PageRank(v)* have noticeable correlation with lower interval time and other two features (*RLR(u)* and *In Ratio*) are inversely corre-

Table 3.3.: Correlation of features with Low and High Interval times

Datasets	*Epinion*		*MC-Emails*		*Enron*	
Features	Low	High	Low	High	Low	High
DirectedDist	-0.00387	-0.04587	0.14453	-0.06022	-0.04023	-0.15364
$Common_{In}$	0.06671	0.33821	0.00018	0.41728	0.22168	0.35640
$Common_{Out}$	0.07231	0.24064	0.06639	0.27446	0.04738	0.20793
$Jaccard_{In}$	0.07765	0.15312	-0.04154	0.29774	0.17726	0.10033
$Jaccard_{Out}$	0.06829	0.13183	-0.07426	0.22517	0.07820	0.06360
RLR(u)	-0.03937	0.06628	-0.17897	0.02467	0.07949	0.06348
ALR(v)	-0.01783	-0.07657	0.15905	0.09455	0.08760	0.06401
PrefAtt	0.03049	0.14439	-0.00163	0.31220	0.06305	0.32053
PrefJacc	0.04258	0.13021	-0.06545	0.23248	0.16523	0.09010
InRatio	-0.01251	-0.01751	-0.15333	-0.04297	-0.10385	-0.09571
OutRatio	-0.00700	-0.02610	0.29600	-0.06979	0.00578	0.12331
PageRank(u)	0.06118	0.20674	-0.07606	-0.09756	-0.00557	-0.12452
PageRank(v)	0.02399	0.14362	0.31830	-0.13432	0.01606	0.03715

lated to lower interval time. One surprising observation for the *MC-Email* dataset is that *PageRank(v)* is the poorest feature (Table 3.2), but highly correlated with both lower and higher interval times, mainly because the feature is positively correlated for lower interval time and inversely correlated with higher interval time. From Table 3.3 I understand that for different datasets user behavior varies and hence a distinct set of features becomes influential to the interval time (especially lower interval time) of that dataset.

3.5 Proposed methodology using survival analysis

Survival analysis is widely used in the medical domain to predict survival time or time to a specific event (such as death) for patient datasets [60], [61]. In the survival analysis setup, for a set of instances under observation, events happen over a time period, from which a survival model learns the temporal patterns of these events and predicts the survival time. Here, I propose a novel method to map the *RLTP* problem to a survival analysis task and explain survival analysis concepts from a reciprocal

link creation perspective. For these concepts, I also provide suitable terminology for the *RLTP* problem to describe my approach clearly.

Beginning of graph expansion and study period: At the first time-stamp, a given directed time-stamped network is static (initialized); the *beginning of graph expansion* is the second time-stamp from when new links are added to the static network. Survival analysis assumes a starting time of the study, from when a model starts to observe for the events. In the *RLTP* problem, the *beginning of graph expansion* serves as the starting time of the study. For the *RLTP* problem, I divide the time-stamps of the network into train and test time periods, and I observe the network for the reciprocal link creation till the end of the train period, so the last time-stamp in the train period is considered to be the **end of the study**. Thus the time window from the beginning of graph expansion to the last time-stamp of train period is considered to be the **study period** which is the same as the train period.

Reciprocal event: For a parasocial link (v, u), if a reciprocal link (u, v) is created during the training period, I call it a *reciprocal event*, which is the event of interest in the *RLTP* problem. In the *RLTP* problem each parasocial link is a data instance, time-stamp of a parasocial link generation is the time when the data instance is considered into the network for study. Hence, the time-stamp of a parasocial link generation is called the **starting time of observation** for that data instance (an ordered pair of vertices).

ever-waiting links: I study the network for a limited time window (train period), and hence for a set of parasocial links, the corresponding reciprocal event may not be observed before the end of the study (last time-stamp of training period). I call these links *ever-waiting links*. *ever-waiting* links carry the information that the reciprocal link creation event did not happen till the end of the train period. In the survival

analysis terminology the *ever-waiting* links are also called censored instances; I use both of these terms interchangeably.

In a traditional regression task, *ever-waiting* links may either be ignored, because the target value (the interval time) for these instances are unknown, or they may be retained with an arbitrarily chosen large interval time, which is higher than the time difference between the end of the study time and the starting time of observation for that parasocial link. The first of the above approaches ignores important information; specifically, the ignored fact is that the interval time for *ever-waiting* links is higher than the time difference between the end of the study and the starting time of observation for that parasocial link. The second approach is simply a crude approximation of the target value. As mentioned before, the main reason to map the *RLTP* problem into survival regression analysis framework is to exploit the important information provided by the *ever-waiting* links.

Target value of survival regression model: The time difference between the starting time of observation (parasocial link generation time) and the time-stamp of the reciprocal edge creation is the interval time which I want to predict in the *RLTP* task. For a reciprocal edge (u, v), the interval time is defined as $Int(u, v)$, as is discussed in Section 3.2. In a traditional survival model, the interval time is called the life-span of an instance as for these models "death" is the event of interest. Hence survival models that predict survival time can be adopted to predict the interval time for the *RLTP* problem. For training the prediction model, I need a feature vector for each data instance, along with the survival time and a binary event indication value (event occurred or not). For the *RLTP* problem, the feature vector of a parasocial edge is $\mathbf{x}_i \in \mathbb{R}^d$, a vector of topological features (Section 3.4) for the i'th parasocial link in training data, where feature dimension d is 13 (number of topological features). For each parasocial links of the training period, if the reciprocal event has occurred during training period then life-span of parasocial link is the interval time with the event indication value set to 1; otherwise, for *ever-waiting* links, the time difference between

the last time-stamp of training and time-stamp of the parasocial link generation is the survival time with event indication value set to 0. Given this training dataset the target value (the interval time) of test instances are predicted by using a trained survival model. I use various survival models, which I discuss in the next subsection.

3.6 Survival models for the *RLTP* problem

As explained in the previous section, any survival model can be adopted to solve the *RLTP* problem. There are two types of widely used survival models: 1) semi-parametric models and 2) parametric models. Parametric models assume that interval time follows a known statistical distribution; hence, if the interval time for a dataset follows a distribution then parametric models perform very good for the dataset compared to a semi-parametric models. However for many real-world datasets, it is difficult to find a suitable statistical distribution that fits well to the interval time, for these datasets semi-parametric models perform better than parametric models, because semi-parametric models do not assume any underlying distribution, rather they try to learn the actual distribution from the data. As I discussed in Section 3.3, some of the datasets are good fit for a statistical distribution but others are not. Hence, I conduct experiments with both semi-parametric and parametric models to offer a comprehensive study of the *RLTP* problem. In this section, I describe these selected semi-parametric and parametric models and their adaptation for solving the *RLTP* problem.

Note that, for a given parasocial link if corresponding reciprocal link is not likely to be created at time t then *Survival function* value for t is high. On the other hand, if the corresponding reciprocal link is highly likely to be created at time t then the *reciprocal event density function* value should be high and that leads to a higher value of the *Hazard function*. We can observe that both *survival function* and *hazard function* are interrelated and we can model either function for the interval time prediction. Next, I describe how semi-parametric Cox regression models the *hazard*

function to solve the *RLTP* problem. Later I discuss parametric methods (BJ-model and AFT models) and their approach for modeling the *survival function* with the help of different statistical distributions.

3.6.1 Cox Regression

For solving the *RLTP* problem, I use regularized Cox model to avoid over-fitting. We observe in Section 3.4.3 that only a few features have a strong correlation with the target variable, so I want to use a sparse regularization model. I use Elastic Net regularization. In literature, a Cox model with Elastic Net regularization is also known as Cox model with Elastic Net (EN) penalty [154]. The penalty term P_{EN} is:

$$P_{EN}(\boldsymbol{\beta}) = \sum_{k=1}^{d} \left[\alpha|\beta_k| + \frac{1}{2}(1-\alpha)\beta_k^2 \right] \tag{3.1}$$

Where, $0 < \alpha \leq 1$ and with EN penalty the objective function of the Cox model (Equation 1.4), becomes

$$\widehat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \quad \frac{1}{n_p} \sum_{i=1}^{n_p} \left[-C_i(\mathbf{x}_i^T \boldsymbol{\beta}) + C_i \log \left(\sum_{j \in R_t} \exp(\mathbf{x}_j^T \boldsymbol{\beta}) \right) \right] + \gamma \cdot P_{EN}(\boldsymbol{\beta}) \tag{3.2}$$

Here, $\gamma > 0$ is a regularization constant. For solving this optimization task, I can use the maximum partial likelihood estimator proposed in [62]; it uses the Newton-Raphson method to iteratively find the estimated $\widehat{\boldsymbol{\beta}}$ which minimizes the Equation (3.2).

3.6.2 Parametric Models

The main idea behind a parametric model is that it assumes that the interval time follows a specific statistical distribution. There are two ways to relate interval

Table 3.4.: Density, Survival and Hazard functions for the distributions used with AFT model. λ is scale parameter and k is shape parameter for both Weibull and log-logistic distribution. For log-normal distribution μ is the mean (location parameter), σ^2 is the variance and Φ is cumulative distribution function of normal distribution.

Distributions	**Density Function**	**Survival Function**	**Hazard Function**
Weibull	$\lambda k t^{k-1} \cdot \exp(-\lambda t^k)$	$\exp(-\lambda t^k)$	$\lambda k t^{k-1}$
Log-Normal	$\frac{1}{\sqrt{2\pi}\sigma t} \exp(-\frac{(log(t)-\mu)^2}{2\sigma^2})$	$1 - \Phi(\frac{log(t)-\mu}{\sigma})$	$\frac{\frac{1}{\sqrt{2\pi}\sigma t} \exp(-\frac{(log(t)-\mu)^2}{2\sigma^2})}{1-\Phi(\frac{log(t)-\mu}{\sigma})}$
Log-Logistic	$\frac{\lambda k t^{k-1}}{(1+\lambda t^k)^2}$	$\frac{1}{1+\lambda t^k}$	$\frac{\lambda k t^{k-1}}{1+\lambda t^k}$

time and a statistical distribution: first, assume that the actual interval time for all parasocial links follows a distribution; and second, assume that the logarithm of the interval time follows a distribution. The models under the first assumption are referred to as linear regression models, and the models under later assumption are called accelerated failure time (AFT) models.

Linear regression model: The statistical linear regression with the least squares estimation is widely used for a variety of regression tasks. However, the issue with the model is that it cannot use information from *ever-waiting* links. For interval time prediction this issue can be handled by using a specific survival model such as the Buckley-James model (BJ model). The BJ model first estimates the interval time of training *ever-waiting* links using the Kaplan-Meier (KM) [63] estimation method and then by using all parasocial links from training period to train a linear model. This linear model can be trained through MLE as described above. For more practical use, Wang et al. [155] proposed twin boosting method with BJ estimator, I use this method to solve the *RLTP* problem.

Accelerated Failure Time (AFT) model: An AFT model assumes that the logarithm of the interval times $\log(\mathcal{T}_{Int})$ follows a statistical distribution and it is linearly

related to the (topological) feature vectors. The general form for AFT regression model is

$$log(\mathcal{T}_{Int}) = \boldsymbol{X} \cdot \boldsymbol{\beta} + \sigma \cdot \epsilon \tag{3.3}$$

where $\boldsymbol{X}$ is the covariate matrix of size $n_p \times d$ where i^{th} row of $\boldsymbol{X}$ is $\mathbf{x}_i$, $\boldsymbol{\beta}$ is a d dimensional coefficient vector (model parameters), σ ($\sigma > 0$) is an unknown scale parameter, and ϵ is an error variable which follows a similar distribution to $\log(\mathcal{T}_{Int})$. For the problem, I use the three most suitable distributions (see Figure 3.3) for interval time, the details of which are given in Table 3.4.

3.7 Algorithmic framework

In Algorithm 6, I describe a general framework of my proposed method. First I divide the time-stamps of the input graph into train and test periods as mentioned in line 1 of Algorithm 6. After that I create training data instances (*train-set*) and test data instance (*test-set*) from the corresponding train and test periods (Lines 2-4). Then I calculate topological features for each parasocial link (data instance) in the *train-set* and *test-set* as described in Lines 5-10 of Algorithm 6. After that I generate target variable for each data instance (Lines 11-26), for which I observe the corresponding reciprocal link in the graph. For a parasocial link $e \in$ *train-set*, if the corresponding reciprocal link is generated during train period then interval time $Int(e)$ (Section 3.2) is the target value with event indicator value $C_e = 1$ otherwise time difference between the link creation and end of training period act as the target value with event indicator value $C_e = 0$.

Algorithm 1: The Framework

For time-stamps (t_0 to t_M) of the input graph, divide the starting $p\%$ time-stamps as training period (t_0 - t_p) and remaining as testing period (t_{p+1} - t_M).
train-set $\leftarrow$ all parasocial links generated before/at time-stamp t_p.
test-set $\leftarrow$ all parasocial links generated after time-stamp t_p and immortal links.
Sort edge in *train-set* and *test-set* based on its edge creation time. {optional}
for each (edge) $e \in$ *train-set* **do**
  $G_{te} \leftarrow$ create a snapshot of the network at $t_e - 1$. {sorting helps in this step}
  $x_e \leftarrow$ generate topological features (Section 3.4) for edge e from the snapshot G_{te}.
  add x_e to $\boldsymbol{X}$.
Similarly, generate topological features for edges in the *test-set* and generate $\boldsymbol{X^{test}}$.
for each $e \in$ *train-set* **do**
  if e has a reciprocal link e_r in dataset **then**
    if $t_{er} \leq t_p$ **then**
      $y_e \leftarrow Int(e) \leftarrow t_{er} - t_e$ {target value for the parasocial edge (data instance)}
      $C_e \leftarrow 1$ {event indicator value for the parasocial edge (data instance)}
    else
      $y_e \leftarrow t_p - t_e$ {target value for the *ever-waiting* link (data instance)}
      $C_e \leftarrow 0$
  else
    $y_e \leftarrow t_p - t_e$ {target value for the *ever-waiting* link (data instance)}
    $C_e \leftarrow 0$
  add y_e to $\mathcal{T}_{Int}^{tr}$.
  add C_e to $\mathcal{C}^{tr}$.
for each $e \in$ *test-set* **do**
  if e has a reciprocal link e_r in dataset **then**
    $y_e \leftarrow Int(e) \leftarrow t_{er} - t_e$ {target value for the parasocial edge (data instance)}
    $C_e \leftarrow 1$
  else
    $y_e \leftarrow t_M - t_e$ {target value for the *ever-waiting* link (data instance)}
    $C_e \leftarrow 0$
  add y_e to $\mathcal{T}_{Int}^{test}$.
  add C_e to $\mathcal{C}^{test}$.
{Use one of the methods among Cox, BJ, and AFT; below I call all three methods}
$\boldsymbol{cox} \leftarrow$ cocktail($\boldsymbol{X}$, $\mathcal{T}_{Int}^{tr}$, $\mathcal{C}^{tr}$) {method of *fastcox* (*R* package)}
{For given distribution *dist*}
$\boldsymbol{AFT_{dist}} \leftarrow$ survreg($\boldsymbol{X}$, $\mathcal{T}_{Int}^{tr}$, $\mathcal{C}^{tr}$,distribution=*dist*) {method of *survival* (*R* package)}
$\boldsymbol{BJmodel} \leftarrow$ bujar($\boldsymbol{X}$, $\mathcal{T}_{Int}^{tr}$, $\mathcal{C}^{tr}$) {method of *bujar* (*R* package)}
The $\boldsymbol{cox}$, $\boldsymbol{AFT_{dist}}$ and $\boldsymbol{BJmodel}$ contain the model parameters $\boldsymbol{\beta}$.
test-res $\leftarrow$ predict interval time y_e for each edge $e \in$ *test-set* using $\boldsymbol{X^{test}}$ and $\boldsymbol{\beta}$.
evaluate *test-res* using $\mathcal{T}_{Int}^{test}$ and $\mathcal{C}^{test}$.

Similarly, I generate target values for data instance of *test-set* as explained in Lines 27-35 of Algorithm 6. Then, I use *R* libraries to train the survival models with training data and predict target values for the test data to generate the test results (*test-res*) and lastly I evaluate that *test-res*.

3.8 Experiments and Results

I conducted a set of rigorous experiments to demonstrate the benefit of using censored information and the superiority of the survival models to solve the *RLTP* problem. I used five survival models: Cox regression model, AFT model with Weibull, log-normal and log-logistic distributions, and Buckley-James (BJ) regression model. To prove the fact that the survival models are better suited for solving the *RLTP* problem, I compared them with traditional regression models such as ridge regression (RidgeReg), lasso regression (LassoReg), feed forward neural networks (FFNN) and support vector regression (SVR). Note that these traditional regression models cannot use censored information (*ever-waiting* links). I also compare Cox regression model with generalized linear model (GML), which is an adopted model from [99].

In addition to the suitability of the survival models for the *RLTP* problem, I also demonstrate the usability of the *ever-waiting* links. For that, we conducted experiments where I train the survival models without censored information and compare the performance of the models on the test dataset. I report the improvement in the performance when the *ever-waiting* links are used for training the survival models.

Lastly, I conduct an experiment to show that reciprocal links with short interval time contain enough information required for training the survival models.

3.8.1 Datasets

For the experiments, I use three real world datasets *Epinion*, *MC-Email* and *Enron*. I discuss these datasets in Section 3.3 and basic statistics of the datasets are shown in Table 3.1.

Generating a synthetic dataset for the *RLTP* problem is a challenging task, because in the literature most of the synthetic graph generation models try to mimic basic real-world properties such as power-law degree distribution [156], community structures [2], etc. All these methods generate directed networks with extremely low reciprocity—generally, less than 1%. Durak et al. have proposed a synthetic network generation algorithm which also considers reciprocity [157]. We use this algorithm for generating three synthetic graphs where the vertex count varies between $10,000$ $(10K)$ to $30,000$ $(30K)$ with increments of $10K$. Edges of these synthetic networks have no time-stamps; hence, I assign random time-stamps between 0 to 100 to parasocial links. The time-stamps of reciprocal links of these synthetic networks are selected by matching the reciprocal link interval time of the *Epinion* dataset through the best fit Weibull distribution.

3.8.2 Experimental Setting

For the experiments, I divide the time-stamps of a dataset into two non-overlapping continuous partitions, where the earlier partition is the train period and the latter is the test period. In three different experiments, I use, respectively, $60\%, 70\%$ and 80% of the earlier time-stamps as the train periods and the remaining time-stamps as the test period. For synthetic datasets, a 70:30 split of the time-stamps is used as the train and test period of the experiments. For calculating the topological features explained in Section 3.4 for a parasocial link (data instance), I take a snapshot of the network until the time-stamp of the corresponding reciprocal link or end of the train period (whichever is earlier).

Like any other link prediction task, *RLTP* also suffers from the class imbalance issue, where the number of positive instances ($C_i = 1$) is much smaller than that of the negative instances. To alleviate this problem I use the well-known majority undersampling [158] strategy as discussed below: all the reciprocal links generated during a train period are considered in the training data pool as positive instances and only 50% of the parasocial links generated during the same period are censored negative instances ($C_i = 0$) in the pool. The test data pool (and their labels) are also generated similarly from the test period. As train and test data instances need to be from their corresponding time periods, I use a modified K-fold cross validation, where each fold contains a random subset of train and test data instances from their respective pools. For all the experiments, I used 5-fold cross validation in this manner.

For minimizing the objective function (Equation (3.2)) of censored problem formulation of *RLTP*, for the Cox regression model, I used cocktail algorithm [159] (the library is provided by the authors of [159]). For AFT models and BJ regression, I used *Survival* package[5] and *Bujar* package[6], respectively, available in **R**. For RidgeReg, LassoReg and SVR, I used scikit-learn python library and for FFNN, I used Matlab NNtoolbox. I used *TopCom* indexing method [160, 161] (Chapter 8) to find shortest directed distance feature. To choose the best parameters of SVR, I used grid search, where the cost parameter C takes values from $\{0.0001, 0.001, 0.01, 0.1, 1.0\}$ and Epsilon (ϵ) takes values from $\{0.0001, 0.001, 0.01, 1.0\}$.

3.8.3 Evaluation Metrics

Datasets generated from directed time-stamped networks are longitudinal data and for the *RLTP* problem the datasets also contain censored information. Evaluating models on these datasets using traditional evaluation metrics is not suitable, instead I use time-dependent AUC (also known as c-Index), which is widely used in longitudinal data analysis [162].

[5]cran.r-project.org/package=survival
[6]cran.r-project.org/web/packages/bujar/index.html

For a pair of data instances, assume (y_i, y_j) and $(\widehat{y}_i, \widehat{y}_j)$ are the target and the predicted values, respectively. The time-dependent AUC is defined as the probability of $\widehat{y}_i > \widehat{y}_j$ given $y_i > y_j$. If target y_i has only 2 possible values, then time-dependent AUC is the same as the popular AUC (Area Under ROC Curve) metric for classification. Similar to the AUC metric, time-dependent AUC takes values between 0 to 1, where 1 is the best possible value for this metric. Time-dependent AUC (TD-AUC) is calculated as follows:

$$TD\text{-}AUC = \frac{1}{n_{cnt}} \sum_{i:C_i=1} \sum_{y_j > y_i} \mathbb{1}(\widehat{y}_j > \widehat{y}_i) \tag{3.4}$$

where, n_{cnt} is total count of (y_i,y_j) pairs such that $C_i = 1$ (the event has occurred) and $y_j > y_i$ holds.

For the Cox regression model, the predicted value is the hazard value and for a higher hazard value the event occurs earlier, hence the time-dependent AUC for Cox can be calculated as:

$$TD\text{-}AUC = \frac{1}{n_{cnt}} \sum_{i:C_i=1} \sum_{y_j > y_i} \mathbb{1}(\mathbf{x}_i^T \widehat{\boldsymbol{\beta}} > \mathbf{x}_j^T \widehat{\boldsymbol{\beta}}) \tag{3.5}$$

3.8.4 Comparison results of survival models and regression models

I compared the selected survival models with four other traditional regression models and the results are shown in Tables 3.5, 3.6 and 3.7, where columns represent different training splits and each row represents a prediction model. A horizontal bar separates the traditional regression models in the upper part and the survival based models in the lower part. Here, each table cell shows mean and standard deviation for TD-AUC values. For most of the cases, the Cox regression model performs the best.

For the *Epinion* dataset, as depicted in Table 3.5, the Cox regression model performs the best with mean TD-AUC 0.7364, 0.7463 and 0.7485 for training period

with 60%, 70% and 80% splits of time-stamps, respectively. Here, with increase in the training data we can clearly see improvement in the performance, which is an expected behavior because with more training examples the model learns better. BJ model is the next best with performance very close to the Cox model. For this model also, the mean TD-AUC improves from 0.7312 to 0.7416 as we increase the training data. Similar behavior is observed for other survival models, but the performance of the AFT models is, unfortunately, not good for the dataset. This can be attributed to the fact that AFT models make strict distribution assumptions on the data and such assumption may not be suitable for the *Epinion* dataset (Figure 3.3).

For the *Epinion* dataset, among the traditional regression based methods, ridge regression performs better than any other competing methods with mean TD-AUC in the range between 0.60 and 0.61. But, when we compare its performance over different training splits, we see that its performance does not improve as we increase the training data. The same behavior holds for other traditional regression methods, such as Lasso regression and FFNN. One possible explanation for this behavior is model under-fitting; that is, the majority of the errors in the traditional regression models are coming from the bias error, so the error does not improve much with a larger dataset which reduces variance error only. On the other hand, survival analysis

Table 3.5.: *Epinion* Dataset: TD-AUC results [mean (±standard deviation)] with different splits used for training period.

Method / Split	60%	70%	80%
RidgeReg	0.6185 (±.0018)	0.6086 (±.0013)	0.6060 (±.0018)
LassoReg	0.6169 (±.0013)	0.6020 (±.0014)	0.6039 (±.0017)
FFNN	0.5510 (±.1296)	0.5048 (±.0822)	0.4456 (±.0725)
SVR	0.4791 (±.0005)	0.4871 (±.0039)	0.4914 (±.0030)
BJ Model	0.7312 (±.0010)	0.7339 (±.0020)	0.7416 (±.0021)
Weibull	0.3807 (±.0763)	0.5210 (±.1446)	0.5232 (±.1282)
logNormal	0.3660 (±.0388)	0.4461 (±.0283)	0.4283 (±.0305)
logLogistic	0.4901 (±.0098)	0.5110 (±.0196)	0.5132 (±.0188)
Cox	**0.7364** (±.0025)	**0.7436** (±.0016)	**0.7485** (±.0028)

Table 3.6.: *MC-Email* Dataset: TD-AUC results [mean (±standard deviation)] with different splits used for training period.

Method / Split	60%	70%	80%
RidgeReg	0.6213 (±.0087)	0.6083 (±.0146)	0.6014 (±.0125)
LassoReg	0.5884 (±.0100)	0.5709 (±.0201)	0.5686 (±.0074)
FFNN	0.4199 (±.0800)	0.4609 (±.0964)	0.5069 (±.0915)
SVR	0.5462 (±.0154)	0.5737 (±.0187)	0.5530 (±.0150)
BJ Model	0.5898 (±.0087)	0.5910 (±.0146)	0.6103 (±.0059)
Weibull	0.6139 (±.0075)	0.6171 (±.0069)	0.6315 (±.0166)
logNormal	0.6391 (±.0053)	0.6463 (±.0015)	0.6695 (±.0116)
logLogistic	0.6380 (±.0121)	0.6494 (±.0062)	0.6747 (±.0201)
Cox	**0.6527** (±.0097)	**0.6558** (±.0125)	**0.6797** (±.0062)

Table 3.7.: *Enron* Dataset: TD-AUC results [mean (±standard deviation)] with different splits used for training period.

Method / Split	60%	70%	80%
RidgeReg	0.5732 (±.0073)	0.5847 (±.0159)	0.5318 (±.0164)
LassoReg	0.5740 (±.0076)	0.5850 (±.0152)	0.5309 (±.0178)
FFNN	0.4900 (±.0258)	0.5407 (±.0434)	0.5363 (±.0561)
SVR	0.5490 (±.0080)	0.5680 (±.0176)	0.5608 (±.0136)
BJ Model	0.5292 (±.0120)	0.6096 (±.0076)	0.5599 (±.0121)
Weibull	0.5710 (±.0168)	**0.6319** (±.0050)	**0.5980** (±.0096)
logNormal	0.5713 (±.0146)	0.6146 (±.0097)	0.5862 (±.0129)
logLogistic	0.5787 (±.0171)	0.6224 (±.0069)	0.5917 (±.0101)
Cox	**0.5854** (±.0166)	0.6311 (±.0110)	0.5919 (±.0084)

based models are more sophisticated, which enables them to design complex functions for predicting the time, thus overcoming the under-fitting issue.

For the *MC-Email* dataset, the overall behavior of the models is very similar to the *Epinion* dataset. Here again the Cox regression model performs the best with mean TD-AUC between 0.65 and 0.68 and its results are improved for larger training data. Performance of different AFT models vary, but they all perform better than all of the traditional regression methods. In particular, AFT with log-logistic and log-normal distributions perform great and their mean TD-AUC is very close to the

Table 3.8.: TD-AUC results [mean (±standard deviation)] for various methods on synthetic datasets.

Method	10K	20K	30K
RidgeReg	0.5210 (±.0029)	0.4949 (±.0018)	0.5203 (±.0023)
LassoReg	0.5150 (±.0034)	0.4876 (±.0059)	0.5091 (±.0048)
FFNN	0.4999 (±.0517)	0.4967 (±.0151)	0.5068 (±.0631)
SVR	0.5379 (±.0021)	0.4963 (±.0026)	0.5473 (±.0015)
BJ Model	0.5589 (±.0011)	0.5232 (±.0013)	0.5557 (±.0008)
Weibull	0.5641 (±.0036)	0.4954 (±.0027)	0.5559 (±.0015)
logNormal	**0.5670** (±.0027)	0.4991 (±.0030)	**0.5618** (±.0011)
logLogistic	0.5597 (±.0029)	0.4985 (±.0042)	0.5576 (±.0019)
Cox	0.5604 (±.0025)	**0.5282** (±.0026)	0.5558 (±.0016)

results of the Cox regression as shown in Table 3.6. The performance of all survival models improve as we provide more training data. On the other hand, best among the competing methods is ridge regression with a mean TD-AUC between 0.60 and 0.62. As I have discussed earlier, this model suffers from under-fitting problem.

For the *Enron* dataset, results are shown in Table 3.7. Here, for the training period with 60% split, Cox regression performs the best with mean TD-AUC 0.58. For the other two splits, AFT model with Weibull distribution performs the best with mean TD-AUC 0.63 and 0.59. The BJ model performs poorly compared to the other survival models with mean TD-AUC ranging from 0.52 to 0.6, but the performance of BJ model is still better than all competing regression methods for training period with 70% and 80% splits of time-stamps. For this dataset, for 80% training split, none of the models have better performance than the other splits. This is due to the fact that this dataset is extremely sparse and it has only $3,007$ links created during 944 time-stamps (Table 3.1). Hence even the 80% split does not provide more informative training samples to perform good prediction on remaining data.

The results for synthetic networks are shown in the Table 3.8 by using the mean TD-AUC and standard deviation metrics. As we observe the results in this table, We can easily conclude that survival models always perform better than traditional

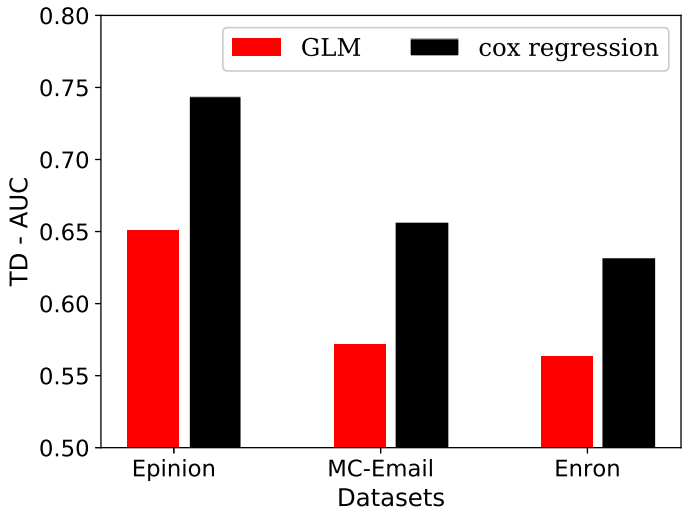


Fig. 3.6.: Comparison of GLM and cox regression

regression methods. For two datasets with $10K$ and $30K$ node instances, the AFT model with log-normal distribution performs the best among all, while for the dataset with $20K$ nodes the Cox regression performs the best. The performance of survival models is consistently very similar except for dataset with $20K$ node where Cox and BJ models clearly perform better than AFT models. Among competing methods, SVR always performs better than others.

3.8.5 Comparison with GLM

Sun et al. [99] proposed a method to predict link generation time in a heterogeneous network, where they design a unique feature for the task and use the feature with generalized linear model (GLM) for the prediction task. This proposed feature is designed based on meta-path (a simple path with link label information) in a heterogeneous network. I adopted this feature for a homogeneous network and the adopted feature can be described as a number of simple paths of size k between two nodes. Counting the number of simple paths is an extremely costly operation especially for a large dataset such as the *Epinion* network; hence for this experiment, I use k upto

5 i.e. $k \in \{2, 3, 4, 5\}$ for all three networks, *Epinion*, *MC-Email* and *Enron*. I provide these homogeneous feature values to GLM (with gamma distribution) to solve the *RLTP* problem. For this experiment I use a 70% split of time-stamps as train period and remaining 30% as test period. The results of this experiment are depicted in Figure 3.6, where GLM is compared with the Cox regression model for all three datasets. From Figure 3.6, I observe that the Cox regression outperforms the GLM model for all three datasets by noticeable margins. I believe, one of the main reasons for the poor performance of the GLM based method is that the feature proposed by [99] is carefully designed for an author-paper based heterogeneous network and its adoption in a homogeneous network is not very useful.

Table 3.9.: Time-Dependent AUC results [mean (±standard deviation)] for survival analysis methods with and without *ever-waiting* Links on real datasets.

Epinion			
model	w/o *ever-waiting*	with *ever-waiting*	%incr
BJ Model	0.4580 (±.0042)	0.7416 (±.0021)	61.94
Weibull	0.4096 (±.0090)	0.5232 (±.1282)	27.73
logNormal	0.4218 (±.0035)	0.4283 (±.0305)	1.53
logLogistic	0.3767 (±.0024)	0.5132 (±.0188)	36.23
Cox	0.4975 (±.0024)	0.7485 (±.0028)	50.45
MC-Email			
model	w/o *ever-waiting*	with *ever-waiting*	%incr
BJ Model	0.4787 (±.0131)	0.6103 (±.0059)	27.51
Weibull	0.5517 (±.0101)	0.6315 (±.0166)	14.46
logNormal	0.6342 (±.0146)	0.6695 (±.0116)	5.56
logLogistic	0.6331 (±.0152)	0.6747 (±.0201)	6.57
Cox	0.6102 (±.0137)	0.6797 (±.0062)	11.38
Enron			
model	w/o *ever-waiting*	with *ever-waiting*	%incr
BJ Model	0.5499 (±.0134)	0.5599 (±.0121)	1.82
Weibull	0.5330 (±.0237)	0.5980 (±.0096)	12.20
logNormal	0.5344 (±.0070)	0.5862 (±.0129)	9.71
logLogistic	0.5379 (±.0053)	0.5917 (±.0101)	10.01
Cox	0.5481 (±.0234)	0.5919 (±.0084)	7.99

Table 3.10.: Time-Dependent AUC results [mean (±standard deviation)] for survival analysis methods with and without *ever-waiting* Links on synthetic datasets.

10K			
model	w/o *ever-waiting*	with *ever-waiting*	%incr
BJ Model	0.5730 (±.0045)	0.5589 (±.0011)	-2.46
Weibull	0.4847 (±.0096)	0.5641 (±.0036)	16.37
logNormal	0.5564 (±.0102)	0.5670 (±.0027)	1.89
logLogistic	0.5546 (±.0128)	0.5597 (±.0029)	0.92
Cox	0.4910 (±.0037)	0.5604 (±.0025)	14.14
20K			
model	w/o *ever-waiting*	with *ever-waiting*	%incr
BJ Model	0.4956 (±.0062)	0.5232 (±.0013)	5.57
Weibull	0.4951 (±.0025)	0.4954 (±.0027)	0.06
logNormal	0.4984 (±.0018)	0.4991 (±.0030)	0.15
logLogistic	0.4965 (±.0055)	0.4985 (±.0042)	0.41
Cox	0.4938 (±.0098)	0.5282 (±.0026)	6.97
30K			
model	w/o *ever-waiting*	with *ever-waiting*	%incr
BJ Model	0.5548 (±.0049)	0.5557 (±.0008)	0.17
Weibull	0.4544 (±.0020)	0.5559 (±.0015)	22.35
logNormal	0.5270 (±.0044)	0.5618 (±.0011)	6.61
logLogistic	0.5243 (±.0042)	0.5576 (±.0019)	6.35
Cox	0.4637 (±.0051)	0.5558 (±.0016)	19.86

3.8.6 Importance of *ever-waiting* links

I conducted experiments to show the importance of *ever-waiting* links and the results are depicted in Tables 3.9 and 3.10. Table 3.9 shows the increment in TD-AUC up to 62% in the real-world datasets, when the survival models are provided with censored information (*ever-waiting* links) during the training, as compared to when the models are trained without censored information. For the *Epinion* dataset, the increment in the results is significant (more than 27% for all models) except AFT with log-normal distribution. Similarly, for the *MC-Email* and the *Enron* datasets the increment is up to 27%, which is substantial. As shown in Table 3.10, for the synthetic datasets we also have very similar increment in the results except for the BJ model with datasets of $10K$ nodes. For the most part the increment in performance is high for the Cox regression and the AFT with Weibull distribution. However, for

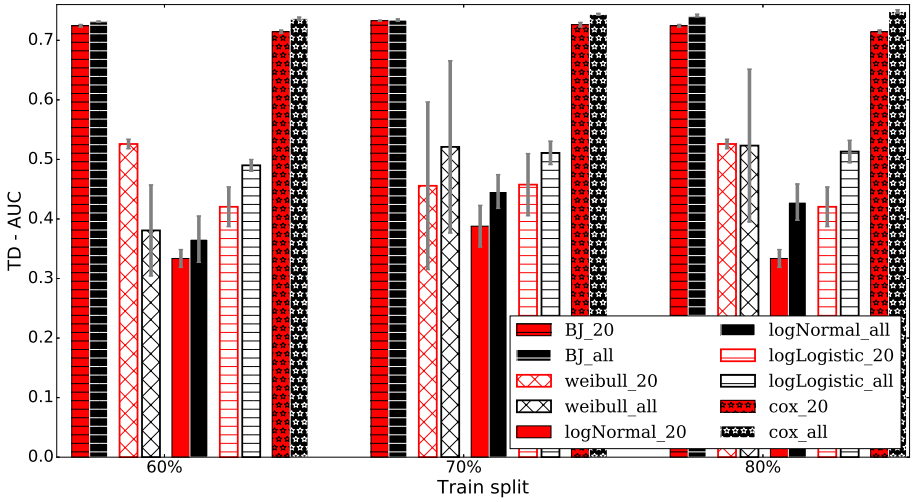


Fig. 3.7.: Epinion Dataset: Comparison of training with top 20% reciprocal links and all reciprocal links.

other models the increment is limited to around 10%. The modest contribution of *ever-waiting* links for the case of synthetic networks can be attributed to the network generation model. I used Durak et al's model [157], which selects pairs of vertices for reciprocal link creation based on only degree distribution without considering any of the social phenomena, so the features that we are using may be not very effective for the synthetic datasets.

3.8.7 Importance of reciprocal links with small interval time

For the *RLTP* problem, reciprocal links carry very useful information and this information is not distributed uniformly over all reciprocal links. I described in Section 3.3 that for most of the reciprocal links the corresponding time interval is relatively small, and very few have high time interval as depicted in Figure 3.2. The reciprocal links for which the corresponding time interval is equal to or less than 20% of the maximum time interval among all the time intervals of reciprocal links in the dataset are called "top 20%" reciprocal links. I trained survival models with top 20%

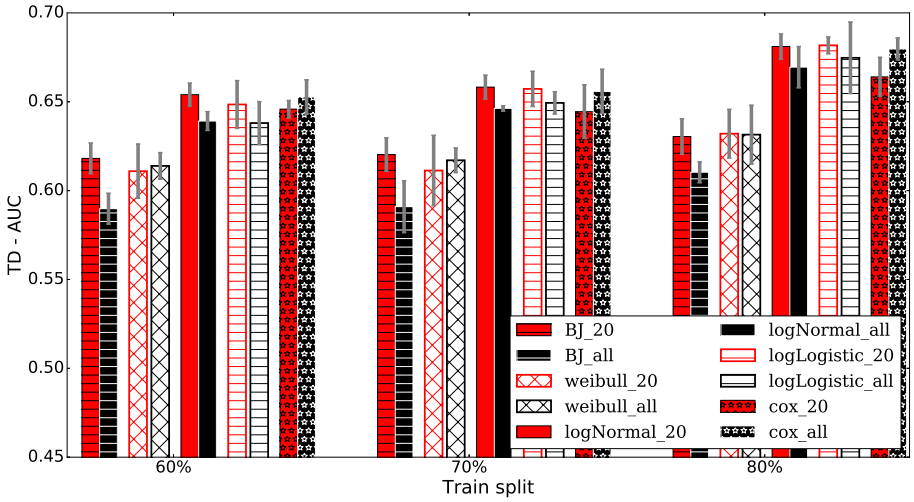


Fig. 3.8.: MC-Email Dataset: Comparison of training with top 20% reciprocal links and all reciprocal links.

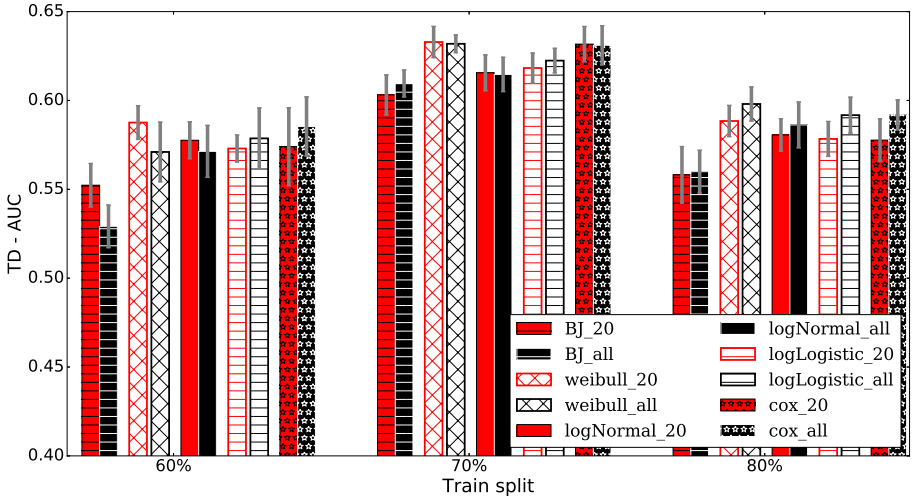


Fig. 3.9.: Enron Dataset: Comparison of training with top 20% reciprocal links and all reciprocal links.

reciprocal links (with *ever-waiting* links) and compared the results of these models with results of models trained with all reciprocal links (with *ever-waiting* links).

Results for these experiments are shown in Figures 3.7, 3.8, 3.9, where all red bars represent different models trained with top 20% reciprocal links and all black bars represent the same models trained using all reciprocal links. We can see that, for all

Table 3.11.: Time-Dependent AUC results [mean (±standard deviation)] for survival analysis methods with top5-features and all features.

Epinion			
model	top5-features	all-features	%incr
BJ Model	0.6541 (±.0027)	0.7339 (±.0020)	12.20
Weibull	0.4878 (±.0872)	0.5210 (±.1446)	6.80
logNormal	0.3157 (±.0062)	0.4461 (±.0283)	41.32
logLogistic	0.3360 (±.0032)	0.5110 (±.0196)	52.10
Cox	0.6292 (±.0056)	0.7436 (±.0016)	18.19
MC-Email			
model	top5-features	all-features	%incr
BJ Model	0.4728 (±.0059)	0.5910 (±.0146)	25.00
Weibull	0.5503 (±.0102)	0.6171 (±.0069)	12.13
logNormal	0.5802 (±.0074)	0.6463 (±.0015)	11.40
logLogistic	0.5917 (±.0125)	0.6494 (±.0062)	9.76
Cox	0.5738 (±.0187)	0.6558 (±.0125)	14.29
Enron			
model	top5-features	all-features	%incr
J Model	0.5995 (±.0061)	0.6096 (±.0076)	1.69
Weibull	0.5985 (±.0140)	0.6319 (±.0050)	5.58
logNormal	0.5972 (±.0130)	0.6146 (±.0097)	2.92
logLogistic	0.6043 (±.0070)	0.6224 (±.0069)	2.99
Cox	0.5964 (±.0069)	0.6311 (±.0110)	5.82

three datasets, survival models trained with top 20% reciprocal links perform very similar or better to the models trained with all reciprocal links. This observation supports my argument that all reciprocal links do not carry same amount of information, but notable amounts of information lie in the reciprocal links with short interval time.

3.8.8 Contribution of Top-5 features

In Section 3.4.3, I study correlation of different features with interval time. Through this experiment, I study the contribution of top five highly correlated features (top5-features) to solve the *RLTP* problem. From the Table 3.2, we can find these top5-features for each real-world dataset. We can see, for *Epinion* dataset $Common_{in}$,

$Common_{out}$, $Jaccard_{In}$, *PrefAtt* and *PageRank(u)* are highly correlated features. Similarly, for both *MC-Email* and *Enron* datasets $Common_{in}$, $Common_{out}$, $Jaccard_{In}$, *PrefAtt* and *PrefJacc* are the top5-features (Table 3.2). For this comparison study, I prepared train and test instances similarly as described in Section 3.8.2 with 70% training split, but the difference is, here each data instance is represented by only corresponding top5-features. I use survival models (Section 3.6) with top5-features data to solve the *RLTP* problem.

Table 3.11 shows results for the comparison experiment with mean TD-AUC value and standard deviation for 5 independent runs. The last column in the Table 3.11 shows the increment in the mean TD-AUC value from top5-features data to all features data. This table clarifies the importance of the other features with lower correlation values (Table 3.2), because for both the *Epinion* and the *MC-Email* datasets the increment in the results is noticeable. But for the *Enron* dataset the increment is not very impressive; I believe low number of data instances and very high correlation of top5-features are the main reasons for this shortcoming.

3.9 Chapter Summary

In this chapter, I proposed a novel problem, namely, reciprocal link time prediction (*RLTP*), which has wide applicability in email, social and other directed networks. I designed various socially meaningful topological features specifically for directed networks, which are useful to solve the *RLTP* problem. I mapped the *RLTP* problem into a survival analysis task and through experiments on three real-life network datasets, we showed that such a framework is better suited than traditional regression based approaches for solving the *RLTP* problem. I demonstrated that using *ever-waiting* links for training adds valuable information to the prediction models. I also investigated the information contributed by the reciprocal links and show that the majority of the required information lies in the top few percent (20%) of the reciprocal links. To the best of my knowledge this is the first study on time interval

prediction for reciprocal links, which is useful to answer response time for emails or friend requests. It can also be used for recommendation in trust networks for suggesting a new connection (parasocial link) for which, the predicted response time is very small.

4. TRIANGLE COMPLETION TIME PREDICTION

4.1 Introduction

In this chapter, I formally define the triangle completion time prediction (*TCTP*) problem and provide a novel framework to solve the *TCTP* problem. As I mentioned in Chapter 1, *TCTP* helps to solve various real-world problems, for example recommendation in an online social network, the user who is going to accept the friend request earlier should be recommended first as illustrated in the Figure 4.1. Also, the knowledge of triangle completion time can also improve the solution of various other network tasks that use triangles, such as, community structure generation [14], designing network generation models [29], and generating link recommendation [30].

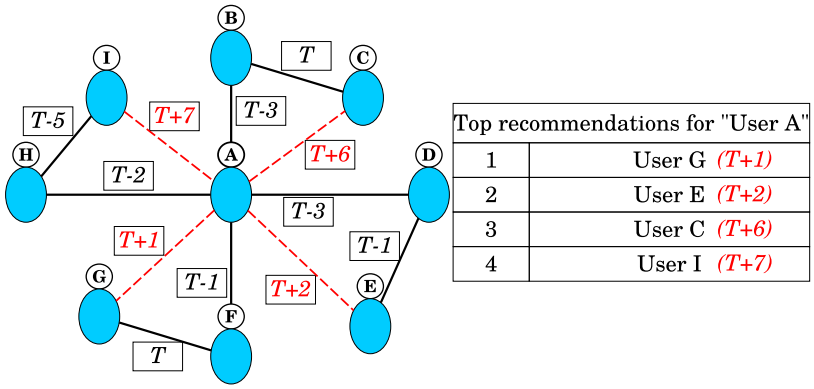


Fig. 4.1.: Simple illustration of the utility of TCTP problem for providing improved friend recommendation. In this figure, user A is associated with 4 triangles, whose predicted completion times are noted as label on the triangles' final edges (red dotted lines). The link recommendation order for A at a time T, based on the earliest triangle completion time, is shown in the table on the right.

Here, I propose a novel framework called *GraNiTE* [1] for solving the *T*riangle *C*ompletion *T*ime *P*rediction (*TCTP*) task. The *GraNiTE* is a network embedding based model, which first obtains latent representation vectors for the triangle completing edges; the vectors are then fed into a traditional regression model for predicting the time for the completion of a triangle. The main novelty of *GraNiTE* is the design of an edge representation vector learning model, which embeds edges with similar triangle completion time in close proximity in the latent space. Obtaining such embedding is a difficult task because the creation time of an edge depends on both local neighborhood around the edge and the time of the past activities of incident nodes. So, existing network embedding models [31] which utilize the neighborhood context of a node for learning its representation vector is inadequate. To achieve the desired embedding, *GraNiTE* develops a novel supervised approach which uses local graphlet frequencies and the edge creation time. The local graphlet frequencies around an edge is used to obtain a part of the embedding vector, which yields a time-ordering embedding. Also, the edge creation time of a pair of edges is used for learning the remaining part of the embedding vector, which yields a time-preserving embedding. Combination of these two brings edges with similar triangle completion time in close proximity of each other in the embedding space. Both the vectors are learned by using a supervised deep learning setup. Through experiments on five real-world datasets, I show that *GraNiTE* reduces the mean absolute error (MAE) to one-forth of the MAE value of the best competing method while solving the *TCTP* problem.

The rest of the chapter is organized as follows. In Section 2, I define the *TCTP* problem formally. In Section 3, I show some interesting observation relating to triangle completion time on five real-world datasets. The *GraNiTE* model is discussed in Section 4. In Section 5, I present experimental results which validate the effectiveness of my model over a collection of baseline models. Section 6 concludes the work.

[1] *GraNiTE* is an anagram of the bold letters in **Gra**phlet and **N**ode based **Ti**me-preserving **E**mbedding.

Table 4.1.: Statistics of datasets (* $\mathcal{T}$ in years for DBLP)

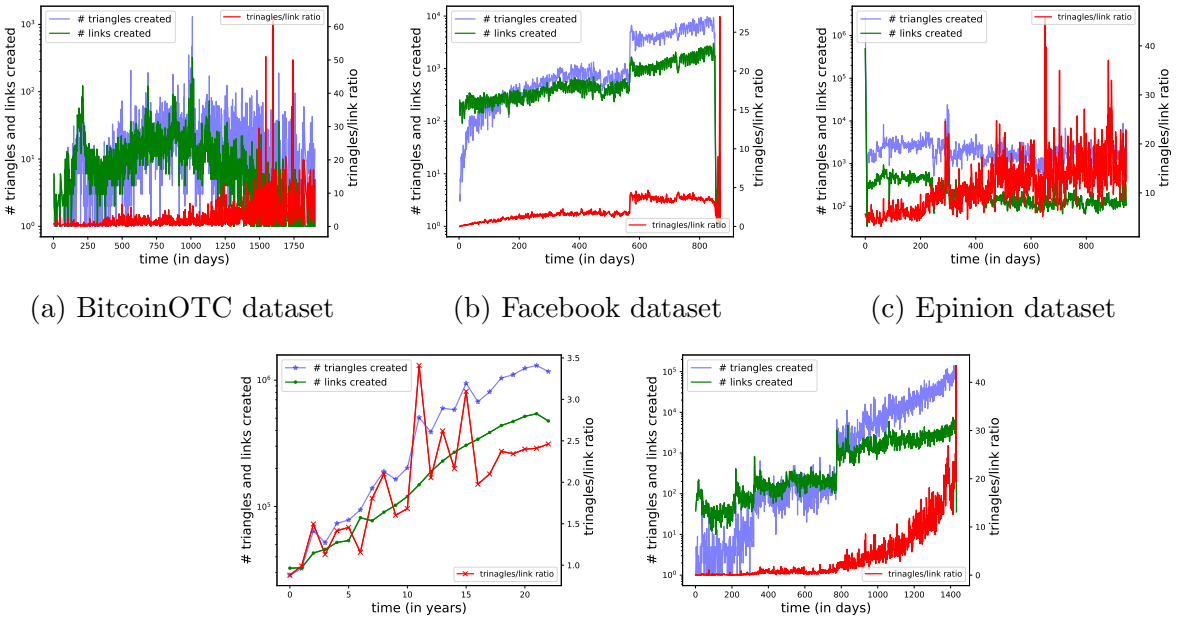
Datasets	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	$\lvert\mathcal{T}\rvert$ (days)	$\lvert\Delta\rvert$
BitcoinOTC	$5,881$	$35,592$	$1,903$	$33,493$
Facebook	$61,096$	$614,797$	869	$1,756,259$
Epinion	$131,580$	$711,210$	944	$4,910,076$
DBLP	$1,240,921$	$5,068,544$	23^*	$11,552,002$
Digg-friend	$279,374$	$1,546,540$	$1,432$	$14,224,759$

4.2 Problem statement

4.2.1 Problem formulation.

Given, a network $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of vertices and $\mathcal{E}$ is a set of edges. For a time-stamped network, given a set of time-stamps $\mathcal{T}$, there also exists a mapping function $\tau : \mathcal{E} \rightarrow \mathcal{T}$, which maps each edge $e = (u, v) \in \mathcal{E}$ to a time-stamp value, $\tau(e) = t_{uv} \in \mathcal{T}$ denoting the creation time of the edge e. A triangle Δ_{abc} in a network is formed by the vertices $a, b, c \in \mathcal{V}$ and the edges $(a, b), (a, c), (b, c) \in \mathcal{E}$. If exactly one of the three edges of a triangle is missing, I call it an open triple. Say, among the three edges above, (a, b) is missing, then the open triple is denoted as Λ^c_{ab}. I use Δ for the set of all triangles in a graph.

Given an open triple Λ^w_{uv}, the objective of *TCTP* is to predict the time-stamp (t_{uv}) of the missing edge (u, v), whose presence would have formed the triangle Δ_{uvw}. But, predicting the future edge creation time from training data is difficult as the time values of training data are from the past. So I make the prediction variable an invariant of the absolute time value by considering the interval time from a reference time value for each triangle, where reference time for a triangle is the time-stamp of the second edge in creation time order. For example, for the open triple Λ^w_{uv} the reference time is the latter of the time-stamps t_{wu}, and t_{wv}. Thus the interval time (target variable) that I want to predict is the time difference between t_{uv} and the reference time, which is $\max(t_{wu}, t_{wv})$. The interval time is denoted by I_{uvw};



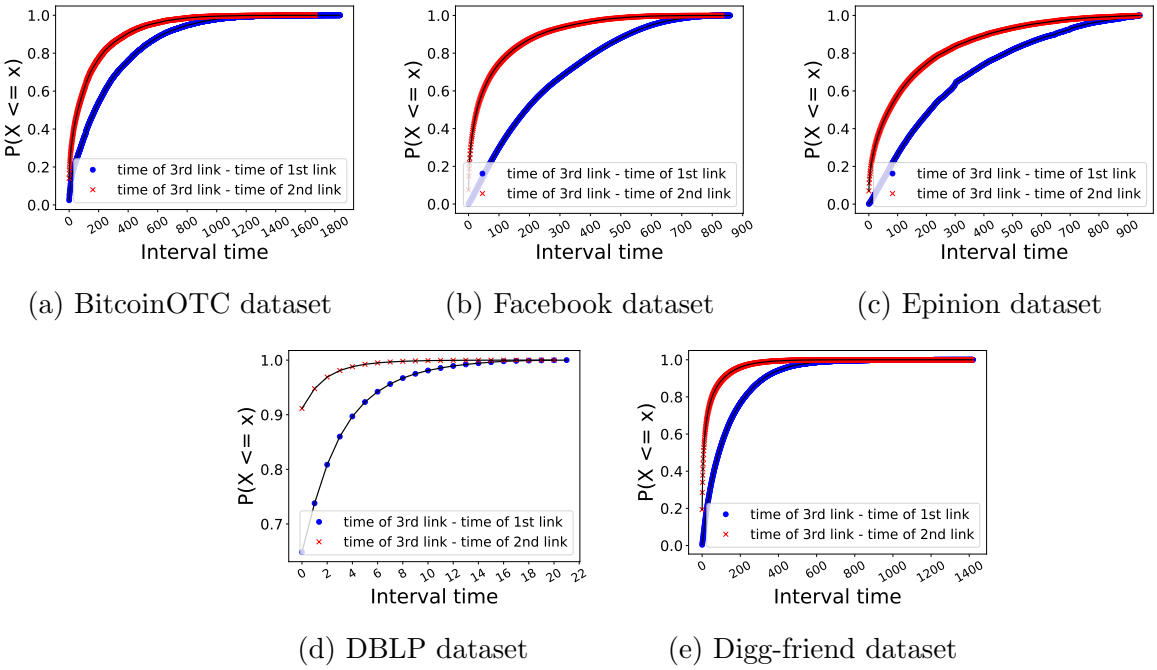
(a) BitcoinOTC dataset (b) Facebook dataset (c) Epinion dataset

(d) DBLP dataset (e) Digg-friend dataset

Fig. 4.2.: Frequency of new edges (green line) and new triangles (blue line) created over time. Ratio of newly created triangle to the newly created link frequency is shown in red line. Y-axis labels on the left show frequency of triangles and link, and the y-axis labels on right show the triangle to link ratio value.

mathematically, $I_{uvw} = t_{uv} - \max(t_{wu}, t_{wv})$. Then the predicted time for the missing edge creation is $t_{uv} = I_{uvw} + max(t_{wu}, t_{wv})$.

Predicting the interval time from a triangle specific reference time incurs a problem, when a single edge completes multiple (say k) open triples, I call such an edge a k-triangle edge. For such a k-triangle edge, ambiguity arises regarding the choice of triples (out of k triples), whose second edge should be used for the reference time—for each of the reference time, a different prediction can be obtained. I solve this problem by using a weighted aggregation approach, a detailed discussion of this is available in Section 4.4.4 “Interval time prediction”.



(a) BitcoinOTC dataset (b) Facebook dataset (c) Epinion dataset

(d) DBLP dataset (e) Digg-friend dataset

Fig. 4.3.: Plots of cumulative distribution function (CDF) for interval times

4.3 Dataset study

The problem of predicting triangle completion time has not been addressed in any earlier works, so before embarking on the discussion of my prediction method, I like to present some observations on the triangle completion time in five real-world datasets. Among these datasets, BitcoinOTC [2] is a trust network of Bitcoin users, other datasets are collected from Konect [3], from which, Facebook and Digg-friend are online friendship networks, Epinion is an online trust network and DBLP is a co-authorship network. Overall information, such as the number of vertices ($|\mathcal{V}|$), edges ($|\mathcal{E}|$), time-stamps ($|\mathcal{T}|$) and triangles ($|\Delta|$) for these datasets are provided in table 4.1.

[2] `http://snap.stanford.edu/data/`
[3] `http://konect.uni-koblenz.de/`

4.3.1 Study of triangle generation rate.

As network grows over time, so do the number of edges and the number of triangles. In this study, my objective is to determine whether there is a temporal correlation between the growth of edges and the growth of triangles in a network. To observe this behavior, I plot the number of new edges (green line) and the number of new triangles (blue line) (y-axis) over different time values (x-axis); Figure 4.2 depicts five plots, one for each dataset. The ratio of newly created triangle count to newly created link count is also shown (red line).

Trend in the plots is similar; as time passes, the number of triangles and the number of links created at each time stamp steadily increase (except Epinion dataset), which represents the fact that the network is growing. Interestingly, triangle to link ratio is also increases. This happens because as a network gets more dense, the probability that a new edge will complete one or more triangles increases. This trend is more pronounced in Digg-friend and DBLP networks. Especially, in Digg-friend network, each link contributes around 20 triangles during the last few time-stamps. On the other hand, for BitcoinOTC, Facebook and Epinion datasets, the triangle to link ratio increases slowly. For Facebook dataset, after slow and steady increase, I observe a sudden hike in all three values around day 570. After investigation, I discovered that, it is a consequence of a newly introduced recommendation feature by "Facebook" in 2008. This feature, exploits common friends information which leads to create many links completing multiple open triples.

4.3.2 Interval time analysis.

For solving *TCTP*, I predict interval time between the triangle completing edge and the second edge in time order. In this study, I investigate the distribution of the interval time by plotting the cumulative distribution function (CDF) of the interval time for all the datasets (blue lines in the plots in Figure 4.3). For comparison, these

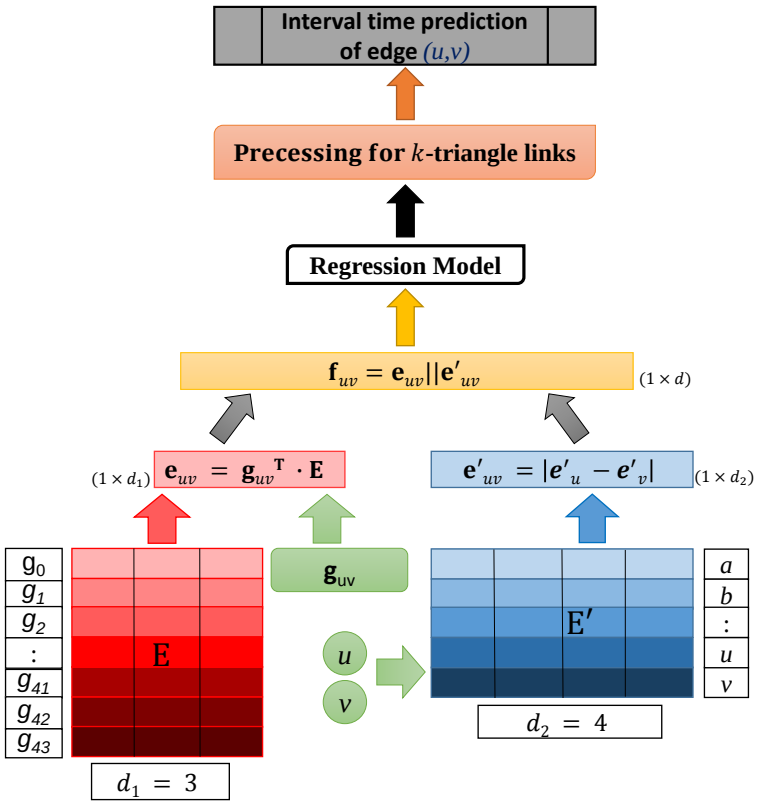


Fig. 4.4.: Interval time prediction for edge (u, v) using Proposed *GraNiTE*.

plots also show the interval time between triangle completing edge and the first edge (red lines).

From Figure 4.3, I observe that for all real-world datasets the interval time between the third link and the second link creation follows a distribution from exponential family; which means most of the third links are created very soon after the generation of the second link. This observation agrees with the social balance theory [20]. As per this theory, triangles and individual links are balanced structures while an open triple is an imbalanced structure. All real-world networks (such as social networks) try to create a balanced structure by closing an open triple as soon as possible; which is validated in Figure 4.3 as the red curve quickly reaches to 1.0 compared to the ascent of the blue curve.

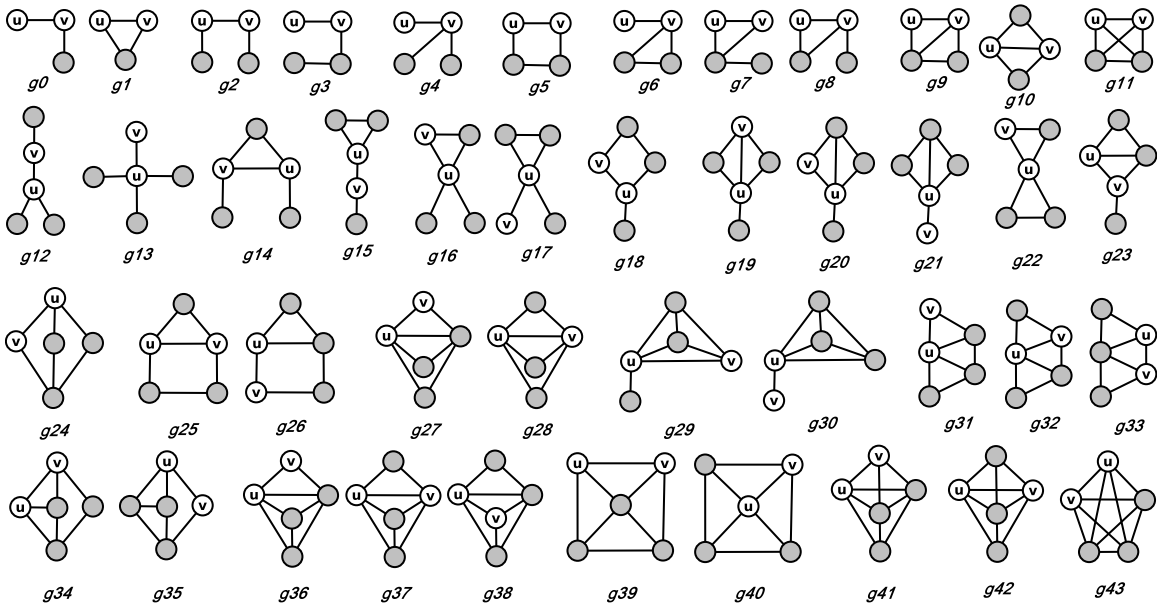


Fig. 4.5.: Local graphlets for given edge (u, v)

4.4 *GraNiTE* Model

GraNiTE model first obtains a latent representation vector for an edge such that edges with similar interval time have latent vectors which are in close proximity. Such a vector for an edge is learned in a supervised fashion by concatenating two kinds of edge embeddings: first, a graphlet-based edge embedding, which embeds the local graphlets into embedding space such that their embedding vectors capture the information of edge ordering based on the interval time. So, I call the edge representation obtained from the graphlet-based embedding method *time-ordering embedding*. Second, a node-based edge embedding that learns node embedding such that proximity of a pair of nodes preserves the interval time of the triangle completing edge. I call the node-based edge embedding *time-preserving embedding*. Concatenation of these two vectors gives the final edge representation vector, which is used to predict a unique interval time for a given edge.

The overall architecture of *GraNiTE* is shown in Figure 4.4. Here nodes u, v and local graphlet frequency vector of edge (u, v) are inputs to the *GraNiTE*. $\mathbf{E}$ and

$\mathbf{E}'$ are graphlet embedding and node embedding matrices, respectively. For an edge (u, v), corresponding time-ordering embedding $\mathbf{e}_{uv} \in \mathbb{R}^{d_1}$ and time-preserving embedding $\mathbf{e}'_{uv} \in \mathbb{R}^{d_2}$ are concatenated to generate final feature vector $\mathbf{f}_{uv} = \mathbf{e}_{uv}||\mathbf{e}'_{uv} \in \mathbb{R}^{d(=d_1+d_2)}$. This feature vector $\mathbf{f}_{uv}$ is fed to a regression model that predicts interval time for (u, v). Lastly, I process the regression model output to return a unique interval time for (u, v), in case this edge completes multiple triangles. In the following subsections, I describe graphlet-based time-ordering embedding and node-based time-preserving embedding.

4.4.1 Graphlet-based Time-ordering Embedding.

In a real world network, local neighborhood of a vertex is highly influential for a new link created at that vertex. In existing works, local neighborhood of a vertex is captured through a collection of random walks originating from that vertex [32], or by first-level and second level neighbors of that node [33]. For finding local neighborhood around an edge I can aggregate the local neighborhood of its incident vertices. A better way to capture edge neighborhood is to use local graphlets (up to a given size), which provide comprehensive information of local neighborhood of an edge [40]. For an edge (u, v), a graphical structure that includes nodes u, v and a subset of direct neighbors of u and/or v is called a local graphlet for the edge (u, v). Then, a vector containing the frequencies of (u, v)'s local graphlets is a quantitative measure of the local neighborhood of this edge. In Figure 4.5, I show all local graphlets of an edge (u, v) up to size-5, which I use in my time-ordering embedding task. To calculate frequencies of these local graphlets, I use *E-CLoG* algorithm [40] of Chapter 5, which is very fast and parallelizable algorithm because graphlet counting process is independent for each edge. After counting frequencies of all 44 graphlets [4], I generate normalized graphlet frequency (NGF), which is an input to my supervised embedding model.

[4]Note that, by strict definition of local graphlet, $g3$ and $g7$ are not local, but I compute their frequencies anyway because these are popular 4-size graphlets.

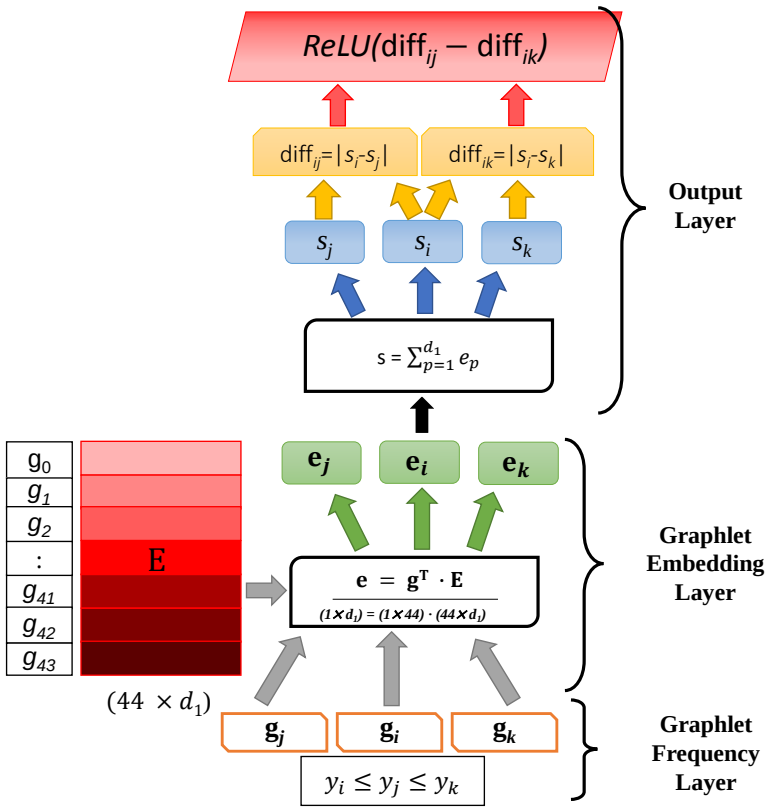


Fig. 4.6.: Learning of the graphlet embedding matrix using three data instances

Graphlet frequencies mimic edge features which are highly informative to capture the local neighborhood of an edge. For instance, the frequency of g_1 is the common neighbor count between u and v, frequency of g_5 is the number of 2-length paths, and frequency of g_{43} is the number of five-size cliques involving both u and v. These features can be used for predicting link probability between the vertex pair u and v. However, these features are not much useful when predicting the interval time of an edge. So, I learn embedding vector for each of the local graphlets, such that edge representation built from these vectors captures the ordering among the edges based on their interval times, so that they are effective for solving the *TCTP* problem. In the following subsection graphlet embedding model is discussed.

Learning Model.

The embedding model has three layers: graphlet frequency layer, graphlet embedding layer and output layer. As shown in the Figure 4.6, graphlet frequency layer takes input, graphlet embedding layer calculates edge embedding for the given set of edges using graphlet embedding matrix and graphlet frequencies, and the output layer calculates my loss function for the embedding, which I optimize by using adaptive gradient descent. The loss function implements the time-ordering objective. Given, three triangle completing edges i, j and k and their interval times, y_i, y_j, and y_k, such that $y_i \leq y_j \leq y_k$, my loss function enforces that the distance between the edge representation vectors of i and j is smaller than the distance between the edge representation vectors of i and k. Thus, the edges which have similar interval time are being brought in a close proximity in the embedding space.

Training data for this learning model is the normalized graphlet frequencies (NGF) for all training instances (triangle completing edges with known interval values), which are represented as $\mathbf{G} \in \mathbb{R}^{m \times g_n}$, where m is the number of training instances and g_n is equal to 44 representing different types of local graphlets. Each row of matrix $\mathbf{G}$ is an NGF for a single training instance i.e. if i^{th} element corresponds to the edge (u, v), $\mathbf{g}_i$ $(= \mathbf{g}_{uv})$ is its normalized graphlet frequency. The target values (interval time) of m training instances are represented as vector $\mathbf{y} \in \mathbb{R}^m$. Now, the layers of the embedding model in Figure 4.6 are explained below:

Graphlet frequency layer: In input layer I feed triples of three sampled data instances i, j and k, such that $y_i \leq y_j \leq y_k$ with their NGF i.e. $\mathbf{g}_i$ $\mathbf{g}_j$, and $\mathbf{g}_k$.

Graphlet embedding layer: This model learns embedding vectors for each local graphlets, represented with the embedding matrix $\mathbf{E} \in \mathbb{R}^{g_n \times d_1}$, where d_1 is the (user-defined) embedding dimension. For any data instance i in training data $\mathbf{G}$, corresponding time-ordering edge representation $\mathbf{e}_i \in \mathbb{R}^{d_1}$ is obtained by vector to matrix multipli-

cation i.e. $\mathbf{e}_i = \mathbf{g}_i^T \cdot \mathbf{E}$. In the embedding layer, for input data instances i, j and k, I calculate three time-ordering embedding vectors $\mathbf{e}_i$, $\mathbf{e}_j$ and $\mathbf{e}_k$ using this vector-matrix multiplication.

Output layer: This layer implements my loss function. For this, first I calculate the score of each edge representation using vector addition i.e. for $\mathbf{e}_i$ the score is $s_i = \Sigma_{p=1}^{d_1} e_i^p$. After that, I pass the score difference between instances i and j (diff_{ij}) and the score difference between i and k (diff_{ik}) to an activation function. The activation function in this layer is ReLU, whose output I minimize. The objective function after regularizing the graphlet embedding matrix is as below:

$$\mathscr{O}_g = \underset{\mathbf{E}}{\boldsymbol{min}} \sum_{\forall(i,j,k)\in T_{ijk}} ReLU(\text{diff}_{ij} - \text{diff}_{ik}) + \lambda_g \cdot \|\mathbf{E}\|_F^2 \tag{4.1}$$

where, $\text{diff}_{ij} = |s_i - s_j|$, λ_g is a regularization constant and T_{ijk} is a training batch of three qualified edge instances from training data.

4.4.2 Time-preserving Node Embedding.

This embedding method learns a set of node representation vectors such that the interval time of an edge is proportional to the l_1 norm of incident node vectors. If an edge has higher interval time, the incident node vectors are pushed farther, if the edge have short interval time, the incident node vectors are close to each other in latent space. Thus, by taking the l_1 norm of node-pairs, I can obtain an embedding vector of an edge which is interval time-preserving and is useful for solving the *TCTP* problem. As depicted in the Figure 4.7, this embedding method is composed of three layers: input layer, node & edge embedding layer, and time preserving output layer. Functionality of each layer is discussed below:

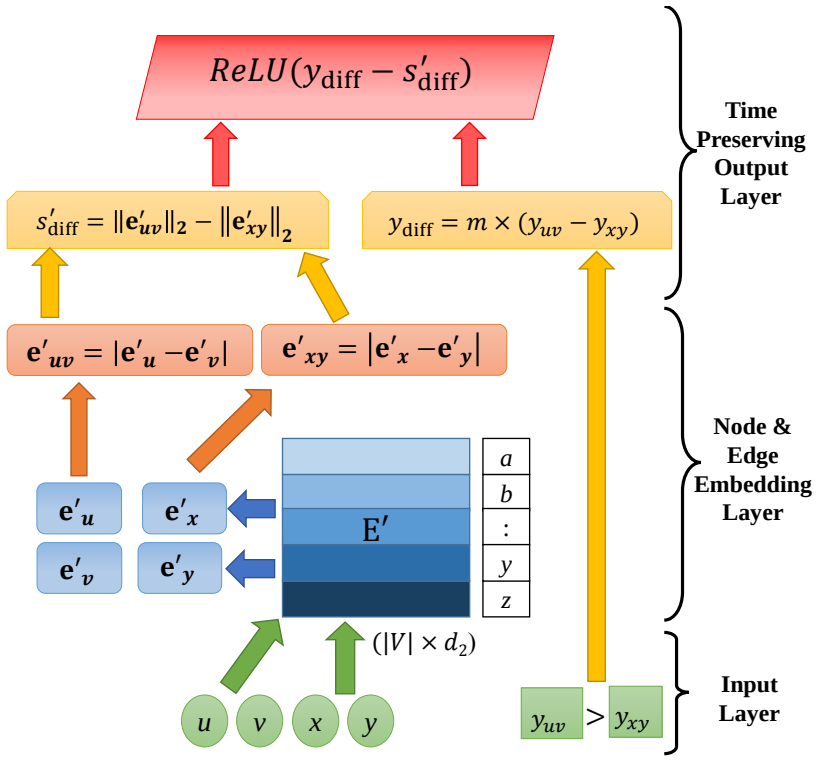


Fig. 4.7.: Learning of the node embedding matrix using two edges (node-pairs).

Input layer: For this embedding approach, input includes two edges, say (u, v) and (x, y) with their interval times, y_{uv} and y_{xy}. The selection of these two edges is based on the criterion that $y_{uv} > y_{xy}$.

Node & edge embedding layer: In this layer, I learn embedding matrix $\mathbf{E}' \in \mathbb{R}^{|\mathcal{V}| \times d_2}$, where d_2 is (user-defined) embedding dimension. From the embedding matrix $\mathbf{E}'$, I find node embedding for a set of 4 nodes incident to the edges (u, v) and (x, y). For any node u, node embedding vector is $\mathbf{e}'_u \in \mathbb{R}^{d_2}$ i.e. u^{th} element of matrix $\mathbf{E}'$. From the node embedding vectors $\mathbf{e}'_u$ and $\mathbf{e}'_v$, I calculate corresponding time-preserving edge embedding vector for (u, v). The time-preserving edge embedding is defined as l_1-distance between the node embedding vectors, i.e. $\mathbf{e}'_{uv} = |\mathbf{e}'_u - \mathbf{e}'_v| \in \mathbb{R}^{d_2}$.

Time-preserving output layer: The objective of this embedding is to preserve the interval time information into embedding matrix, such that time-preserving edge vectors are proportional to their interval time. For that, I calculate an edge score using l_2-norm of an edge embedding, i.e. (u, v) edge score $s'_{uv} = \|\mathbf{e}'_{uv}\|_2$. I design the loss function such that edge score difference $s'_{\text{diff}} = s'_{uv} - s'_{xy}$ between edges (u, v) and (x, y) is proportional to their interval time difference $y_{uv} - y_{xy}$. The objective function of the embedding is

$$\mathscr{O}_n = \min_{\mathbf{E}'} \sum_{\forall (u,v),(x,y) \in T_{uv,xy}} ReLU(y_{\text{diff}} - s'_{\text{diff}}) + \lambda_n \cdot \|\mathbf{E}'\|_F^2 \tag{4.2}$$

where, $y_{\text{diff}} = m \times (y_{uv} - y_{xy})$, λ_n is a regularization constant, $T_{uv,xy}$ is a training batch of edge pairs, and m is a scale factor.

4.4.3 Model inference and optimization.

I use mini-batch adaptive gradient decent (AdaGrad) to optimize the objective functions (Equations 4.1 and 4.2) of both embedding methods. Mini-batch AdaGrad is a modified mini-batch gradient decent approach, where learning rate of each dimension is different based on gradient values of all previous iterations [163]. This independent adaption of learning rate for each dimension is especially well suited for graphlet embedding method as graphlet frequency vector is mostly a sparse vector which generates sparse edge embedding vectors. For time-preserving node embedding, independent learning rate helps to learn the embedding vectors more efficiently such that two node can maintain its proximity in embedding space proportional to interval time.

For mini-batch AdaGrad, first I generate training batch, say T, from training instances. For each mini-batch, I uniformly choose training instances that satisfy the desired constrains: for graphlet embedding, a training instance consists of three

edges i, j and k, for which $y_i \leq y_j \leq y_k$ and for time-preserving node embedding, a training instance is an edge pair, $i = (u, v)$ and $j = (x, y)$, such that, $y_i \leq y_j$ During an iteration, AdaGrad updates each embedding vector, say $\mathbf{e}$, corresponding to all samples from training batch using the following equation:

$$e_i^{t+1} = e_i^t - \alpha_i^t \times \frac{\partial \mathscr{O}}{\partial e_i^t}$$

where, e_i^t is an i^{th} element of vector $\mathbf{e}$ at iteration t. Here I can see that at each iteration t, AdaGrad updates embedding vectors using different learning rates α_i^t for each dimension.

For time complexity analysis, given a training batch T, the total cost of calculating gradients of objective functions ($\mathscr{O}_g$ and $\mathscr{O}_n$) depends on the dimension of embedding vector i.e. $\Theta(d_i)$, $d_i \in \{d_1, d_2\}$. Similarly, calculating learning rate and updating embedding vector also costs $\Theta(d_i)$. In graphlet embedding, I need to perform vector to matrix multiplication, which costs $\Theta(44 \times d_1)$. Hence, total cost of the both embedding methods is $\Theta(44 \times d_1 + d_2) = \Theta(d_1 + d_2)$. As time complexity is linear to embedding dimensions, both embedding methods are very fast in learning embedding vectors even for large networks.

4.4.4 Interval time prediction.

I learn both time-ordering graphlet embedding matrix and time-preserving node embedding matrix from training instances. I generate edge representation for test instances from these embedding matrices, as shown in Figure 4.4. This edge representation is fed to a traditional regression model (I have used Support Vector Regression) which predicts an interval time. However, predicting the interval time of a k-triangle link poses a challenge, as any regression model predicts multiple (k) creation times for such an edge. The simplest approach to overcome this issue is to assign the mean of k predictions as the final predicted value for the k-triangle link. But, as we know mean is highly sensitive to outliers especially for the small number of samples (mostly

$k \in [2, 20]$), so using a mean value does not yield the best result. From the discussion in Section 4.3.2 "Interval time analysis", we know that triangle interval time follows exponential distribution. Hence I use exponential decay $W(I_{uvw}) = w_0 \cdot \exp(-\lambda \cdot I_{uvw})$ as a weight of each prediction, where λ is a decay constant and w_0 is an initial value. I calculate weighted mean which serves as a final prediction value for a k-triangle link.

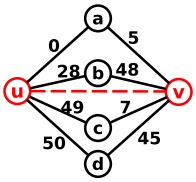


Fig. 4.8.: (u, v) as 4-triangle link

In Figure 4.8, I show a toy graph with creation time of each link and (u, v) is a 4-triangle link. Let's assume this model predicts 4 interval times $(40, 3, 1, 1)$ corresponding to four open triangles $(\Lambda^a_{uv}, \Lambda^b_{uv}, \Lambda^c_{uv}, \Lambda^d_{uv})$ respectively. Hence, I have 4 predicted creation times i.e. $(5 + 40 = 45, 51, 50, 51)$ for link (u, v). So, the final prediction for the edge (u, v) is calculated by using the equation below:

$$\widehat{t_{uv}} = \frac{W(40) \times 45 + W(3) \times 51 + W(1) \times 50 + W(1) \times 51}{W(40) + W(3) + W(1) + W(1)} \tag{4.3}$$

4.5 Experiments and results

I conduct experiments to show the superior performance of the proposed *GraN-iTE* in solving the *TCTP* problem. No existing works solve the *TCTP* problem, so I build baseline methods from two approaches described as below:

The first approach uses features generated directly from the network topology.

1. **Topo. Feat.** (Topological features) This method uses traditional topological features such as common neighbor count, Jaccard coefficient, preferential attachment, adamic-adar, Katz measure with five different β values $\{0.1, 0.05, 0.01,$

$0.005, 0.001\}$. These features are well-known for solving the link prediction task [9]. I generate topological features for an edge (last edge of triangle) from the snapshot of the network when the second link of the triangle appears; triangle interval time is also computed from that temporal snapshot.

2. **Graphlet Feat.** In this framework I use local graphlet frequencies of an edge (last edge of triangle) as a feature set for the time prediction task. These graphlet frequencies are also calculated from the temporal snapshot of the network as mentioned previously in Topo. Feat.

The second approach uses well known network embedding approaches.

3. **LINE** [33]: LINE embeds the network into a latent space by leveraging both first-order and second-order proximity of each node.

4. **Node2vec** [31]: Node2vec utilizes Skip-Gram based language model to analyze the truncated biased random walks on the graph.

5. **GraphSAGE** [58]: It presents an inductive representation learning framework that learns a function and generates embeddings by sampling and aggregating features from a node's local neighborhood.

6. **AROPE** [164]: AROPE is a matrix decomposition based embedding approach, which preserves different higher-order proximity for different input graphs and it provides global optimal solution for a given order.

7. **VERSE** [165]: It is a versatile node embedding method that preserves specific node similarity measure(s) and also captures global structural information.

4.5.1 Experiment settings.

For this experiment, I divide the time-stamps of each dataset into three chronologically ordered partitions with the assumption that initial partition is network growing

Table 4.2.: User parameters for the embedding methods

Datasets	Learning-Rate (α)	Regularization (λ)	Scale-Factor (m)
BitcoinOTC	0.1	0.00001	0.0001
Facebook	0.1	0.0001	1.0
Epinion	0.1	0.000001	0.1
DBLP	0.0001	0.00001	0.01
Digg-friend	0.0001	0.00001	0.0001

period, which spans from the beginning up to 50% of total time-stamps. The second partition, which spans from 50% to 70% of the total time-stamps, is the train period, and finally, from 70% till the end is the test period. I select the edges completing triangles during the train period as training instances and the edges completing triangles during the test period as test instances. I also retain 5% of test instances for parameter tuning (performance on these instances are excluded in the reported results).

There are a few user defined hyper-parameters in the proposed *GraNiTE*. For both embedding approaches, I fix the embedding dimensions as 50, i.e. $d_1 = d_2 = 50$. Hence, the final embedding dimension is $d = 100$ as discussed in Section 4.4 "*GraNiTE* Model". The regularization rates for both embedding methods are set as same value $\lambda_g = \lambda_n$ from set $\{0.000001, 0.00001, 0.0001\}$ using grid search approach. Similarly, initial learning rate for AdaGrad optimization is selected from set $\{0.1, 0.01, 0.001, 0.0001\}$ for both embedding methods. For time preserving node embedding, the scale factor (m) is selected from set $\{1.0, 0.1, 0.01, 0.001, 0.0001\}$ using grid search method. For each dataset, actual parameter values used for the comparison experiments are mentioned in Table 4.2. The training batch size is 100 and the number of epochs is set to 50. Additionally, for predicting time of k-triangle links, decay constant (λ) and initial weight (w_0) are set to 1.0 for calculating exponential decay weights. Lastly, I use support vector regression (SVR) with linear kernel and penalty $C = 1.0$ as a regression method for *GraNiTE* and for all competing methods. For fair comparison, SVR is identically configured for all methods.

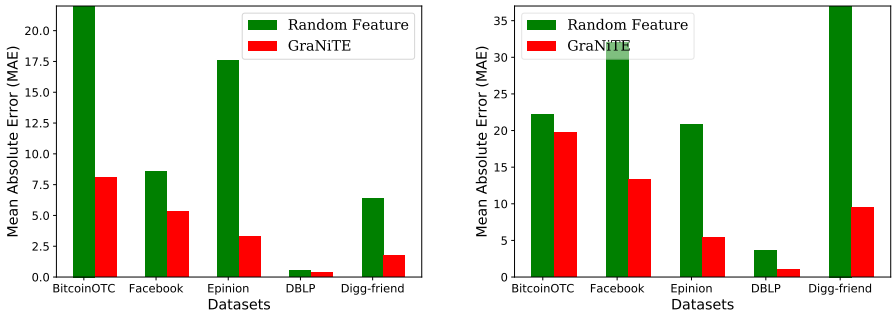
Table 4.3.: Comparison experiment results using MAE for interval times in 1^{st} ($\leq 30days$) and 2^{nd}-month (31-60$days$). [for DBLP dataset: 0-2 years and 3-7 years]. For *GraNiTE*, % improvement over the best competing method (underlined) is shown in brackets.

Dataset		Topo. Feat.	Graphlet Feat.	LINE	Node2vec	GraphSAGE	AROPE	VERSE	*GraNiTE*
Bitcoin-OTC	$\leq 30d$	17.22	17.7	8.86	26.68	11.99	28.62	25.81	**8.08** (0.09%)
	31-60d	21.92	18.29	34.03	**16.55**	28.84	21.59	20.56	19.75 (−19.34%)
	Avg.	19.57	17.995	21.445	21.615	20.415	25.105	23.185	**13.911** (22.7%)
Facebook	$\leq 30d$	7.78	7.93	8.36	7.95	8.37	7.93	7.98	**5.39** (32.46%)
	31-60d	32.04	30.9	31.98	32.87	31.96	32.55	32.73	**13.32** (56.89%)
	Avg.	19.91	19.415	20.17	20.41	20.165	20.24	20.355	**9.355** (51.82%)
Epinion	$\leq 30d$	15.88	14.31	12.52	17.09	13.79	14.3	19.85	**3.28** (73.8%)
	31-60d	22.02	24.82	25.18	20.17	23.45	23.22	17.9	**5.36** (70.06%)
	Avg.	18.95	19.565	18.85	18.63	18.62	18.76	18.875	**4.32** (76.8%)
DBLP	$\leq 30d$	0.526	0.525	0.527	0.527	0.526	0.526	0.5267	**0.379** (27.79%)
	31-60d	3.623	3.618	3.624	3.623	3.623	3.624	3.623	**0.973** (73.11%)
	Avg.	2.0745	2.0715	2.0755	2.075	2.0745	2.075	2.0748	**0.6759** (67.37%)
Digg-friends	$\leq 30d$	6.75	6.25	6.03	7.73	5.95	7.37	6.95	**1.75** (70.59%)
	31-60d	41.06	37.34	38.77	32.66	38.85	34.34	34.75	**9.52** (70.85%)
	Avg.	23.905	21.795	22.4	20.195	22.4	20.855	20.85	**5.635** (72.1%)

For all competing embedding methods the embedding dimensions are set as 100, same size of the feature vector ($d = 100$). I grid search the different tunning parameters to find the best performance of these embedding methods. I select learning rate from set $\{0.0001, 0.001, 0.01, 0.1\}$ for all methods. For Node2vec, I select walk bias factors p and q from $\{0.1, 0.5, 1.0\}$ and number of walks per node is selected from $\{5, 10, 15, 20\}$. For AROPE, the order of proximity is selected from set $\{1, 2, 3, 4, 5\}$. For VERSE, I select personalized pagerank parameter α from set $\{0.1, 0.5, 0.9\}$.

4.5.2 Comparison results.

I evaluate the models using mean absolute error (MAE) over two groups of interval times: 1-month ($\leq$ 30 days) and 2-months (31 to 60 days) for all datasets, except DBLP, for which the two intervals are 0-2 years and 3-7 years. Instances that have higher than 60 days of interval time are outlier instances, hence they are excluded. Besides, for real-life social network applications, predicting an interval value beyond two months is probably not very interesting. Within 60 days, I show results in two



(a) Comparing random feature based method with *GraNiTE*for predicting up to 30 days.

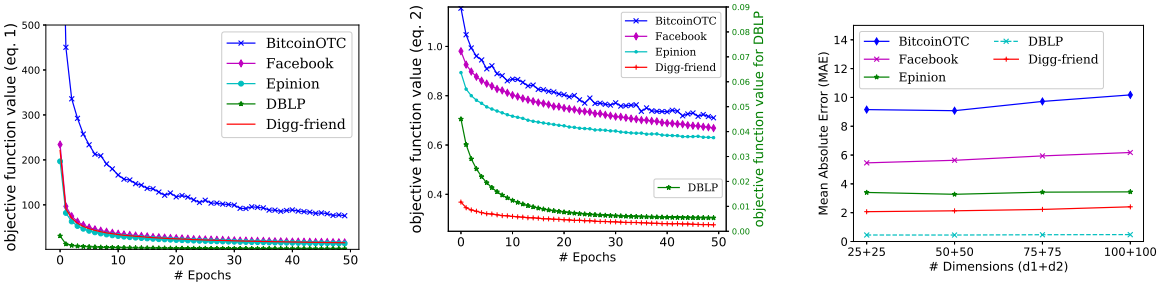
(b) Comparing random feature based method with *GraNiTE* for predicting between 31-60 days.

Fig. 4.9.: Comparing random feature based method with *GraNiTE*.

groups: 1-month, and 2-month, because some of the competing methods work well for one group, but not the other.

First, I compare proposed method with a naive baseline, which is random vectors of the same dimensions (100) as features to solve *TCTP*. This comparison results are shown in Figure 4.9 and we can observe that *GraNiTE* outperforms random features based method for both small and large interval ranges for all dataset. For other baseline methods, comparison results for all five datasets are shown in Table 4.3, where each column represents a prediction method. Rows are grouped into five, one for each dataset; each dataset group has three rows: small interval ($\leq 30d$), large interval (30-60d) and Average (Avg.) over these two intervals. Results of the proposed method (*GraNiTE*) is shown in the last column; besides MAE, in this column I also show the percentage of improvement of *GraNiTE* over the best of the competing methods(underlined). The best results in each row is shown in bold font.

I can observe from the table that the proposed *GraNiTE* performs the best for all the datasets considering the average. The improvements over the competing methods, at a minimum, 22.7% for the BitcoinOTC dataset, and, to the maximum, 76.8% for the Epinion dataset. If I consider short and long intervals ($\leq 30d$ and 30-60d) independently, *GraNiTE* performs the best in all datasets, except BitcoinOTC dataset.



(a) Convergence trends of graphlet embedding

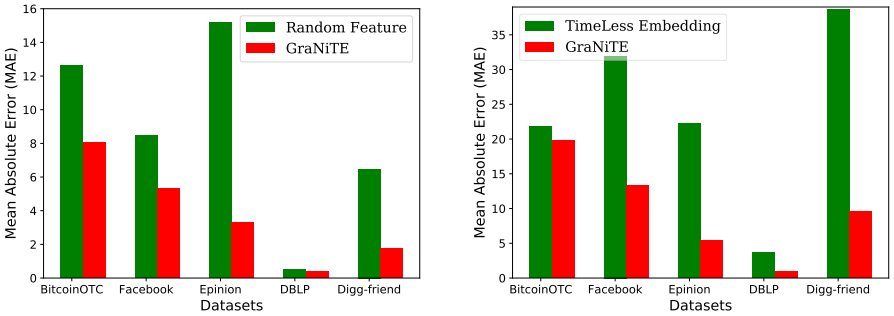
(b) Convergence trends of time preserving node embedding

(c) *GraNiTE* behavior for different embedding dimensions for $\leq 30d$.

Fig. 4.10.: Convergence patterns and dimensionality study

However, notice that for BitcoinOTC dataset, although Node2vec performs the best for large interval times, for small interval times its performance is extremely poor (almost thrice MAE compared to *GraNiTE*). Only *GraNiTE* shows consistently good results for both small and large interval ranges over all the datasets.

Another observation is that, for all datasets, results of large interval times (31-60 days) is worse than the results of small interval time ($\leq$ 30 days). For competing methods, these values are sometimes very poor that it is meaningless for practical use. For instance, for Epinion, each of the competing methods have an MAE around 20 or more for large interval, whereas *GraNiTE* has an MAE value of 5.36 only. Likewise, for Digg-friends, each of the competing methods have an MAE more than 20, but *GraNiTE*'s average MAE is merely 5.63. Overall, for both intervals over all the datasets, *GraNiTE*shows significantly (t-test with p-value $\ll 0.01$) lower MAE than the second best method. The main reason for poor performance of competing methods is that, those methods can capture the local and/or global structural information of nodes/edges but fail to capture temporal information. While for *GraNiTE*, the graphlet embedding method is able to translate the patterns of local neighborhood into time-ordering edge vector; at the same time, time preserving node embedding method is able to capture the interval time information into node embedding vector. Both of the features help to enhance the performance of *GraNiTE*.



(a) Comparing TimeLess embedding method with *GraNiTE*for predicting up to 30 days.

(b) Comparing TimeLess embedding method with *GraNiTE* for predicting between 31-60 days.

Fig. 4.11.: Comparing TimeLess embedding method with *GraNiTE*.

4.5.3 Convergence and dimensionality study.

Here, I study the convergence trend of both embedding methods. As shown in Figure 4.10a, for time-ordering graphlet embedding method each dataset converges quickly i.e. after 15 epochs the objective function value doesn't change much. Similarly, for time-preserving node embedding method all datasets achieve the convergence by 25 epochs as shown in Figure 4.10b.

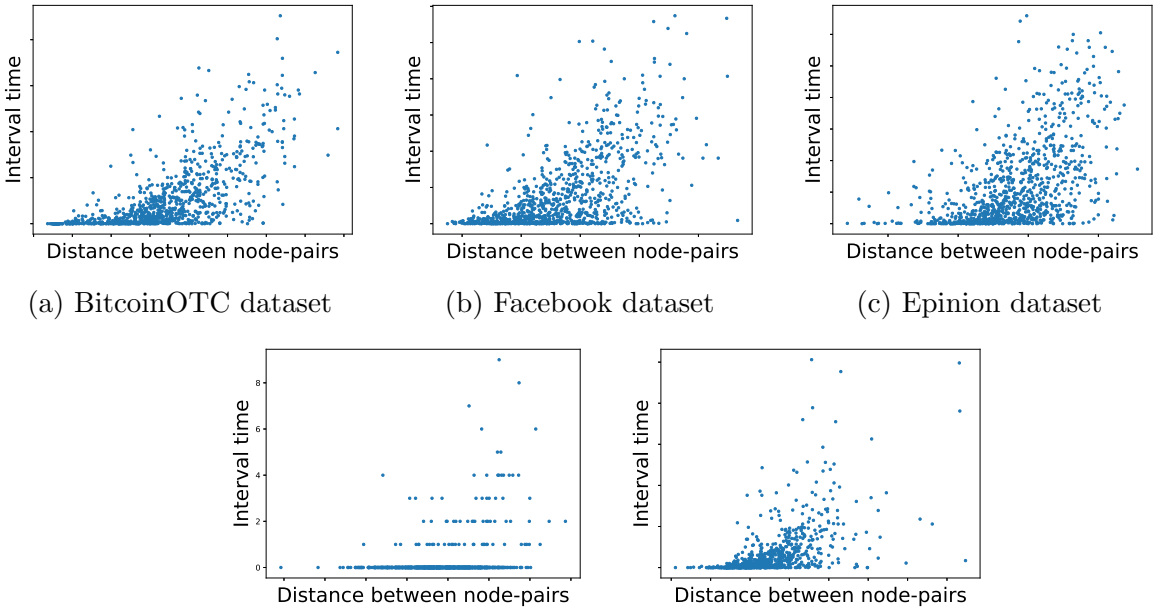
I also study the behavior of the proposed *GraNiTE* for different embedding dimensions. For this study, again I keep the same dimensions for both embedding i.e. $d_1 = d_2$ and select their values from set $\{25, 50, 75, 100\}$. The prediction results for small interval times using each dimension are depicted in Figure 4.10c. From the figure, I can observe that performance for lower dimensions are very similar, but with higher dimensions the model gradually starts having higher test errors, mainly because the more complex model overfits the training instances. I observe that results for large interval times are very similar.

4.5.4 Importance of inclusion of time while learning embedding.

The proposed framework *GraNiTE* uses time information while learning the embedding and hence temporal patterns are encoded into the learned representation vectors. I modified the proposed embedding approach such that it doesn't incorporate timing information during learning. This modified embedding method is similar to the embedding method discussed in Chapter 6, where the triple $\langle x, y, z \rangle$ are sampled such that (x, y) closes a triangle(s) and (x, z) does not. I call this modified method *TimeLess* embedding method. I solve the *TCTP* using the *TimeLess* embedding vectors as features and compare the results with the proposed method *GraNiTE*. The comparison results are depicted in Figure 4.11. The figure shows the superiority of the proposed *GraNiTE* over *TimeLess* embedding method. The performance of the *TimeLess* embedding method is very similar to GraphSAGE method discussed previously in the Section 4.5.2. However, it fails to perform significantly better than all existing baseline methods and *GraNiTE* outperforms the *TimeLess* embedding method. I believe the key reason for this behavior is the absence of the timing information from the representation vectors.

Table 4.4.: Pearson correlation between l_1 distance (in embedding space) and interval times.

Datasets	Train set	Test set
BitcoinOTC	0.62	0.38
Facebook	0.60	0.38
Epinion	0.55	0.38
DBLP	0.24	−0.02
Digg-friend	0.55	0.27



(a) BitcoinOTC dataset (b) Facebook dataset (c) Epinion dataset

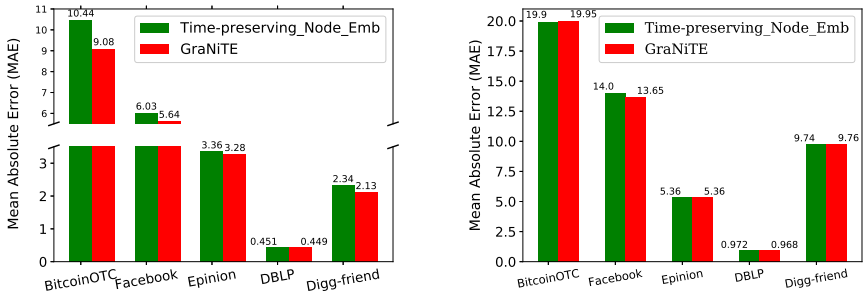
(d) DBLP dataset (e) Digg-friend dataset

Fig. 4.12.: Scatter plots of 1000 random instances to show correlation of l_2-distance with interval times.

4.5.5 Study importance of each embedding approach.

In this section, first I verify my claim that proximity of time-preserving node embedding captures interval time by studying the time to distance relation. Next, I compare the performance of time-preserving node embedding (only) with *GraNiTE* to learn importance of graphlet-based time-ordering embedding.

For the first task, I calculate l_1-distance between node-pair (of a triangle completing edge) in the embedding space and find the Pearson correlation between the l_1-distances and interval times. The results for the experiment are shown in Table 4.4, where I can see that l_1-distance and interval time is highly correlated for all the datasets and for both train and test instances; except for DBLP dataset. For DBLP dataset, the biggest interval time is 9 years, which is a very small number; so this correlation value can be improved by tunning the scale factor (m). For other datasets, I can see the similar behavior of high correlation in Figure 4.12, which de-



(a) Comparing Time-preserving Node embedding performance for predicting upto 30 days.

(b) Comparing Time-preserving Node embedding performance for predicting between 31-60 days.

Fig. 4.13.: Comparing Time-preserving Node embedding with *GraNiTE*.

picts the distance to interval time scatter plots for 1000 random instance from all five datasets.

For the second task, I conduct comparison experiment between *GraNiTE* and time-preserving node embedding (only). In this experiment, for time-preserving node embedding (only), I drop graphlet-based time-ordering embedding from proposed *GraNiTE*, i.e. I consider $\mathbf{f}_{uv} = \mathbf{e}'_{uv}$ in Figure 4.4. For this experiment, I keep the same parameter values and same dimensionality for *GraNiTE* ($d_1 + d_2 = 100$) and time-preserving node embedding ($d_1 = 0$ and $d_2 = 100$). The results for the experiments are depicted in Figure 4.13. I can learn from the Figure 4.13a that for smaller interval time the graphlet-based time-ordering embedding adds important information that reduces the prediction error for *GraNiTE*; while for larger interval times, both methods performs very similar.

4.6 Chapter Summary

Here, I propose a novel problem of triangle completion time prediction (*TCTP*) and provide an effective and robust framework *GraNiTE* to solve this problem by using time-ordering graphlet embedding and time-preserving node embedding methods. Through experiments on five datasets, I show the superiority of my proposed method

compared to baseline methods which use known graph topological features, graphlet frequency features or popular and state-of-art network embedding approaches. To the best of my knowledge, I am the first to formulate the *TCTP* problem and to propose a novel framework for solving this problem.

5. COUNTING EDGE-CENTRIC LOCAL GRAPHLETS

5.1 Introduction

Frequency distribution of small induced subgraphs (aka graphlets) captures key connectivity patterns in a given network, hence this distribution is increasingly being used for various network analysis tasks. For instance, Rahman et al. [34] and Ahmed et al. [38] have used graphlet frequencies for network classification; Ugander et al. [37] have used the same for modeling network structures. Besides these, graphlet frequencies have also been used for solving problems in various other disciplines, examples include biological network comparisons [35,36], image classification [166], and building graph kernels for chemoinformatics [167]. In all the above applications, a vector representing the frequency (normalized or unnormalized) of small-sized graphlets (typically 3 to 5 vertices) induced in a network is used as a signature for capturing the local connectivity patterns of the entire network as a whole. Obtaining such a vector is a costly task, but several recent algorithms have been proposed for solving it efficiently [34, 38, 39]. To overcome the lack of scalability issue, parallel [38, 168] and sampling based approximation algorithms [103, 169] have also been proposed.

Global graphlet frequencies are useful for network analysis tasks where the entire network as a whole needs to be modeled for the task of network-level classification or comparison. Unfortunately, the majority of the real-life network analysis tasks do not consider the network as a whole, rather they consider vertices or edges as first-class entities, and perform prediction on the attributes of nodes or edges. For example, Popular tasks on social networks, such as, expert search, and role discovery, are performed by some form of node classification [64]. On the other hand, tasks like link prediction [9], relationship-type prediction, and product recommendation are solved by edge classification [65]. For solving such tasks, a global graphlet count considering

the entire network is not much useful. I rather want to obtain graphlet counts within the local contex of a vertex or an edge for capturing the topological neighborhood around that vertex or edge. This variant of graphlet counting is called local graphlet counting, which is increasingly being used for network analysis tasks. For instance, V. Vacic et al. [170] have used local graphlet counts to classify protein residues, A. Nabhan et al. [171] have used local graphlet counts for keyword identification from text. Many of the link prediction features, such as Adamic-Adar, Jaccard-Coefficient, and Preferential Attachment are functions of local frequency of a specific graphlet, namely, a triangle.

Although very useful, there are not many works that have considered counting the local graphlet frequencies, beyond triangles. An ad-hoc solution to this deficiency is to restrict a large graph within a local context by building a contextual subgraph and then apply global graphlet count algorithm on that subgraph. For instance Hermansson et al. [172] have built an ego network of a given node and then used global graphlet counting on the ego network to obtain a local graphlet counter. Such a method is an approximation, and more importantly they are overkill because a global graphlet counting method needs to be called for all vertices (for obtaining node-centric graphlet counts) or for all edges (for obtaining edge-centric counting). There exist several works which address local graphlet counting for the case of triangles. For example, Y. Lim et al. [106] and L. De Stefani et al. [107] have proposed estimation method to count local triangles from streaming data. Recently, Ahmed et al. [173] have proposed an efficient parallel algorithm for exact and approximate local graphlet counting upto size 4-vertex graphlets. However, the number of 4-size topologies is usually too small to capture comprehensive topological patterns. Another important shortcoming of the existing local graphlet counting methods is that they do not find the local graphlet counts for different vertex and/or edge orbits. This is a severe shortcoming because orbits represent the role of a vertex or an edge within a given graphlet. An aggregated count of a graphlet, without considering orbits, misrepresents the local topological configuration around a vertex or an edge.

I provide an efficient method, namely *E-CLoG* [1] for obtaining local graphlet counts with respect to a given edge in a graph. *E-CLoG* counts all $3, 4$, and 5-sized local graphlets considering all possible edge orbits of a graphlet. By considering edge orbits I count 8 size-4 and 32 size-5 graphlets. The method is efficient because it does not enumerate all of the above graphlets, rather it only enumerates 4 out of 8 size-4 graphlets, and 14 out of 32 size-5 graphlets. The remaining graphlet counts are obtained in constant time by combinatorial calculation over carefully-designed data structures. I also provide a parallel implementation of *E-CLoG*, which is highly scalable. In this chapter,

1. I propose *E-CLoG*, a highly efficient method for counting edge-centric local graphlets up-to size-5 considering edge orbits. To the best of my knowledge, *E-CLoG* is the first work that obtains local graphlet counts for size-5 graphlets. It is also the first work which considers edge orbits in its counts.

2. I also provide a shared-memory, multi-core implementation of *E-CLoG*, which makes it even more scalable for very large real-world networks.

3. I show experiments which validate *E-CLoG*'s effectiveness as local features for network analysis, specifically for the task of link prediction.

5.2 Problem formulation

Let $G(\mathcal{V}, \mathcal{E})$ be a simple, undirected graph, where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of edges. For a vertex $u \in \mathcal{V}$, neighbors of u are represented by $\Gamma(u) = \{v | (u, v) \in \mathcal{E}\}$ and degree of u is represented by $d(u) = |\Gamma(u)|$.

Definition 5.2.1 (Induced graphlet) *An undirected graph $g(\mathcal{V}', \mathcal{E}')$ is an induced graphlet of $G(\mathcal{V}, \mathcal{E})$, if V' is a subset of $\mathcal{V}$ and $\mathcal{E}'$ consists of all of the edges in $\mathcal{E}$ that have both endpoints in $\mathcal{V}'$. If $|\mathcal{V}'| = k$, we call it k-size induced graphlet, or in-short, k-graphlet in G. I also consider that all k-graplets are connected.* ■

[1] *E-CLoG* stands for **E**dge **C**entric **Lo**cal **G**raphlet.

In this work, given a graph $G(\mathcal{V}, \mathcal{E})$ and a specific edge $e = (u, v) \in \mathcal{E}$, we count the frequency of edge centric local graphlets. Such an edge centric local graphlet (say, g) must include the given edge e; besides, the remaining vertices of g must be a neighbor of u and/or a neighbor of v. Below is a formal definition of edge-centric local graphlets.

Definition 5.2.2 (Edge centric local graphlet) *Say, $g(\mathcal{V}', \mathcal{E}')$ is a k-graphlet of a graph $G(\mathcal{V}, \mathcal{E})$. Now, given an edge $(u, v) \in \mathcal{E}$, g is called an edge centric local graphlet, if $(u, v) \in \mathcal{E}'$ and $w \in \mathcal{V}' \setminus \{u, v\} \Rightarrow w \in \Gamma(u) \cup \Gamma(v)$.* ■

Say, $\mathcal{G}_k(u, v)$ is the set of all connected graphical topologies such that each member of this set is a possible k-size edge centric local graphlet with respect to an edge (u, v). In Figures 5.1 we show all members of $\mathcal{G}_3(u, v)$ and $\mathcal{G}_4(u, v)$, and in Figure 5.3, I show all members of $\mathcal{G}_5(u, v)$. In each of these graphlets, the edge (u, v) is clearly identified. Notice that, if we ignore the vertex labels u and v, there are multiple graphlets in these figures, which are topologically identical. For examples, in Figure 5.1 graphlets $g7$ and $g8$ have the same 4 size graph structure (lets call it two-triangles), but they are represented as two different graphlets ($g7$ and $g8$). I identify these structurally identical graphlets differently, based on edge orbit of the given edge (u, v). Below I formally define edge orbit and other necessary terminologies.

Definition 5.2.3 (Graphlet Isomorphism) *A graphlet $g_1(\mathcal{V}_1, \mathcal{E}_1)$ is said to be isomorphic to another graphlet $g_2(\mathcal{V}_2, \mathcal{E}_2)$, if there exists a bijective function $f_{iso} : \mathcal{V}_1 \to \mathcal{V}_2$ such that $(u, v) \in \mathcal{E}_1 \Leftrightarrow (f_{iso}(u), f_{iso}(v)) \in \mathcal{E}_2$ and the bijection function f_{iso} is called graphlet isomorphism from g_1 to g_2.* ■

Definition 5.2.4 (Graphlet Automorphism) *A graphlet automorphism is graphlet isomorphism with itself, i.e. a bijection function f_{auto} that maps the set of vertices $\mathcal{V}_1$ back to $\mathcal{V}_1$ with a different permutation and satisfies the prosperities of graphlet isomorphism.* ■

Definition 5.2.5 (Edge orbit) *For a given graphlet* $g_1(\mathcal{V}_1, \mathcal{E}_1)$*, edges* $(u, v), (u', v') \in \mathcal{E}_1$ *are in same orbit if an automorphism* f_{auto} *of* g_1 *exists such that* $u = f_{auto}(u')$ *and* $v = f_{auto}(v')$. ■

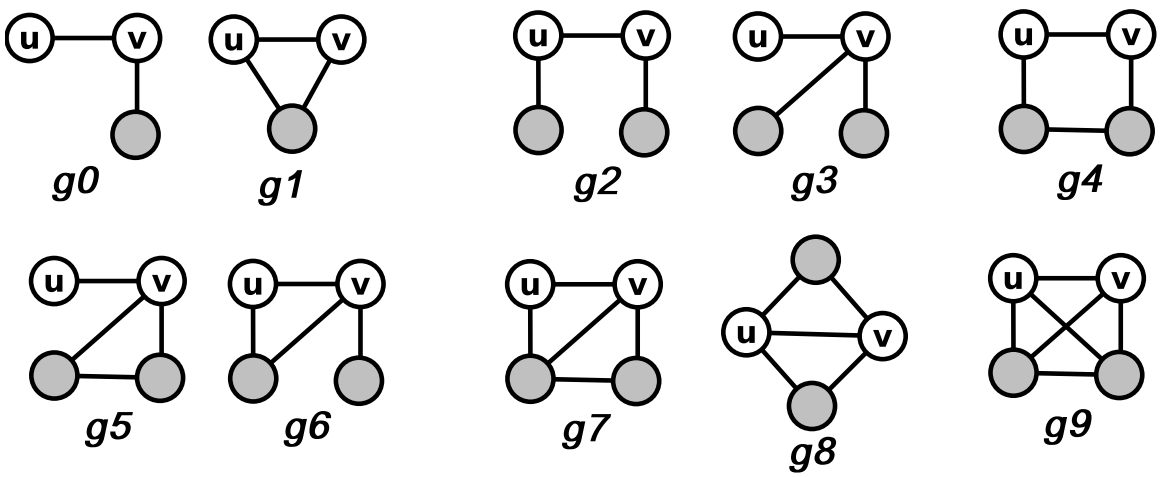


Fig. 5.1.: 3-4 size local graphlets

Fig. 5.2.: Example of 4 size non local graphlets. Left: a non-local edge orbit of 4-path, structurally identical to $g2$; right: a non-local edge orbit of tailed-triangle, structurally identical to $g5$ and $g6$.

Notice that, if two edges are in the same orbit they must have the same pair of degrees for their incident vertices. On the other hand, if two edges do not have the same pair of degrees then they are definitely in different orbits. For example, in Figure 5.1 graphlet types $g7$ and $g8$ are separated based on the orbit of (u, v), here $d(u) = 2, d(v) = 3$ in $g7$ and $d(u) = 3, d(v) = 3$ in $g8$.

In this study, each local graphlet, separated by structure or edge orbit of the given edge, is referred as a graphlet type or a graphlet, and I obtain its count. For instance in Figure 5.1, there are 10 ($g0 - g9$) graphlet types. Notice that, I do not count some graphlet types which have different edge orbits because they are not identified as a local graphlet. For example in Figure 5.2 both 4-path and tailed-triangle graphlets have different edge orbits than corresponding graphlet types in Figure 5.1, but they are not local because the vertex y in both the graphlets in Figure 5.2 is neither neighbor of u nor neighbor of v. Hence, I do not count frequency for such non-local graphlet types.

5.2.1 Problem definition

Given an edge (u, v) of an undirected graph $G(\mathcal{V}, \mathcal{E})$, my goal is to count the number of appearances of each k $\{3, 4, 5\}$ size local graphlet type for the edge (u, v) in the graph G.

5.3 Proposed method

In existing works, there are two distinct philosophies for counting graphlets: counting by enumeration and counting with algebraic expressions. In the first philosophy, each of the graphlet instances are enumerated at least once. It becomes very costly for large graphs as the number of graphlets in these graphs easily exceeds hundreds of billions. On the other hand, counting with algebraic expression is cheap as counting is performed by using a combinatorial approach. Unfortunately, combinatorial counting is easy for size-3 or size-4 graphlets, but it becomes difficult for size-5 graphlets because the number of graphlet configurations is substantially higher for the case of size-5 than size-4. Also, for the case of local graphlet counting, edge orbits need to be considered, which makes it even more difficult.

The proposed method for local graphlet counting is a hybrid approach, which enumerates only a subset of local graphlets and obtains the count of the remaining

local graphlets in constant time by using algebraic expression. Another key innovation of my method is that it utilizes the count of sub-graphlets to efficiently compute the count of a larger size graphlet, which contains the sub-graphlet. Thus, my method first counts the size-3 local graphlets, and then use these to count size-4 local graphlets, and then iteratively use those counts to count size-5 local graphlets. Finally, the proposed method uses a generic counting algorithm, which counts the frequency of different graphlets by setting the value of a small number of template variables, which makes the method easy to understand and implement. In Section 5.3.1, I first discuss the method used to count 3- and 4-size local graphlet types. Then, in Section 5.3.3, I discuss the method for counting 5-size graphlets. In Table 5.1 I show all the notations that are used in the discussion. Note that, for remaining discussion in this chapter, not all sets are shown in calligraphic uppercase letter, for example $\Gamma(u), T, N_u, N_v, T_i, N_{ui}, N_{vi}, S_1, S_2$ and S_3 are all various sets as shown in Table 5.1.

5.3.1 3 and 4 sized local graphlet counting

Given an edge (u, v), counting size-3 local graphlets is easy. There are only two such graphlets, open triple (g_0) and triangle (g_1), and both the structures have only one edge orbit. I count them from $\Gamma(u)$ and $\Gamma(v)$ by finding three disjoint sets of vertices: T, N_u, and N_v, where $T = \Gamma(u) \cap \Gamma(v)$, $N_u = \Gamma(u) \setminus T$, and $N_v = \Gamma(v) \setminus T$. Note that, vertices u and v are not included in any of these sets, i.e. $u, v \notin (T \cup N_u \cup N_v)$, this is because I consider only simple graph without any self loop so no vertex is a neighbor of itself. Also T, N_u and N_v are pair-wise disjoint i.e.

$$N_u \cap T = \phi \qquad \& \qquad N_v \cap T = \phi \qquad \& \qquad N_u \cap N_v = \phi$$

T contains the vertices which are neighbors of both u and v, thus forming a triangle and N_u contains neighbors of u which are not neighbors of v, so these vertices are terminal vertices of open triples centered at u, and identically, N_v contains the ter-

Table 5.1.: Summary of the notations

Notations	Meaning
(u, v)	Edge for which local graphlets are being computed
$d(u)$	Degree of the node u
$\Gamma(u)$	Set of neighboring nodes of the node u
T	Set of nodes creating triangles with edge (u, v)
N_u	Set of nodes that is only neighbor of u not v
N_v	Set of nodes that is only neighbor of v not u
i, j, k	Variables, instantiated as 3^{rd} , 4^{th} and 5^{th} nodes (after u and v) of 5 size graphlets.
T_i	Set of neighboring nodes of i which are also in T
N_{ui}	Set of neighboring nodes of i which are also in N_u
N_{vi}	Set of neighboring nodes of i which are also in N_v
S_1, S_2, S_3	Set variables, which take set as a value, depending on the membership of i, j and k, respectively.
l_1, l_2, l_3	Binary variables, $l_1 = 1$ if i, j is connected, otherwise 0 similarly, $l_2 = 1 \mid l_3 = 1$ if $i, k \mid j, k$ is connected, otherwise 0.
$C(i, j)^{S_1,S_2,S_3}_{l_1,l_2,l_3}$	Count of set of graphlet type(s) for given i, j and different values of S_3, l_2, l_3.
t	Takes values from $\{1, 2, 3, 4\}$, based on values of l_2 and l_3 (1 for 11,2 for 01, 3 for 10, and 4 for 00).
b_i	Bias value need to be deducted to maintain the $i \neq j \neq k$ property.
d_i, d	Dividing factor(s) to handle duplicate counting.

minal vertices of open triples centered at v. Then $f0 = |N_u| + |N_v|$ and $f1 = |T|$. This completes the counting of size-3 graphlets.

There are total eight size-4 local graphlets, of which I enumerate 4 of them, and I obtain the count of the remaining graphlets in constant time by using algebraic expression. Besides, for efficient enumeration, I use the set T, N_u and N_v which I obtain while counting the size-3 local graphlets.

Counting $g9$ (4-clique): Any vertex $x \in T$ forms a triangle with (u, v). Now, if another vertex $y \in \Gamma(x)$ also forms a triangle with (u, v), the induced topology (u, v, x, y) forms a 4-clique. To avoid double counting, I use the condition that the identifier of x is higher than that of y.

Counting $g4$ (4-cycle): If $x \in N_u$, y is a neighbor of x, and $y \in N_v$, then (u, v, x, y) forms a 4-cycle.

Counting $g5$ (tailed-triangle): This graphlet has two edge orbits, $g5$ and $g6$, of which I enumerate $g5$ (tailed-triangle with (u, v) as tail), and obtain $g6$ in constant time. If, $x \in N_u$, and y is a neighbor of x, and $y \in N_u$, then (v, u, x, y) forms a $g5$, in which v is the tail vertex. To avoid double counting, I use the constraint $x > y$. A symmetric enumeration where $x \in N_v$ obtains tailed-triangle with u as the tail vertex.

Counting $g7$ (two-triangles): This graphlet has two edge orbits, $g7$ and $g8$, of which I enumerate $g7$ and obtain $g8$ in constant time. If $x \in N_u$, and $y \in \Gamma(x)$, and $y \in T$, (u, x, y, v) forms a two triangle with (u, y) as the diagonal edge. A symmetric enumeration where $x \in N_v$ obtains two-triangles with (v, y) as the diagonal edge.

The frequencies of the remaining four graphlet types $(g2, g3, g6, g8)$ can be calculated in constant time by using the following equations:

$$f2 \leftarrow |N_u| \times |N_v| - f4 \tag{5.1}$$

$$f3 \leftarrow \binom{|N_u|}{2} + \binom{|N_v|}{2} - f5 \tag{5.2}$$

$$f6 \leftarrow |T| \times f0 - f7 \tag{5.3}$$

$$f8 \leftarrow \binom{|T|}{2} - f9 \tag{5.4}$$

The detailed steps for the method are available in Algorithm 2. In this algorithm I use f to represent the frequency vector that contains frequency of all the graphlet types (size $3, 4$ and 5) and fN represents frequency of a specific graphlet type gN. In the algorithm lines 5-14 generates three distinct sets T, N_u and N_v. The map data structure, *status_map*, maps a vertex to values α, β, or γ to represent the membership from sets T, N_u, or N_v, respectively. In the Algorithm 2, lines 17-20 shows iteration over elements of T and counting 4-clique frequency and lines 21-28 (29-34) show iteration over elements of N_u (N_v) to count frequencies of graphlet types $g4, g5$, and

Algorithm 2: GET_3 − 4_GRAPHLETCOUNT(G, u, v)

```
initialize frequencies                                          // f0 − f41 ← 0
initialize unique neighbor sets of u and v                      // N_u ← {} , N_v ← {}
initialize common neighbor set                                  // T ← {}
initialize status of each vertex in G                           // for each x do status_map(x) ← φ
for each x ∈ Γ(u) do
    if x ≠ v then
        N_u ← N_u ∪ x, status_map(x) ← α                        // α represents membership of N_u
for each x ∈ Γ(v) do
    if x ≠ u then
        if status_map(x) = α then
            T ← T ∪ x, status_map(x) ← γ                        // overwrite α with γ (membership of T)
            N_u ← N_u \ x                                       // remove the element x from N_u
        else
            N_v ← N_v ∪ x, status_map(x) ← β                    // β represents membership of N_v
f0 ← |N_u| + |N_v|                                              // number of unique neighbors (no triangles)
f1 ← |T|                                                        // number of triangles
for each x ∈ T do
    for each y ∈ Γ(x) do
        if status_map(y) == γ and y < x  then
            f9 ← f9 + 1                                         // 4clique
for each x ∈ N_u do
    for each y ∈ Γ(x) do
        if status_map(y) == α and y < x then
            f5 ← f5 + 1                                         // tail-triangle with (u, v) as tail
        else if status_map(y) == β then
            f4 ← f4 + 1                                         // 4cycle
        else if status_map(y) == γ then
            f7 ← f7 + 1                                         // 2triangles with (u, y) as diagonal link
for each x ∈ N_v do
    for each y ∈ Γ(x) do
        if status_map(y) == β and y < x then
            f5 ← f5 + 1                                         // tail-triangle with (u, v) as tail
        else if status_map(y) == γ then
            f7 ← f7 + 1                                         // 2triangles with (v, y) as diagonal link
calculate f2, f3, f6, f8 using Equations 5.1, 5.2, 5.3, 5.4.
return  f, T, N_u, N_v, status_map
```

$g7$. Note that, this algorithm has some similarity to the algorithm of Ahmed et al. [173]. However, their work does not consider counting all edge orbits. Further, their work only considers upto size 4-vertex graphlets, which is substantially simpler than the case of size-5 graphlets, which I discuss next.

5.3.2 5-size Local Graphlet Counting

For counting 5-size local graphlet types, I utilize the enumeration of 4-size local graphlets. Given edge (u, v), and a 4-size local graphlet, we first obtain all feasible fifth nodes, and based on the connections of the fifth node with the 4-size graphlet nodes, I identify and count different 5-size graphlet types. This method appears

straight-forward, but counting all 5 size local graphlet types efficiently and avoiding enumeration of all graphlet types is a challenging task. This method enumerates only 14 local graphlet types of size 5 and calculates other 18 graphlet types in constant time using algebraic expressions. See the Table 5.2.

Table 5.2.: List of enumerating and non-enumerating 5 sized local graphlet types

Enumerated	$g11, g14, g17, g20, g22, g23, g26, g27, g28,$ $g33, g34, g37, g39, g41$
Not enumerated	$g10, g12, g13, g15, g16, g18, g19, g21, g24,$ $g25, g29, g30, g31, g32, g35, g36, g38, g40$

A key observation regarding a size-5 local graphlet for a given edge (u, v) is that the remaining three vertices of this graphlet must be from T, N_u, or N_v. This is due to the definition of local edge-centric graphlet, which requires that the remaining three vertices to be neighbors or u, or v or both (see Definition 5.2.2). I denote these vertices as i, j and k, such that they are distinct, i.e., $i \neq j \neq k$; the 3^{rd} vertex is represented as i, the 4^{th} as j and the 5^{th} as k. Now, there are total 9 different combinations based on the joint membership of i, j, and k within the sets T, N_u, and

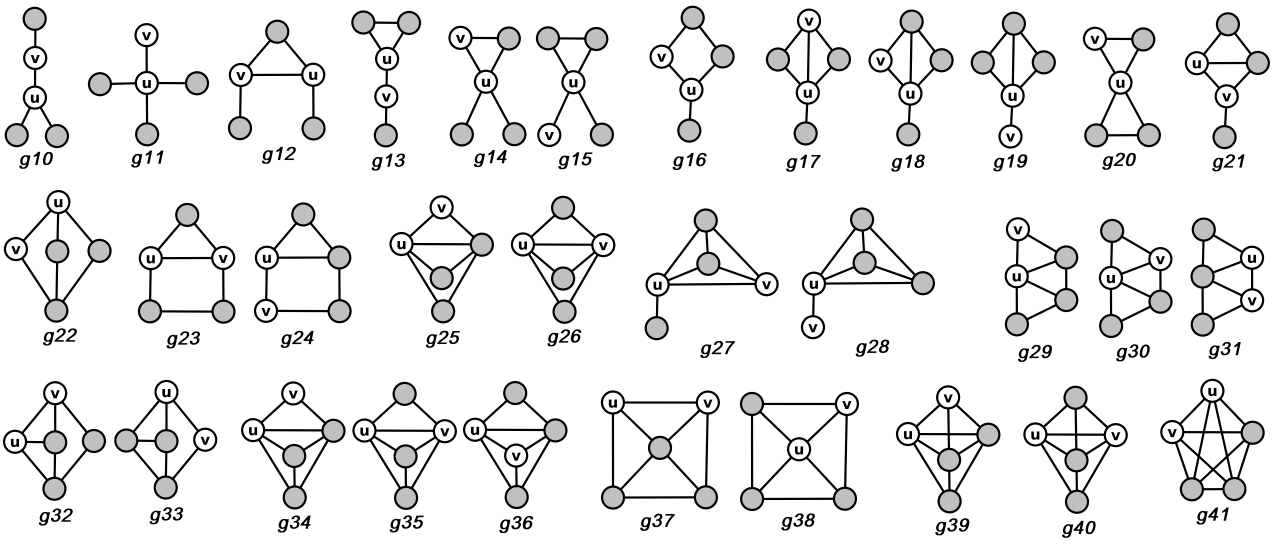


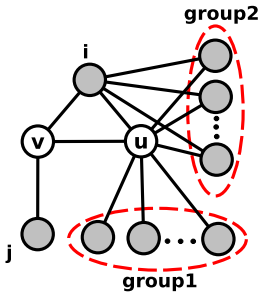
Fig. 5.3.: 5 size local graphlets

N_v and each of these combinations lead to different sets of local graphlets. Besides, these combinations are exhaustive i.e., they cover each and every local edge-centric graphlet given the edge (u, v). Also, there is no false-positive, that is every choice of three vertices i, j, and k within these the sets T, N_u and N_v leads to a valid graphlet which I need to consider in my counting. So, for counting all size-5 local graphlets given (u, v), if suffices that I count all graphlets by efficiently enumerating i, j, and k over the three sets T, N_u and N_v.

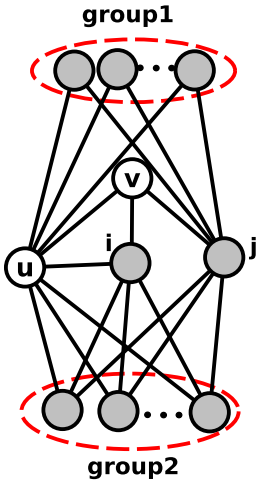
To write a generic algorithm that enumerates and counts graphlets within each of the above 9 combinations, I use three selector variables S_1, S_2, S_3, which take values from sets T, N_u, and N_v based on the membership of i, j, k within these sets. For instance, for an enumeration, if $i, j, k \in T$ I have $S_1 = S_2 = S_3 = T$. Thus, values of S_1, S_2, and S_3 denote an equivalence class, and local size-5 graphlets within one equivalence class can be counted efficiently. Now, conditioned on the equivalence class, edges between the vertices $\{u, v\}$ and $\{i, j, k\}$ are already fixed. But, based on the existence of edges between i, j, and k one equivalence class may lead to more than one local graphlet. I use the binary variable $l_1, l_2, l_3 \in \{1, 0\}$ for denoting edge existence between i, j, and k; $l_1 = 1$ if $(i, j) \in E$ else 0, $l_2 = 1$ if $(i, k) \in E$ else 0 and $l_3 = 1$ if $(j, k) \in E$ else 0. This leads to 8 possible choices within an equivalent class.

From the above discussion, I can obtain a simple size-5 local graphlet enumerator, using three nested for loops for vertices i, j, and k, each iterating over the set T, N_u and N_v and within the body of the innermost for loop, the enumerator determines the graphlet type based on the value of l_1, l_2, and l_3. However, such a method enumerates all local graphlet instances and hence is not efficient.

My approach for counting 5-size graphlets is that I enumerate over each 4-size graphlets and then without enumeration I count the number of different graphlets based on the topological disposition of possible 5^{th} node. This saves a large number of enumerations and yields a vastly improved local graphlet counting algorithm. Besides, the degree of freedom of search space becomes smaller, because when a 4-graphlet is given, the third and fourth vertices, i, j are fixed, and the values of S_1, S_2



(a) Example of graphlet types created using variable $l_2 = 0/1$ and fixed $S_3 = N_u$ and $l_3 = 0$



(b) Example of graphlet types created using variable $l_2 = 0/1$ and fixed $S_3 = N_u$ and $l_3 = 1$

Fig. 5.4.: Illustration of how different values of l_2 and l_3 generates different graphlet types

and l_1 are known; then the values of S_3, l_2 and l_3 decide the specific type of the size-5 graphlet.

Example: In Figure 5.4, I show a specific enumeration, in which $i \in T$ and $j \in N_v$, and I would like to count all local graphlets for which the fifth vertex k belongs to N_u, i.e., the selector variable, $S_3 = N_u$. The possible candidates for the k vertices are shown within dotted ovals. Also, in this example the vertex i, and j are not connected, so $l_1 = 0$. Now, the four possible values of l_2, l_3 create four different types of size-5 graphlets, which are $g12, g21, g23$, and $g32$. The left figure shows the case for $l_3 = 0$ (k is not connected with j) and the right figure shows the case for $l_3 = 1$ (k and j are connected). For both the left and the right figures, the vertices shown in the dotted oval labeled as "group1" exhibit $l_2 = 0$ and produce $g12$ (on left figure) and $g23$ (on the right figure). Likewise, the vertices in "group2" stands for $l_2 = 1$ and they produce $g21$ (on left figure) and $g32$ (on right figure). ■

To facilitate effective counting by iterating over 4-graphlets, I compute a few more vertex-sets beyonds T, N_u and N_v, by utilizing $\Gamma(i)$, the adjacency vector of the third vertex i. For different values of l_1, the 4^{th} vertex j belongs to different subsets of the sets, T, N_u or N_v. For example, if $S_2 = T$, then for $l_1 = 1$, j must belong to $\Gamma(i) \cap T$ and for $l_1 = 0$, j belong to $T \setminus \Gamma(i)$. I represent these sets as below:

$T_i = \Gamma(i) \cap T, \qquad N_{ui} = \Gamma(i) \cap N_u \ , \qquad N_{vi} = \Gamma(i) \cap N_v$

$\overline{T_i} = T \setminus \Gamma(i), \qquad \overline{N_{ui}} = N_u \setminus \Gamma(i), \qquad \overline{N_{vi}} = N_v \setminus \Gamma(i)$

In the template algorithm I refer to the above sets by using their selector variable. For example, for the selector variable S_3, I will write the set as S_{3i} which is equal to $\Gamma(i) \cap S_3$; thus S_{3i} can be T_i, or N_{ui} or N_{vi} depending on whether S_3 is T, N_u or N_v.

As mentioned earlier, for a given 4-size graphlets, say, (u, v, i, j), I count (without enumeration) the number of different graphlets based on the topological disposition of possible 5^{th} node . To facilitate this, we define the term $C(i,j)^{S_1,S_2,S_3}_{l_1,l_2,l_3}$. It represents the total count of size-5 graphlets which contain (u, v, i, j) as their sub-graphlet. In $C(i,j)^{S_1,S_2,S_3}_{l_1,l_2,l_3}$, i and j are given, thus S_1, S_2, and l_1 are already fixed. Thus the value of $C(i,j)^{S_1,S_2,S_3}_{l_1,l_2,l_3}$ is the number of 5-graphlets for the assigned values of S_3, l_2 and l_3. By fixing these three variables, I can obtain the count of a specific graphlet. For example, $C(i,j)^{S_1,S_2,S_3}_{l_1,1,0}$ represents the count of a graphlet type generated from the current i, j vertices and all possible $k \in S3$ such that $l2 = 1$, $l3 = 0$. More specific example is $C(i,j)^{T,T,T}_{1,1,1}$, which represents count of 5 size clique for given i, j values. Because, here all three vertices i, j, k are connected to both u and v and they themselves are also pair-wise connected($l1 = l2 = l3 = 1$), which creates fully connected 5 size subgraph (5-clique). Similarly, $C(i,j)^{T,T,T}_{1,0,1}$ represents count of graphlet type $g40$ for given i, j values.

For unknown values of l_2 and/or l_3, I use $*$, hence total count of graphlet types for given values of i, j, S_3 and l_2 but unknown value of l_3 can be represented as $C(i,j)^{S_1,S_2,S_3}_{l_1,l_2,*}$. For example, total count of graphlet types $\{g12, g18\}$ in Figure 5.4a can be represented as $C(i,j)^{T,N_v,N_u}_{0,*,0}$ and total count from Figure 5.4b is $C(i,j)^{T,N_v,N_u}_{0,*,1}$. Similarly, if both l_2 and $l3$ are unknown then I use $C(i,j)^{S_1,S_2,S_3}_{l_1,*,*}$, so $C(i,j)^{T,N_v,N_u}_{0,*,*}$ is

the total count for all the four kinds of graphlets in Figure 5.4. Now the value of $C(i,j)^{S_1,S_2,S_3}_{l_1,*,*}$ can be obtained by using the following theorem.

Theorem 5.3.1

$$C(i,j)^{S_1,S_2,S_3}_{l1,*,*} = \begin{cases} |S_3|, & \textit{if } (S_3 \neq S_1 \wedge S_3 \neq S_2) \\ |S_3|{-}1, & \textit{if } ((S_3 = S_1 \wedge S_3 \neq S_2) \\ & \quad \vee (S_3 \neq S1 \wedge S_3 = S_2)) \\ |S_3|{-}2, & \textit{if } (S_3 = S_1 \wedge S_3 = S_2) \end{cases}$$

Proof For each value of $k \in S_3$, the induced graphlet (u, v, i, j, k) matches a specific local graphlet type, but irrespective of the graphlet type it always adds a count to $C(i,j)^{S_1,S_2,S_3}_{l_1,*,*}$. Hence, count of the $C(i,j)^{S_1,S_2,S_3}_{l_1,*,*}$ is same as the cardinality of S_3. Now, if vertices i and/or j are also from the same set as S_3 i.e. $S_1 = S_3$ and/or $S_2 = S_3$ (note that S_1, S_2, S_3 can take only three different values from $\{T, N_u, N_v\}$, and they can have same values), then I need to deduct the total count by 1 if only i or only j is from the same set and deduct the count by 2 if both i, j are from the same set as S_3. Because all three vertices (i, j, k) need to be distinct $(i \neq j \neq k)$ to generate a 5-size local graphlet. ■

For the case when the value of l_2, or l_3 is also known, I have the following theorem.

Theorem 5.3.2

$$C(i,j)^{S_1,S_2,S_3}_{l_1,1,*} = \begin{cases} |S_{3i}|{-}1, & \textit{if } (S_3 = S_2 \wedge l_1 = 1) \\ |S_{3i}|, & \textit{otherwise} \end{cases}$$

$$C(i,j)^{S_1,S_2,S_3}_{l_1,*,1} = \begin{cases} |S_{3j}|{-}1, & \textit{if } (S_3 = S_1 \wedge l_1 = 1) \\ |S_{3j}|, & \textit{otherwise} \end{cases}$$

Proof For $C(i,j)^{S_1,S_2,S_3}_{l_1,1,*}$, I know that $k \in S_3 \cap \Gamma(i)$, hence total possible count is equal to the size of the set $S_{3i} = S_3 \cap \Gamma(i)$. I also need to ensure that $j \neq k$, hence if

$S_2 = S_3$ and j is also a neighbor of i (to make a symmetry to the condition that k is a neighbor of i), I need to deduct the count by one. A similar argument also holds for $C(i,j)^{S_1,S_2,S_3}_{l_1,*,1}$. ■

Now, the following Theorem and corollaries help us to count several graphlets in constant time simply by using algebraic expression.

Theorem 5.3.3

$$C(i,j)^{S_1,S_2,S_3}_{l_1,*,*} = C(i,j)^{S_1,S_2,S_3}_{l_1,0,0} + C(i,j)^{S_1,S_2,S_3}_{l_1,0,1} + C(i,j)^{S_1,S_2,S_3}_{l_1,1,0} + C(i,j)^{S_1,S_2,S_3}_{l_1,1,1}$$

Corollary 5.3.4

$$C(i,j)^{S_1,S_2,S_3}_{l_1,0,0} = C(i,j)^{S_1,S_2,S_3}_{l_1,*,*} - C(i,j)^{S_1,S_2,S_3}_{l_1,*,1} - C(i,j)^{S_1,S_2,S_3}_{l_1,1,*} + C(i,j)^{S_1,S_2,S_3}_{l_1,1,1}$$

Corollary 5.3.5

$$C(i,j)^{S_1,S_2,S_3}_{l_1,1,0} = C(i,j)^{S_1,S_2,S_3}_{l_1,1,*} - C(i,j)^{S_1,S_2,S_3}_{l_1,1,1}$$

Corollary 5.3.6

$$C(i,j)^{S_1,S_2,S_3}_{l_1,0,1} = C(i,j)^{S_1,S_2,S_3}_{l_1,*,1} - C(i,j)^{S_1,S_2,S_3}_{l_1,1,1}$$

5.3.3 Generic Counting Algorithm

In Algorithm 3, I provide a generic algorithm for counting all 5 size local graphlets. In this generic algorithm, I consider the following variables, $S_1, S_2, S_3, t, b_i, d_i$ and d as template variables and a specific set of values for these variables gives the frequency of a specific graphlet type. Among these S_1, S_2 and S_3 are selector variables (for vertices i, j and k), t is an integer between 1 and 4 depending on the joint value of l_2 and l_3, b_i is the bias which is the count adjustment when multiple selector variables

have the same value. The bias values are computed by using Theorem 5.3.1 and Theorem 5.3.2. Even after addressing for the obvious duplication by maintaining order among the vertices i, j, and k, some graphlets are generated multiple times, d_i and d are normalizing factors to ensures that each of the local graphlets are counted once and exactly once.

The detailed information on graphlet type and the associated values of the template variables are shown in Table 5.3. As shown in this table, the variable S_1 takes values from N_u or T, S_2 takes values conditioning on the fact whether the second vertex is adjacent to first vertex or not. So, it takes values from T_i, N_{ui}, N_{vi} or their complements $\overline{T_i}, \overline{N_{ui}}$, $\overline{N_{vi}}$. And S_3 takes value from N_u, N_v, or T. The bias and normalizing factor values are also shown in this table. In this table, I also have three other variables, S_{1r}, S_{2r} and S_{3r}. They will be discussed in Section 5.3.4.

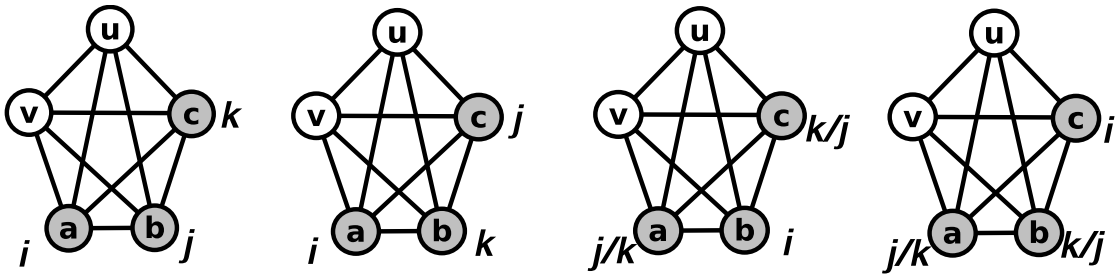
Algorithm 3 uses a sub-routine, Algorithm 4, where $|S_{3ij}|$ represents value of $C(i,j)_{l_1,1,1}^{S_1,S_2,S_3}$. Using Theorem 5.3.2, Corollary 5.3.5 and 5.3.6, I calculate frequencies of specific graphlet types represented as $f2_{ij}$ and $f3_{ij}$ in the algorithm. Similarly, line 5 of the algorithm gives frequency of a graphlet type where k is not connected to either i or j using Theorem 5.3.1 and Corollary 5.3.4. The deduction from the

Algorithm 3: Template Algorithm for Graphlet number N

1: assign S_1, S_2, S_3 to the right set for graphlet N from Table 5.3
2: select $t \in \{1,2,3,4\}$, b_i, d_i, and d for graphlet $N$from Table 5.3
3: $\langle l1_i, l2_i, l3_i, l4_i \rangle \leftarrow \mathbf{0}$ // initialization
4: **for all** $i \in S_1$ **do**
5: $\quad \langle l1_{ij}^{old}, l2_{ij}^{old}, l3_{ij}^{old}, l4_{ij}^{old} \rangle \leftarrow \mathbf{0}$ // initialization
6: $\quad$ **for all** $j \in S_2$ **do**
7: $\quad\quad \langle l1_{ij}^{new}, l2_{ij}^{new}, l3_{ij}^{new}, l4_{ij}^{new} \rangle \leftarrow$
$\quad\quad\quad \langle l1_{ij}^{old}, l2_{ij}^{old}, l3_{ij}^{old}, l4_{ij}^{old} \rangle +$ get_freq_k(i,j,S_3)
8: $\quad\quad \langle l1_{ij}^{old}, \cdots, l4_{ij}^{old} \rangle \leftarrow \langle l1_{ij}^{new}, \cdots, l4_{ij}^{new} \rangle$
$\quad$ // end of loop for j
9: $\quad lt_i \leftarrow lt_i + (lt_{ij}^{new} - b_i)/d_i$
// end of loop for i
10: $fN \leftarrow fN + lt_i/d$

Algorithm 4: get_freq_k(i, j, S_3)

1: $S_{3ij} \leftarrow S_{3i} \cap S_{3j}$
2: $v1 = |S_{3ij}|$
3: $v2 = |S_{3i}| - |S_{3ij}|$
4: $v3 = |S_{3j}| - |S_{3ij}|$
5: $v4 = |S_3| - |S_{3i}| - |S_{3j}| + |S_{3ij}|$
6: **return** $\langle v1, v2, v3, v4 \rangle$



(a) 5 clique graphlet ($i = a$) (b) 5 clique graphlet ($i = b/c$)

Fig. 5.5.: 5 clique graphlet counting

total count (in both Theorem 5.3.1 and 5.3.2) for maintaining $i \neq j \neq k$ property, is adopted as a bias value b_i in the Algorithm 3.

Example: Frequency of the 5 size clique i.e. graphlet type $g41$ can be calculated using template variable values $S_1 = T$, $S_2 = T_i$, $S_3 = T$, $t = 1$, $b_i = 0$, $d_i = 2$ and $d = 3$. For 5 size clique, as shown in the Figure 5.5, three vertices (a, b, c) other than u and v need to be from set T, hence $S_1 = T$, $S_2 = T$ and $S_3 = T$ and all three vertices need to be interconnected, so I use $S_2 = T_i$ ($l_1 = 1$) and $t = 1$ ($f1_{ij} \Rightarrow l_2 = 1, l_3 = 1$). Now, as we can see for each i, selection of j and k are interchangeable i.e. in Figure 5.5a j and k are interchangeable between vertices b and c. Therefore this graphlet will be counted twice for each i, which leads to $d_i = 2$. Similarly, from Figure 5.5b, we can see that I select all of the three vertices (a, b, c) as an i one by one (Algorithm 3: line 1), hence I need to divide the total count by 3 i.e. $d = 3$.

Table 5.3.: Set of values for template variables to count various 5 size local graphlets

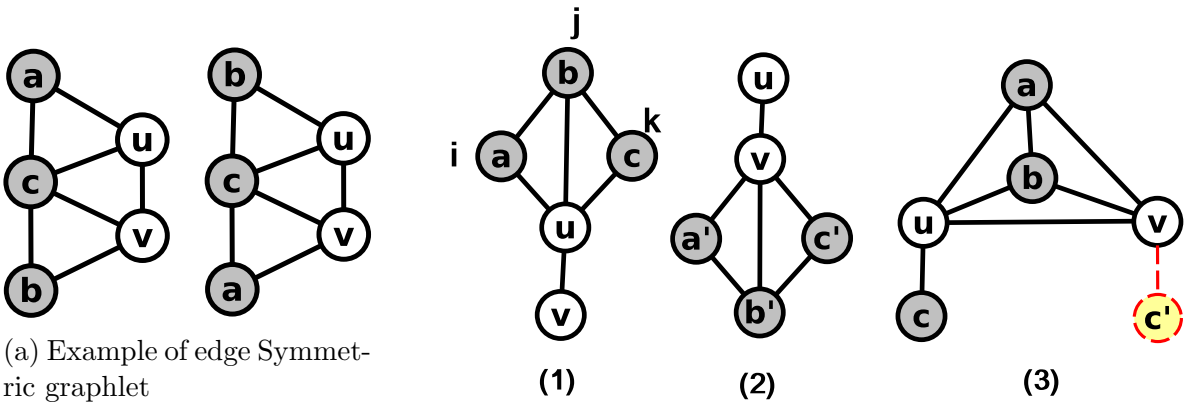
graphlet	S_1	S_2	S_3	t	b_i	d_i	d	S_{1r}	S_{2r}	S_{3r}
$g10$	N_u	$\overline{N_{ui}}$	N_v	4	0	1	2	N_v	$\overline{N_{vi}}$	N_u
$g11$	N_u	$\overline{N_{ui}}$	N_u	4	$2 \cdot \lvert S_2 \rvert$	2	3	N_v	$\overline{N_{vi}}$	N_v
$g12$	T	$\overline{N_{ui}}$	N_v	4	0	1	1	–	–	–
$g13$	N_u	$\overline{N_{ui}}$	N_v	4	0	1	2	N_v	$\overline{N_{vi}}$	N_u
$g14$	T	$\overline{N_{ui}}$	N_u	4	$\lvert S_2 \rvert$	2	1	T	$\overline{N_{ui}}$	N_v
$g15$	N_u	$\overline{N_{ui}}$	N_u	4	0	1	2	N_v	$\overline{N_{vi}}$	N_v
$g16$	N_u	$\overline{N_{ui}}$	N_v	3	0	1	1	N_v	$\overline{N_{vi}}$	N_u
$g17$	T	$\overline{T_i}$	$N_u + N_v$	4	0	1	2	–	–	–
$g18$	T	N_{ui}	N_u	4	0	1	1	T	N_{vi}	N_v
$g19$	N_u	N_{ui}	N_u	3	$\lvert S_2 \rvert$	1	2	N_v	N_{vi}	N_v
$g20$	N_u	N_{ui}	T	4	0	1	2	N_v	N_{vi}	T
$g21$	T	$\overline{N_{ui}}$	N_v	4	0	1	1	T	$\overline{N_{vi}}$	N_u
$g22$	N_u	$\overline{N_{ui}}$	N_v	1	0	1	2	N_v	$\overline{N_{vi}}$	N_u
$g23$	T	$\overline{N_{ui}}$	N_v	3	0	1	1	–	–	–
$g24$	N_u	N_{ui}	N_v	3	0	1	1	N_v	N_{vi}	N_u
$g25$	T	N_{ui}	N_u	2	0	2	1	T	N_{vi}	N_v
$g26$	T	$\overline{T_i}$	T	4	$2 \cdot \lvert S_2 \rvert$	2	3	–	–	–
$g27$	T	T_i	$N_u + N_v$	4	0	1	2	–	–	–
$g28$	N_u	N_{ui}	N_u	1	0	2	3	N_v	N_{vi}	N_v
$g29$	T	N_{ui}	N_u	3	0	1	1	T	N_{vi}	N_v
$g30$	T	N_{ui}	T	4	0	1	1	T	N_{vi}	T
$g31$	T	N_{ui}	N_v	2	0	1	1	–	–	–
$g32$	T	N_{ui}	N_v	3	0	1	1	T	N_{vi}	N_u
$g33$	N_u	N_{ui}	N_v	1	0	1	2	N_v	N_{vi}	N_u
$g34$	T	N_{ui}	N_u	1	0	2	1	T	N_{vi}	N_v
$g35$	T	T_i	T	4	0	1	2	–	–	–
$g36$	T	N_{ui}	T	2	0	1	1	T	N_{vi}	T
$g37$	T	N_{ui}	N_v	1	0	1	1	–	–	–
$g38$	T	N_{ui}	T	3	$\lvert S_2 \rvert$	1	2	T	N_{vi}	T
$g39$	T	N_{ui}	T	1	0	1	2	T	N_{vi}	T
$g40$	T	T_i	T	3	$\lvert S_2 \rvert$	1	2	–	–	–
$g41$	T	T_i	T	1	0	2	3	–	–	–

5.3.4 Counting frequency of non-symmetric local graphlets

Definition 5.3.1 (Vertex orbit) *For a given graphlet $g(V, E)$, vertexes $u, v \in V$ are in same orbit if an automorphism f_{auto} of g exists such that $u = f_{auto}(v)$.* ■

Definition 5.3.2 (Symmetric local graphlets) *For a given edge (u, v) a local graphlet g is called symmetric if vertices u and v are in the same vertex orbit for g.* ■

For symmetric local graphlets I do not need to count frequency for reverse sequence of the vertices v, u (instead of u, v). For a symmetric local graphlet, $d(u) = d(v)$ and



(a) Example of edge Symmetric graphlet

(b) Examples of edge non-symmetric graphlets

Fig. 5.6.: Example of edge symmetric and non-symmetric graphlets

graphlet structure remains the same after exchanging neighborhood of the vertices u and v. For example, as shown in Figure 5.6a for graphlet type $g31$, $d(u) = d(v) = 3$ and when I exchange neighbors of u and v the graphlet structure remains the same, because vertex u and v are in the same orbit. On the other hand, Figure 5.6b shows two examples of non-symmetric graphlets of type $g19$ and $g27$.

To calculate the correct frequency for non-symmetric graphlets, I need to count occurrence of the graphlet with reverse sequence of the vertices. For example, in Figure 5.6b(1) a, b and c all are neighbors of u and $d(u) = 4$. However to count the correct frequency of the graphlet type $g19$, we need to count frequency for the graphlet shown in Figure 5.6b(2) where $d(v) = 4$ and a', b' and c' all are neighbors of v. To count the correct frequency for the graphlet in Figure 5.6b(1), template variables take values $S_1 = N_u$, $S_2 = N_{ui}$ and $S_3 = N_u$. But to count frequency of the graphlet in Figure 5.6b(2), the variable values become $S_{1r} = N_v$, $S_{2r} = N_{vi}$ and $S_{3r} = N_v$, where S_{1r} represents reverse version of S_1 as shown in Table 5.3. Template variables t, b_i, d_i and d remain the same for both cases. Because when I calculate $f3_{ij}$ using Algorithm 4, N_{uj} (S_{3j}) also includes vertex represented by i (here a), hence I need to subtract 1 from $f3_{ij}$ for each j. I subtract this value combined using bias

$b_i = |S_2|$. Lastly, we also count the same graphlet two times considering $i = a$ and $i = c$, hence we divide the global count by $d = 2$.

Another example of non-symmetric graphlet is graphlet type $g27$. To count correct frequency for this type, I just need to sum up possible unique neighbors of u and unique neighbors of v (with red dotted line). To calculate both cases as shown in Table 5.3, I just need to put two different values (N_u and N_v) of template variable S_3 and sum the resultant frequency values.

5.3.5 Complexity Analysis

For 5-size local graphlets counting, any regular algorithm generally takes $O(\Delta^3)$ time for each edge [34], where Δ is maximum degree in the graph G. However, time complexity of this method for counting 5 sized local graphlets (Algorithm 3) is $O((T^{max} + N_u^{max} + N_v^{max})^3)$, where T^{max} is the largest value of $|T|$ out of all edges of the graph. Similarly N_u^{max} and N_v^{max} represents largest value of $|N_u|$ and $|N_v|$ respectively. For any edge (u, v), $|N_u| = d(u) - |T|$ and $|N_v| = d(v) - |T|$, and for any real-world sparse network $0 \leq T^{max} \ll \Delta$. Also, for any node with highest degree there is very low probability that $|T| = 0$, hence $|T| > 0$ and $N_u^{max} \leq \Delta$ and $N_v^{max} \leq \Delta$.

5.3.6 Parallelizing the *E-CLoG*

E-CLoG is embarrassingly parallelizable, as the main work is performed over two for loops and operations inside the loops are independent for each iteration. Additionally, each edge can have unique independent data structure (T, N_u, N_v) values, hence parallel computation over edges is highly effective for large number of edges. I use only second strategy in my implementation.

Table 5.4.: Dataset statistics

Datasets	$\|V\|$	$\|E\|$	*Avg.Deg*
arenas-meta	453	2025	8.94
arenas-pgp	10680	24316	4.55
as-caida	26475	53381	4.03
ca-AstroPh	18771	198050	21.10
com-amazon	334863	925872	5.53
com-dblp	317080	1049866	6.62
douban	154908	327161	4.22
facebook-wosn	63731	817035	25.64
loc−brightkite	58228	214078	7.35
maayan-vidal	3023	6149	4.07
opsahl-powergrid	4941	6594	2.67
petster-hamster	2426	16630	13.71
reactome	6229	146160	46.93
roadNet-CA	1965206	2766607	2.82
roadNet-PA	1088092	1541898	2.83
roadNet-TX	1379917	1921660	2.79
topology	34761	107719	6.20
wordnet-words	146005	656999	9.00

5.4 Experiments and Results

I conducted three different experiments to show efficiency, scalability and usability of the proposed method. In the first experiment, I compare running time of my method with GRAFT [34]. In the second experiment I show that, nearly linear speed-up can be achieved for the parallel version of the *E-CLoG*. Lastly, in the third experiment, I show the utility of local graphlet frequencies for solving link prediction problem.

I collected 18 different graph datasets from different domains from KONECT [2]. I have 3 biological networks: *arenas-meta* is a metabolic network of the roundworm, *maayan-vidal* and *reactome* are networks of protein-protein interactions in humans. *Arenas-png* is an interaction network of users of the Pretty Good Privacy (PGP)

[2]Koblenz Network Collection: `http://konect.uni-koblenz.de/`

algorithm. *As-caida* and *topology* are networks of autonomous systems of the Internet. *ca-AstroPh* and *com-dblp* are co-authorship graphs, and *com-amazon* is a co-purchase network of Amazon. There are four different types of online social networks: *douban* is an online recommendation based social network, $facebook\text{-}wosn$ is an online friendship network, *loc-brightkite* is a location based social network and *petster-hamster* is a friendship network of pet owners. I have few infrastructure based networks, such as *opsahl-powergrid* is a power-grid network, and three road networks $roadNet\text{-}CA$, $roadNet\text{-}PA$ and $roadNet\text{-}TX$ for three different states of the USA. $Wordnet\text{-}words$ is a lexical network of words from the WordNet dataset. Some basic statistics such as number of vertices ($|V|$), number of edges ($|E|$), and average degree ($Avg.Deg$) for the datasets are shown in the Table 5.4.

5.4.1 Runtime Comparison

There exist no methods that perform local edge-centric graphlet counting for size-5 graphlets. Two recent global methods for exact counting of size-5 graphlets exist, GRAFT [34], and ESCAPE [174]; from which, GRAFT iterates over each of the edges of the input graph and aggregates the sub-counts of the graphlets that are incident to the edge of that iteration. At the end, it divides the duplicity factors of each graphlets to obtain the global graphlet counts. Thus GRAFT, indirectly, is an edge centric local graphlet counting method, which produces count of local 5-graphlets for each edge. In this experiment, I compare GRAFT's running time with *E-CLoG*'s running time by finding the total time of computing local graphlets over all the edges. Note that, This comparison is a bit unfair for *E-CLoG* as it generates frequencies of local 5-graphlets of all edge orbits, totalling 32 graphlets, but GRAFT generates frequency of only 21 size-5 graphlets. Also note, among the 21 topologies that GRAFT counts, two topologies 5-path and 5-cycle are not counted by *E-CLoG* as they are not local graphlets as per definition. For this comparison, I extend *E-CLoG* and compute these two counts also. Lastly, GRAFT does not provide parallel implementation,

Table 5.5.: Runtime comparison between E-CLoG and GRAFT
(d = days, h = hours, m = minutes, s = seconds)

Datasets	E-CLoG (serial)	GRAFT	E-CLoG (parallel)
arenas-meta	5.2**s**	2.67**m**	0.36**s**
arenas-pgp	14.6**s**	13.49**m**	1.06**s**
as-caida	160.98**m**	> 8.51**d**	8.56**m**
ca-AstroPh	23.51**m**	3**d**	1.58**m**
com-amazon	4.3**m**	9.82**h**	24.22**s**
com-dblp	14.32**m**	1.56**d**	56.31**s**
douban	5.41**m**	10.02**h**	15.43**s**
facebook-wosn	210.94**m**	> 4.01**d**	13.03**m**
loc−brightkite	43.93**m**	> 7.15**d**	2.54**m**
maayan-vidal	1.71**s**	47.39**s**	0.11**s**
opsahl-powergrid	0.0556**s**	0.3292**s**	0.01**s**
petster-hamster	43.44**s**	54.79**m**	3.24**s**
reactome	98.9**m**	> 7.15**d**	6.72**m**
roadNet-CA	17.71**m**	2.2**m**	5.1**m**
roadNet-PA	5.52**m**	1.07**m**	1.57**m**
roadNet-TX	8.64**m**	1.38**m**	2.49**m**
topology	510.7**m**	> 7.15**d**	26.17**m**
wordnet-words	93.13**m**	> 4.01**d**	5.43**m**

hence I use single thread computation for my method for fair comparison. The other exact global graphlet counting method, ESCAPE, does not iterate over the edges so it cannot produce counts for local edge graphlets and is not comparable with my method.

Table 5.5 shows the runtime comparison between *E-CLoG* and GRAFT over all 18 different graph datasets listed in Table 5.4. First column is the dataset name, the second and third columns show the *E-CLoG*'s time and the GRAFT time, respectively. The reported time for both the methods is the time for counting local graphlets for all edges for each dataset after averaging the run-time over 5 runs. For a better understanding, I write different time units (such as, **s** for seconds, **m** for minutes, **h** for hour, and so on.) besides the time values. As we can see, for most of the datasets *E-CLoG* runs one or two orders of magnitude faster than GRAFT. For 6 datasets,

GRAFT is unable to complete the counting even after few days (> in the Table 5.5), while *E-CLoG* completed the counting in few hours on those datasets. A specific example can be wordnet-words dataset, for which *E-CLoG* took only 93.13 minutes (= 1.5 hours), but GRAFT did not finish in 4 days! In 13 of 16 graphs, *E-CLoG* performs very good. Other 3 graphs in which GRAFT did better than *E-CLoG*, interestingly, are all road networks. A possible reason for this is during enumeration GRAFT first aligns an edge of a tree graphlet with the given edge, and then performs costly checks for counting cyclic graphlets. Since, road networks are mostly tree networks, they run very fast on GRAFT. However, most of the real-life networks have a substantial number of cycles, for those graphs, GRAFT is very poor.

In this table, I also show the runtime of parallel version of *E-CLoG* in Column 4. This parallel version uses 72 threads. For most of the graphs, the parallel version improves the runtime significantly. For instance, for facebook graph the parallel version runs in 13 minutes whereas the single-thread version runs in 211 minutes.

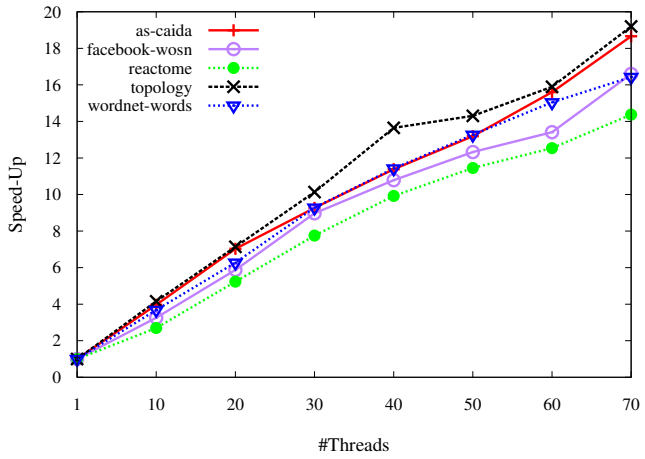


Fig. 5.7.: Strong scaling results for a variety of graphs. I obtain 14x–20x speedup using 70 threads.

5.4.2 Scalability

This section investigates the parallel performance of the proposed algorithm. The parallel algorithm for local graphlet counting has lock-free updates due to the partitioning of edges across the processing units and each edge is guaranteed to be processed by a single worker. For these experiments, I used a machine with two Intel Xeon E5-2699 v3 platform with 2.30GHz CPUs. Each processor has 18 cores with 46MB of L3 cache and 256KB of L2 cache. The machine has 256GB of memory, however, *E-CLoG* never came close to using all of it. *E-CLoG* scales well as the number of processing units increase. In particular, strong scaling is observed in Figure 5.7 for the 5 graphs having the worst running time for serial execution. I obtain 14x–20x speedup using 70 threads for most graphs.

5.4.3 Link Prediction

Given a pair of non-adjacent vertices u, and v in a social network, link prediction task predicts whether the vertices will form a link in future. In a supervised classification setup, link prediction task is typically solved by using a fix set of topological features which determine the topological similarity between u and v. In this experiment I will demonstrate that local graphlet frequency distribution (LGFD) for the edge (u, v) captures topological properties (around the vertices u, v) that have substantially more predictive power than the traditionally used topological features for link prediction.

For this experiment, I use three datasets which are time-stamped networks. From Table 5.4, only two datasets ($facebook\text{-}wosn$ and *topology*) have time-stamps affiliated with edges. Here, $facebook\text{-}wosn$ has $333,923$ unique time-stamps and *topology* has $23,768$ unique time stamps, which I use. I also create $DBLP$ co-authorship network with time-stamps (yearly) from AMiner [4]. To create this network, I select a set of authors who have published 2 or more papers in database and data mining

[4] `https://aminer.org/data`

Table 5.6.: Comparison results for Link Prediction Problem

	ROC-AUC		PR-AUC	
Datasets	Topo-feat	LGFD	Topo-feat	LGFD
facebook-wosn	0.5519	**0.9101**	0.6784	**0.9351**
topology	0.8179	**0.9366**	0.8837	**0.9415**
DBLP	0.6897	**0.7411**	0.7703	**0.7822**

conferences and from these selected authors I generate induced co-authorship graph. This network has $4,545$ authors with $20,491$ connections over 45 years.

For experiment, I divide the time-stamps of each dataset into three chronologically ordered partitions, network growing period, train period and test period; from the beginning up to 70% of total time-stamps is network growing period, 70% to 85% is train period, and from 85% till the end is the test period. An edge created in train period is a positive train instance and an edge created (for the first time) during this test period is a positive test instance. I take random node pairs which do not have a connecting edge till the end of the training period as negative train instances and random disconnected node pairs as negative test instances. In this experiment, we use local graphlet frequencies normalized over all local graphlet types (42) as a features set. For comparison with these local graphlet frequencies, I use 10 traditional topological features : number of common neighbors, Jaccard's coefficient, preferential attachment, adamic-adar and Katz for 5 different β values $(0.1, 0.05, 0.01, 0.005, 0.001)$. Note that, Katz, adamic-adar and Jaccard's coefficient are proven to be the best topological features for link prediction. Also, computing some of these link prediction features on a large network is substantially more costly than computing local graphlet frequency. For instance, for all datasets computing Katz takes several days! For supervised classification I use linear SVM (Support Vector Machine), for which the regularization coefficient C is chosen by grid-search from values $\{0.001, 0.01, 0.1, 1.0, 10.0, 100.0\}$ using a random 20% of test instances as

Table 5.7.: Useful graphlets for link prediction

Datasets	Graphlets with high individual AUC
facebook-wosn	$g0, g5, g12, g14, g16, g18, g20, g21, g25, g29, g34$
topology	$g0, g5, g11, g14, g15, g18, g19, g20, g25, g29, g34$
DBLP	$g0, g2, g5, g12, g14, g18, g20, g21, g25, g27, g29, g34, g36, g39$

a validation set. I evaluate the link prediction results using Area Under Curve ROC (*ROC-AUC*) and Precision-Recall AUC (*PR-AUC*)

In Table 5.6 I show the link prediction comparison results. For both the metrics, LGFD substantially improve the link prediction performance, typically 10% to 45% improvement in both kinds of AUC has been observed. This is not surprising because local graphlet frequency actually encodes information that traditional topological features encode. These results show high discriminative power of graphlet frequencies to identify future links, but not all local graphlet types are equally important. So I conducted an analysis study to find out frequencies for which specific local graphlets are highly impactful. For the study, I find normalized graphlet frequency of each graphlet which is treated as a predicted probability of a classifier to calculate individual AUC value for each graphlet type. In Table 5.7, I show list of graphlet types for which AUC is above 85% for *facebook-wosn* and *topology* datasets and above 70% for *DBLP* dataset. This table shows which graphlet type is an important feature by itself, and it also highlights the fact that 5 size local graphlets are commonly very good features for link prediction task.

5.5 Chapter Summary

In this work, I present a very efficient algorithm for computing the frequencies of edge-centric local graphlets of size upto 5. The experimental results show that the proposed method is very efficient, and scalable for large real-life networks. I also show

the utility of local graphlet count for predicting future links in a social or collaboration network.

6. ATTRIBUTED NETWORK EMBEDDING FOR RELATED ATTRIBUTES

6.1 Introduction

In this chapter, I present a representation learning framework for joint learning of representation vectors of both nodes and its attributes in a single vector space. Note that, networks with a finite set of attributes can be represented as two separate networks; first is a network of nodes and second is a bipartite graph of nodes and attributes. In this embedding approach, I treat the input network with nodal attributes as two networks as described in next section of this chapter. Additionally, this embedding method assumes that node attributes are inter-related and these relations are partially known, which creates another network of attributes. For the proposed representation learning method, I generate three networks, node-node network, node-attribute bipartite network and attribute-attribute network, from the input attributed network. Specifically, I use these three networks as inputs to the representation learning model and use both Bayesian personalized ranking and margin based loss functions to learn the vector representations. I conducted experiments on a real-world dataset from a well-known job portal company, where the input networks are job-job transition graph, job-skill graph, and skill-skill graph. I provided job and skill recommendations using the proposed embedding approach and compare with several baseline methods to show that the proposed representation learning framework yields better representation vectors.

6.2 Problem Formulation

Let $G = (\mathcal{V}, \mathcal{E}, \mathcal{A})$ be an attributed network, where $\mathcal{V}$ is a set of n nodes, and $\mathcal{E}$ is a set of edges, and $\mathcal{A}$ is a set of attributes. There exist a many to many mapping function $f : \mathcal{V} \rightarrow \mathcal{A}$ to identify the connection between nodes and corresponding attributes. I can convert this attributed network into two separate networks as below:

Definition 6.2.1 (Node-node network) *A node-node network is represented as $G^{vv}(\mathcal{V}, \mathcal{E}^v)$, where $\mathcal{E}^v = \mathcal{E}$ is a set of edges between the node.*

For nodes $u, v \in \mathcal{J}$, an edge from u to v is represented as $e^v_{uv} = (u, v) \in \mathcal{E}^v$. The total number of nodes is denoted as $n^v = |\mathcal{V}|$.

Definition 6.2.2 (Node-attribute network) *A bipartite graph $G^{va}(\mathcal{V} \cup \mathcal{A}, \mathcal{E}^{va})$, where $\mathcal{V}$ is a set of nodes, $\mathcal{A}$ is a set of attributes with number of attributes $n^a = |\mathcal{A}|$ and $\mathcal{E}^{va} = \bigcup_{i=1}^{n_v}\{f(i)\}$ is a set of edges from a node to a corresponding attribute.*

As discussed before, this method assumes attributes are inter-related and create a network

Definition 6.2.3 (Attribute-attribute network) *Let $G^{aa}(\mathcal{A}, \mathcal{E}^a)$ be an undirected attribute relation network, where $\mathcal{A}$ is a set of attributes, $\mathcal{E}^a$ is a set of edges between the related attributes.*

Formally, given node-node network G^{vv}, node-attribute network G^{va} and attribute-attribute network G^{aa}, my goal is to obtain k-dimensional representation of nodes ($\mathbf{W}$) and attributes ($\mathbf{W}'$) into a shared latent space. Here, $\mathbf{W} = [\mathbf{w}_1^T, \mathbf{w}_2^T, ..., \mathbf{w}_{n^j}^T]^T \in \mathbb{R}^{n^v \times k}$, where $\mathbf{w}_i$ is i^{th} column of embedding matrix $\mathbf{W}$, which is the representation of the i^{th} node. Similarly, $\mathbf{W}' = [\mathbf{w}_1'^T, \mathbf{w}_2'^T, ..., \mathbf{w}_{n^s}'^T]^T \in \mathbb{R}^{n^a \times k}$ is the attribute representation matrix. The embedding matrices $\mathbf{W}$ and $\mathbf{W}'$ should preserve the connectivity information from graphs G^{vv} and G^{aa}, respectively. Additionally, these matrices also leverage signals from graphs G^{va} and through G^{va}, node similarity information propagates to $\mathbf{W}'$ and attribute similarity information propagates to $\mathbf{W}$.

6.3 Methodology

In this section, I discuss our proposed representation learning model. My goal is to encode the local neighborhood structures captured by the three networks into k-dimensional node and attribute embedding matrices that captures the structural and conceptual similarities between nodes and attributes.

6.3.1 Model Design

First, I capture node-node similarity information from graph G^{vv}. The main intuition behind the proposed embedding model is that similar nodes need to be closer in latent space compared to non-similar nodes. For example, for an edge $e_{xy} \in \mathcal{E}^v$ the vector representation $\mathbf{w}_x$ of node x should be closer to $\mathbf{w}_y$ compared to $\mathbf{w}_z$ when $e_{xz} \notin \mathcal{E}^v$. I calculate the affinity score between two embedding vectors using a dot product operation, hence the affinity score between node x and node y is represented as $A^v_{xy} = \langle \mathbf{w}_x, \mathbf{w}_y \rangle$. More precisely, we are interested in having higher affinity score between node x and node y compared to node x and node z given $e_{xy} \in \mathcal{E}^v$ and $e_{xz} \notin \mathcal{E}^v$, i.e. $A^v_{xy} > A^v_{xz}$. We can model the probability function that preserves the order $A^v_{xy} > A^v_{xz}$ for given $\mathbf{w}_x$, $\mathbf{w}_y$ and $\mathbf{w}_z$. Specifically, I utilize two functions for this modeling task, 1) sigmoid function $\sigma(v) = \frac{1}{1+e^{-v}}$ and 2) $ReLU$ function $ReLU(v) = max(0, v)$. In the following, I show the formulation only for sigmoid function, and it is very similar for the $ReLU$ function. The probability that the order $A^v_{xy} > A^v_{xz}$ is preserved can be formulated as below:

$$P(A^v_{xy} > A^v_{xz} \mid \mathbf{w}_x, \mathbf{w}_y, \mathbf{w}_z) = \sigma(A^v_{xyz}) \tag{6.1}$$

where,

$$A^v_{xyz} = A^v_{xy} - A^v_{xz} = \langle \mathbf{w}_x, \mathbf{w}_y \rangle - \langle \mathbf{w}_x, \mathbf{w}_z \rangle \tag{6.2}$$

From equation 6.1, it is clear that the higher the value of A^v_{xyz}, the better the ordering is preserved. Hence, our goal is to maximize the probability for preserving all the ranking orders of all training triplets (x, y, z), where $e_{xy} \in \mathcal{E}^v$ and $e_{xz} \notin \mathcal{E}^v$. I assume that all training triplets are sampled independently, thus the joint probability of preserving all training ranking orders, $P(>|\ \mathbf{W})$, can be represented as below:

$$\begin{aligned} P(>|\ \mathbf{W}) &= \prod_{(x,y,z)\in\mathcal{D}^{vv}} P(A^v_{xy} > A^v_{xz} \mid \mathbf{w}_x, \mathbf{w}_y, \mathbf{w}_z) \\ &= \prod_{(x,y,z)\in\mathcal{D}^{vv}} \sigma(A^v_{xyz}) \\ &= \prod_{(x,y,z)\in\mathcal{D}^{vv}} \sigma(A^v_{xy} - A^v_{xz}) \end{aligned} \tag{6.3}$$

Where $\mathcal{D}^{vv}$ is a set of training triplets from graph G^{vv}. I aim to maximize the joint probability of training triplets (equation 6.3). For the computational simplicity, I minimize the negative log of this joint probability instead, which is shown as follows:

$$\begin{aligned} O^{vv} &= \min_{\mathbf{W}} \ -\ln P(>|\ \mathbf{W}) \\ &= \min_{\mathbf{W}} \ -\sum_{(x,y,z)\in\mathcal{D}^{vv}} \ln \sigma(A^v_{xy} - A^v_{xz}) \end{aligned} \tag{6.4}$$

The optimization objective shown in equation 6.4 helps to achieve desirable node embedding, where similar nodes have higher affinity score than non-similar nodes.

Similarly, I obtain the attribute embedding matrix $\mathbf{W}' \in \mathbb{R}^{n^a \times k}$, which preserve the attribute similarity information. For the attribute-attribute network G^{aa}, our

goal is to obtain higher affinity scores for related attribute. This can be achieved using the following optimization objective:

$$
\begin{aligned}
O^{aa} &= \underset{\mathbf{W}'}{\boldsymbol{min}} \; -\ln P(>|\, \mathbf{W}') \\
&= \underset{\mathbf{W}'}{\boldsymbol{min}} \; -\sum_{(x,y,z)\in\mathcal{D}^{ss}} \ln \sigma(A^a_{xy} - A^a_{xz}) \\
&= \underset{\mathbf{W}'}{\boldsymbol{min}} \; -\sum_{(x,y,z)\in\mathcal{D}^{ss}} \ln \sigma(\langle \mathbf{w}'_x, \mathbf{w}'_y\rangle - \langle \mathbf{w}'_x, \mathbf{w}'_z\rangle)
\end{aligned} \tag{6.5}
$$

Where, w'_i is the i^{th} column of the attribute embedding matrix $\mathbf{W}'$, $\mathcal{D}^{aa}$ is a set of training triplets sampled from graph G^{aa} and affinity score between attributes x and y is denoted as $A^a_{xy} = \langle \mathbf{w}'_x, \mathbf{w}'_y\rangle$. The objective function shown in the equation 6.5 is to maximize the difference in terms of affinity scores between related attributes and non-relevant attributes in the graph G^{aa}.

Lastly, I incorporate information from node-attribute bipartite graph G^{va} into the node embedding matrix $\mathbf{W}$ and attribute embedding matrix $\mathbf{W}'$. In order to achieve that, I sample two attribute y^a and z^a such that the attribute y^a is associated with node x^v and the attribute z^a is not connected to node x^v. Here I compute the affinity score between node x^v and attribute y^a as $A^{va}_{x^v y^a} = \langle \mathbf{w}_{x^v}, \mathbf{w}'_{y^a}\rangle$. Note that for a given node x^v, I select its corresponding node latent vector from $\mathbf{W}$ and for the attribute y^a I select its corresponding latent vector from $\mathbf{W}'$. The objective function is formulated as below:

$$
\begin{aligned}
O^{va} &= \underset{\mathbf{W}\mathbf{W}'}{\boldsymbol{min}} \; -\ln P(>|\, \mathbf{W}, \mathbf{W}') \\
&= \underset{\mathbf{W}\mathbf{W}'}{\boldsymbol{min}} \; -\sum_{(x^v,y^a,z^a)\in\mathcal{D}^{va}} \ln \sigma(A^{va}_{x^v y^a} - A^{va}_{x^v z^a}) \\
&= \underset{\mathbf{W}\mathbf{W}'}{\boldsymbol{min}} \; -\sum_{(x^v,y^a,z^a)\in\mathcal{D}^{va}} \ln \sigma(\langle \mathbf{w}_{x^v}, \mathbf{w}'_{y^a}\rangle - \langle \mathbf{w}_{x^v}, \mathbf{w}'_{z^a}\rangle)
\end{aligned} \tag{6.6}
$$

Where $\mathcal{D}^{va}$ is a set of training triplets sampled from graph G^{va}.

The goal of the proposed network embedding framework is to unify these three types of relations (G^{vv}, G^{va}, G^{aa}) together to learn high quality node and attribute embedding matrices. An intuitive manner is to collectively embed these three graphs, which can be achieved by minimizing the following objective function:

$$\mathbf{O}(\mathbf{W}, \mathbf{W}') = \underset{\mathbf{W}\mathbf{W}'}{\boldsymbol{min}} \quad O^{vv} + O^{va} + O^{aa} + \lambda \cdot (\|\mathbf{W}\|_F^2 + \|\mathbf{W}'\|_F^2) \tag{6.7}$$

Where λ is a regularization co-parameter and $\|\cdot\|_F^2$ is l_2 regularization for both embedding matrices to avoid over-fitting.

I call the proposed method with sigmoid function as $Joint\text{-}BPR$, as equations 6.4, 6.5 and 6.6 are in the similar spirit to Bayesian Personalized Ranking (BPR) [44, 175]. Similarly, I call the proposed method with $ReLU$ function as $Joint\text{-}Margin$.

6.3.2 Model Optimization

Our proposed embedding framework has two model parameters $\mathbf{W}$ and $\mathbf{W}'$, which are node and attribute embedding matrices. I learn these matrices using mini-batch gradient decent. Specifically, I sample triples (x, y, z) from each of the three graphs (G^{vv}, G^{va}, G^{aa}), where (x, y) are connected and (x, z) are dis-connected pair of nodes in the corresponding graphs as discussed in the previous section. For each mini-batch, I compute derivative of the objective function shown in equation 6.7 with respect to $\mathbf{W}$ and $\mathbf{W}'$ and update the matrix values using the following equations:

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \alpha \times \frac{\partial \mathbf{O}(\mathbf{W}, \mathbf{W}')}{\partial \mathbf{W}} \tag{6.8}$$

$$\mathbf{W}'_{t+1} = \mathbf{W}'_t - \alpha \times \frac{\partial \mathbf{O}(\mathbf{W}, \mathbf{W}')}{\partial \mathbf{W}'} \tag{6.9}$$

where α is the learning rate. Additionally, I initialize both matrices from normal distribution with 0.0 mean and 0.1 standard deviation. For better understanding of the proposed methodology, I provide the pseudo-code of our method in Algorithm 1.

Algorithm 5: Pseudo-code for the proposed embedding framework

Input: G^{vv}, G^{va}, G^{aa}, embedding dimension k, batch size b, learning rate α, regularization coefficient λ

Output: Node embedding matrix $\mathbf{W}$ and attribute embedding matrix $\mathbf{W}'$.

1: Initialize $\mathbf{W}, \mathbf{W}'$ as k-dimensional matrices with 0 mean and 0.1 standard deviation from normal distribution
2: Given G^{vv}, G^{va}, G^{aa}, construct training triplet sets $\mathcal{D}^{vv}$, $\mathcal{D}^{va}$, and $\mathcal{D}^{aa}$ respectively using uniform sampling technique
3: **for** each training instance in training sample sets **do**
4: Update involved parameters using min-batch gradient descent as described in Sections 3.1, 3.2
5: **return** $\mathbf{W}, \mathbf{W}'$.

6.4 Experiments and Results

I conducted experiments on real-world dataset that I received from a popular job portal company. I generated three graphs from the data

1. job transition graph: It is a network of jobs that has connection between jobs based on transition from one job to other. This graph is treated as node-node network.

2. job-skill graph: It is a bipartite graph between jobs and related skills, here the jobs are node and corresponding skills are attributes. So this is a node-attribute network.

3. skill co-occurrence graph: It is a graph inter-connecting skills based on its frequent co-occurrence on job posting. This graph is treated as attribute-attribute network.

Below, I briefly describe how these three graphs are generated.

6.4.1 Data Preparation

This method relies on utilizing three networks, namely, (i) job transition graph, (ii) job-skill graph, and (iii) skill co-occurrence graph. In order to build these networks, 20

million resumes are collected from one of the largest human capital solution company in the US. Then parse each resume to extract the work history section which is then parsed to extract the job title and employment date. After that the job titles in a resume are ordered in the temporal order such that job x is placed before job y if the employment date of x is earlier than the employment date of y. Therefore, in the job transition network an edge e_{xy} represents that job x listed in a resume before job y in the work history section. Lastly, we normalized the set of job titles using Carotene [176], an in-house job classification tool in CareerBuilder. This step reduces the number of unique jobs to 4325 with $2,432,231$ distinct job transitions. Note that although job transition graph is directed, for learning the representation vector, it is treated as an undirected graph.

For the skill co-occurrence network, the skills extracted from each resume are connected, so an edge $e_{s_1 s_2}$ means both skills s_1 and s_2 are listed in the same resume at least once. Then any edge with co-occurrence value smaller than 10 is removed to avoid poor quality relations, that left us with 6214 unique skills co-occurring in $11,760,132$ different ways. To further reduce the noisy connections, the weights on the edges are calculated using Point-wise Mutual Information (PMI) and use reasonable weight threshold to filter poor quality edges.

Lastly, to build the job-skill bipartite graph, each skill is extracted from the description of job x in all the job postings that has job x. Then job to skill connections are created by using another method described in [177], which creates a graph that connects 4325 jobs to 6214 skills using $103,073$ edges.

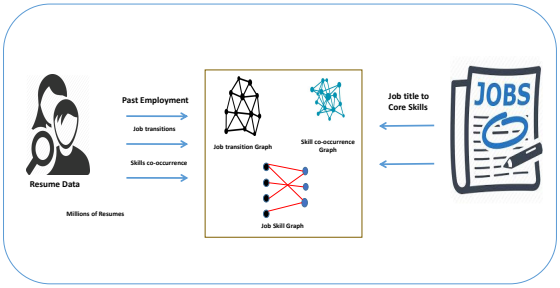


Fig. 6.1.: Data Preparation

6.4.2 Comparison Works

I conduct comparison experiments to show superiority of the proposed methods on our real-world dataset discussed in the previous section. For this experiment, I compare with three baseline methods.

- **Bigram:** It estimates the job transition probability based on first-order Markov assumption i.e. it uses popularity of the target job in the network to calculate the transition probability. It is considered as one of the most competitive method in practice for densely connected graphs.
- **AANE [53]:** Accelerated Attributed Network Embedding is a state-of-art embedding method that incorporates nodal attributes with topology using joint matrix factorization to learn low-dimensional network representation.
- **PTE [113]:** Predictive Text Embedding incorporates all three graphs, namely Job transition, Job skill and Skill co-occurrence, into single network representations using matrix factorization. The objective of this method is to minimize the distance between empirical similarity distribution and embedding similarity distribution using KL-divergence.

6.4.3 Experiment Settings

For conducting these experiments, I create train and test set using leave-one-out strategy i.e. for each job in G^{vv} I randomly keep one of the neighboring job as positive test instance and the remaining neighboring jobs as positive train instance. Ranking all jobs is a costly task, hence I uniformly select 100 jobs (over all jobs in the dataset) as negative test instances. For evaluation, I use Hit-Rate (HR@10) from top 10 ranked jobs and calculate Normalized Discounted Cumulative Gain (NDCG@10) up to position 10 for evaluation. I also calculate pair-wise/local AUC (Equation 6.10)

Table 6.1.: Comparison results for job transition recommendation. (Embedding dimension = 50)

Metrics	Bigram	AANE	PTE	*Joint-BPR*	*Joint-Margin*
HR@10	0.2742	0.8964	0.9260	0.9055	**0.9575**
NDCG@10	0.1479	0.7151	0.7332	0.7060	**0.7826**
local AUC	0.6682	0.9622	0.9722	0.9698	**0.9835**

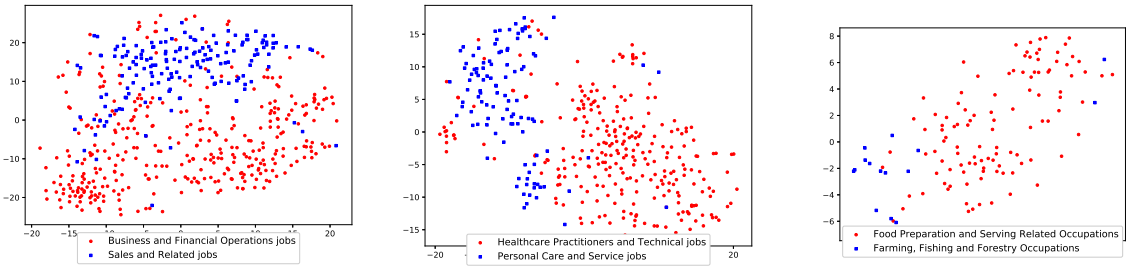
for test set $\mathcal{T}$ which includes pair (x, y), where x is a test node and y is a positive target job. I uniformly sample (100) negative jobs (TJ^{x-}) for each test node x.

$$\text{local AUC} = \frac{1}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} \frac{1}{|TJ^{x-}|} \sum_{z\in TJ^{x-}} \mathbb{1}(A^v_{xy} > A^v_{xz}) \tag{6.10}$$

For the proposed method, I use mini-batch gradient descent for the optimization. Specifically, I set learning rate as 0.1, regularization coefficient value as 0.0001, batch-size as 100 and number of epochs as 20. I use the same configuration for PTE for fair comparison. Also, I keep the embedding dimension as 50 for all embedding methods (*Joint-BPR*, *Joint-Margin*, AANE and PTE). For AANE, I perform grid search to select regularization parameter $\lambda = 0.01$ from set $\{0.0001, 0.001, 0.01, 0.1\}$ and penalty parameter $\rho = 0.1$ from set $\{0.05, 0.1, 0.5, 1\}$.

6.4.4 Comparison Results

The comparison results are depicted in the Table 6.1. I observe that both proposed methods (*Joint-BPR* and *Joint-Margin*) outperform the Bigram method substantially. Additionally, the proposed methods also improve the performance over AANE; for example, *Joint-Margin* achieves around 7% increment in hit-rate and 10% increment in NDCG over AANE. Notably, PTE performs better than other baselines and also outperforms one of the proposed methods (*Joint-BPR*) by small margin. One of the possible reasons for good performance by PTE is that it uses all three graphs to learn the job embedding. However, our proposed method *Joint-Margin*



(a) Bussiness and Finance VS. Sales

(b) Healthcare practitioners VS. Personal care service

(c) Food and serving VS. Farming and fishing

Fig. 6.2.: Clustering of similar job-categories in embedding space

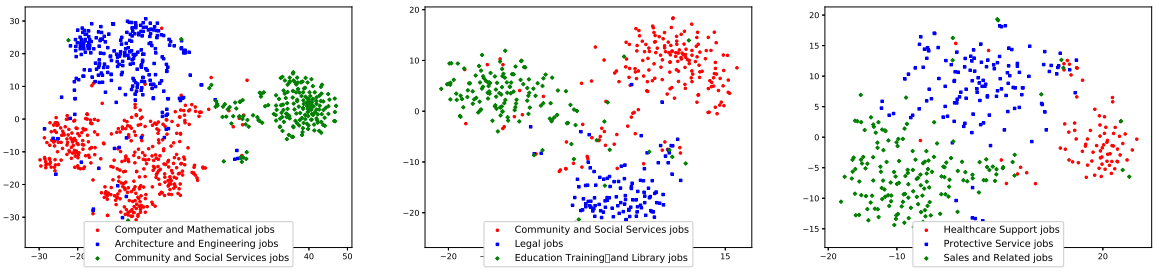


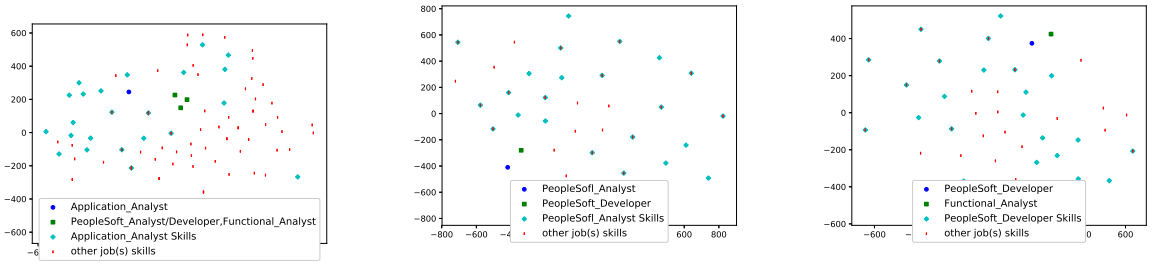
Fig. 6.3.: Clustering of non similar job-categories in embedding space

outperforms all competing methods and improves the Hit-rate by 3.4% and NDCG by 6.74% compared to the second best method, which is PTE.

6.4.5 Job Clustering

I categorize all jobs into different groups using O*Net [1] job-categories. I plot our job embeddings generated from our proposed *Joint-Margin* to check the clustering behavior of the jobs into the latent space. As I have many different job-categories, I plot two similar categories into same figure to show that proposed embeddings can distinguish similar categories in the latent space, for example I plot "Bussiness and Finance jobs" with "Sales and related jobs" in Figure 6.2a , "Healthcare Practitioners jobs" with "Personal care and service jobs" in Figure 6.2b and "Food preparation

[1] `https://www.onetonline.org`



(a) Distance between Application analyst and other jobs

(b) Relation of PeopleSoft analyst and PeopleSoft developer

(c) Relation of PeopleSoft developer and Functional analyst

Fig. 6.4.: Comparison of job transitions of a user

and serving jobs" with "Farming, fishing and forestry jobs" in Figure 6.2c. I also plot three non-similar groups together to show strong clustering behavior among jobs from the same job-category as depicted in Figure 6.3. Notice that, though I do not use these O*Net job-category information to generate job embedding, still the proposed embedding generates correct clusters in the latent space.

6.4.6 Case Study

I study a professional, who switched 3 jobs during her career. She started working as "Application analyst" in August 1993 and switched to a new job as "PeopleSoft analyst" in January 1994. She worked there almost four years and then switched to a new job as "PeopleSoft developer" in October 1998. Finally after three years, in July 2001 she switched to a "Functional analyst" job and worked there until 2015. I observe that in the first job transition, the professional switched the job very quickly, probably because the job domain was not very suitable for her. However, judging from her longer job duration in the later jobs, I suspect that those jobs were satisfying and the later job transitions were merely for better career prospect and career progression.

To check the quality of the proposed *Joint-Margin*, I check recommendation for these jobs, i.e., "Application analyst", "PeopleSoft analyst" and "PeopleSoft devel-

Table 6.2.: Top 10 job and skill recommendations for 3 different job owners

Civil Engineer		Photographer		Cafe Manager	
Jobs	Skills	Jobs	Skills	Jobs	Skills
Structural Engineer	Civil Engineering	Creative Manager	Graphic Design	Operations Manager	Customer Service
Geotechnical Engineer	Surveying	Graphic Artist	Adobe InDesign	Food Service Manager	Management
Transportation Engineer	MicroStation	Production Artist	Printing	Restaurant Manager	Training
Traffic Engineer	Geotechnical Eng.	Art Director	Adobe Photoshop	General Manager	Leadership
Staff Engineer	Professional Eng.	Graphic Designer	Photography	Banquet Manager	Operations
Surveyor	AutoCAD	Production Director	Adobe Flash	Restaurant Supervisor	Rotation
CAD Technician	Land Development	Media Assistant	Adobe Dreamweaver	Store Director	Food Services
Transmission Engineer	Elevation	Package Designer	Adobe Creative Suite	Operations Manager	Retailing
Water and Wastewater Eng	Site Planning	Layout Designer (Arts)	Web Design	Field Service Manager	Sales
Architectural Designer	Physical Education	Videographer	Adobe Illustrator	Department Head (Sales)	Merchandising

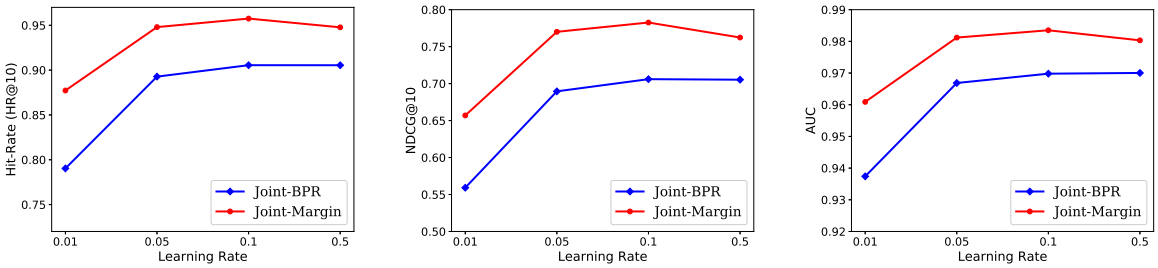
oper" using our job embedding. For "Application analyst" jobs, our recommendation suggests "PeopleSoft analyst" job at 45^{th} position. On the other hand, for "PeopleSoft analyst" job, our method suggests "PeopleSoft developer" job at 4^{th} rank. For the "PeopleSoft developer" job, the rank of "Functional analyst" job is 2. As we can see, the recommendation using the proposed embedding methodology provides good ranking for jobs that are satisfactory, and leads to better career prospect.

In Figure 6.4, I plot all four jobs and corresponding skills using job and skill embeddings to understand the recommendation behavior of our method. We observe from Figure 6.4a that "Application analyst" is far in latent space compared to other three jobs and skill set for "Application analyst" is also not well aligned with skill set of the other three jobs. However, there are a few overlapping skills which qualify "Application analyst" as a precursor for the other three jobs. Note that "Application analyst" is a generic job, from which 880 distinct job transitions happened in our dataset; among them "PeopleSoft analyst" is ranked 45 in recommendation, possibly because of the common skills between these two jobs. This explains the benefit of using skill information in the proposed embedding. In the remaining two plots in this figure, I show relation between "PeopleSoft analyst", "PeopleSoft developer" and "Functional analyst" jobs. As can be seen in Figures 6.4b, 6.4c, their skill sets are highly indistinguishable in the latent space as these jobs are well aligned in the career progression trajectory in that job sector.

6.4.7 Example of Job and Skill Recommendations

Here, I study recommendations for three job owners from different domains and show that the proposed method provides highly relevant suggestions. As I mentioned before, the proposed method embeds the jobs and skills into a shared latent space, thus I can provide job recommendation as well as skill recommendation. I list top 10 recommended jobs and skills for three different jobs in Table 6.2. From the table, I observe that the recommended jobs are typically more specialized job or a job that one can achieve after being promoted. For instance, a "photographer" is recommended "graphic artist" and "production artist" jobs, which are more specialized jobs than the photographer. He is also suggested jobs, such as "creative manager" and "art director", which are jobs that a photographer can obtain after a career advancement. For skill recommendation, I provide highly relevant (mandatory) and advanced skills required in the domain as shown in the Table 6.2. For example, for a good "photographer", knowing "photography", "printing" and "Adobe photoshop" are mandatory, but he could get suitable higher level jobs if he acquires skills such as "graphic design", "web designing" and different Adobe tools. Similar observations can be made for the recommendation of other two jobs that I have studied.

Note that, by sharing information among three different input graphs our proposed model can recommend skills which have not been associated to a job in the job-skill bipartite graph. For instance, the graph G^{js} does not contain any connection between the job"civil engineer" and the skill "site planning". However, our proposed method is able to learn the relation between these two and put them nearer in the latent space such that the skill appears as top skills for the "civil engineer" job. This verifies our claim that while learning the job and skill representations, I leverage information from all three graphs and this method provides higher quality job and skill embedding.



(a) Learning Rate vs HR (b) Learning Rate vs NDCG (c) Learning Rate vs AUC

Fig. 6.5.: Performance of the proposed models for different learning rate values

6.4.8 Parameter Study

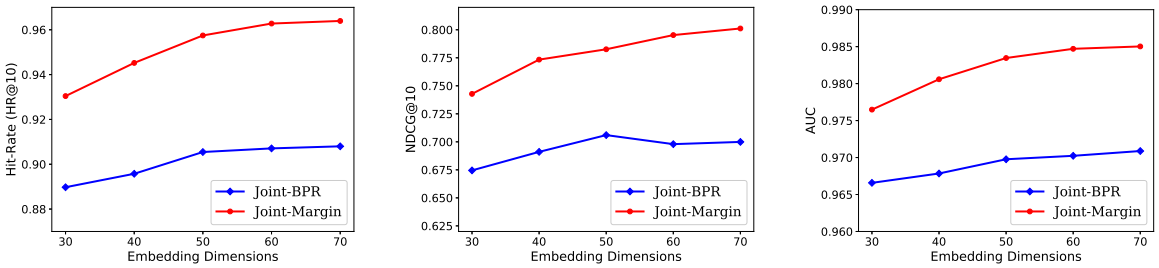
For the proposed method, there are two major influential parameters, first is the learning rate (α) and second is the embedding dimension (k). In this study, I analyze the influence of both parameters separately over the recommendation performance of the proposed methods. For both experiments, I keep the regularization coefficient, batch-size and epoch count the same as mentioned in Section 6.4.3.

Learning Rate Study

For this experiment, I keep the embedding dimensions as 50 and select the learning rate from set $\{0.01, 0.05, 0.1, 0.5\}$. I report the performance using Hit-Rate, NDCG and AUC in Figure 6.5. In the figure, we can observe that the performance of the proposed models improve drastically from learning rate 0.01 to 0.05, precisely 27.5% and 22.5% improvement in terms of NDCG@10 for *Joint-BPR* and *Joint-Margin* respectively. The possible explanation is due to the fact that with lower learning rate, the model converges slowly, while with higher learning rate, the model will converge to its optimum quickly.

Embedding Dimension Study

For this experiment, I keep the learning rate as 0.1 and select the embedding dimensions from the set $\{30, 40, 50, 60, 70\}$. I report the performance using Hit-Rate, NDCG and AUC in the Figure 6.6. This figure shows continuous improvement in the performance for larger embedding dimensions, however, this improvement is not



(a) Embedding Dimensions vs HR (b) Embedding Dimensions vs NDCG (c) Embedding Dimensions vs AUC

Fig. 6.6.: Performance of the proposed models for different embedding dimensions

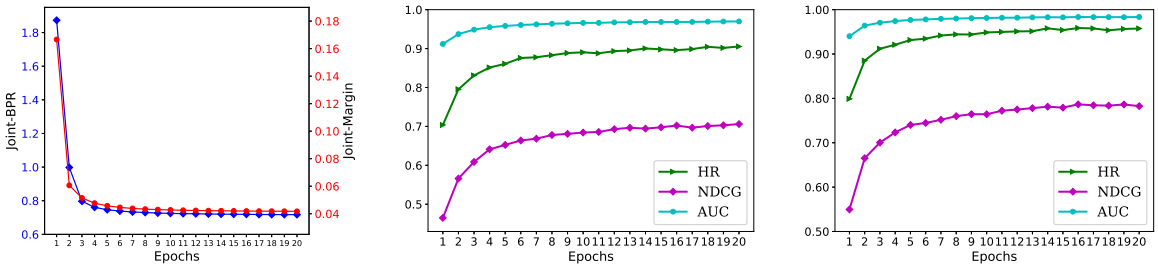
noticeable after embedding dimensions reach 50. For example, there is around 5% improvement in terms of NDCG@10 when the embedding dimension increases from 30 to 50 for *Joint-BPR*, but performance degrades in terms of NDCG@10 when the embedding dimension increases from 50 to 70.

6.4.9 Convergence Study

Finally I study the convergence behavior of the proposed methods. In particular, I calculate objective function value (Equation 6.7) at each epoch and plot the results in Figure 6.7a, where blue line represents *Joint-BPR* and red line shows the behavior of *Joint-Margin*. Figure 6.7a also shows that both models converge in 3-4 epochs. I also evaluate performance of the proposed models after each epoch using hit-rate, NDCG and AUC. For this evaluation, results are depicted in Figures 6.7b and 6.7c. From both figures, I observe that the recommendation performance of both embedding models become steady after a few epochs.

6.5 Chapter Summary

I propose a novel representation learning based solution for attributed network with inter-related attributes. The proposed representation learning model utilizes



(a) Convergence of the objective function

(b) *Joint-BPR* performance over epochs

(c) *Joint-Margin* performance over epochs

Fig. 6.7.: Convergence study for both proposed embedding methods

the pairwise ranking objective which learns job and skill vector representations into a shared latent space using three pre-processed graphs. The embedding approach utilize node-node connection and attribute-attribute connection information to elevate attribute and node embedding vectors,respectively. I show through real-world dataset that this joint embedding approach not only allows us to provide high quality job recommendation but also provides skill suggestions required to obtain the new job. The comparison experiments and case studies demonstrate that our proposed methodology consistently outperforms several existing state-of-the-arts for the job and skill recommendation.

7. ATTRIBUTED NETWORK EMBEDDING FOR SPARSE INDEPENDENT ATTRIBUTES

7.1 Introduction

The past few years, network embedding is attracting many researchers for its superior ability to solve different network analysis problems. This embedding based local node features play a vital role in performance while solving node based network analysis tasks such as node classification [41], link prediction [31], and community detection [42]. Because of wide applicability, researchers proposed many network embedding methods such as DeepWalk [32], LINE [33], Node2Vec [31], and SDNE [45]. Most of these existing embedding methods capture neighborhood information by distributing topologically similar nodes closely in the learned vector space. But these methods are unable to use auxiliary information of a network that is available as node/edge attributes.

Recently, a few works have been proposed which consider attributed network embedding [52,53,114]; however, the majority of these methods use a matrix factorization approach, which suffers from some crucial limitations. For example, earliest among these works is Text-Associated DeepWalk (TADW) [52], which incorporates the text features of nodes into DeepWalk by factorizing a matrix $\mathbf{M}$ constructed from the summation of a set of graph transition matrices. But, SVD based matrix factorization is both time and memory consuming, which restricts TADW to scale up to large datasets. Furthermore, obtaining an accurate matrix $\mathbf{M}$ for factorization is difficult and TADW instead factorizes an approximate matrix, which reduces its representation capacity. Huang et al. [53] proposed another matrix factorization (MF) based method, known as, Accelerated Attributed Network Embedding (AANE). It suffers from the same limitation as TADW. Another crucial limitation of the above methods

is that they have a design matrix which they factorize, but such a matrix cannot deal with nodal attributes of rich types. In summary, the representation power of a matrix factorization based method is found to be poorer than a neural network based method, as I will show in the experiment section of this chapter.

I found two most recent attributed network embedding methods, GraphSAGE and Graph2Gauss, which use deep neural network methods. To generate embedding of a node, GraphSAGE [58] aggregates embedding of its multi-hope neighbors using a convolution neural network model. GraphSAGE has a high time complexity, besides such ad-hoc aggregation may introduce noise which adversely affects its performance. Recently, Bojchevski et al. [56] proposed the Graph2Gauss (G2G), where they embed each node as a Gaussian distribution. G2G uses a neural network based deep encoder to process the nodal attributes and obtains an intermediate hidden representation, which is then used to generate the mean vector and the covariance matrix of the learned Gaussian distribution of a node. As a result, in G2G's learning, the interaction between the attribute information and the topology information of a node is poor. On the other hand, the learning pipeline of the proposed *Neural-Brane* enables effective information exchange between the attribute and topology of a node, making it much superior than G2G while learning embedding for attributed networks. It is worth noting that some recent works have proposed semi-supervised attributed network embedding considering the availability of node labels [54, 178], but the focus in this embedding method is unsupervised attributed network embedding, for which vertex labels are not available.

7.1.1 The solution and contribution.

I present *Neural-Brane*, a novel method for attributed network embedding. For a vertex of the input network, *Neural-Brane* infuses its network topological information and nodal attributes by using a custom neural network model, which returns a single representation vector capturing both the aspects of that vertex. The loss function

of *Neural-Brane* utilizes BPR [175] to capture attribute and topological similarities between a pair of nodes in their learned representation vectors. Specifically, the BPR objective elevates the ranking of a vertex-pair having similar attributes and topology by embedding the vertices in close proximity in the representation space, in comparison to other vertex-pairs which are not similar. The key aspects of this chapter are:

1. I propose *Neural-Brane*, a custom neural network based model for learning node embedding vectors by integrating local topology structure and nodal attributes. The source code (with datasets) of the *Neural-Brane* is available at: `https://git.io/fNF6X`

2. *Neural-Brane* has a novel neural network architecture which enables effective mixing of attribute and structure information for learning node representation vectors capturing both the aspects of a node. Besides, it uses Bayesian personalized ranking as its objective function, which is superior than cross-entropy based objective function used in several existing network embedding works.

3. Extensive validations on four real-world datasets demonstrate that *Neural-Brane* consistently outperforms 10 state-of-the-art methods, which results in up to 25% Macro-F1 lift for node classification and more than 10% NMI gain for node clustering respectively.

7.2 Problem Statement

Let $G = (\mathcal{V}, \mathcal{E}, \mathbf{A})$ be an attributed network, where $\mathcal{V}$ is a set of n nodes, and $\mathcal{E}$ is a set of edges, and $\mathbf{A}$ is a $n \times m$ binary attribute matrix such that the row $\mathbf{a}_i$ denotes a row attribute vector associated with node i in G. Each edge $(i, j) \in \mathcal{E}$ is associated with a weight w_{ij}. The neighbors of node i is represented as $\mathcal{N}(i)$. m is the number of node attributes in $\mathbf{A}$. I use $\mathcal{A}(i)$ to denote the non-zero attribute set of node i.

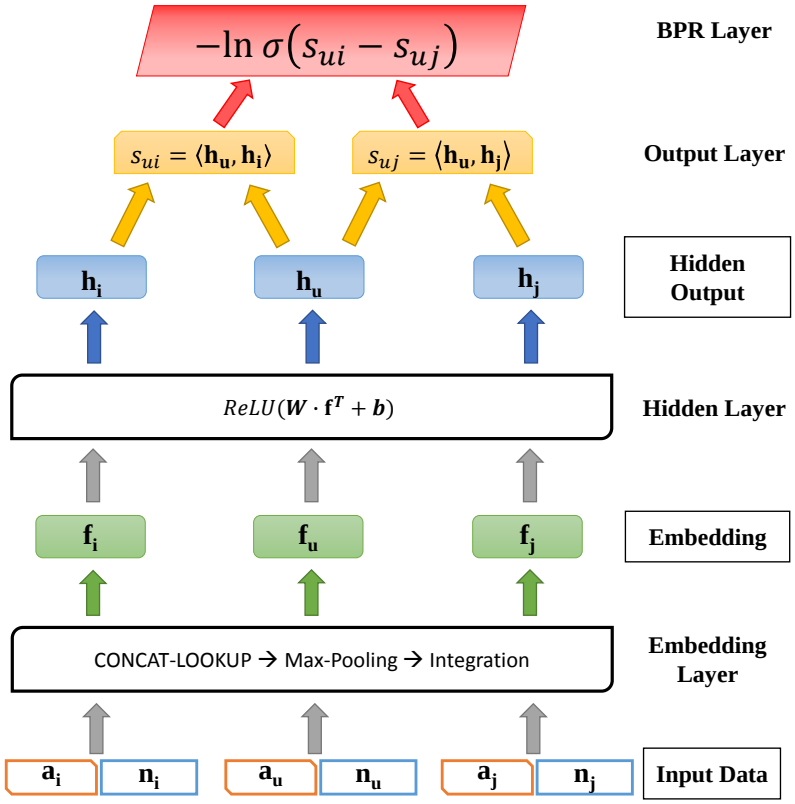


Fig. 7.1.: *Neural-Brane* architecture. Given a node u, $\mathbf{a}_u$ is its binary attribute vector and $\mathbf{n}_u$ is its adjacency vector. The model training uses node-triplets (u, i, j), such that $(u, i) \in \mathcal{E}$ and $(u, j) \notin \mathcal{E}$.

The attributed network embedding problem is formally defined as follows: given an attributed network $G = (\mathcal{V}, \mathcal{E}, \mathbf{A})$, I aim to obtain the representation of its vertices as a $n \times d$ matrix $\mathbf{F} = [\mathbf{f}_1^T, ..., \mathbf{f}_n^T]^T \in \mathbb{R}^{n \times d}$, where $\mathbf{f}_i$ is the row vector representing the embedding of node i. The representation matrix $\mathbf{F}$ should preserve the node proximity from both network topological structure $\mathcal{E}$ and node attributes $\mathbf{A}$. Eventually, $\mathbf{F}$ serves as feature representation for the vertices of G, as such, that they can be used for various downstream network mining tasks.

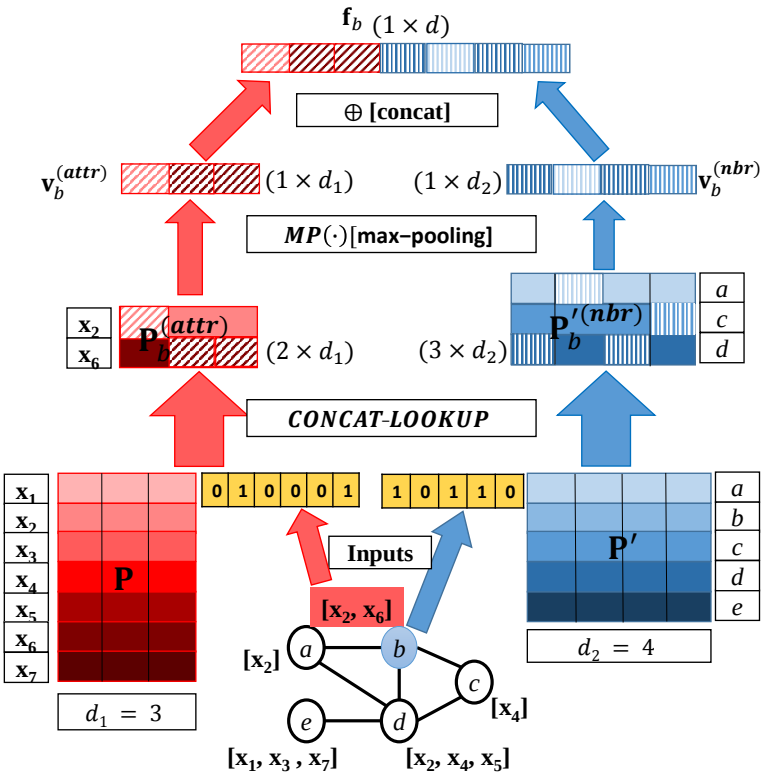


Fig. 7.2.: The figure shows the mechanism of the embedding layer for the vertex b of a toy attributed graph. The graph contains 5 vertices and 6 edges, where each vertex is associated with a collection of nodal attributes. For example, vertex b is connected to vertices $\{a, c, d\}$ and associated with attributes $\{\mathrm{x}_2, \mathrm{x}_6\}$, respectively. The cardinality of the attribute set $\{\mathrm{x}_1, \cdots, \mathrm{x}_7\}$ is 7.

7.3 *Neural-Brane*: Attributed Network Embedding Framework

In this section, I discuss the proposed neural Bayesian personalized ranking model for attributed network embedding. The model uses a neural network architecture with embedding layer, hidden layer, output layer, and BPR layer from bottom to top, as illustrated in Figure 7.1. Specifically, the embedding layer learns a unified vector representation of a node from the vector representation of its nodal attributes and neighbors; the hidden layer applies nonlinear dimensionality reduction over the embedding vectors of the nodes, the output layer and the BPR layer enable model inference through back-propagation.

7.3.1 Embedding Layer

The embedding layer has two embedding matrices $\mathbf{P}$, and $\mathbf{P}'$; each row of $\mathbf{P}$ is a d_1 dimensional vector representation of an attribute, and each row of $\mathbf{P}'$ is a d_2 dimensional vector representation of a vertex (both d_1 and d_2 are user-defined parameter). These matrices are updated iteratively during the learning process. For a given vertex u, embedding layer produces u's latent representation vector $\mathbf{f}_u$ by learning from embedding vectors of u's attributes and neighbors, i.e., corresponding rows of $\mathbf{P}$ and $\mathbf{P}'$, respectively; thus the neighbors and attributes of u are jointly involved in the construction of u's latent representation vector ($\mathbf{f}_u$), which enables *Neural-Brane* to bring the latent representation vectors of nodes with similar attributes and neighborhood in close proximity in the latent space.

I illustrate the vector construction process using a toy attributed graph in Figure 7.2. Given the vertex b from the toy graph, the embedding layer first takes its attribute and adjacency vectors (from $\mathbf{P}$ and $\mathbf{P}'$) as input and then generates its corresponding attributional and nodal embedding matrices ($\mathbf{P}_b^{(attr)}$ and $\mathbf{P}_b'^{(nbr)}$) by using the *CONCAT-LOOKUP*$(\cdot)$ function. After that, attributional and neighborhood embedding vectors are obtained from $\mathbf{P}_b^{(attr)}$ and $\mathbf{P}_b'^{(nbr)}$ by using the max-pooling operation respectively. Finally, the learned attributional and neighborhood embedding vectors are concatenated together to obtain the final embedding representation of the vertex b. Below I provide more details of the operations in embedding layer.

Encoding attributional information.

Given a node $u \in \mathcal{V}$ and the attribute matrix $\mathbf{A}$, $\mathbf{a}_u \in \mathbb{R}^{1\times m}$ is $\mathbf{A}$'s row corresponding to u's binary attribute vector. I apply a row-wise concatenation based embedding lookup layer to transform $\mathbf{a}_u$ into a latent matrix, $\mathbf{P}_u^{(attr)}$, as shown below:

$$\mathbf{P}_u^{(attr)} = \textit{CONCAT-LOOKUP}(\mathbf{P}, \mathbf{a}_u), \tag{7.1}$$

where $\mathbf{P} \in \mathbb{R}^{m \times d_1}$ is the attribute embedding matrix in which each row is a d_1 (user defined parameter) sized vector representation of an attribute. Lookup is performed by *CONCAT-LOOKUP*$(\cdot)$ function which first performs a row projection on $\mathbf{P}$ by selecting the rows corresponding to the attribute-set $\mathcal{A}(u)$ and then stacks the selected vectors row-wise into the matrix $\mathbf{P}_u^{(attr)} \in \mathbb{R}^{|\mathcal{A}(u)| \times d_1}$. Then I apply a max-pooling operation on the generated $\mathbf{P}_u^{(attr)}$ matrix in order to transform it into a single vector. Specifically, max-pooling operation retains the most informative signal by extracting the largest value in each dimension (i.e., column) of the matrix $\mathbf{P}_u^{(attr)}$ to obtain $\mathbf{v}_u^{attr}$.

$$\mathbf{v}_u^{attr} = MP(\mathbf{P}_u^{(attr)}), \tag{7.2}$$

where $\mathbf{v}_u^{attr} \in \mathbb{R}^{1 \times d_1}$ is the latent vector representation of node u based on its attributional signals, and $MP(\cdot)$ denotes the max-pooling operation.

Encoding network topology.

Given a node u, I describe its neighborhood by using a binary adjacency vector, denoted as $\mathbf{n}_u \in \mathbb{R}^{1 \times n}$, in which u's neighbors are set to 1, and the rest of entries are set as 0. Similar to the operations I use for encoding the attributional information, I apply a row-wise concatenation based lookup layer to transform $\mathbf{n}_u$ into a latent matrix $\mathbf{P}_u'^{(nbr)}$ and then apply max-pooling operation on the obtained latent matrix. Thus,

$$\mathbf{P}_u'^{(nbr)} = \textit{CONCAT-LOOKUP}(\mathbf{P}', \mathbf{n}_u) \tag{7.3}$$

$$\mathbf{v}_u^{nbr} = MP(\mathbf{P}_u'^{(nbr)}), \tag{7.4}$$

where $\mathbf{P}' \in \mathbb{R}^{n \times d_2}$ is the neighborhood embedding matrix for lookup (similar to matrix $\mathbf{P}$), and $\mathbf{P}_u'^{(nbr)} \in \mathbb{R}^{|\mathcal{N}(u)| \times d_2}$ is the obtained latent matrix generated from the *CONCAT-LOOKUP*$(\cdot)$ function. Moreover, $\mathbf{v}_u^{nbr} \in \mathbb{R}^{1 \times d_2}$ obtained from the $MP(\cdot)$

operation is the latent vector representation of node u based on its neighborhood topology.

Integration component.

Once I obtain the vector representation of node u from both its attributional information and topological structure as developed in Equations 7.1, 7.2, 7.3 and 7.4, I further integrate both latent vectors into a unified vector representation by vector concatenation, as shown below:

$$\mathbf{f}_u = \mathbf{v}_u^{attr} \;||\; \mathbf{v}_u^{nbr} := [\mathbf{v}_u^{attr} \;\; \mathbf{v}_u^{nbr}], \tag{7.5}$$

where $\mathbf{f}_u \in \mathbb{R}^{1\times d}$ $(d_1 + d_2 = d)$, and "||" denotes the vector concatenation operation.

7.3.2 Hidden Layer

Given the obtained embedding vector $\mathbf{f_u} \in \mathbb{R}^{1\times d}$ for node u in the attributed network G, the hidden layer aims to transform its embedding vector into another representation $\mathbf{h}_u$, in which signals from attributes and neighborhood of a vertex interact with each other. Formally, given $\mathbf{f}_u$, the hidden layer produces $\mathbf{h}_u \in \mathbb{R}^{1\times h}$ by the following formula:

$$\mathbf{h}_u^T = ReLU(\mathbf{W}\mathbf{f}_u^T + \mathbf{b}) \tag{7.6}$$

Here I use rectified linear function $ReLU(x)$, defined as $\max(0, x)$, as the activation function for achieving better convergence speed. Parameters $\mathbf{W} \in \mathbb{R}^{h\times d}$ and $\mathbf{b} \in \mathbb{R}^{h\times 1}$ are weights and bias for the hidden layer, respectively; h is a user-defined parameter denoting the number of neurons in the hidden layer. It is worth mentioning that in the hidden layer, all the nodes share the same set of parameters $\{\mathbf{W}, \mathbf{b}\}$, which enables information sharing across different vertices (see the box denoted as "Hidden Layer" in Figure 7.1).

7.3.3 Output and BPR Layers

Given a node pair u and i, I use their corresponding representations $\mathbf{h}_u$ and $\mathbf{h}_i$ from hidden layer (Equation 7.6) as input for the output layer. The task of this layer is to measure the similarity score between a pair of vertices by taking the dot product of their representation vectors. Since this computation uses the vector representation of the vertices from the hidden layer, it encodes both attribute similarity and neighborhood similarity jointly. The similarity score between vertices u and i, defined as s_{ui}, is calculated as $\langle \mathbf{h}_u, \mathbf{h}_i \rangle$.

BPR layer implements the Bayesian personalized ranking objective. For the embedding task, the ranking objective is that the neighboring nodes in the graph should have more similar vector representations in the embedding space than non-neighboring nodes. For example, the similarity score between two neighboring vertices u and i, should be larger than the similarity score between two non-neighboring nodes u and j. As shown in Figure 7.1, given the vertex triplet (u, i, j), I model the probability of preserving ranking order $s_{ui} > s_{uj}$ using the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$. Mathematically,

$$\begin{aligned} P\big(s_{ui} > s_{uj} | \mathbf{h}_u, \mathbf{h}_i, \mathbf{h}_j\big) &= \sigma\big(s_{ui} - s_{uj}\big) \\ &= \frac{1}{1 + e^{-\big(\langle \mathbf{h_u}, \mathbf{h}_i \rangle - \langle \mathbf{h}_u, \mathbf{h}_j \rangle\big)}} \end{aligned} \tag{7.7}$$

As I observe from Equation 7.7, the larger the difference between s_{ui} and s_{uj}, the more likely the ranking order $s_{ui} > s_{uj}$ is preserved. By assuming that all the triplet based ranking orders generated from the graph G to be independent, the probability of all the ranking orders being preserved is defined as follows:

$$\prod_{(u,i,j) \in \mathcal{D}} P(i >_u j) = \prod_{(u,i,j) \in \mathcal{D}} \sigma\big(s_{ui} - s_{uj}\big), \tag{7.8}$$

where $\mathcal{D}$ represents training triplet sets generated from G and $i >_u j$ is a shorthand notation denoting $s_{ui} > s_{uj}$; the notation is motivated from the concept that i is larger than j considering the partial order relation $>_u$.

The goal of my attributed network embedding is to maximize the expression in Equation 7.8. For the computational convenience, I minimize the sum of negative-likelihood loss function, which is shown as below:

$$\mathcal{L}(\Theta) = - \sum_{(u,i,j)\in\mathcal{D}} \ln \sigma\big(s_{ui} - s_{uj}\big) + \lambda \cdot ||\Theta||_F^2 \tag{7.9}$$

where $\Theta = \{\mathbf{P}, \mathbf{P}', \mathbf{W}, \mathbf{b}\}$ are model parameters used in all different layers, and $\lambda \cdot ||\Theta||_F^2$ is a regularization term to prevent model overfitting.

7.3.4 Model inference and optimization

I employ the back propagation algorithm by utilizing mini-batch gradient descent to optimize the parameters $\Theta = \{\mathbf{P}, \mathbf{P}', \mathbf{W}, \mathbf{b}\}$ in this model. First step of mini-batch gradient descent is to sample a batch of triplets from G. Specifically, given an arbitrary node u, I sample one of its neighbors i, i.e., $i \in \mathcal{N}(u)$, with the probability proportional to the edge weight w_{ij}. On the other hand, I sample its non-neighboring node j, i.e., $j \notin \mathcal{N}(u)$, with the probability proportional to the node degree in the graph. Next, for each mini-batch training triplets, I compute the derivative and update the corresponding parameters Θ. For that, first I find the gradient of the objective function in Equation 7.9 with respect to model parameter,

$$\begin{aligned}\frac{\partial \mathcal{L}(\Theta)}{\partial \Theta} &= - \sum_{(u,i,j)\in\mathcal{D}} \frac{\partial \ln \sigma\big(s_{ui} - s_{uj}\big)}{\partial \Theta} + \lambda \frac{\partial ||\Theta||_F^2}{\partial \Theta} \\ &= - \sum_{(u,i,j)\in\mathcal{D}} \big(1 - \sigma(s_{ui} - s_{uj})\big) \cdot \frac{\partial}{\partial \Theta}\big(s_{ui} - s_{uj}\big) + 2\lambda ||\Theta||_F \end{aligned} \tag{7.10}$$

Now, for each model parameter I find $\frac{\partial}{\partial \Theta}(s_{ui} - s_{uj})$ using the chain rule. In particular, by back-propagating from Bayesian personalized ranking layer to hidden

Algorithm 6: *Neural-Brane* Framework

Input: $G = (\mathcal{V}, \mathcal{E}, \mathbf{A})$, embedding dimensions d_1, d_2, batch size b, learning rate α, regularization coefficient λ.

Output: Attributional embedding matrix $\mathbf{P}$ and neighborhood embedding matrix $\mathbf{P}'$.

Initialize all model parameters $\Theta = \{\mathbf{P}, \mathbf{P}', \mathbf{W}, \mathbf{b}\}$ with 0 mean and 0.01 standard deviation from the Gaussian distribution.
repeat
  Construct the mini-batch of node-triples (u, i, j).
  Calculate $\mathbf{f}_u, \mathbf{f}_i, \mathbf{f}_j$ using Equations 7.1, 7.2, 7.3, 7.4, 7.5.
  Calculate $\mathbf{h}_u, \mathbf{h}_i, \mathbf{h}_j$ based on the Equation 7.6.
  Calculate $s_{ui} = \langle \mathbf{h}_u, \mathbf{h}_i \rangle$ and $s_{uj} = \langle \mathbf{h}_u, \mathbf{h}_j \rangle$
  Calculate $\mathcal{L}(\Theta)$ using Equation 7.9.
  Update the gradients of $\Theta = \{\mathbf{P}, \mathbf{P}', \mathbf{W}, \mathbf{b}\}$ using the back-propagation.
until Convergence
return $\mathbf{P}, \mathbf{P}'$.

layer, I update the gradients w.r.t. weight matrix $\mathbf{W}$ and bias vector $\mathbf{b}$ accordingly. Then in the embedding layer, I update the gradients of the corresponding embedding vectors (i.e., rows) in $\{\mathbf{P}, \mathbf{P}'\}$ associated with all the neighboring nodes and attributes involved in each mini-batch training triplets respectively. Mathematically,

$$\Theta^{t+1} = \Theta^t - \alpha \times \frac{\partial \mathcal{L}(\Theta)}{\partial \Theta} \tag{7.11}$$

where α is the learning rate. In addition, I initialize all model parameters Θ by using a Gaussian distribution with 0 mean and 0.01 standard deviation. The pseudo-code of the proposed *Neural-Brane* framework is summarized in Algorithm 6.

7.3.5 Model complexity analysis.

For the time complexity analysis, given the sampled training triplet set $\mathcal{D}$, the total costs of calculating and updating gradients of $\mathcal{L}$ w.r.t. corresponding embedding vectors involved in $\{\mathbf{P}, \mathbf{P}'\}$ are $\mathcal{O}(d)$. Similarly, the total costs of computing and updating gradients of $\mathcal{L}$ w.r.t. parameters $\{\mathbf{W}, \mathbf{b}\}$ in the hidden layer are $\mathcal{O}(hd + h)$.

To generate training mini-batch, I use degree proportional sampling and its time complexity is $\mathcal{O}(n)$. Therefore, the total computational complexity of the proposed methodology for *Neural-Brane* is $|\mathcal{D}| * \big(\mathcal{O}(d) + \mathcal{O}(hd + h) + \mathcal{O}(n)\big)$. As time complexity of the *Neural-Brane* is linear to the embedding size, hidden layer dimension and input graph size, it is extremely fast. For example, it takes around 15 minutes to learn embedding for the largest dataset *Arnetminer* (see Table 7.1). We can easily observe that the space complexity for the proposed *Neural-Brane* is proportional to input graph size and embedding size i.e. $\mathcal{O}(n \cdot d)$.

7.4 Experiments and Results

In this section, I first introduce the datasets and baseline comparisons used in this work. Then I thoroughly evaluate the proposed *Neural-Brane* through two downstream data mining tasks (node classification and clustering) on four real-world networks, for which node attributes are available. Finally, I analyze the quantitative experimental results, investigate parameter sensitivity, convergence behavior, and the effect of pooling strategy of *Neural-Brane*.

7.4.1 Experimental Setup

Datasets. I perform experiments on four real-world datasets, whose statistics are shown in Table 7.1. The largest among these networks has around 15.75K vertices, and 109.5K edges. Note that, publicly available networks exist, which are larger than the networks that I use in this work, but those larger networks are neither attributed nor they have class label for the vertices, so I cannot use those in the experiment. Nevertheless, the largest dataset *Arnetminer*, has more nodes, edges and attributes than datasets used by recent attribute embedding papers [52, 114]. More description of the datasets is given below.

Table 7.1.: Statistics of Four Real-World Datasets

Dataset	# Nodes	# Edges	# Attributes	# Classes
CiteSeer	3,312	4,732	3,703	6
Arnetminer	15,753	109,548	135,647	5
Caltech36	671	15,645	64	2
Reed98	895	17,631	64	2

CiteSeer[1] is a citation network, in which nodes refer to papers and links refer to citation relationship among papers. Selected keywords from the paper are used as nodal attributes. Additionally, the papers are classified into 6 categories according to its research domain, namely Artificial Intelligence (AI), Database (DB), Information Retrieval (IR), Machine Learning (ML), Human Computer Interaction (HCI), and Multi-Agent Analysis.

Arnetminer[2] is a paper relation network consisting of scientific publications from 5 distinct research areas. Specifically, I select a list of representative conferences and journals from each of them. 1) *Data Mining* (KDD, SDM, ICDM, WSDM, PKDD); 2) *Medical Informatics* (JAMIA, J. of Biomedical Info., AI in Medicine, IEEE Tran. on Medical Imaging, IEEE Tran. on Information and Technology in Biomedicine); 3) *Theory* (STOC, FOCS, SODA); 4) *Computer Vision and Visualization* (CVPR, ICCV, VAST, TVCG, IEEE Visualization and Information Visualization) 5) *Database* (SIGMOD, VLDB, ICDE). Authors and keywords similarity between two papers are used for building edges. Keywords from paper title and abstract are used as attributes.

Caltech36 and *Reed98* [179] are two university Facebook networks. Specifically, each node represents a user from the corresponding university and edge represents user friendship. The attributes of each node is represented by a 64-dimensional one-hot vector based on gender, major, second major/minor, dorm/house, and year. I use student/faculty status of a node as the class label.

[1]https://linqs.soe.ucsc.edu/data
[2]https://aminer.org/topic_paper_author

Baseline Comparison.

To validate the benefit of the proposed *Neural-Brane*, I compare it against 10 different methods. Among all the competing methods, DeepWalk, LINE, and Node2Vec are topology-oriented network embedding approaches. NNMF, DeepWalk + NNMF, GraphSAGE, PTE-KL, TADW, AANE and G2G are state-of-the-arts for combining both network structure and nodal attributes for network representation learning. Note that PTE-KL is a semi-supervised embedding approach, and I hold the label information out for a fair comparison.

1. **DeepWalk** [32]: It utilize Skip-Gram based language model to analyze the truncated uniform random walks on the graph.

2. **LINE** [33]: It embeds the network into a latent space by leveraging both first-order and second-order proximity of each node.

3. **Node2Vec** [31]: Similar to DeepWalk, Node2Vec designs a biased random walk procedure for network embedding.

4. **Non-Negative Matrix Factorization (NNMF):** The model captures both node attributes and network structure to learn topic distributions of each node.

5. **DW+NNMF:** It simply concatenates the vector representations learned by DeepWalk and NNMF.

6. **GraphSAGE** [58]: GraphSAGE presents an inductive representation learning framework that leverages node feature information (e.g., text attributes) to efficiently generate node embeddings in the network.

7. **PTE-KL** [113]: Predictive Text Embedding framework aims to capture the relations of paper-paper and paper-attribute under matrix factorization framework. The objective is based on KL-divergence between empirical similarity distribution and embedding similarity distribution.

8. **TADW** [52]: Text-associated DeepWalk combines the text features of each node with its topology information and uses the MF version of DeepWalk.

9. **AANE** [53]: Accelerated Attributed Network Embedding learns low-dimensional representation of nodes from network linkage and content information through a joint matrix factorization.

10. **G2G** [56]: Graph2Gauss learns node representation such that each node vector is a Gaussian distribution.

Parameter Setting and Implementation Details.

There are a few user-defined hyper-parameters in the proposed embedding model. I fix the embedding dimension $d = 150$ (same for all baseline methods) with $d_1 = d_2 = 75$. For the number of neurons in hidden layer h, I set it to be 150. For the regularization coefficient λ in the embedding model (see Equation 7.9), I set it as 0.00005. In addition to that, I fix the learning rate $\alpha = 0.5$ (see Equation 7.11) and batch size to be 100 during the model learning and optimization. For baseline methods such as GraphSAGE, PTE-KL, AANE, G2G and others, I select learning rate α from the set $\{0.01, 0.05, 0.1, 0.5\}$[3] using grid search. Similarly for PTE-KL, TADW and other baseline methods regularization coefficient λ is selected from the set $\{0.01, 0.001, 0.0001\}$. For random walk based baselines (DeepWalk and Node2Vec), I select the best walk length from the set $\{20, 40, 60, 80\}$. For the rest of hyper-parameters, I use default parameter values as suggested by their original papers.

[3]For GraphSAGE I also check smaller values of α i.e. $\{10^{-4}, 10^{-5}, 10^{-6}\}$ as suggested in the paper [58].

Table 7.2.: Quantitative results of Macro-F1 between the proposed *Neural-Brane* and other baselines for the node classification task using logistic regression on various datasets (embedding dimension = 150). [*GraphSAGE for Arnetminer is not able to complete after 2 days.]

Citeseer											
Train%	DeepWalk	LINE	Node2Vec	NNMF	DW+NNMF	GraphSAGE	PTE-KL	TADW	AANE	G2G	*Neural-Brane*
30%	0.4952	0.4304	0.5462	0.4367	0.5185	0.4418	0.5456	0.5756	0.5684	0.5860	**0.6375**$_{\pm.0075}$
50%	0.5199	0.4590	0.5632	0.4619	0.5598	0.4621	0.5647	0.5900	0.5844	0.5939	**0.6450**$_{\pm.0026}$
70%	0.5318	0.4600	0.5743	0.4711	0.5780	0.4662	0.5732	0.6106	0.5996	0.6003	**0.6508**$_{\pm.0115}$
Arnetminer											
Train%	DeepWalk	LINE	Node2Vec	NNMF	DW+NNMF	GraphSAGE*	PTE-KL	TADW	AANE	G2G	*Neural-Brane*
30%	0.7281	0.5364	0.7729	0.6087	0.6968	-	0.5341	0.7969	0.7902	0.8062	**0.8693**$_{\pm.0016}$
50%	0.7336	0.5422	0.7837	0.6541	0.7016	-	0.5426	0.8031	0.8009	0.8145	**0.8713**$_{\pm.0017}$
70%	0.7389	0.5485	0.7877	0.6748	0.7044	-	0.5519	0.8079	0.8065	0.8186	**0.8759**$_{\pm.0034}$
Caltech36											
Train%	DeepWalk	LINE	Node2Vec	NNMF	DW+NNMF	GraphSAGE	PTE-KL	TADW	AANE	G2G	*Neural-Brane*
30%	0.7824	0.8023	0.7859	0.5243	0.8480	0.7233	0.8701	0.8748	0.8527	0.8523	**0.9219**$_{\pm.0121}$
50%	0.7949	0.8079	0.8080	0.5953	0.8552	0.7712	0.8697	0.8866	0.8843	0.8691	**0.9285**$_{\pm.0134}$
70%	0.8217	0.8112	0.8131	0.6445	0.8712	0.8220	0.8786	0.8929	0.9008	0.8977	**0.9456**$_{\pm.0139}$
Reed98											
Train%	DeepWalk	LINE	Node2Vec	NNMF	DW+NNMF	GraphSAGE	PTE-KL	TADW	AANE	G2G	*Neural-Brane*
30%	0.7662	0.7195	0.7682	0.6472	0.8055	0.6325	0.8333	0.8460	0.8285	0.7515	**0.8788**$_{\pm.0105}$
50%	0.7774	0.7195	0.7805	0.7123	0.8275	0.7012	0.8413	0.8519	0.8433	0.7772	**0.8916**$_{\pm.0176}$
70%	0.7927	0.7446	0.7925	0.7695	0.8321	0.7682	0.8590	0.8636	0.8660	0.7925	**0.9033**$_{\pm.0146}$

7.4.2 Quantitative Results

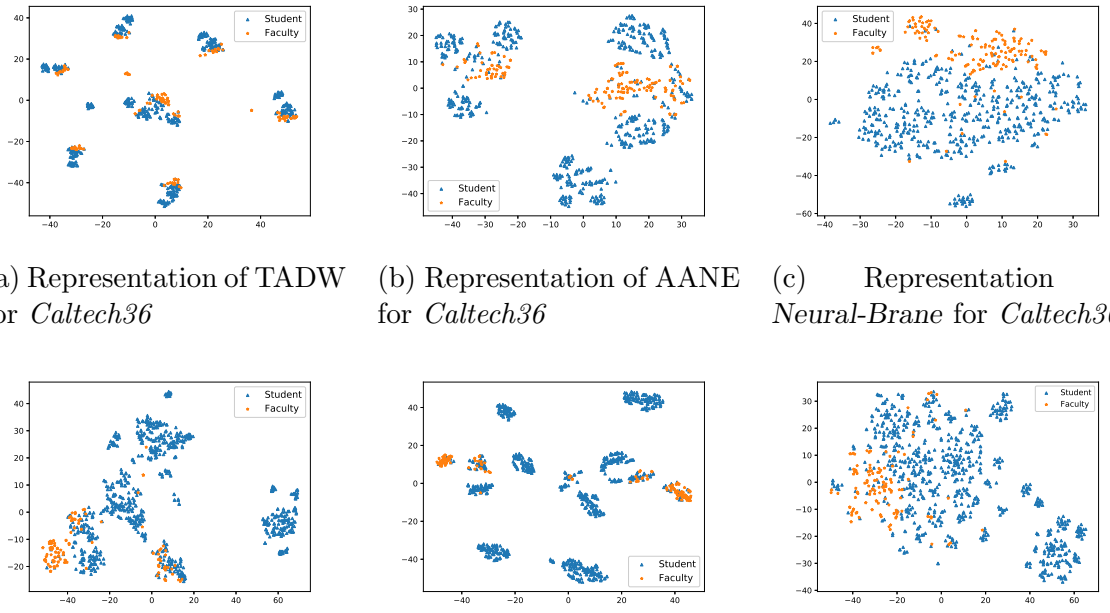
Node Classification.

For fair comparison between network embedding methods, I purposely choose a linear classifier to control the impact of complicated learning approaches on the classification performance. Specifically, I treat the node representations learned by different approaches as features, and train a logistic regression classifier for multi-class / binary classification. In each dataset, $p\% \in \{30\%, 50\%, 70\%\}$ of nodes are randomly selected as training set and the rest as test set. I use the widely used metric Macro-F1 [180] for classification assessment. Each method is executed 10 times and the average value is reported. For *Neural-Brane*, I also report standard deviation. For better visual comparison, I highlight the best Macro-F1 score of each training ratio (p) with bold font.

Table 7.2 shows results for node classification, where each column is an embedding method and rows represent different train splits (p). As I observe from Table 7.2,

performance of the last four (PTE-KL, TADW, AANE, G2G) baseline methods are highly competitive among each others. But, the proposed *Neural-Brane* consistently outperforms all these and other baseline methods under all training ratios. Moreover, the overall performance improvement that the *Neural-Brane* delivers over the second best method is significant. For example, in Citeseer dataset, when training ratio p ranges from 30% to 70%, *Neural-Brane* outperforms the G2G by 8.8%, 8.6%, 8.4% in terms of Macro-F1, respectively. Furthermore, the improvement over G2G is statistically significant (paired t-test with p-value $\ll 0.01$). The relatively good performance of the proposed *Neural-Brane* across various training ratios is due to the fact that the proposed neural Bayesian personalized ranking framework is able to generate high-quality latent features by capturing crucial ordering information between nodes and incorporating nodal attributes and network topology into network embedding. Furthermore, BPR is shown to be better suited than other loss functions, such as point-wise square loss in TADW and K-L divergence based objective in LINE and PTE-KL, for placing similar nodes in the embedding space for the downstream node classification task.

Among the competing methods, topology-oriented network embedding approaches such as LINE and DeepWalk perform fairly poor on all datasets. This is mainly because the network structure is rather sparse and only contains limited information. On the other hand, TADW is much better than DeepWalk due to the fact that textual contents contain richer signals compared to the network structure. When concatenating the embedding vectors from DeepWalk and NNMF, the classification performance is relatively improved compared to a single DeepWalk. However, the naive combination between DeepWalk and NNMF is far from optimal, compared to the proposed *Neural-Brane*. Note that, GraphSAGE for Arnetminer dataset is not able to complete after 2 days on contemporary server having 64 cores with 2.3 GHz and 132 GB memory.



(a) Representation of TADW for *Caltech36*

(b) Representation of AANE for *Caltech36*

(c) Representation of *Neural-Brane* for *Caltech36*

(d) Representation of TADW for *Reed98*

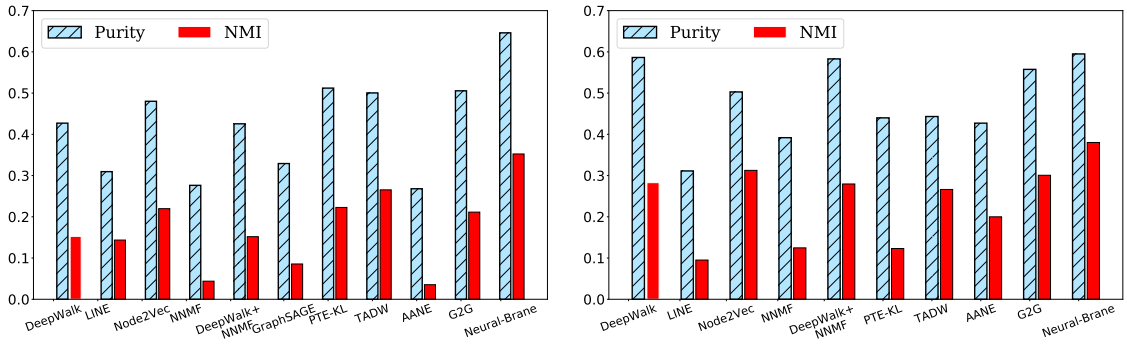
(e) Representation of AANE for *Reed98*

(f) Representation of *Neural-Brane* for *Reed98*

Fig. 7.3.: The visualization comparison among various embedding methodologies for *Caltech36* and *Reed98* datasets

Visualization and Node Clustering.

The primary goal of graph embedding approaches is to put similar nodes closer in their corresponding latent space, hence a desirable embedding method should generate clusters of similar nodes in the embedding space. Visualization for large number of classes in two dimensional space is impractical. Instead, in Figure 7.3, we plot $2D$ representation of learned vector representations for *Caltech36* and *Reed98* datasets. Note that both of these datasets contain only 2 classes and hence provide interpretable visualization. Specifically, I plot embedding representations of *Neural-Brane* along with two best competing methods, namely TADW and AANE. These figures clearly demonstrate that *Neural-Brane* provides better discrimination of classes through clustering in the latent space compared to both TADW and AANE.

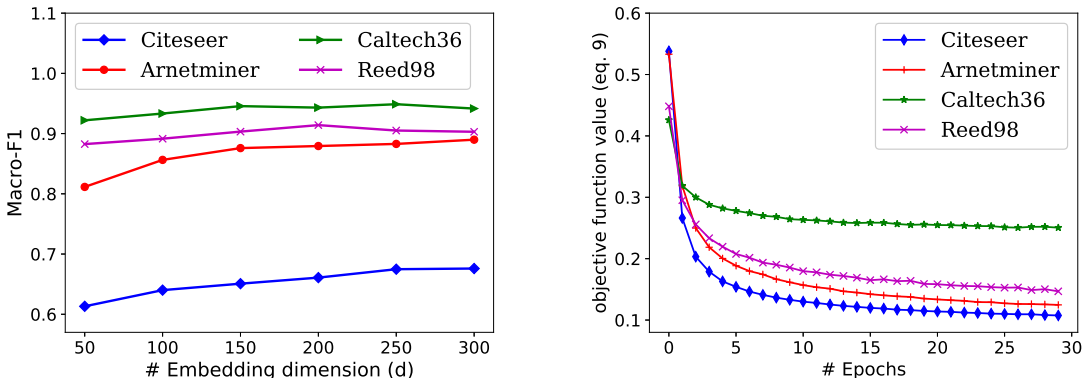


(a) *CiteSeer* Dataset (b) *Arnetminer* Dataset

Fig. 7.4.: The performance of node clustering

For the other two larger datasets (CiteSeer and Arnetminer), I use k-means clustering approach to the learned vector representations of nodes and utilize both Purity and Normalized Mutual Information (NMI) [180] to assess the quality of clustering results. Furthermore, I match the ground-truth number of clusters as input for running k-means, execute the clustering process 10 times to alleviate the sensitivity of centroid initialization, and report the average results.

The clustering results for both *CiteSeer* and *Arnetminer* datasets are depicted in Figure 7.4. As we can see, the proposed *Neural-Brane* consistently achieves the best clustering results in contrast to all competing baselines. For example, in Citeseer dataset, the proposed *Neural-Brane* achieves 0.3524 NMI. However, the best competing method PTE-KL only obtains 0.2653 NMI, indicating more than 32.8% gains. Similarly, for Arnetminer dataset, *Neural-Brane* obtains 34.5% improvements over the best competing approach DeepWalk in terms of NMI. The possible explanation for higher performance of *Neural-Brane* could be due to the fact that the proposed Bayesian ranking formulation directly optimizes the pairwise distance between similar and dissimilar nodes, thus making their corresponding vectors cluster-aware in the embedded space.



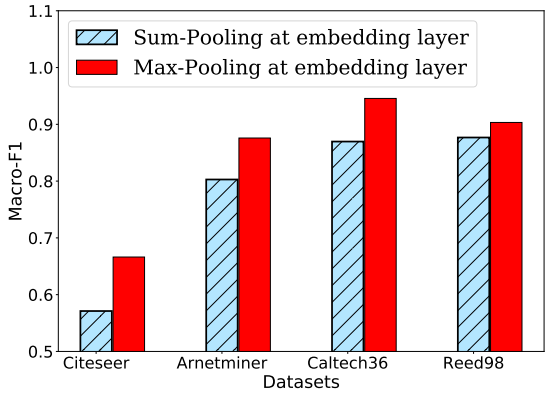
(a) The effect of embedding dimension for node classification

(b) Convergence analysis of the ranking objective function shown in Equation 7.9

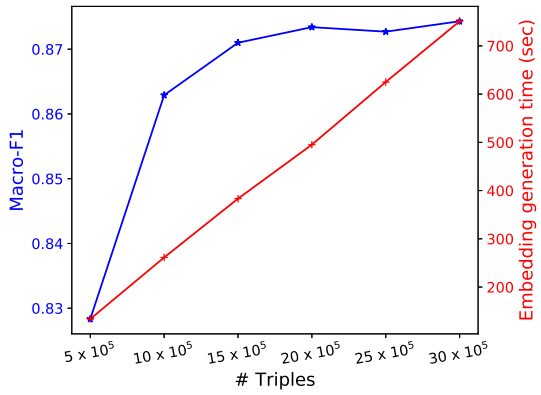
Fig. 7.5.: Analysis of the embedding dimension and convergence

7.4.3 Analysis of Parameter Sensitivity and Algorithm Convergence

I conduct experiments to demonstrate how the embedding dimension affects the node classification task using the proposed *Neural-Brane*. Specifically, I vary the number of embedding dimension parameter d as $\{50, 100, 150, 200, 250, 300\}$ and set the training ratio $p = 70\%$. I report the Macro-F1 results on all four datasets, which is shown in Figure 7.5a. As we observe, as the embedding dimension d increases, the classification performance in terms of Macro-F1 first increases and then tends to stabilize. The possible explanation could be that when the embedding dimension is too small, the embedding representation capability is not sufficient. However, when the embedding dimension becomes sufficiently large, it captures all necessary information from the data, leading to the stable classification performance. Furthermore, I investigate the convergence trend of *Neural-Brane*. As shown in Figure 7.5b, *Neural-Brane* converges approximately within 10 epochs and achieves promising convergence results in terms of the objective function value on all four datasets.



(a) The pooling strategy comparison for the task of node classification



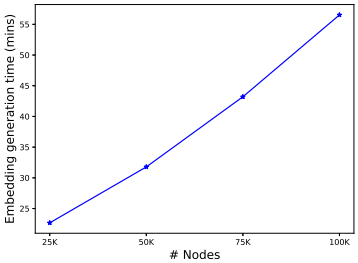
(b) Effect of # triples used for training on performance and time (*Arnetminer* dataset)

Fig. 7.6.: Study effects of pooling strategy and # training triples

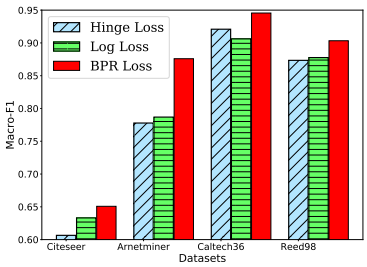
7.4.4 Effect of Pooling Strategy and Number of Training Triples

I investigate the effect of the pooling strategy in the embedding layer for the task of node classification. For the comparison, I consider to take the sum rather than max pooling and hold the rest of neural architecture and hyper-parameter settings constant. I report the Macro-F1 results on all four datasets with training ratio $p = 70\%$, which is shown in Figure 7.6a. As we observe, max pooling consistently outperforms the alternative sum pooling strategy for the task of node classification across all datasets. The possible explanation is due to the fact that the max-pooling operation returns the strongest signal for each embedding dimension, which alleviates noisy signals. On the other hand, the sum pooling operation considers accumulated signals from each input embedding dimension, which leads to inaccurate information aggregation.

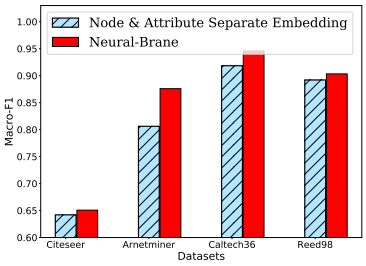
Finally, to verify the efficiency of the *Neural-Brane*, I study how embedding generation time and node classification performance varies with count of training triples. For that, I use *Arnetminer* dataset and plot macro-F1 results and embedding generation time over different counts of training triples in Figure 7.6b. I can see that for half a million triples the *Neural-Brane* doesn't render the optimal result as the



(a) Results for scalability study



(b) Results of performance using different loss functions



(c) Combined and independent learning of attribute and neighborhood embedding.

Fig. 7.7.: Scalability Study and importance of BPR loss and other layers of the *Neural-Brane*.

method is not converged. However, it converges with 1.5 million triples and consistently provides very good performance (high Macro-F1) for higher triple counts. Notice that, for this biggest dataset (*Arnetminer*), *Neural-Brane* takes around 6 minutes ($<$ 400 seconds) to sample the 1.5 million triples and train with those triples. This observation also proves that *Neural-Brane* is highly scalable because of its linear time complexity (Section 7.3.5).

7.4.5 Scalability study

To check the scalability of the proposed *Neural-Brane*, I conducted experiment to check run-times of various large synthetic networks. To generate these synthetic networks, I use popular Barabási-Albert preferential attachment model [16]. I generate low density random binary vector of size 500 as a synthetic attributes for each node. For this experiment, I vary size of the networks such that they have nodes in range of 25000 to 100000 with 25000 increment. The running time of these networks are depicted in Figure 7.7a, which shows a linear increase in run-time with the increase in size of the network. The empirical linear increase in run-time with respect to the size of the network is consistent with this model complexity analysis in Section 7.3.5.

7.4.6 Effectiveness of BPR loss and contribution of other *Neural-Brane* layers

As I discussed before, the ranking BPR loss as an objective function highly contributes towards the remarkable performance of the proposed *Neural-Brane*. To support this claim, I conduct comparison experiment where, I replace the objective function of the *Neural-Brane* with traditional Hinge loss and Cross-entropy (Log) loss. For fair comparison, I run the modified models with the same set of parameters discussed in Section 7.4.1. The performance of the modified methods and proposed *Neural-Brane* is shown in Figure 7.7b, where I can see that *Neural-Brane* with BPR loss always outperforms both Log loss and Hinge loss based methods.

Though, BPR loss helps in performance improvement of the *Neural-Brane*, I need to check the importance of embedding and hidden layers which are responsible for information fusion of topology and attributes. For this experiment, I feed attribute vector (v_b^{attr} for node b) directly to the output layer to learn attribute embedding ($\mathbf{P}$). Similarly, I feed neighborhood vector (v_b^{nbr} for node b) to output layer to learn neighborhood based embedding ($\mathbf{P}'$). I concatenate these vectors for each node as a final node representation vectors, I call this method as Node & attribute separate embedding. I compare the classification performance of this embedding method with proposed *Neural-Brane* and results are shown in Figure 7.7c. This comparison result shows that the embedding and hidden layers of the proposed method contributes towards improvement of the performance. Hence, From these results, I can conclude that both BPR loss as an objective function and advanced approach of information fusion using embedding and hidden layers, jointly produce superior performance for the proposed *Neural-Brane*.

7.5 Chapter Summary

I present a novel neural Bayesian personalized ranking formulation for attributed network embedding, which I call *Neural-Brane*. Specifically, *Neural-Brane* combines

a designed neural network model and a novel Bayesian ranking objective to learn informative vector representations that jointly incorporate network topology and nodal attributions. Experimental results on the node classification and clustering tasks over four real-world datasets demonstrate the effectiveness of the proposed *Neural-Brane* over 10 baseline methods.

8. INDEX FOR SHORTEST DISTANCE QUERY IN A DIRECTED NETWORK

8.1 Introduction

AS discussed in the Introduction chapter, finding shortest distance between two nodes in a directed graph is one of the most useful operations in graph analysis. While there are various traditional methods to solve the shortest path distance problem, the classical algorithms deem inefficient for providing real-time answers for a huge graphs. So, there is a growing interest for the discovery of more efficient methods for solving this task.

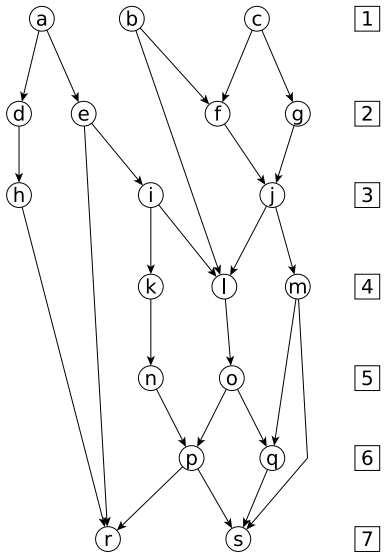
Various approaches are considered for obtaining an efficient distance query method for large graphs. One of them is to exploit topological properties of real-life networks that adhere to some specific characteristics. For instance, many researchers exploit the spatial and planar properties of road networks [119, 121, 123] to obtain efficient solutions for distance queries in road networks. However, for a general network from any other domain, such methods perform poorly [134]. The second approach is to perform pre-processing on the host graph and build an index data structure which can be used at runtime to answer the distance query between an arbitrary pair of nodes more efficiently. Several indexing ideas are used, but two are the most common, landmark-based indexing [127,129,130,135] and 2-hop-cover indexing [181]. Methods adopting the former idea identify a set of landmark nodes and pre-compute all-single source shortest paths from these landmark nodes. During query time, distances between a pair of arbitrary nodes are answered from their distances to their respective closest landmark nodes. Most of these methods deliver an approximation of shortest path distance except a method presented in [135]. Methods adopting the two-hop cover indexing generally find the exact solution for a distance query [25, 132, 182].

These methods store a collection of hops (paths starting from that node), such that the shortest path between a pair of arbitrary vertices can be obtained from the intersection of the hops of those vertices.

A related work to the shortest path problem is the reachability problem. Given a directed graph $G(V, E)$, and a pair of vertices u and v, the reachability problem answers whether a path exists from u to v. This problem can be solved in $O(|V|+|E|)$ time using graph traversal, where V is the set of vertices and E is the set of edges. However, using a reachability index, a better runtime can be obtained in practice. All the existing solutions [183, 184] of the reachability problem solve it for a directed acyclic graph (DAG). This is due to the fact that any directed graph can be converted to a DAG such that a DAG node is a strongly connected component (SCC) of the original graph; since any nodes in an SCC is reachable to each other, the reachability solution in the DAG easily answers a reachability query in the original graph. The indexing idea that we propose in this work also exploits the SCC, but unlike existing works we solve the distance query problem instead of reachability.

In this work, we propose *TopCom* [1], an indexing based method for obtaining exact solution of a distance query in an arbitrary directed graph. In principle, *TopCom* uses a 2-hop-cover solution, but its indexing is different from other existing indexing methods. Specifically, the basic indexing scheme of *TopCom* is designed for a DAG and it is inspired from the indexing solution of the reachability queries proposed in [185]. Due to its design, *TopCom* exhibits a very attractive performance for a DAG or general graph in which SCCs are relatively small. However, we also extend the basic indexing scheme so that it also solves the distance query for an arbitrary directed graph. We show experiment results that validate *TopCom*'s superior performance over IS-Label [25] and TreeMap [26] which are two of the fastest known indexing based shortest path methods in the recent years. Following other recent works, we also compare our method with bi-directional Dijkstra, which is a well-accepted baseline method for distance query solutions in a directed graph. Note that, this journal

[1] *TopCom* stands for **Top**ological **Com**pression which is the fundamental operation that is used to create the index data structure of this method.



Edge	*distance*
(d, h')	2
(e, k)	2
(e, l)	2
(f, l)	2
(f, m)	2
(g, l)	2
(g, m)	2
(k, p)	2
(l, p)	2
(l, q)	2

Fig. 8.1.: Pre-processing of DAG before Compression: (a) Original DAG G and (b) Modified DAG G_m. The dummy edges data structure ($DummyEdges$) associated with this modified DAG is shown to the right.

article is an extended version of a published conference article [186]; the conference article works for DAG only, but this work solves distance query indexing for arbitrary directed graphs.

8.2 Method

In this section, we discuss the shortest distance indexing of *TopCom* for a DAG. In subsequent section, we will show how this can be adapted for a general directed graph.

8.2.1 Topological compression

The main idea of *TopCom* is based on topological compression of DAG, which is performed during the index building step. During the compression, additional distance information is preserved in a data structure which *TopCom* uses for answering

a distance query efficiently. For the sake of simplicity, in subsequent discussion we assume that the given graph is unweighted for which the weight of each edge is 1 and the distance between two vertices is the minimum hop count between them. We will discuss the necessary adaptations that are needed for a weighted graph in the subsection 8.2.3.

Topological Level: Given a DAG G, we use $\mathcal{V}_G$ and $\mathcal{E}_G$ to represent set of vertices and edges of G, respectively. The topological level of any vertex $v \in \mathcal{V}_G$, defined as $topo(v)$, is 1 if v has no incoming edge, otherwise it is at least one higher than the topological level of any of v's parents. Mathematically,

$$topo(v) = \begin{cases} \max\limits_{(u,v)\in\mathcal{E}_G} \; topo(u) + 1, & \text{if } v \text{ has incoming edges} \\ 1, & \text{otherwise} \end{cases}$$

For a vertex v, if $topo(v)$ is even, we call v an even-topology vertex, otherwise v is an odd-topology vertex. An edge, $e = (u, v) \in \mathcal{E}_G$, is a single-level edge if $topo(v) - topo(u) = 1$, otherwise it is a multi-level edge. For a DAG G, its topological level is the largest value of $topo(v)$ over the vertices in G, i.e.:

$$topo(G) = \max_{v\in\mathcal{V}_G} \; topo(v)$$

Example: Consider the DAG G in Figure 8.1(a). Topological level of vertices, a, b, and c is 1, as the vertices have no incoming edge. The topological level of vertex l is 4, as one of the predecessor node of l is i, which has a topological level value of 3. $Topo(G)$ is equal to 7, because 7 is the largest topological level value for one of the vertices in G. ■

Topological compression of a DAG is performed iteratively, such that the compressed output of one iteration is the input of subsequent iteration. For an input DAG G, one iteration of topological compression removes all odd-topology vertices from G

along with the edges that are incident to the removed vertices. All single-level edges are thus removed, as one of the adjacent vertices of these edges is an odd-topology vertex. A multi-level edge is also removed if at least one of the endpoints of the edge is an odd-topology vertex. As a result of this compression, the topological level of G reduces by half. For the purpose of shortest distance index building, starting from $G = G^0$, we apply this compression process iteratively to generate a sequence of DAGs $G^1, G^2, \cdots, G^t$ such that the topological level number of each successive DAG is half of that of the previous DAG, and the topological level number of the final DAG in this sequence is 1; i.e., $topo(G^{i+1}) = \lfloor topo(G^i)/2 \rfloor$, and $topo(G^t) = 1$, where $t = \lfloor \log_2 topo(G) \rfloor$.

Example: Consider the same DAG $G = G^0$ in Figure 8.1(a). Its topological compression in the first iteration, G^1 is shown in Figure 8.2(a), and in the second iteration, G^2 is shown in Figure 8.2(c). G^2 is the last compression state of G^0, as topological level of G^2 is 1. Note that, in G^1, all odd-topology vertices of G^0, such as, a, b, c, h, i, j, etc. are removed. All single-level edges of G^0, such as, $(e, i), (k, n), (p, s)$, etc. are removed. Multi-level edges, such as, (b, l) and (m, s) are also removed. However, there are newly added vertices in G^1, such as $b', h', r1$, and $s1$, along with newly added edges, such as, (b', l) and $(e, r1)$. More discussion about these additional vertices and edges are given in the following paragraphs. ■

Each iteration of topological compression of a DAG causes loss of information regarding the connectivity among the vertices; for correctly answering distance queries *TopCom* needs to preserve the connectivity information as the input DAG is being compressed. The preservation process gives rise to additional vertices and edges in G^1, which we have seen in the above example. The connectivity preservation process is discussed in detail below.

The most common information loss is caused by the removal of single-level edges. However, such edges are also easily recoverable from the lastly compressed graph in which the edges were present before their removal. So, *TopCom* does not perform

any action for explicit preservation of single-level edges. To preserve the information that is lost due to the removal of multi-level edges, *TopCom* inserts additional even-topology vertices, together with additional edges between the even-topology vertices to prepare the DAG for the compression. The insertion of additional vertices and edges for preserving the information of a removed DAG multi-level edge $e = (u, v)$ is discussed below along with an example given in Figure 8.1. In this figure, the topological levels are mentioned in rectangular boxes. On the left side we show the original graph, and on the right side we show the modified graph which preserves information that is lost due to compression.

There are four possible cases for an edges (u, v) that is being removed due to topological compression.

Case 1: ($topo(u)$ is odd and $topo(v)$ is even). Compression removes the vertex u, so we add a **fictitious** vertex u' such that $topo(u') = topo(u) + 1$. Then we remove the multi-level edge (u, v) and replace it with with two edges (u, u') and (u', v). Since topological level number of both u' and v are even, the topological compression does not delete the edge (u', v). For example, consider the multi-level edge (b, l) in figure 8.1(a), $topo(b) = 1$ (odd), and $topo(l) = 4$ (even). In the modified graph Figure 8.1(b) this edge is replaced by two edges (b, b') and (b', l), where b' is the fictitious node.

Case 2: ($topo(u)$ is even and $topo(v)$ is odd). This case is symmetric to Case 1 as compression removes v instead of u. We use a similar approach like Case 1 to handle this case. We create v_1, a copy of the vertex v such that $topo(v_1) = topo(v) - 1$ and replace the multi-level edge (u, v) with two edges (u, v_1) and (v_1, v). To distinguish the vertices added in these two cases, the newly added vertex is called **fictitious** for Case 1, and it is called **copied** for Case 2. The justification of such naming will be clarified in latter part of the text. Example of Case 2 in Figure 8.1(a) is edge (m, s), where $topo(m) = 4$ (even) and $topo(s) = 7$ (odd). In modified graph, we add copied

node s_1 and replace the original edge with two edges shown in Figure 8.1(b).

Case 3: ($topo(u)$ is odd and $topo(v)$ is odd). In this case we use the combination of above two methods and add two new vertices u' and v_1. We set topological level numbering of new vertices as mentioned above. Also we replace multi level edge (u, v) with three different edges (u, u') , (u', v_1), and (v_1, v). Multi level edge (h, r) in Figure 8.1(a) is an example of this case. As shown in Figure 8.1(b), we add two new vertices h' and r_1 and three new edges, (h, h'), (h', r_1), and (r_1, r) after deleting the original edge (h, r). Note that, if $topo(u) = topo(v) - 2$, $topo(u') = topo(v_1)$. In this case, we treat it as Case 1 by adding only u' (but not v_1) and following the Case 1. It generates a single-level edge (u', v), which we do not need to handle explicitly.

Case 4: ($topo(u)$ is even and $topo(v)$ is even). This is the easiest case as both u and v are not removed by the compression process and we do not make any change in the graph. Also note that the changes in the above three cases convert those cases into this Case 4. For example, applying Case 1 for edge (b, l) in Figure 8.1 creates new multi-edge (b', l) which is an occurrence of Case 4. Similarly Case 2 creates the Case 4 multi-edge (m, s_1).

Dummy edges data structure: We described earlier, we do not need to handle single-level edges separately. However if two continuous single-level edges are removed, we still need to maintain the logical connection between the even-topology vertices. For example, in Figure 8.1(a) edges $(e, i), (i, k)$, and (i, l) are single-level edges which will be deleted after the first compression iteration because $topo(i) = 3$. Now, information of logical (indirect) connection between e to k and l needs to be maintained, because all three vertices will exist after the compression. To handle this, we add new **dummy edges** (e, k) and (e, l); dummy edges are shown as dotted lines in Figure 8.1(b). Note that, for any dummy edge (u, v), $topo(v) - topo(u) = 2$ in the current DAG and the edges for which the node-topology difference is higher

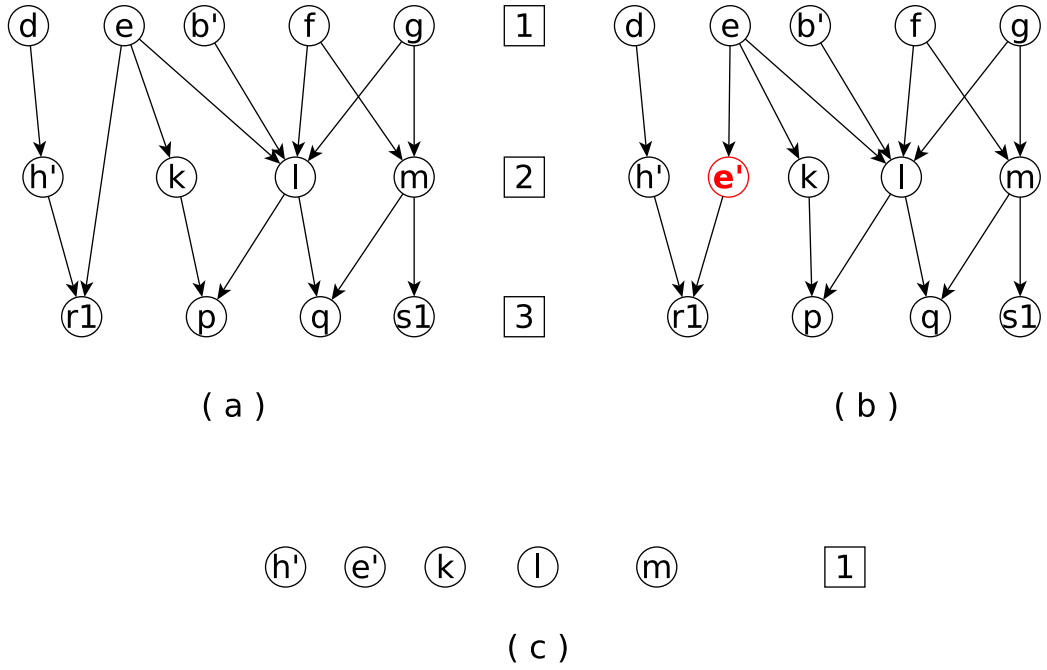


Fig. 8.2.: (a) 1-Compressed Graph G^1, (b) Modified 1-compressed Graph G^1_m, (c) 2-Compressed Graph G^2

than 2 are handled by the above 4 multi-level edge cases. For same start and end nodes, if there are multiple dummy edges, *TopCom* considers edge with the smallest distance. To find the dummy edges, we scan through all odd-topology vertices and find their single-level incoming and outgoing edges. We store all these dummy edges along with the corresponding *distance* value in a list called $DummyEdges$ as shown in Figure 8.1, which we use during the index generation step. For example dummy edge (d, h') has a distance 2 in the Figure 8.1, then $[(d, h'), 2]$ is stored in $DummyEdges$.

At each compression iteration, we first obtain a modified graph, with fictitious vertices, copied vertices, and dummy edges and then apply compression to obtain the compressed graph of the subsequent iteration. The fictitious vertices, copied vertices, and dummy edges of the modified graph in earlier iteration become regular vertices and edges of the compressed graph in subsequent iteration. The above modification and compression proceeds iteratively until we reach t-compressed graph, G^t, for which the topological level number is 1. We use G_m to denote the modified uncompressed

Algorithm 7: Outgoing Index *value* Generation

Input: G^*_m (set of modified graphs), DummyEdges (set of dummy edges and corresponding distance)
Output: I^{out}_* (set of out going indexes for all nodes)

```
1: for all G_m^curr ∈ {G_m^topo(G), ..., G_m^1, G_m} do
2:    O = {u ∈ V_{G_m^curr} | topo(u) = odd_number}
3:    for all v ∈ O do
4:       org_v = GetOriginal(v)
5:       for all (v, w) ∈ E_{G_m^curr} do
6:          org_w = GetOriginal(w)
7:          if org_v == org_w then
8:             Continue
9:          if (v, w) == Dummy_Edge then
10:            distance = GetDummyDistance(DummyEdges, v, w)
11:         if w == fictitious_vertex then
12:            distance = distance - 1
13:         RecursiveInsert(I^out_{org_v}, org_w, distance, out)
```

graph, G^1_m to denote the modified 1-compressed graph, G^2_m to denote the modified 2-compressed graph and so on. For example, Figure 8.2(a) shows G^1 which is obtained by compressing the modified graph G_m in Figure 8.1(b). Figure 8.2(b) shows G^1_m, the modified 1-compressed graph, and Figure 8.2(c) shows G^2, the 2-compressed graph. We refer the set of all modified compressed graphs as G^*_m, i.e. $\{ G^t_m, ..., G^2_m, G^1_m \}$.

8.2.2 Index generation

TopCom's index data structure is represented as a table of *key-value* pairs. For each *key*, (vertex) v of the input graph, the *value* contains two lists: (i) outgoing index value I^{out}_v , which stores shortest distances from v to a set of vertices reachable from v; and (ii) incoming index value I^{in}_v, which stores shortest distances between v and a set of vertices that can reach v. Both the lists contain a collection of tuples, $\langle vertex\_id, distance \rangle$, where *vertex_id* is the id of a vertex other than v, and *distance* is the corresponding shortest path distance between v and that vertex.

At the beginning of the indexing step, for each vertex v, *TopCom* initializes I^{out}_v and I^{in}_v with an empty set. It generates index from G^t_m and repeats the process in

reverse order of graph compression i.e. from graph G_m^t to G_m. In i'th iteration of index building, it uses G_m^{t-i} and inserts a set of tuples in I_v^{out} and I_v^{in}, only if v is an odd-topology vertex in G_m^{t-i}. Thus, during the first iteration, for every odd-topology vertex v of G_m^{t-1}, for an incoming edge (u, v) *TopCom* first checks whether (u, v) is in *DummyEdges* data structure, if so, it inserts $\langle u, d\rangle$ in I_v^{in}, where the distance d value is obtained from the *DummyEdges* data structure. Otherwise, it inserts $\langle u, 1\rangle$ in I_v^{in}. Similarly, for an outgoing edge (v, w) *TopCom* inserts $\langle w, d\rangle$ in I_v^{out}, if (v, w) is in *DummyEdges*, otherwise it inserts $\langle w, 1\rangle$ in I_v^{out}. *TopCom* also inserts (Line 16 in Algorithm 7) elements of I_u^{in} and I_w^{out} into I_v^{in} and I_v^{out}, respectively, using recursive calls.

Algorithm 7 shows the pseudo-code of the index generation procedure for outgoing index *values* only. An identical piece of code can be used for generating incoming index *values* also, but for that we need to exchange the roles of fictitious and copied vertex, and change the I_*^{out} with I_*^{in} in Line 5-21 (more discussion on this is forthcoming).

As shown in Line 2 of Algorithm 7, *TopCom* first collects all odd-topology vertices in variable O and builds out-indexes for each of these vertices using outgoing edges from these vertices (the edge (v, w) in Line 5 of Algorithm 7). Note that, vertices v and w in G_m^{curr} can be fictitious or copied vertex; *TopCom* uses the subroutine GetOriginal() which returns original vertex corresponding to any fictitious or copied vertex, if necessary (Line 4 and 6). Using the data structure *DummyEdges* (discussed in section 8.2.1), it first checks whether the edge (v, w) is a dummy edge (Line 10); if so, it obtains the actual *distance* from the data structure. In case the end-vertex w is a fictitious vertex, *TopCom* decrements the distance value by 1 (Line 14), because for each fictitious vertex, an extra edge with distance 1 is added from the original vertex to the fictitious vertex which has increased the distance value by one. For instance, in the graph in Figure 8.1, the actual distance from a to h is 2, but the fictitious vertex h' records the distance to be 3, which should be corrected. On the other hand, if w is a copied vertex, *TopCom* does not make this subtraction,

Table 8.1.: Intermediate index generated from the DAG in Figure 8.2(b)

key	Out Index *value*	***key***	In Index *value*
b	$\{\langle l, 1\rangle\}$	**p**	$\{\langle k, 2\rangle, \langle l, 2\rangle\}$
d	$\{\langle h, 1\rangle\}$	**q**	$\{\langle m, 1\rangle, \langle l, 2\rangle\}$
e	$\{\langle k, 2\rangle, \langle l, 2\rangle\}$	**r**	$\{\langle h, 1\rangle, \langle e, 1\rangle\}$
f	$\{\langle l, 2\rangle, \langle m, 2\rangle\}$	**s**	$\{\langle m, 1\rangle\}$
g	$\{\langle l, 2\rangle, \langle m, 2\rangle\}$		

because when a copied vertex is used as destination instead of the original vertex, the distance between the source vertex and the copied vertex correctly reflects the actual distance. For an example, in the same graph, the distance between e and r is 1; when we use the copied vertex $r1$ instead of r, distance between e and $r1$ is recorded as 1, which is the correct distance between e and r; so no distance correction is needed during the out index building when the destination vertex is a copied vertex. This is the reason why we make a distinction between the fictitious vertices and the copied vertices.

Finallly note that, after generating indexes for each vertex there may be multiple entries for some vertices; from these multiple entries we need to get the smallest value (entry) and remove others. For building incoming index *values*, *TopCom* subtracts 1 for a copied vertex, but does not subtract 1 for a fictitious vertex, as the roles of start and end vertices flip for the incoming index *values*. Below, we give a complete index building example using the vertex a of the graph in Figure 8.1.

Example: We want to find the outgoing index (*value*) for vertex a (*key*) of the graph G in Figure 8.1(a). $topo(G) = 2$, so we start building index using the graph G^1_m, which is shown in Figure 8.2(b). In the first iteration, *TopCom* builds $I^{out}_d = \{\langle h, 1\rangle\}$; the distance value of 1 comes as follows: *TopCom* uses distance of dummy edge (d, h') that is 2 (Figure 8.1-II) and then it replaces the fictitious vertex h' with h and obtains a distance of 1 by subtracting 1 from 2 (Line 14). It also inserts the following entries under the *key* e; i.e., $I^{out}_e = \{\langle k, 2\rangle, \langle l, 2\rangle\}$. The resulting indexes after this iteration

Table 8.2.: Index for the DAG in Figure 8.1

key	Out Index *value*	In Index *value*
a	$\{\langle d,1\rangle, \langle e,1\rangle, \langle h,2\rangle, \langle k,3\rangle, \langle l,3\rangle\}$	$\emptyset$
b	$\{\langle l,1\rangle, \langle f,1\rangle, \langle m,3\rangle\}$	$\emptyset$
c	$\{\langle f,1\rangle, \langle g,1\rangle, \langle l,3\rangle, \langle m,3\rangle\}$	$\emptyset$
d	$\{\langle h,1\rangle\}$	$\emptyset$
e	$\{\langle k,2\rangle, \langle l,2\rangle\}$	$\emptyset$
f	$\{\langle l,2\rangle, \langle m,2\rangle\}$	$\emptyset$
g	$\{\langle l,2\rangle, \langle m,2\rangle\}$	$\emptyset$
h	$\emptyset$	$\{\langle d,1\rangle\}$
i	$\{\langle k,1\rangle, \langle l,1\rangle\}$	$\{\langle e,1\rangle\}$
j	$\{\langle l,1\rangle, \langle m,1\rangle\}$	$\{\langle f,1\rangle, \langle g,1\rangle\}$
n	$\{\langle p,1\rangle\}$	$\{\langle k,1\rangle\}$
o	$\{\langle p,1\rangle\}$	$\{\langle l,1\rangle\}$
p	$\emptyset$	$\{\langle k,2\rangle, \langle l,2\rangle\}$
q	$\emptyset$	$\{\langle m,1\rangle, \langle l,2\rangle\}$
r	$\emptyset$	$\{\langle h,1\rangle, \langle e,1\rangle, \langle p,1\rangle, \langle k,3\rangle, \langle l,3\rangle\}$
s	$\emptyset$	$\{\langle m,1\rangle, \langle p,1\rangle, \langle k,3\rangle, \langle l,3\rangle, \langle q,1\rangle\}$

is presented in Table 8.1; incoming index *values* for keys b, d, e, f, g are empty (not presented in the table) and similarly outgoing index *values* for keys p, q, r, s are empty. For next iteration considering G_m, *TopCom* inserts $\langle d,1\rangle$ in I_a^{out}; using recursive calls of algorithm 8 (Line 11), this function also inserts $\langle h,2\rangle$ in I_a^{out}, recursion stops at h because I_h^{out} is empty (Line 6). Similarly, $\langle e,1\rangle$ and $\langle k,3\rangle, \langle l,3\rangle$ are inserted in I_a^{out} recursively from I_e^{out}. At the end of the algorithm 7 we remove duplicate entries from indexes. For example, incoming index for key s has two entries for vertex m, $\langle m,1\rangle$ and $\langle m,2\rangle$, one corresponding to edge $(m, s1)$ in G_m^1 and the other is a recursive result from q to s in G_m. *TopCom* considers $\langle m,1\rangle$ and discards the other entry from I_s^{in}. For the graph in Figure 8.1(a), corresponding indexes are presented in Table 8.2.

8.2.3 Index for weighted graph

For weighted graph, *TopCom* makes some minor changes in the above algorithm. First, distance values are stored both in the indexes and in the *DummyEdge* data

Algorithm 8: RECURSIVEINSERT($I_v^{io}, a, distance, in\_or\_out$)

```
1: if in_or_out == in then
2:     I_a^io = I_a^in
3: else
4:     I_a^io = I_a^out
5: if I_a^io == ∅ then
6:     add_tuple(I_v^io, a, distance)
7: else
8:     add_tuple(I_v^io, a, distance)
9:     for all (x, dist) ∈ I_a^io do
10:        RecursiveInsert(I_v^io, x, distance + dist, in_or_out)
```

structure. Many of these distances are implicitly 1 for unweighted graph, which is not true for weighted graph, so, for the latter *TopCom* stores the distance explicitly. Also, it ensures that the distance value between fictitious (or copied) vertices and an original vertex is one, so that the Algorithm 7 works as it is.

8.2.4 Query processing

For query processing, *TopCom* uses the distance indexes that is built during the indexing stage. For a given distance query from u to v, i.e. to compute $\delta(u, v)$, *TopCom* intersects outgoing index *value* of *key* u i.e. I_u^{out} and incoming index *value* of *key* v i.e. I_v^{in} and finds common *vertex_id* in I_u^{out} and I_v^{in}, along with the *distance* values. To cover the cases, when v is in the outgoing index *value* of u, or u is in the incoming index *value* of v, the tuples $\langle u, 0\rangle$ and $\langle v, 0\rangle$ are also added in I_u^{out} and I_v^{in} respectively and then the intersection set of the indexes is found. If the intersection set size is 0, there is no path from u to v and hence the distance is infinity. Otherwise, the distance is simply the sum of the *distances* from u to *vertex_id* and *vertex_id* to v. If multiple paths exist, we take the one that has the smallest distance value.

Example: We want to find $\delta(a, s)$ in Figure 8.1. From table 8.2, $I_a^{out} \cap I_s^{in} = \{k, l\}$. Now, we need to sum up the corresponding *distance* values, that gives $\{\langle k, 6\rangle, \langle l, 6\rangle\}$.

Now we need to find smallest distance value; in this case both the values are same, hence we can provide any one as a result.

8.2.5 Theoretical proofs for correctness

In this section, we prove the correctness of *TopCom*, through the claim that *TopCom*'s index is based on 2-hop covers of the shortest distance in a graph and method described in Section 8.2.4 gives correct shortest distance value. For shortest path, such a cover is a collection S of shortest paths such that for every two vertices u and v, there is a shortest path from u to v that is a concatenation of atmost two paths from S. [181]. That is, shortest path from u to v is stored in S or there is an intermediate node x such that shortest paths from u to x and from x to v are stored in S. For *TopCom*'s index also, the shortest distance from any node u to node v is the 2-hop cover such that the index itself has shortest distance value from u to v or there is an intermediate node x which would be present in both I_u^{out} and I_v^{in}.

Example: In DAG G in Figure 8.1(a) to find distance from a to s, we need to check the outgoing index value for vertex a and the incoming index value for vertex s in Table 8.2. This gives us two possible shortest paths passing through intermediate node k or l, because distance in both cases is same. Thus, there can be multiple shortest paths however, atmost one intermediate node in the index.

In the Theorem 8.2.1, we try to identify the topological layer of an intermediate node x. We identify a unique topological level for each pair of u and v, which tells there is atmost one intermediate node in a shortest path from u to v because in DAG there cannot be a directed edge within topological layer. We begin with the following lemmas, which will be useful for constructing the proof of the theorem.

Lemma 1 *In G_m, if a node u is at topological level 2^i, it will be at topological level 1 in G^i.*

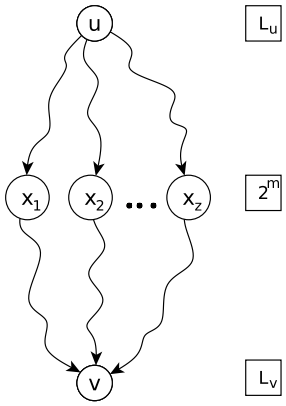


Fig. 8.3.: Shortest path from u to v passing through x

Proof *TopCom* compression method removes all odd-topology nodes and carries over nodes from the even topological levels to the next compression iteration. Thus any node from an even topological level $2x$ in some compressed graph will be at topological level x in the compressed graph of next iteration. Say, the node u is at topological level 2^i in G_m, then it will be at topology level 2^{i-1} in G^1. Since 2^{i-1} is also even, no fictitious or copied vertex will be added for u, and in G^1_m, it will remain at 2^{i-1} level. In the next compression iteration, u will simply be moved to 2^{i-2} level in G^2 and so on. Hence, it will be at level $2^{i-i} = 2^0 = 1$ level in G^i graph. ■

Example In the graph G_m shown in Figure 8.1(b), the node d is at topological level 2 and the node k is at topological level 4. In G^1 shown in Figure 8.2(a) the node d is at topological level 1; similarly, in G^2 shown in Figure 8.2(c), the node k is at topological level 1.

Lemma 2 *In TopCom's index, for all keys, the values contain vertices, which are only from even topological level in the modified DAG G_m.*

Proof As per Line 2 of Algorithm 7, *TopCom*'s index *keys* are nodes from only an odd topological level, and the *values* of index are built using the incident edges of those key nodes. In the modified graph G_m, all the edges from/to an odd topology vertex connects with an even topology vertex, through the use of fictitious/copied

nodes (if needed). Hence, if any node in DAG G_m is at an odd topological level, it cannot be included as an index *value*. Additionally when we compress G_m^i to get G^{i+1}, we only include nodes from even topological levels, hence nodes from odd levels will never be included as a *value* for index building at compressed levels also. ■

Example: See the completely built index of the graph G in Figure 8.1(a) as shown in Table 8.2. The nodes that appear as *values* are $\{d, e, b'(b), f, g, h'(h), k, l, m, r_1(r), p, q, s_1(s)\}$. All of these are from the even topology nodes in G_m as shown in Figure 8.1(b).

Theorem 8.2.1 *For finding shortest distance from u to v, assume that u has topological level number L_u and v has topological level number L_v in G_m. We define*

$$n = \underset{i}{\text{argmax}}(L_u \leq 2^i \leq L_v) \tag{8.1}$$

Now, if there is a shortest path from u to v, for each shortest path, exclusively, one of the following is true.

Case 1: No intermediate node x i.e. I_u^{out} includes v or I_v^{in} includes u.

Case 2: There is an intermediate node x, and

$$topo(x) = 2^n + C$$

for some constant offset C.

Proof We prove this theorem using mathematical induction on n.

Base case: $n = 1$. If there is a direct edge from u to v then *case 1* is true because if $L_u = 2$ then I_v^{in} includes u or if $L_v = 2$ then I_u^{out} includes v. If there is an intermediate node x then $L_u = 1$ and $L_v = 3$, hence $topo(x) = 2$ that shows *case 2* is

true. From Lemma 2 both I_u^{out} and I_v^{in} include the node x if there is a path from u to v. In this case constant offset C would be zero.

Induction hypothesis: Here we assume that for $n = d$ given theorem is true.

Induction step: We want to prove, for $n = d + 1$ given theorem is true.

If there is no intermediate node x then *case 1* is true. Hence, we discuss the only scenario where there is an intermediate node x and we want to show that x is in both I_u^{out} and I_v^{in}. We sub-divide the proof for zero and non-zero values of constant offset C.

Constant offset C is zero:

If there are 2^{d+1} levels, then compression step would be conducted at least one more time than 2^d levels. From Lemma 1 at the d'th step of compression, nodes at topological level 2^d in graph G_m are at 1^{st} topological level in G^d and nodes from topological level 2^{d+1} would be at 2^{nd} topological level.

Hence, *TopCom* will build index for *keys* (nodes) from topological level 2^d in graph G_m, and those index *values* include nodes from topological level 2^{d+1}. As our method recursively includes already built index *values*, the nodes from topological level 2^{d+1} would be recursively included to corresponding outgoing index *values* for *keys* at lower compression levels. Hence, if $topo(x) = 2^{d+1}$ then it is present in outgoing index value of u.

The similar argument works for incoming index of v.

Constant offset C is non-zero:

If we cannot find n that satisfies equation 8.1 then constant offset C is non-zero. In this case offset can be calculated as :

$$\begin{aligned} C &= 2^{n_{low}} \\ where, \qquad n_{low} &= \operatorname*{argmax}_i(2^i < L_u) \end{aligned} \tag{8.2}$$

Now, we define modified topological level number of u is L_u^m, where $L_u^m = L_u - C$ and similarly modified topological level number of v is $L_v^m = L_v - C$. We use L_u^m and L_v^m in equation 8.1 to get n

$$n = \underset{i}{\text{argmax}}(L_u^m \leq 2^i \leq L_v^m)$$

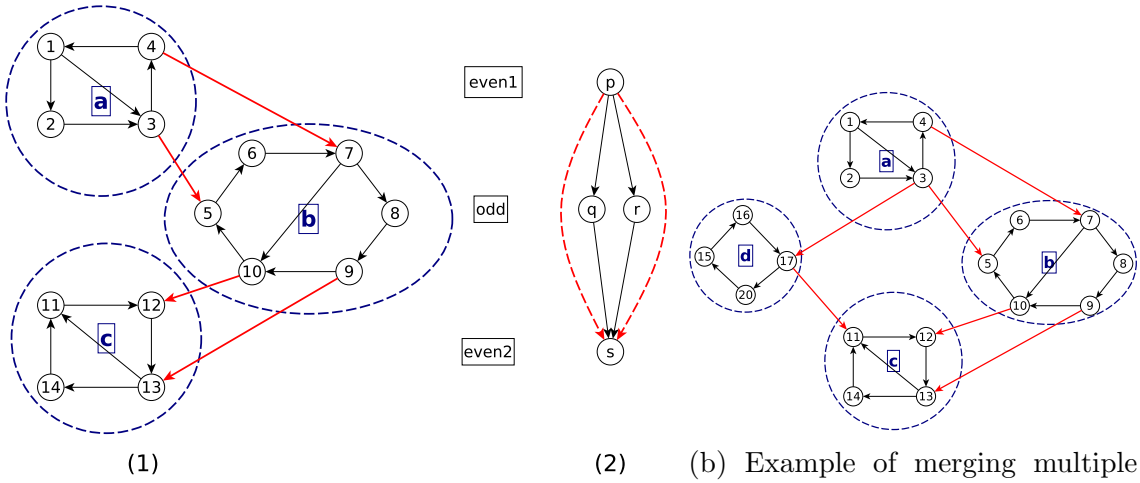
Now, with $topo(x) = 2^n + C$, argument works similarly as zero offset.

■

Example of non-zero offset C: In Figure 8.1(b), we want to know the shortest distance from n to r where corresponding topological levels are $L_n = 5$ and $L_r = 7$ respectively. For this, we can not find any n that satisfies the equation 8.1. From equation 8.2, we can calculate $n_{low} = 2$, using which modified topological levels $L_n^m = 1(5 - 2^2)$ and $L_r^m = 3$ can be obtained. From L_n^m and L_r^m we get $m = 1$. Now, $2^1 + 2^2 = 6$ is the topological level of intermediate node p, which is present in both I_n^{out} and I_r^{in} (Table 8.2). If we look carefully L_n^m and L_r^m is a base case in the mathematical induction proof of Theorem 8.2.1, and $L_n(5)$, $L_r(7)$ with offset C behave exactly the same as the base case.

Note: If there is no node from topological level 2^n in the shortest path from u to v, then there must be one multilevel edge which skips that level. For a node incident to that multilevel edge, at some step of the compression, we need to prepare fictitious/copied node. That new fictitious/copied node works as a node from topological level 2^n and will be included in both I_u^{out} and I_v^{in}. Thus, theorem works fine for this case.

For example, as depicted in Figure 8.1(b), shortest path from a to r doesn't pass through any node from topological level $4(2^2)$ in G_m, but it has a multilevel edge (e, r_1). In G_m^2 (Figure 8.2(b)), this edge causes a fictitious node e' at topological level 2 which is (logically) topological level 4 in G_m. The resulting index in Table 8.2 shows



(1) (2)

(a) (1)Example for Distance within Middle DAG node of G_d and (2)Example of multiple dummy edges in modified G_d

(b) Example of merging multiple dummy DAG edges

Fig. 8.4.: Dummy edge handling

that, the node $e'(e)$ is included as a *value* in the incoming index of r (I_r^{in}) and also included in I_a^{out}.

8.3 Indexing for general directed graph

Any directed graph G can be converted to a Directed Acyclic Graph (DAG) G_d, by considering each strongly connected component (SCC) of G as a node of G_d. Thus in DAG, the edges within a SCC are collapsed within the corresponding node. However, if an edge in G connects two vertices from two distinct SCCs, in G_d those SCCs are connected by a DAG edge. To build the shortest path index for a general directed graph, *TopCom* first uses Tarjan's algorithm [187] to convert G to a DAG G_d by finding all SCCs of G. It also maintains a necessary data structure that keeps the mapping from a DAG node to a set of graph vertices, and vice-versa. We call this a parent-child mapping, i.e., a DAG node is the parent of graph vertices which are part of the corresponding SCC. A DAG edge connects two vertices, one from a distinct SCC. We call such vertices terminal vertices. A single DAG edge between a

pair of SCCs may encapsulate multiple paths (one edge or multiple edges) of Graph G such that the end vertices of these paths are terminal vertices in those pair of SCCs. *TopCom*'s DAG edge data structure contains a set of tuples, each representing one of these paths. A tuple has three elements: node-id of start terminal vertex, node-id of end terminal vertex, and the distance between these two vertices in G. For example consider Figure 8.4a(1), a DAG edge (a, b) stores $\{(3, 5, 1), (4, 7, 1), (3, 7, 2), (4, 5, 3)\}$, first two tuples represent a single-edge path, but the last two represent multi-edge paths. 3, 4 are terminal vertices of DAG node a, and 5,7 are terminal vertices of DAG node b. For each tuple, the third field stores the shortest distance between the pair of terminal vertices in the first two fields of that tuple. To compute distance between an arbitrary pair of vertices within an SCC, *TopCom* also pre-computes all-pair shortest paths among all graph nodes belonging to a single SCC and store them in shortest path index. In real-life directed networks, the size of SCCs are generally not very large, so storing all-pair distances within a SCC in the shortest path index is feasible.

8.3.1 Distance for dummy edges:

For an unweighted DAG two consecutive edges yield a distance value of 2, but for DAG which is a compressed representation of a general unweighted directed graph, two consecutive DAG edges may constitute an arbitrary distance value. This is due to the fact that the shortest path may visit a large number of vertices which are part of the start, middle, and end SCC. For an example, see Figure 8.4a(1); in this figure, the rectangles "even1", "odd" and "even2" are the topological level numbers; a, b and c are the DAG nodes (ellipses), and nodes with numeric ids are nodes of G. Two consecutive DAG edges are (a, b) and (b, c) connecting SCCs a, b and SCCs b, c, respectively. The shortest distance between a and c depends on the terminal nodes of a and c that are being used. If terminal node of a is 3 and terminal node of c is 13, the distance is 5, following the path $3 \rightarrow 4 \rightarrow 7 \rightarrow 10 \rightarrow 12 \rightarrow 13$. In this path, besides the distance 2 over the DAG edges, there are three within-SCC edges,

one in each of SCCs. Thus, the total distance for a dummy edge is the sum of (i) distance from a terminal vertex of staring SCC to a terminal vertex in the middle SCC, (ii) distance between a pair of terminal vertices in the middle SCC, and (iii) distance from a terminal node in the middle SCC to a terminal node in the end SCC. To account for this, *TopCom* computes the dummy edge distance by considering all possible combination of terminal nodes in each SCCs.

Example: Say, *TopCom* wants to find dummy DAG edge a to c which would be set of tuple $\{(3, 12, *), (3, 13, *), (4, 12, *), (4, 13, *)\}$, (all sources to all destinations) where $*$ represents the shortest *distance* values that it needs to find. To find the distance from node 3 to 13, it finds the distance for all combinations of terminal nodes in the middle SCCs and takes the minimum. From starting SCC to middle SCC ($\delta(3, 5) = 1$ and $\delta(3, 7) = 2$), within middle SCC ($\delta(5, 9) = 4$, $\delta(5, 10) = 3$, $\delta(7, 9) = 2$ and $\delta(7, 10) = 1$) and finally from middle SCC to end SCC ($\delta(9, 13) = 1$, $\delta(10, 13) = 2$). In this example, $\delta(3, 7) + \delta(7, 10) + \delta(10, 13)$ gives the minimum value 5 which generates the tuple $(3, 13, 5)$. Distance for all other tuples are also calculated similarly.

Multiple dummy edges: Another issue is, there could be multiple dummy edges having same starting and ending DAG nodes as shown in Figures 8.4a(2). If the original graph itself is a DAG, *TopCom* considers the dummy edge with the lowest *distance*. But for converted DAG G_d applying this solution is more complex, because distance within middle SCC can be different for different SCCs. For this, we need to merge all possible tuples of all dummy edges, and recalculate the distances by taking the minimum distance from the merged set of tuples.

Example in Figure 8.4b we extend the example of Figure 8.4a(1) with one more DAG node d, which also connects node a to node c. Dummy edge through the middle node d is $\{(3, 11, 2), (4, 11, 4), (3, 12, 3), (4, 12, 5), (3, 13, 4), (4, 13, 6)\}$, and through the middle node b is $\{(3, 11, 6), (4, 11, 5), (3, 12, 4), (4, 12, 3), (3, 13, 5), (4, 13, 4)\}$. For dummy

edge (a, c), we combine both the sets of tuples and obtain the smallest distance. Thus the final representation of dummy edge (a, c) is the following: $\{(3, 11, 2), (4, 11, 4), (3, 12, 3), (4, 12, 3), (3, 13, 4), (4, 13, 4)\}$.

8.3.2 Modification in index and query processing

TopCom Index for general graph stores the bidirectional mapping between the DAG nodes and the vertices of the input graph. For every DAG edge (and also for DAG dummy edges), it stores the set of tuple based representation that we have discussed in the above subsection. It also stores all pair distance between each of the vertices within an SCC. Above all, it prepares and stores the 2-hop cover DAG indexes for the DAG representation of the input graph using the methodologies that we discussed in Section 8.2.

For query processing, given a query (u, v), *TopCom* first identifies the corresponding SSE nodes in the DAG using the bidirectional map. Say, these SSEs are s_u and s_v, respectively. If $s_u = s_v$, *TopCom* simply uses the within SSE all-pair index and return the distance between u and v. Otherwise, it first finds the set of out-terminal SSE nodes of s_u (say, X), and in-terminal SSE nodes of s_v (say, Y). Then it uses the 2-hop cover indexing for finding the shortest path distance between each pair of nodes—one from X, and the other from Y. It also considers within SCC distances in three SCCs:

Starting DAG node: Distance from starting node of the query to start terminal node of the DAG node. For example consider figure 8.4a(1), where query is $\delta(1, 14)$. Now all outgoing edges from DAG a are from nodes 3 and 4, hence we need to get distances from 1 to 3 and 4 i.e. $\delta(1, 3) = 1$ and $\delta(1, 4) = 2$.

Middle DAG node: If there is a middle node (from the 2-hop cover index) then distance from incoming edge terminal node to outgoing edge terminal node within middle DAG node needs to be calculated. In our example suppose b is middle node, then we need distance from 5 to 9 and 10, i.e. $\delta(5, 9) = 4$, $\delta(5, 10) = 3$ and similar

for node 7.

Ending DAG node: Distance from end terminal node to ending node of the query within the DAG node.:w That means in our example distance from 12 and 13 to 14 i.e. $\delta(12, 14) = 2$ and $\delta(13, 14) = 1$.

Here again our task is to get minimum distance among all, and we use the similar strategy as section 8.3.1, which is to minimize the summation of above three distances along with edge distances.

As we can see, *TopCom*'s principle indexing process works with DAG and it performs well on real world datasets (Table 8.5). One reason is, real world complex graph becomes less complex when converted as DAG. For example, average degrees of DAG, AD_{DAG} (Table 8.3) are always smaller than AD, mostly an order of magnitude smaller. However because of DAG, we also need to handle a challenging task, i.e. maintaining distance information within SCC for each DAG node. To keep this information, most common ways are to maintain distance matrix or to keep set of edges and calculate distance at run time. Both of these methods have their own pros and cons; keeping distance matrix is the fastest access method but the size of SCC leads to space limitation i.e. we need space for $O(n^2)$ elements and for huge SCC this may be a notable problem. On the other hand keeping set of edges is a memory efficient way, however all distance finding algorithms are polynomial time in terms of $|V|$ and $|E|$; and for huge SCC, finding distance between nodes at run time would be much slower. This represents a well known phenomenon in Computer Science called space-time trade-off. Here for our task, time gets priority over space, hence we selected first method, where we are maintaining distance matrix for each DAG node. For large SCC, this may take high memory, however we observed that our index is still not very large for contemporary machine.

8.3.3 Correctness revisited

In this Section 8.3, we explain how to adopt the proposed indexing method for a general directed graph. For that, first we convert a general directed graph into DAG and then build index on the DAG. We describe the methods to maintain the information at both steps.

We should be able to calculate shortest distance from one node to any other node within the same DAG node. When we convert a general directed graph to DAG, we maintain this information by creating appropriate data structures during conversion (Section 8.3).

The index generation method for DAG is further divided into two steps: 1) Topological Compression and 2) Index Generation. We need to maintain information only during the first step, because the index generation step only builds index from the graphs generated in the topological compression (first) step. The topological compression step is described in Section 8.2.1, where DAG is compressed iteratively by removing all odd-topology vertices and incident vertices. This compression process maintains loss of information using dummy edges, to keep the correct information we explicitly handle distance information for dummy edges as explained in Section 8.3.1.

Lastly, as the structure of a converted DAG is different, we need to handle the queries a little differently. We have explained the modification of query processing in Section 8.3.2. Hence, all the required logical modifications are handled and *TopCom* maintains correctness for general directed graph.

8.4 Experimental evaluation

We compare performance of *TopCom* with two of the recent methods (IS-Label and TreeMap) for answering distance query. We also compare *TopCom* with baseline method Bidirectional Dijkstra's algorithm, which is one of the fastest online methods for single source shortest distance queries. For both IS-Label and TreeMap, codes are provided by their authors. For these experiments we use a machine with Intel 2.4

GHz processor, 8 GB RAM and Ubuntu 14.04 LTS OS. In [26] the author has claimed that TreeMap works for weighted directed graphs, however we are provided with the code of unweighted version for TreeMap, hence all comparisons with TreeMap are for unweighted graphs. Additionally as Y. Xiang mentioned in the paper, TreeMap needs huge memory if tree width is above threshold (1000). The only dataset we are able to run using above machine is WikiVote. Hence for comparison with TreeMap, we used machine with AMD 2.3 GHz processor, 132 GB RAM and Red Hat Enterprise Server Release 6.6 OS. We also perform comparison to IS-Lable using same machine for two datasets (Email_Eu and Epinion). Using synthetic graphs of different sizes and degrees, we show that TreeMap is not scalable for higher degree graphs. To generate these synthetic graphs we use python package networkx (procedure name, FAST_GNP_RANDOM_GRAPH()).

8.4.1 Datasets

Table 8.3.: Real world datasets and basic information

Name	| V |	| E |	AD	MD	$\| V_{DAG} \|$	$\| E_{DAG} \|$	AD_{DAG}	MD_{DAG}	Largest SCC
Email_Eu	265,214	420,045	1.58	7,636	231,000	223,004	0.97	168,815	34,203
Epinion	49,289	487,183	9.88	2,631	16,264	16,497	1.01	15,789	32,417
Gnutella09	8,114	26,013	3.21	102	5,491	6,495	1.18	5,147	2,624
Gnutella31	62,586	147,892	2.36	95	48,438	55,349	1.14	43,928	14,149
WikiVote	7,116	103,689	14.57	1,167	5,817	19,540	3.36	4869	1,300

Here for our experiments, we used seven real world datasets (Table 8.3) from different domains to show wide applicability of *TopCom* . $|V|$ and $|E|$ are the number of vertices and the number of edges respectively. Similarly $|V_{DAG}|$ and $|E_{DAG}|$ are the number of vertices and the edges in the DAG of the corresponding graph. AD and AD_{DAG} are average degree values for the graph and its DAG counterpart, respectively. MD and MD_{DAG} are maximum degrees i.e. maximum in or out degree in the graph and its DAG, respectively. *Largest SCC* is a size of the biggest DAG node which encapsulate the maximum number of input graph nodes.

We collected all datasets from SNAP (Stanford Network Analysis Project) web page[3] except *Epinion* trust network dataset, which we collected from [188]. *Email_Eu* is a snapshot of an email network generated by European Research Institute. *Epinion* dataset is a trust network generated from social network users, it represents which user trusts whom. *Gnutella* is a peer-to-peer file sharing network where *Gnutella*09 is a snapshot of the network on 9th August 2002 and *Gnutella*31 is a snapshot of the same network on 31st August 2002. $WikiVote$ is a network generated from Wikipedia admin voting history data.

Note: All these datasets are unweighted, however we assign random positive weights to each edge of every datasets and use same set of random weights for all the experiments.

8.4.2 Results and Discussion

AS per expectation for DAG *TopCom* outperforms IS-Label method for all datasets depicted in figure 8.5a. Results are average query time over 10 times execution in micro-second (μs), where each method calculated 10K random queries in every execution. For more detailed comparison, if we look at table 8.4, we can see that for datasets *Epinion*, *Gnutella*09 and *Gnutella*31 *TopCom* outperforms IS-Label and TreeMap by an order of magnitude. For other datasets also *TopCom* performs 2-3 times better than both of the competing methods. If we look at the Bi-Dijkstra results, *TopCom* performs multiple orders of magnitude better for all datasets.

In figure 8.5b results for general graphs are plotted, which clearly demonstrate superiority of *TopCom* over IS-Label for general graph. Here also we used Average Query time over 10 times execution with each run calculating 10K random queries. As shown in the Table 8.5, *TopCom* outperforms IS-Label for all the datasets. For *Gnutella*31 dataset *TopCom* performs an order of magnitude better than IS-Label and surprisingly IS-Label performs really poor on *Epinion* dataset such that *TopCom*

[3]http://snap.stanford.edu/data/index.html
[4]Unweighted Graph results

Table 8.4.: Average Query Time for DAG (μs)

Name	**TopCom**	**IS-Label**	**Bi-Djk**	**TreeMap** [4]
Email_Eu	0.1059	0.3865	1657.46	0.2674
Epinion	0.0360	0.2388	14.83	0.1722
Gnutella09	0.0345	0.3292	7.27	0.115
Gnutella31	0.0752	0.2095	50.74	0.254
WikiVote	0.1551	0.3494	43.11	0.2131

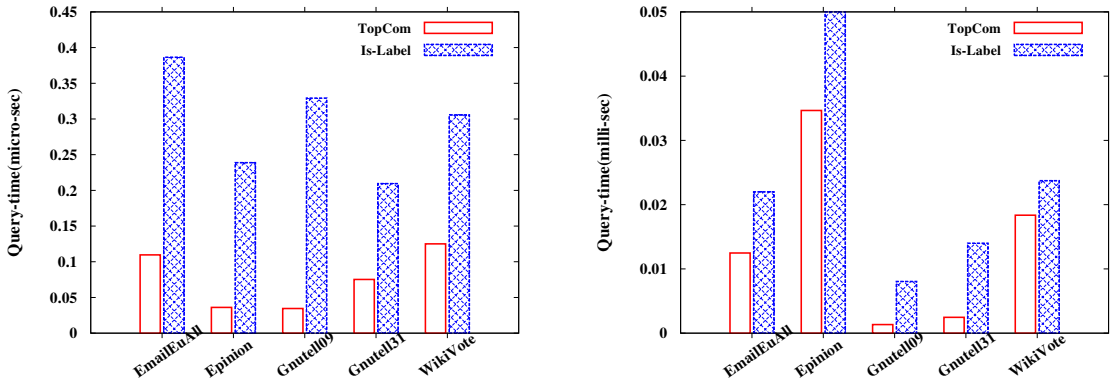
outperforms IS-Label by two orders of magnitude. For *Guntella*09 *TopCom* performs almost 7 times better and for remaining datasets *TopCom* performs almost two times better than IS-Label. Here once again *TopCom* is multiple orders of magnitude faster than Bi-Dijkstra for all datasets.

Table 8.5.: Average Query Time for General Graph (μs)

Name	**TopCom**	**IS-Label**	**Bi-Djk**	**TreeMap** [4]	**TopCom** [5]
Email_Eu	12.4708	21.98527	1482.41	0.8102	10.45136
Epinion	34.6582	2114.02	3570.8667	5.727	33.1907
Gnutella09	1.3405	8.0429	110.6521	1.6255	1.29084
Gnutella31	2.46202	13.9999	299.423	5.804	2.44672
WikiVote	18.3593	23.72954	183.4193	8.371	19.06744

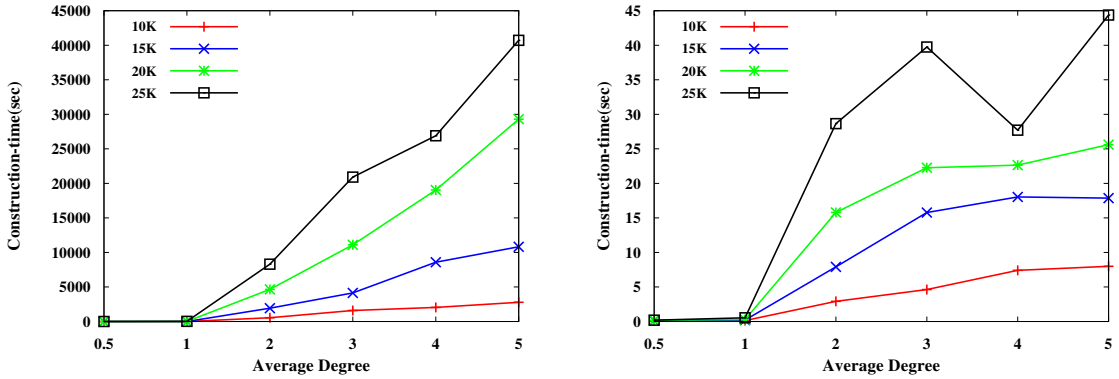
Table 8.5 shows result of TreeMap and *TopCom* comparisons on unweighed graph. It is clear that *TopCom* is competitively better for *Gnutella*09 and *Gnutella*31 datasets, but TreeMap performs an order of magnitude better for other three datasets. However, when we run the TreeMap for building index it took hours to build the index for some datasets, for example average index building time for *Epinion* dataset is more than 9 hours, while *Gnutella*31 takes more than 26 hours. We believe one of the reasons is a bigger graph with higher average degree. To find out the actual cause, we generated synthetic graphs of different sizes (10000-25000) and degrees (0.5-5) and tried to build indexes using TreeMap. In figure 8.6a the index building time is shown

[5]Unweighted Graph: Avg. over 5 times execution for 10K queries



(a) For DAG (μs) (b) For General Graph (ms)

Fig. 8.5.: Average Query time comparison



(a) TreeMap Results on Synthetic Graphs (b) TopCom Results on Synthetic Graphs

Fig. 8.6.: Index Building time for Synthetic graphs

in seconds, after degree 1 all graphs started taking higher time and for bigger graphs the slope of the curve is very large. We compare construction time of *TopCom* for the same set of synthetic graphs shown in figure 8.6b, and as shown *TopCom* hardly takes few seconds for index construction. Highest time taken by *TopCom* is 44 sec for 25K nodes graph with average degree 5, which is almost thousand times faster compared to TreeMap. Hence, it shows that it is difficult for TreeMap to scale for large graphs with higher degree, as many real world graphs generally has higher tree width with larger graphs [189].

8.5 Chapter Summary

In this work we proposed *TopCom* : a unique indexing method to answer distance query for directed real-world graphs. This method uses topological ordering property of DAG and describes a novel method for distance preserving compression of DAG. We compared *TopCom* with IS-Label and found the our method performs better than IS-Label for both weighted DAG and weighted general graph. We strongly believe our method should perform similar or better for unweighed graphs, because we store distance information in label irrespective of weighted/unweighted edges. We do not compare *TopCom* with other recent methods such as HCL [132] and state-of-art 2-Hop [181], because Fu et al. have compared IS-Label with HCL and proved superiority of IS-Label in [25]. We also compare *TopCom* with the recent TreeMap method, which performs better for some datasets, however, we show that this method is not scalable for huge graphs with higher degree. We plan to study further to build index for dynamic large graphs that can answer exact distance query in an acceptable time.

9. CONCLUSION AND FUTURE WORKS

In this thesis, I address a challenging open problem of time prediction in a network. Particularly, I solve two different problems in directed (*RLTP*) and undirected networks (*TCTP*). Additionally, I propose efficient graphlet counting algorithm and novel embedding methods to assist innovative framework to solve *TCTP* problem.

First in Chapter 3, I design and carefully study an interesting problem of reciprocal link time prediction (*RLTP*) in a directed network. To solve it accurately, I study various datasets to understand relation between reciprocal links and different social theories and based on this study, I find suitable topological features. Then I transform the problem into survival analysis framework such that *RLTP* can be efficiently solved using various survival models. Through experiments I show that survival models out-performs traditional regression models in solving the *RLTP* problem.

Next in Chapter 4, I design a triangle completion time prediction (*TCTP*) problem in an undirected network and to solve this *TCTP* accurately I propose a novel framework that uses graphlet and node embedding based features. For this framework, in Chapter 5, I design a highly efficient and parallel algorithm that counts edge-centric local graphlet upto size-5. After that in Chapters 6 and 7, I develop unique attributed network embedding approaches to incorporate these graphlet information in representation vectors. Lastly, through experiments I demonstrate that proposed framework is highly accurate in solving the *TCTP* problem and it out-performs all traditional topological, graphlet and embedding feature based approaches.

The time prediction is highly challenging network analysis problem and this thesis is the first step towards solving this tough problem. Hence, it has wide range of options available to tackle this problem. In Chapter 3, we provided initial social study with topological features, however further study on the replying pattern and corresponding timing patterns can lead to more sophisticated features to solve the *RLTP* problem.

Also, we solve the problem using survival models, but other popular event prediction models such as Point process or agent based reinforcement learning can solve the *RLTP* problem. In Chapter 4, we designed an efficient framework based on supervised embedding methods, a further study on the relation of link creation time and social theories can be explored to find temporal and social features. Similar to the *RLTP* problem this problem can also be solved using more sophisticated regression models. Additionally, both the *RLTP* and *TCTP* are restricted time prediction problem in a network, the more general time prediction problem is to predict link creation time of a random node in a network. For this, first next logical step would be to solve the time prediction for larger units of networks such as 4-size graphlet structures.

REFERENCES

REFERENCES

[1] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: Membership, growth, and evolution," in *ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '06, 2006, pp. 44–54.

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

[3] C. Aggarwal and K. Subbian, "Evolutionary network analysis: A survey," *ACM Comput. Surv.*, vol. 47, no. 1, pp. 10:1–10:36, May 2014.

[4] B. Dumba, G. Golnari, and Z.-L. Zhang, *Analysis of a Reciprocal Network Using Google+: Structural Properties and Evolution.* Springer International Publishing, 2016, pp. 14–26.

[5] K. R. Harrison, M. Ventresca, and B. M. Ombuki-Berman, "A meta-analysis of centrality measures for comparing and generating complex network models," *Journal of computational science*, vol. 17, pp. 205–215, 2016.

[6] M. Golosovsky, "Mechanisms of complex network growth: Synthesis of the preferential attachment and fitness models," *Phys. Rev. E*, vol. 97, p. 062310, Jun 2018.

[7] N. Attar and S. Aliakbary, "Automatic generation of adaptive network models based on similarity to the desired complex network," *arXiv preprint arXiv:1810.01921*, 2018.

[8] M. Bell, S. Perera, M. Piraveenan, M. Bliemer, T. Latty, and C. Reid, "Network growth models: A behavioural basis for attachment proportional to fitness," *Scientific reports*, vol. 7, p. 42431, 2017.

[9] M. A. Hasan and M. J. Zaki, *A Survey of Link Prediction in Social Networks.* Springer US, 2011, pp. 243–275.

[10] D. Liben-Nowell and J. Kleinberg, "The link prediction problem for social networks," in *International Conference on Information and Knowledge Management*, ser. CIKM '03, 2003, pp. 556–559.

[11] V. Martínez, F. Berzal, and J.-C. Cubero, "A survey of link prediction in complex networks," *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, pp. 69:1–69:33, Dec. 2016.

[12] M. Li, H. Zou, S. Guan, X. Gong, K. Li, Z. Di, and C.-H. Lai, "A coevolving model based on preferential triadic closure for social media networks," *Scientific reports*, vol. 3, p. 2512, 2013.

[13] H. Huang, J. Tang, L. Liu, J. Luo, and X. Fu, "Triadic closure pattern analysis and prediction in social networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 12, pp. 3374–3389, 2015.

[14] G. Bianconi, R. K. Darst, J. Iacovacci, and S. Fortunato, "Triadic closure as a basic generating mechanism of communities in complex networks," *Phys. Rev. E*, vol. 90, p. 042806, Oct 2014.

[15] D. J. Watts and S. H. Strogatz, "Collective dynamics of small-world networks," *nature*, vol. 393, no. 6684, p. 440, 1998.

[16] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *science*, vol. 286, no. 5439, pp. 509–512, 1999.

[17] P. Holme and B. J. Kim, "Growing scale-free networks with tunable clustering," *Physical review E*, vol. 65, no. 2, p. 026107, 2002.

[18] R. L. Trivers, "The evolution of reciprocal altruism," *Quarterly Review of Biology*, vol. 46, pp. 33–57, 1971.

[19] S. Leider, M. M. Mobius, T. Rosenblat, and Q.-A. Do, "How much is a friend worth? directed altruism and enforced reciprocity in social networks," *revision of NBER Working Paper 13135, Cambridge, Mass., National Bureau of Economics Research.*, 2007.

[20] T. Antal, P. Krapivsky, and S. Redner, "Social balance on networks: The dynamics of friendship and enmity," *Physica D: Nonlinear Phenomena*, vol. 224, pp. 130 – 136, 2006.

[21] D. Easley, J. Kleinberg *et al.*, "Networks, crowds, and markets: Reasoning about a highly connected world," *Significance*, vol. 9, pp. 43–44, 2012.

[22] T. Lou, J. Tang, J. Hopcroft, Z. Fang, and X. Ding, "Learning to predict reciprocity and triadic closure in social networks," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 7, no. 2, pp. 5:1–5:25, Aug. 2013.

[23] E. Estrada and F. Arrigo, "Predicting triadic closure in networks using communicability distance functions," *SIAM Journal on Applied Mathematics*, vol. 75, no. 4, pp. 1725–1744, 2015.

[24] J. Hopcroft, T. Lou, and J. Tang, "Who will follow you back?: Reciprocal relationship prediction," in *ACM on Conference on Information and Knowledge Management*, ser. CIKM '11, 2011, pp. 1137–1146.

[25] A. W.-C. Fu, H. Wu, J. Cheng, and R. C.-W. Wong, "Is-label: An independent-set based labeling scheme for point-to-point distance querying," *The VLDB Endowment*, vol. 6, pp. 457–468, April 2013.

[26] Y. Xiang, "Answering exact distance queries on real-world graphs with bounded performance guarantees," *The VLDB Journal*, vol. 23, no. 5, pp. 677–695, Oct. 2014.

[27] V. S. Dave, M. Al Hasan, and C. K. Reddy, "How fast will you get a response? predicting interval time for reciprocal link creation." in *AAAI International Conference on Web and Social Media (ICWSM)*, 2017, pp. 508–511.

[28] V. S. Dave, M. A. Hasan, B. Zhang, and C. K. Reddy, "Predicting interval time for reciprocal link creation using survival analysis," *Social Network Analysis and Mining*, vol. 8, no. 1, p. 16, Mar 2018.

[29] J. Leskovec, L. Backstrom, R. Kumar, and A. Tomkins, "Microscopic evolution of social networks," in *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '08, 2008, pp. 462–470.

[30] Y. Dong, J. Tang, S. Wu, J. Tian, N. V. Chawla, J. Rao, and H. Cao, "Link prediction and recommendation across heterogeneous social networks," in *Proceedings of the 2012 IEEE 12th International Conference on Data Mining*, ser. ICDM '12, 2012, pp. 181–190.

[31] A. Grover and J. Leskovec, "Node2vec: Scalable feature learning for networks," in *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16, 2016, pp. 855–864.

[32] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '14, 2014, pp. 701–710.

[33] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding," in *Proceedings of the 24th International Conference on World Wide Web*, ser. WWW '15, 2015, pp. 1067–1077.

[34] M. Rahman, M. A. Bhuiyan, and M. A. Hasan, "Graft: An efficient graphlet counting method for large graph analysis," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 10, pp. 2466–2478, Oct 2014.

[35] W. Hayes, K. Sun, and N. Pržulj, "Graphlet-based measures are suitable for biological network comparison," *Bioinformatics*, vol. 29, no. 4, p. 483, 2013.

[36] N. Pržulj, "Biological network comparison using graphlet degree distribution," *Bioinformatics*, vol. 23, no. 2, p. e177, 2007.

[37] J. Ugander, L. Backstrom, and J. Kleinberg, "Subgraph frequencies: Mapping the empirical and extremal geography of large graph collections," in *Proceedings of the 22Nd International Conference on World Wide Web*, 2013, pp. 1307–1318.

[38] N. K. Ahmed, J. Neville, R. A. Rossi, and N. Duffield, "Efficient graphlet counting for large networks," in *Proceedings of the 2015 IEEE International Conference on Data Mining (ICDM)*, ser. ICDM '15, 2015, pp. 1–10.

[39] D. Marcus and Y. Shavitt, "Rage - a rapid graphlet enumerator for large networks," *Comput. Netw.*, vol. 56, no. 2, pp. 810–819, Feb. 2012.

[40] V. S. Dave, N. K. Ahmed, and M. A. Hasan, "E-clog: Counting edge-centric local graphlets," in *2017 IEEE International Conference on Big Data (Big Data)*, Dec 2017, pp. 586–595.

[41] C. Tu, W. Zhang, Z. Liu, and M. Sun, "Max-margin deepwalk: Discriminative learning of network representation," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2016, pp. 3889–3895.

[42] X. Wang, P. Cui, J. Wang, J. Pei, W. Zhu, and S. Yang, "Community preserving network embedding," in *AAAI Conference on Artificial Intelligence*, 2017.

[43] V. S. Dave, B. Zhang, M. A. Hasan, K. A. Jadda, and M. Korayem, "A combined representation learning approach for better job and skill recommendation," in *ACM on Conference on Information and Knowledge Management*, ser. CIKM '18, 2018.

[44] B. Zhang and M. Al Hasan, "Name disambiguation in anonymized graphs using network embedding," in *ACM on Conference on Information and Knowledge Management*, 2017, pp. 1239–1248.

[45] D. Wang, P. Cui, and W. Zhu, "Structural deep network embedding," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16, 2016, pp. 1225–1234.

[46] K. Okamoto, W. Chen, and X.-Y. Li, "Ranking of closeness centrality for large-scale social networks," in *Frontiers in Algorithmics*, ser. Lecture Notes in Computer Science, 2008, vol. 5059, pp. 186–195.

[47] A. Erdem Sariyuce, K. Kaya, E. Saule, and U. Catalyurek, "Incremental algorithms for network management and analysis based on closeness centrality," *ArXiv e-prints*, march 2013.

[48] D. Kempe, J. Kleinberg, and E. Tardos, "Maximizing the spread of influence through a social network," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '03, 2003, pp. 137–146.

[49] M. Kargar and A. An, "Keyword search in graphs: Finding r-cliques," *The VLDB Endowment*, vol. 4, no. 10, pp. 681–692, Jul. 2011.

[50] A. Ukkonen, C. Castillo, D. Donato, and A. Gionis, "Searching the wikipedia with contextual information," in *ACM on Conference on Information and Knowledge Management*, ser. CIKM '08, 2008, pp. 1351–1352.

[51] M. Fang, J. Yin, X. Zhu, and C. Zhang, "Trgraph: Cross-network transfer learning via common signature subgraphs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 9, pp. 2536–2549, 2015.

[52] C. Yang, Z. Liu, D. Zhao, M. Sun, and E. Y. Chang, "Network representation learning with rich text information," in *Proceedings of the 24th International Conference on Artificial Intelligence*, ser. IJCAI'15, 2015, pp. 2111–2117.

[53] X. Huang, J. Li, and X. Hu, "Accelerated attributed network embedding," in *Proceedings of the SIAM International Conference on Data Mining*, 2017, pp. 633–641.

[54] S. Pan, J. Wu, X. Zhu, C. Zhang, and Y. Wang, "Tri-party deep network representation," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, ser. IJCAI'16, 2016, pp. 1895–1901.

[55] S. Cao, W. Lu, and Q. Xu, "Grarep: Learning graph representations with global structural information," in *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, 2015, pp. 891–900.

[56] A. Bojchevski and S. Gnnemann, "Deep gaussian embedding of graphs: Unsupervised inductive learning via ranking," in *International Conference on Learning Representations (ICLR)*, 2018.

[57] Z. Yang, W. W. Cohen, and R. Salakhutdinov, "Revisiting semi-supervised learning with graph embeddings," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ser. ICML'16, 2016, pp. 40–48.

[58] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems (NIPS) 30*, 2017, pp. 1024–1034.

[59] J. Li, H. Dani, X. Hu, J. Tang, Y. Chang, and H. Liu, "Attributed network embedding for learning in a dynamic environment," in *ACM on Conference on Information and Knowledge Management*, ser. CIKM '17, 2017, pp. 387–396.

[60] B. Vinzamuri and C. K. Reddy, "Cox regression with correlation based regularization for electronic health records," in *IEEE International Conference on Data Mining (ICDM)*, 2013, pp. 757–766.

[61] P. Wang, Y. Li, and C. K. Reddy, "Machine learning for survival analysis: A survey," *ACM Computing Surveys*, 2017.

[62] D. R. Cox, "Regression models and life-tables," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 34, no. 2, pp. 187–220, 1972.

[63] E. L. Kaplan and P. Meier, "Nonparametric estimation from incomplete observations," *Journal of the American Statistical Association*, vol. 53, no. 282, pp. 457–481, 1958.

[64] S. Bhagat, G. Cormode, and S. Muthukrishnan, *Node Classification in Social Networks*. Boston, MA: Springer US, 2011, pp. 115–148.

[65] C. Aggarwal, G. He, and P. Zhao, "Edge classification in networks," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, 2016, pp. 1038–1049.

[66] S. Fortunato and D. Hric, "Community detection in networks: A user guide," *Physics Reports*, vol. 659, pp. 1 – 44, 2016, community detection in networks: A user guide.

[67] M. A. Hasan, V. Chaoji, S. Salem, and M. Zaki, "Link prediction using supervised learning," in *In Proceedings of SDM 06 workshop on Link Analysis, Counterterrorism and Security*, 2006.

[68] S. Bhagat, G. Cormode, and I. Rozenbaum, "Applying link-based classification to label blogs," in *International Workshop on Social Network Mining and Analysis*. Springer, 2007, pp. 97–117.

[69] A. Azran, "The rendezvous algorithm: Multiclass semi-supervised learning with markov random walks," in *Proceedings of the 24th international conference on Machine learning*. ACM, 2007, pp. 49–56.

[70] J. Leskovec, D. Huttenlocher, and J. Kleinberg, "Predicting positive and negative links in online social networks," in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10, 2010, pp. 641–650.

[71] A. K. Menon and C. Elkan, "Link prediction via matrix factorization," in *Machine Learning and Knowledge Discovery in Databases*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 437–452.

[72] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systa survey of link prediction in complex networksems," *Computer*, no. 8, pp. 30–37, 2009.

[73] S. Cao, W. Lu, and Q. Xu, "Deep neural networks for learning graph representations." in *AAAI*, 2016, pp. 1145–1152.

[74] Z. Liaghat, A. H. Rasekh, and A. Mahdavi, "Application of data mining methods for link prediction in social networks," *Social Network Analysis and Mining*, vol. 3, no. 2, pp. 143–150, Jun 2013.

[75] Z. Wang, C. Chen, and W. Li, "Predictive network representation learning for link prediction," in *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '17, 2017, pp. 969–972.

[76] Y. Sun, R. Barber, M. Gupta, C. C. Aggarwal, and J. Han, "Co-author relationship prediction in heterogeneous bibliographic networks," in *International Conference on Advances in Social Networks Analysis and Mining*, July 2011, pp. 121–128.

[77] X. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang, "Knowledge vault: A web-scale approach to probabilistic knowledge fusion," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '14, 2014, pp. 601–610.

[78] B. Zhang, S. Choudhury, M. A. Hasan, X. Ning, K. Agarwal, S. Purohit, and P. G. P. Cabrera, "Trust from the past: Bayesian personalized ranking based link prediction in knowledge graphs," in *SDM Workshop on Mining Networks and Graphs (MNG 2016)*, 2016.

[79] D. Song and D. A. Meyer, "Link sign prediction and ranking in signed directed social networks," *Social Network Analysis and Mining*, vol. 5, no. 1, p. 52, Sep 2015.

[80] P. Symeonidis and N. Mantas, "Spectral clustering for link prediction in social networks with positive and negative links," *Social Network Analysis and Mining*, vol. 3, no. 4, pp. 1433–1447, Dec 2013.

[81] I. Esslimani, A. Brun, and A. Boyer, "Densifying a behavioral recommender system by social networks link prediction methods," *Social Network Analysis and Mining*, vol. 1, no. 3, pp. 159–172, Jul 2011.

[82] N. Z. Gong and W. Xu, "Reciprocal versus parasocial relationships in online social networks," *Social Network Analysis and Mining*, vol. 4, no. 1, pp. 1–14, 2014.

[83] J. Cheng, D. M. Romero, B. Meeder, and J. Kleinberg, "Predicting reciprocity in social networks," in *Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on*, 2011, pp. 49–56.

[84] X. Feng, J. Zhao, Z. Fang, and K. Xu, "Time-aware reciprocity prediction in trust network," in *Advances in Social Networks Analysis and Mining (ASONAM)*, 2014, pp. 234–237.

[85] K. Zhao, X. Wang, M. Yu, and B. Gao, "User recommendations in reciprocal and bipartite social networks–an online dating case study," *IEEE Intelligent Systems*, vol. 29, no. 2, pp. 27–35, Mar 2014.

[86] P. Xia, B. Liu, Y. Sun, and C. Chen, "Reciprocal recommendation system for online dating," in *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, ser. ASONAM '15. ACM, 2015, pp. 234–241.

[87] P. Xia, S. Zhai, B. Liu, Y. Sun, and C. Chen, "Design of reciprocal recommendation systems for online dating," *Social Network Analysis and Mining*, vol. 6, no. 1, p. 32, Jun 2016.

[88] K. Tu, B. Ribeiro, D. Jensen, D. Towsley, B. Liu, H. Jiang, and X. Wang, "Online dating recommendations: Matching markets and learning preferences," in *Proceedings of the 23rd International Conference on World Wide Web*, ser. WWW '14 Companion. ACM, 2014, pp. 787–792.

[89] X. Zang, T. Yamasaki, K. Aizawa, T. Nakamoto, E. Kuwabara, S. Egami, and Y. Fuchida, "You will succeed or not? matching prediction in a marriage consulting service," in *2017 IEEE Third International Conference on Multimedia Big Data (BigMM)*, April 2017, pp. 109–116.

[90] S. Lagraa and H. Seba, "An efficient exact algorithm for triangle listing in large graphs," *Data Mining and Knowledge Discovery*, vol. 30, no. 5, pp. 1350–1369, 2016.

[91] M. K. Rasel, E. Elena, and Y.-K. Lee, "Summarized bit batch-based triangle listing in massive graphs," *Information Sciences*, vol. 441, pp. 1 – 17, 2018.

[92] K. Shin, J. Kim, B. Hooi, and C. Faloutsos, "Think before you discard: Accurate triangle counting in graph streams with deletions," in *Machine Learning and Knowledge Discovery in Databases*. Cham: Springer International Publishing, 2019, pp. 141–157.

[93] M. Al Hasan and V. S. Dave, "Triangle counting in large networks: a review," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2017.

[94] A. Sala, L. Cao, C. Wilson, R. Zablit, H. Zheng, and B. Y. Zhao, "Measurement-calibrated graph models for social network experiments," in *ACM International World Wide Web Conference*, 2010, pp. 861–870.

[95] D. Romero and J. Kleinberg, "The directed closure process in hybrid social-information networks, with an analysis of link formation on twitter," *International AAAI Conference on Web and Social Media*, 2010.

[96] H. Huang, J. Tang, S. Wu, L. Liu, and X. fu, "Mining triadic closure patterns in social networks," in *Proceedings of the 23rd International Conference on World Wide Web*, ser. WWW '14 Companion. ACM, 2014, pp. 499–504.

[97] H. Huang, Y. Dong, J. Tang, H. Yang, N. V. Chawla, and X. Fu, "Will triadic closure strengthen ties in social networks?" *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 12, no. 3, p. 30, 2018.

[98] M. Zignani, S. Gaito, G. P. Rossi, X. Zhao, H. Zheng, and B. Y. Zhao, "Link and triadic closure delay: Temporal metrics for social network dynamics," in *Eighth International AAAI Conference on Weblogs and Social Media*, 2014.

[99] Y. Sun, J. Han, C. C. Aggarwal, and N. V. Chawla, "When will it happen?: Relationship prediction in heterogeneous information networks," in *ACM International Conference on Web Search and Data Mining*, ser. WSDM, 2012, pp. 663–672.

[100] M. Li, Y. Jia, Y. Wang, Z. Zhao, and X. Cheng, "Predicting links and their building time: A path-based approach," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16. AAAI Press, 2016, pp. 4228–4229.

[101] T. Hocevar and J. Demsar, "A combinatorial approach to graphlet counting," *Bioinformatics*, vol. 30, no. 4, p. 559, 2014.

[102] S. Wernicke and F. Rasche, "Fanmod: A tool for fast network motif detection," *Bioinformatics*, vol. 22, no. 9, pp. 1152–1153, May 2006.

[103] M. Jha, C. Seshadhri, and A. Pinar, "Path sampling: A fast and provable method for estimating 4-vertex subgraph counts," in *Proceedings of the 24th International Conference on World Wide Web*, ser. WWW '15, 2015, pp. 495–505.

[104] E. R. Elenberg, K. Shanmugam, M. Borokhovich, and A. G. Dimakis, "Distributed estimation of graph 4-profiles," in *Proceedings of the 25th International Conference on World Wide Web*, ser. WWW '16, 2016, pp. 483–493.

[105] X. Chen, Y. Li, P. Wang, and J. C. S. Lui, "A general framework for estimating graphlet statistics via random walk," *The VLDB Endowment*, vol. 10, no. 3, pp. 253–264, Nov. 2016.

[106] Y. Lim and U. Kang, "Mascot: Memory-efficient and accurate sampling for counting local triangles in graph streams," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '15, 2015, pp. 685–694.

[107] L. De Stefani, A. Epasto, M. Riondato, and E. Upfal, "TriÈst: Counting local and global triangles in fully-dynamic streams with fixed memory size," in *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16, 2016, pp. 825–834.

[108] C. Seshadhri, A. Pinar, and T. G. Kolda, "Triadic measures on graphs: The power of wedge sampling," in *Proceedings of the 2013 SIAM International Conference on Data Mining*. SIAM, 2013, pp. 10–18.

[109] N. K. Ahmed, T. Willke, and R. A. Rossi, "Exact and estimation of local edge-centric graphlet counts," in *Proceedings of the 5th International Workshop on Big Data, Streams and Heterogeneous Source Mining (KDD BigMine)*, 2016, pp. 1–10.

[110] S. Chang, W. Han, J. Tang, G.-J. Qi, C. C. Aggarwal, and T. S. Huang, "Heterogeneous network embedding via deep architectures," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '15, 2015, pp. 119–128.

[111] W. L. Hamilton, R. Ying, and J. Leskovec, "Representation learning on graphs: Methods and applications," *IEEE Data Eng. Bull.*, vol. 40, no. 3, pp. 52–74, 2017.

[112] A. García-Durán and M. Niepert, "Learning graph representations with embedding propagation," in *Advances in neural information processing systems (NIPS)*, 2017, pp. 5125–5136.

[113] J. Tang, M. Qu, and Q. Mei, "Pte: Predictive text embedding through large-scale heterogeneous text networks," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '15, 2015, pp. 1165–1174.

[114] D. Zhang, J. Yin, X. Zhu, and C. Zhang, "User profile preserving social network embedding," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 2017, pp. 3378–3384.

[115] P.-Y. Chen, S. Choudhury, and A. O. Hero, "Multi-centrality graph spectral decompositions and their application to cyber intrusion detection," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2016, pp. 4553–4557.

[116] S. T. Roweis and L. K. Saul, "Nonlinear dimensionality reduction by locally linear embedding," *Science*, vol. 290, no. 5500, pp. 2323–2326, 2000.

[117] U. Zwick, "Exact and approximate distances in graphs a survey," in *Algorithms ESA 2001*, ser. Lecture Notes in Computer Science, F. auf der Heide, Ed. Springer Berlin Heidelberg, 2001, vol. 2161, pp. 33–48.

[118] C. Sommer, "Shortest-path queries in static networks," *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, pp. 45:1–45:31, Mar. 2014.

[119] D. Yan, J. Cheng, W. Ng, and S. Liu, "Finding distance-preserving subgraphs in large road networks," in *Data Engineering, ICDE '13. Proceedings of the International Conference on*, 2013, pp. 625–636.

[120] M. Rice and V. J. Tsotras, "Graph indexing of road networks for shortest path queries with label restrictions," *The VLDB Endowmen*, vol. 4, no. 2, pp. 69–80, Nov. 2010.

[121] Y. Tao, C. Sheng, and J. Pei, "On k-skip shortest paths," in *ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11, 2011, pp. 421–432.

[122] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou, "Shortest path and distance queries on road networks: Towards bridging theory and practice," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13, 2013, pp. 857–868.

[123] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck, "A hub-based labeling algorithm for shortest paths in road networks," in *International Conference on Experimental Algorithms*, ser. SEA'11, 2011, pp. 230–241.

[124] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, "Contraction hierarchies: Faster and simpler hierarchical routing in road networks," in *Proceedings of the 7th International Conference on Experimental Algorithms*, ser. WEA'08. Springer-Verlag, 2008, pp. 319–333.

[125] P. Sanders and D. Schultes, "Highway hierarchies hasten exact shortest path queries," in *Algorithms ESA 2005*, ser. Lecture Notes in Computer Science, G. Brodal and S. Leonardi, Eds. Springer Berlin Heidelberg, 2005, vol. 3669, pp. 568–579.

[126] S. Jung and S. Pramanik, "An efficient path computation model for hierarchically structured topographical road maps," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 5, pp. 1029–1046, Sep. 2002.

[127] K. Tretyakov, A. Armas-Cervantes, L. García-Bañuelos, J. Vilo, and M. Dumas, "Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs," in *ACM on Conference on Information and Knowledge Management*, ser. CIKM '11, 2011, pp. 1785–1794.

[128] A. Gubichev, S. Bedathur, S. Seufert, and G. Weikum, "Fast and accurate estimation of shortest paths in large graphs," in *ACM on Conference on Information and Knowledge Management*, ser. CIKM '10. ACM, 2010, pp. 499–508.

[129] M. Qiao, H. Cheng, L. Chang, and J. Yu, "Approximate shortest distance computing: A query-dependent local landmark scheme," *IEEE Transactions, Knowledge and Data Engineering*, vol. 26, no. 1, pp. 55–68, Jan 2014.

[130] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis, "Fast shortest path distance estimation in large networks," in *ACM on Conference on Information and Knowledge Management*, ser. CIKM '09, 2009, pp. 867–876.

[131] A. D. Zhu, X. Xiao, S. Wang, and W. Lin, "Efficient single-source shortest path and distance queries on large graphs," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '13, 2013, pp. 998–1006.

[132] R. Jin, N. Ruan, Y. Xiang, and V. Lee, "A highway-centric labeling approach for answering distance queries on large sparse graphs," in *ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12, 2012, pp. 445–456.

[133] J. Cheng and J. X. Yu, "On-line exact shortest distance query processing," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, ser. EDBT '09. ACM, 2009, pp. 481–492.

[134] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck, "Hierarchical hub labelings for shortest paths," in *Annual European Conference on Algorithms*, ser. ESA'12, 2012, pp. 24–35.

[135] T. Akiba, Y. Iwata, and Y. Yoshida, "Fast exact shortest-path distance queries on large networks by pruned landmark labeling," in *ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13, 2013, pp. 349–360.

[136] J. Gao, R. Jin, J. Zhou, J. X. Yu, X. Jiang, and T. Wang, "Relational approach for shortest path discovery over large graphs," *The VLDB Endowment*, vol. 5, no. 4, pp. 358–369, Dec. 2011.

[137] F. Wei, "Tedi: Efficient shortest path query answering on graphs," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. ACM, 2010, pp. 99–110.

[138] J. Cheng, Y. Ke, S. Chu, and C. Cheng, "Efficient processing of distance queries in large graphs: A vertex cover approach," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. ACM, 2012, pp. 457–468.

[139] S. Anand, R. Chandramouli, K. P. Subbalakshmi, and M. Venkataraman, "Altruism in social networks: good guys do finish first," *Social Network Analysis and Mining*, vol. 3, no. 2, pp. 167–177, Jun 2013.

[140] J. Valverde-Rebaza and A. de Andrade Lopes, "Exploiting behaviors of communities of twitter users for link prediction," *Social Network Analysis and Mining*, vol. 3, no. 4, pp. 1063–1074, Dec 2013.

[141] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan, "On compressing social networks," in *ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '09, 2009, pp. 219–228.

[142] Y.-X. Zhu, X.-G. Zhang, G.-Q. Sun, M. Tang, T. Zhou, and Z.-K. Zhang, "Influence of reciprocal links in social networks," *PloS one*, vol. 9, no. 7, 2014.

[143] V. Zlatić and H. Štefančić, "Influence of reciprocal edges on degree distribution and degree correlations," *Physical Review E*, vol. 80, no. 1, 2009.

[144] M. R. Kuhnt and O. A. Brust, "Low reciprocity rates in acquaintance networks of young adults: fact or artifact?" *Social Network Analysis and Mining*, vol. 4, no. 1, p. 167, Feb 2014.

[145] H. Akaike, *Information Theory and an Extension of the Maximum Likelihood Principle*. New York, NY: Springer New York, 1998, pp. 199–213.

[146] G. Schwarz, "Estimating the dimension of a model," *The Annals of Statistics*, vol. 6, no. 2, pp. 461–464, 03 1978.

[147] S. Adalı and J. Golbeck, "Predicting personality with social behavior: a comparative study," *Social Network Analysis and Mining*, vol. 4, no. 1, p. 159, Feb 2014.

[148] T. Tuna, E. Akbas, A. Aksoy, M. A. Canbaz, U. Karabiyik, B. Gonen, and R. Aygun, "User characterization for online social networks," *Social Network Analysis and Mining*, vol. 6, no. 1, p. 104, Nov 2016.

[149] M. Shahriari, O. A. Sichani, J. Gharibshah, and M. Jalili, "Sign prediction in social networks based on users reputation and optimism," *Social Network Analysis and Mining*, vol. 6, no. 1, p. 91, Oct 2016.

[150] M. Roshanaei and S. Mishra, "Studying the attributes of users in twitter considering their emotional states," *Social Network Analysis and Mining*, vol. 5, no. 1, p. 34, Jul 2015.

[151] P. Bogdanov, M. Busch, J. Moehlis, A. K. Singh, and B. K. Szymanski, "Modeling individual topic-specific behavior and influence backbone networks in social media," *Social Network Analysis and Mining*, vol. 4, no. 1, p. 204, Jun 2014.

[152] P. Devineni, D. Koutra, M. Faloutsos, and C. Faloutsos, "Facebook wall posts: a model of user behaviors," *Social Network Analysis and Mining*, vol. 7, no. 1, p. 6, Feb 2017.

[153] P. Xia, B. Ribeiro, C. Chen, B. Liu, and D. Towsley, "A study of user behavior on an online dating site," in *2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2013)*, Aug 2013, pp. 243–247.

[154] H. Zou and T. Hastie, "Regularization and variable selection via the elastic net," *Journal of the Royal Statistical Society, Series B*, vol. 67, pp. 301–320, 2005.

[155] Z. Wang and C. Wang, "Buckley-james boosting for survival analysis with high-dimensional biomarker data," *Statistical Applications in Genetics and Molecular Biology*, vol. 9, no. 1, 2010.

[156] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *ACM SIGCOMM computer communication review*, vol. 29, no. 4. ACM, 1999, pp. 251–262.

[157] A. P. C. S. Nurcan Durak, Tamara G Kolda, "A scalable null model for directed graphs matching all degree distributions: In, out, and reciprocal," in *IEEE Workshop on Network Science*, 2013, pp. 23–30.

[158] C. Bunkhumpornpat, K. Sinapiromsaran, and C. Lursinsap, "Mute: Majority under-sampling technique," in *International Conference on Information, Communications and Signal Processing (ICICS)*. IEEE, 2011, pp. 1–4.

[159] Y. Yang and H. Zou, "A cocktail algorithm for solving the elastic net penalized cox regression in high dimensions," *Statatistics and Its Interface*, vol. 6, no. 2, pp. 167–173, 2013.

[160] V. S. Dave and M. A. Hasan, *TopCom: Index for Shortest Distance Query in Directed Graph*. Springer International Publishing, 2015, pp. 471–480.

[161] ——, "Topcom: Index for shortest distance query in directed graph," *CoRR*, vol. abs/1602.01537, 2016. [Online]. Available: http://arxiv.org/abs/1602.01537

[162] M. J. Pencina and R. B. D'Agostino, "Overall-c as a measure of discrimination in survival analysis: model specific population value and confidence interval estimation," *Statistics in Medicine*, vol. 23, no. 13, pp. 2109–2123, 2004.

[163] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[164] Z. Zhang, P. Cui, X. Wang, J. Pei, X. Yao, and W. Zhu, "Arbitrary-order proximity preserved network embedding," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '18, 2018, pp. 2778–2786.

[165] A. Tsitsulin, D. Mottin, P. Karras, and E. Müller, "Verse: Versatile graph embeddings from similarity measures," in *Proceedings of the 2018 World Wide Web Conference*, ser. WWW '18, 2018, pp. 539–548.

[166] L. Zhang, R. Hong, Y. Gao, R. Ji, Q. Dai, and X. Li, "Image categorization by learning a propagated graphlet path," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 27, no. 3, pp. 674–685, March 2016.

[167] H. Kashima, H. Saigo, M. Hattori, and K. Tsuda, "Graph kernels for chemoinformatics," *Chemoinformatics and adv. machine learning perspectives: complex computational methods and collaborative techniques*, p. 1, 2010.

[168] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: A system for distributed graph pattern mining," in *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Oct 2015.

[169] M. Bhuiyan, M. Rahman, M. Rahman, and M. A. Hasan, "GUISE: uniform sampling of graphlets for large graph analysis," in *12th IEEE International Conference on Data Mining, Brussels, Belgium*, 2012, pp. 91–100.

[170] V. Vacic, L. M. Iakoucheva, S. Lonardi, and P. Radivojac, "Graphlet kernels for prediction of functional residues in protein structures," *Journal of Computational Biology*, vol. 17, no. 1, pp. 55–72, 2010.

[171] A. R. Nabhan and K. Shaalan, "Keyword identification using text graphlet patterns," in *International Conference on Applications of Natural Language to Information Systems*. Springer, 2016, pp. 152–161.

[172] L. Hermansson, T. Kerola, F. Johansson, V. Jethava, and D. Dubhashi, "Entity disambiguation in anonymized graphs using graph kernels," in *International Conference on Information & Knowledge Management*, ser. CIKM '13, 2013, pp. 1037–1046.

[173] N. K. Ahmed, T. Willke, and R. A. Rossi, "Estimation of local subgraph counts," in *Proceedings of the IEEE International Conference on Big Data (IEEE Big Data)*, 2016, pp. 1–10.

[174] A. Pinar, C. Seshadhri, and V. Vaidyanathan, "Escape: Efficiently counting all 5-vertex subgraphs," in *Proceedings of the 24th International Conference on World Wide Web*, ser. WWW '17, 2017.

[175] S. Rendle, C. Freudenthaler, Z. Gantner, and L. Schmidt-Thieme, "Bpr: Bayesian personalized ranking from implicit feedback," in *Conference on Uncertainty in Artificial Intelligence*, ser. UAI '09, 2009, pp. 452–461.

[176] F. Javed, Q. Luo, M. McNair, F. Jacob, M. Zhao, and T. S. Kang, "Carotene: A job title classification system for the online recruitment domain," in *2015 IEEE First International Conference on Big Data Computing Service and Applications*, March 2015, pp. 286–293.

[177] W. Zhou, Y. Zhu, F. Javed, M. Rahman, J. Balaji, and M. McNair, "Quantifying skill relevance to job titles," in *2016 IEEE International Conference on Big Data (Big Data)*, 2016, pp. 1532–1541.

[178] X. Huang, J. Li, and X. Hu, "Label informed attributed network embedding," in *ACM International Conference on Web Search and Data Mining*, 2017, pp. 731–739.

[179] A. L. Traud, P. J. Mucha, and M. A. Porter, "Social structure of facebook networks," *Physica A: Statistical Mechanics and its Applications*, vol. 391, no. 16, pp. 4165 – 4180, 2012.

[180] M. J. Zaki and W. M. Jr, *Data Mining and Analysis: Fundamental Concepts and Algorithms*. Cambridge University Press, 2014.

[181] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," in *Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '02, 2002, pp. 937–946.

[182] M. Jiang, A. W.-C. Fu, R. C.-W. Wong, and Y. Xu, "Hop doubling label indexing for point-to-point distance querying on scale-free networks," *The VLDB Endowment*, vol. 7, no. 12, pp. 1203–1214, Aug. 2014.

[183] H. Yildirim, V. Chaoji, and M. Zaki, "Grail: a scalable index for reachability queries in very large graphs," *The VLDB Journal*, vol. 21, no. 4, pp. 509–534, 2012.

[184] A. D. Zhu, W. Lin, S. Wang, and X. Xiao, "Reachability queries on large dynamic graphs: A total order approach," in *ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14, 2014, pp. 1323–1334.

[185] J. Cheng, S. Huang, H. Wu, and A. W.-C. Fu, "Tf-label: A topological-folding labeling scheme for reachability querying in a large graph," in *ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13, 2013, pp. 193–204.

[186] V. Dave and M. Hasan, "Topcom: Index for shortest distance query in directed graph," in *Database and Expert Systems Applications*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2015, vol. 9261, pp. 471–480.

[187] R. Tarjan, "Depth first search and linear graph algorithms," *SIAM Journal on Computing*, 1972.

[188] P. Massa and P. Avesani, "Trust-aware bootstrapping of recommender systems," in *ECAI Workshop on Recommender Systems*, 2006, pp. 29–33.

[189] Y. Gao, "Treewidth of erdsrnyi random graphs, random intersection graphs, and scale-free random graphs," *Discrete Applied Mathematics*, vol. 160, no. 45, pp. 566 – 578, 2012.

VITA

VITA

Vachik S. Dave

Education.

- Bachelor of Engineering (B.E.) Computer Engineering
  Dharmsinh Desai University, India 2005-2009
- Master of Technology (M.Tech.) Computer and Information Science
  National Institute of Technology - Hamirpur (NIT-H), India 2009-2011
  Thesis Title: Neural Networks based Model for Software Effort Estimation.
- Doctor of Philosophy (Ph.D.) Computer and Information Science
  Purdue University, USA 2013 - Present
  Thesis Title: Solving time prediction problems using graphlets and embedding based local features.

Academic employment and internships.

- Graduate Teaching Assistant Jan, 2013 - Present
  Courses: Data Mining, Data Structures, Theory of Computation & Analysis of Algorithms
- R&D Data Science, Intern @ CareerBuilder, Atlanta, GA Summer, 2016
  Project title: company name normalization (SIGKDD-2017).
- Data Science, Intern @ Conversant Media, Chicago, IL Summer, 2017
  Identifier team: To identify set of attributes as an individual person from knowledge graph.

Academic Award: *Best Teaching Assistant* in Computer Science Department, Purdue School of Science.

Publications

Conference papers.

1. Vachik S. Dave, and Mohammad Al Hasan. "Triangle Completion Time Prediction using Time-conserving Embedding" The European conference on Machine Learning and Principles of Knowledge Discovery in Databases (ECML-PKDD), Sept-2019.
2. Vachik S. Dave, Baichuan Zhang, Mohammad Al Hasan, Khalifeh AlJadda and Mohammed Korayem. "A combined representation learning approach for better job and skill recommendation" ACM Conference on Information and Knowledge Management (CIKM), Oct-2018.
3. Vachik S. Dave, Nesreen Ahmed and Mohammad Al Hasan. "E-CLoG: Counting Edge-Centric Local Graphlets" IEEE International Conference on Big Data (IEEE BigData), 2017.
4. Qiaoling Liu, Faizan Javed, Vachik S. Dave and Ankita Joshi. "Supporting Employer Name Normalization at both Entity and Cluster Level" ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), Aug-2017.
5. Vachik S. Dave and Mohammad Al Hasan. "How Fast Will You Get a Response? Predicting Interval Time for Reciprocal Link Creation" AAAI International Conference on Web and Social Media (ICWSM), May-2017.
6. Vachik S. Dave and Mohammad Al Hasan. "TopCom: Index for Shortest Distance Query in Directed Graph" Database and Expert Systems Applications (DEXA), Sept-2015.
7. Vachik S. Dave and Kamlesh Dutta. "Neural Network based Software Effort Estimation Evaluation criterion MMRE" International Conference on Computer Communication Technology (ICCCT), September 2011.

8. Vachik S. Dave and Kamlesh Dutta. "Application of Feed-Forward Neural Network in estimation of Software Effort", International Symposium on Devices MEMS, Intelligent Systems and Communication (ISDMISC), Jan 2011.

Journal papers.

1. Vachik S. Dave, Baichuan Zhang, Pin-Yu Chen and Mohammad Al Hasan. "Neural-Brane: Neural Baysian Personalized Ranking for Attributed Network Embedding" Data Science and Engineering a Springer Open Journal, June-2019.
2. Kamlesh Dutta, Varun Gupta and Vachik S. Dave. "Analysis and comparison of Neural Network models for software development effort estimation" Journal of Cases on Information Technology (JCIT), Volume 21, Issue 2, April-2019.
3. Vachik S. Dave and Mohammad Al Hasan. "Predicting Interval Time for Reciprocal Link Creation" Social Network Analysis and Mining, March-2018.
4. Mohammad Al Hasan and Vachik S. Dave. "Triangle counting in large networks: a review" Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, Oct-2017.
5. Baichuan Zhang, Noman Mohammed, Vachik S. Dave, and Mohammad Al Hasan. "Feature Selection for Classification under Anonymity Constraint" Transactions on Data Privacy (TDP), Volume 10, Issue 1, April-2017.
6. Vachik S. Dave and Kamlesh Dutta. "Neural network based models for software effort estimation: a review" Artificial Intelligence Review, Springer Publication, May-2014.
7. Vachik S. Dave and Kamlesh Dutta. "Comparison of Regression model, Feed-forward Neural Network and Radial Basis Neural Network for Software Development Effort Estimation" ACM SIGSOFT Software Engineering Notes, September-2011.