Otto-von-Guericke-Universität Magdeburg

Faculty of Computer Science

Master's Thesis

Graph Sketches and Embeddings: A Study of their Applications in Graph Databases

Author:

Sindhuja Madabushi

March 19th, 2019

Advisors:

M.Sc. Gabriel Campero Durand

Data and Knowledge Engineering Group

Prof. Dr. rer. nat. habil. Gunter Saake

Data and Knowledge Engineering Group

Madabushi, Sindhuja:
Graph Sketches and Embeddings: A Study of their Applications in Graph Databases
Master's Thesis, Otto-von-Guericke-Universität Magdeburg, 2019.

Contents

List of Figures

List of Tables

Abstract

Large-scale graph processing has evolved to be an important research area, thanks to the proliferation of on-line social networks and growing uses of linked data. Graph-based models are used in many applications, including search engines and recommendation engines, within and outside social networks. An important practical consideration while processing large volumes of data is creating summaries of the data, since it is usually impractical to perform operations on the entire dataset. These summarization procedures attempt to preserve important data features with a significantly reduced memory footprint.

To be purposeful, the data structures used for summarization should be easy to build while also producing good approximations of relevant properties of the data. Although there is existing work that evaluates different graph summarization techniques over some generic graph properties (such as community counts and similarity measures), there is no clear understanding of how these solutions behave when integrated with off-the-shelf graph databases.

In this thesis, we consider two summarization techniques, namely sparsifiers and graph embeddings, examining their effects on data generated using the LDBC social network benchmark, which is loaded onto the Neo4j graph database. We present detailed empirical results that investigate the efficacy of several sparsification and embedding techniques in preserving graph properties of interest, viz., community detection, node centrality, page rank, and cosine similarity.

Our results can be considered as a first step towards guiding the choice of the sparsification/embedding technique to be used in practical scenarios with graph databases, while obtaining informative and fast query results.

Acknowledgements

By submitting this thesis, my long term association with Otto von Guericke University will come to an end.

First and foremost, I am grateful to my advisor M.Sc. Gabriel Campero Durand for his guidance, patience and constant encouragement without which this may not have been possible. I take this as an opportunity to express my deep gratitude especially for all the long discussions, running the entire experiments for embeddings on another machine in my absence from Magdeburg, and providing me with the graph visualizations that helped me to understand the different results of the sparsification techniques. Thank you for the constant support and encouragement.

I would like to thank Prof. Dr. rer. nat. habil. Gunter Saake for giving me the opportunity to write my Master's thesis at his chair.

I would like to thank my friends(family) at Magdeburg who supported me and helped me grow personally and professionally, throughout my masters studies at OvGU.

Most part of research and writing for this thesis has been done during my weekly commutes from Magdeburg to Berlin. I would like to thank the Deutsche bahn authorities for making it a comfortable and memorable experience!

Finally, I would like to thank my mother and brother for constant encouragement to achieve my career goals and helping me be the best version of myself in all facets of life.

Declaration of Academic Integrity

I hereby declare that this thesis is solely my own work and I have cited all external sources used. *Magdeburg, March 19th, 2019*

Sindhuja Madabushi

1. Introduction

It's a small world

"Networks are everywhere" is perhaps a statement that has become seemingly usual, thanks to the rise in the number of real world applications, which use network paradigms to model complex systems. The ubiquity of networks has become so apparent, given the fact that they can be relevantly studied, by modeling real world networks as graphs. This approach is highly applicable. For example, protein structures within each of our cells can be modeled as graphs by casting nodes to proteins and links to interactions between them. It is extremely important that these networks work right for our very existence and the study of these structures could help us cure a lot of diseases. On the other hand, the universe can be modeled as a network too, showing how galaxies are connected to each other by gravitational forces and other influences, a concept called as "cosmic wrap" which astrophysicists are increasingly talking about.[1]
But the study of networks is not just about how they facilitate modeling. With network analysis it is possible to gain insights into real world phenomena. Studies have shown that real world networks are not random[B^+16]. They are often connected following a small world pattern, in which any node can reach any other node in a relatively small number of steps (when compared to the size of the graph). Hence, detecting this type of networks (i.e., with small world patterns) form a backbone for understanding the dynamics of a real world system; capturing, the spread of fashions, cultures, languages and perhaps even wars.[2]
In the 21st century, for all one knows, the most evident example of networks are online social networks. Popular social networks like Facebook, Twitter, LinkedIn, and Instagram are being used by people across the globe to connect with each other. Since these connections are made by a number of people across the globe, the real time

[1]https://www.youtube.com/watch?v=c867FlzxZ9Y
[2]https://www.technologyreview.com/s/534576/how-network-science-is-changing-our-understanding-of-law/

social network data is very large, consisting of millions of nodes and edges, where nodes represents people and edges represent relationships between them. Analyzing this data means analyzing relationships between people, detecting communities, detecting the spread of opinions, facts, and rumors for the good. As this technology advances, efficient tools and techniques are required to store, process and analyze this data, especially since the data is being generated rapidly every second. Some of the technologies used include graph databases, a specialized kind of data management system that helps users to store and query data with a network paradigm.// Apart from specialized storage and query solutions, the magnitude of the generated data calls for so called *summarization techniques*. When dealing with massive data sets, it is much impractical to perform operations on the entire dataset as a whole. So, the main idea is to capture some relevant properties of data in a structure with a reduced memory footprint (when compared to the original data), which contains important information about the data, information that can be derived from just the synopsis at any time. Since this process helps in identification of structure and meaning in data, the data mining community has taken a strong interest in this area of research.
The concept of graph summarization cannot be easily defined[LSDK18]. This is because, summaries are application-dependent and are formed depending on what properties of a graph the user wants to retain within the summary. For example, there are structure preserving summaries and node preserving summaries, summaries that preserve spectral properties[BSST13], and summaries that preserve statistical properties. Therefore, this problem is being studied algorithmically in the fields of graph mining, theoretic computer science, and data management. According to a recent survey by Y Liu et al.[LSDK18], graph summarization has the following benefits and challenges:

Benefits

- *Reduction of data volume and storage.* As mentioned earlier, the real-world graph data is very large. Summaries preserve relevant properties with relatively lower memory footprint compared to the original data.

- *Speedup of graph algorithms and queries.* Summaries provide notable information from the original graph. The summary graph can be more efficiently queried, analyzed and understood with already existing algorithms and tools.

- *Interactive analysis support.* As the size of graph data increases, visualization of this data also becomes a challenge. The resultant graph summaries make it easier to visualize these datasets, avoiding the "hairball" visualization problem.

- *Noise elimination.* Large graph datasets also tend to contain unnecessary links between nodes making the data noisy. Graph summaries retain only the important information avoiding all the noise in the data.

Challenges

- *Complexity of data.* As the graph data is interconnected, partitioning the data and parallelization of operations on data is not as straight-forward as for other cases.

Also, the data could be heterogeneous with information from different sources (text, image) which might require special data structures. These aspects make the building of summaries challenging.

- *Data volume.* The most successful social networking site, Facebook had about two billion monthly users as of June 2017[3]. The goal of summarization techniques is to reduce the size of the graphs to consider. However, the summarization techniques themselves, face the challenge of processing large data. The design of these techniques are often steered by how well they can scale up with the input graph.

- *Definition of interestingness.* The definition of interestingness is subjective, requiring both domain knowledge and user preferences. The cutoff between interestingness and uninterestingness is determined by considering the tradeoffs between time, space and the information preserved in the summaries.

- *Evaluation.* The evaluation of summaries often varies according to perspectives. From a database perspective, a summary can be said to be good if it supports local or global queries with high efficiency. In terms of compressed sensing, a good summary minimizes the number of bits needed to describe the input graph. From a theoretic computer science perspective the evaluation focus would be on how accurately can dense graphs be reduced to sparse graphs with smaller time and space complexities.

1.1 Research aim

By now, it is quite evident that network analysis and graph technologies are becoming increasingly prevalent, and have established their usefulness across various domains. Recently, a team from University of Waterloo conducted an extensive survey[SMS+17] aimed at understanding the types of graphs users have, computations they run on their graphs, software they use, and challenges they face while processing their data.

Their survey was conducted with participants from both industry and academia across the globe. Table 1.1 shows the detailed list of challenges faced by participants. R indicates researchers and P indicates practitioners. Since real world graphs are often large, and consist of diverse range of entities, scalability and visualization are the most pressing challenges reported by participants.

In this thesis, we consider a summarization techniques, which are a solution that could simultaneously help with scalability and visualization. In addition we aim to purposefully evaluate such techniques with an off-the-shelf graph database, to help data management teams to understand better the gains that can be expected from these techniques in real world software stacks.

[3]https://www.telegraph.co.uk/technology/2017/06/27/facebook-now-has-2-billion-users-mark-zuckerberg-announces/

Challenge	Total	R	P
Scalability (i.e., software that can process larger graphs)	45	20	25
Visualization	39	17	22
Query Languages / Programming APIs	39	18	21
Faster graph or machine learning algorithms	35	19	16
Usability (i.e., easier to deploy, configure, and use)	25	10	15
Benchmarks	22	12	10
Extract & Transform	20	6	14
More general purpose graph software (e.g., that can process offline, online, and streaming computations)	20	9	11
Graph Cleaning	17	7	10
Debugging & Testing	10	2	8

Table 1.1: The graph processing challenges selected by the participants from the survey conducted at University of Waterloo [SMS+17]

With this aim we propose to study two types of structure-preserving summarization techniques, namely, graph sparsification and graph embeddings on static graphs of different scale factors. We study the summarization process (incl. alternative approaches for each selected technique), and the performance of queries over the summarized data:

- We compare then the runtime of cypher queries for betweenness centrality, community detection, page rank, partition size and strongly connected components on both sparsified and unsparsified original data.
- We run a query for pairwise cosine similarity of nodes on both embedded and unembedded data.

Overall we propose a novel performance study of chosen summarization techniques on a graph database, thereby aiming to help practitioners understand better solutions to deal with the problem of scalability and visualization.

1.2 Research methodology (CRISP-DM)

CRISP-DM stands for cross industry process for data mining. It is a process model that is being used for data mining projects since the late 1990s. This methodology consists of a series of project steps which allow and might require the user to navigate back and forth. Figure 1.1[4] shows the different phases and the relationships between them, of a typical data mining project. As mentioned earlier, the order of occurrence of the phases in this methodology is not rigid. However, overall, the results of the current phase are expected to determine the occurrence the next phase or particular task of the next phase. Henceforth, we discuss different phases of CRISP-DM approach.

[4]https://www.kdnuggets.com/2017/01/four-problems-crisp-dm-fix.html

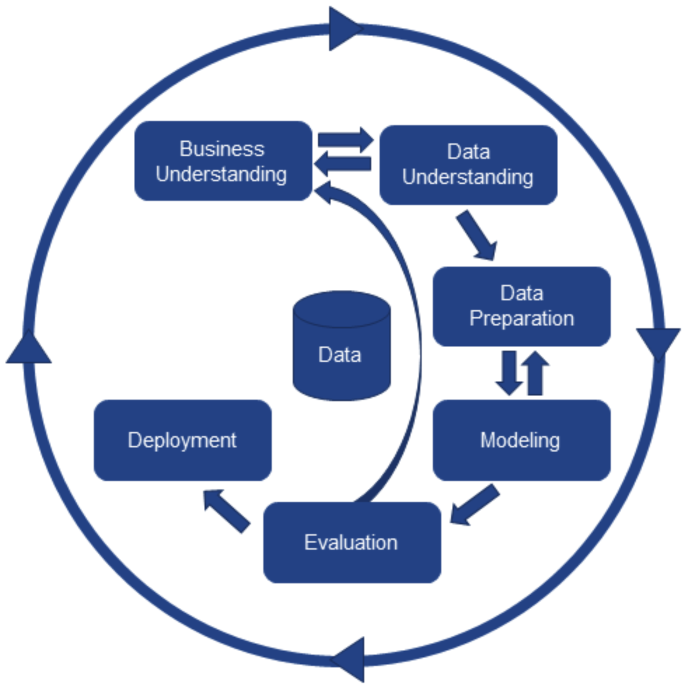


Figure 1.1: Stages of CRISP-DM methodology

Business understanding
This is the initial phase that focuses on understanding of project goals from a business perspective. The knowledge is then converted into a data mining problem definition, and an initial plan to achieve the goals. In our study the results of this phase are collected in our background chapter.

Data understanding
The data understanding phase starts with collection of data and then, carries forward with activities that aid the user to get familiarized with the data and its quality. This phase involves Collecting initial data, describing the data, exploring the data, and verifying the data quality. In our phase this corresponded to understanding an cleaning the datasets selected for our evaluation. The results are partially described in our experimental setup chapter.

Data preparation
In this phase we already have the dataset(s) produced by the previous phase. At this stage, the data should be preprocessed and cleaned. This preprocessed data will be used further for modeling and analysis. Since, we already have the clean data, it is always advisable to have dataset description handy. This phase typically involves selecting data, cleaning data, constructing data, integrating data, and formating data. This phase was carried out for our evaluations, but the results are not described in this thesis.

Modeling
In this phase several modeling techniques are used on the previously processed data. There are several techniques which can be used for a particular data mining problem.

Also, these modeling techniques require tuning of parameters to optimal values and hence, going back to data processing phase might be necessary. The following tasks should be done in this phase: selecting modeling technique, generating test design, building model, and assessing model. This phase comprised the setup of the summarization techniques we selected.

Evaluation

In a data analysis perspective, the model is already largely evaluated. But before deployment, it is important to thoroughly review the steps executed to build it, to be certain that the model achieves the business objectives. A key objective is to determine if there is a key business question that has not been sufficiently addressed. At the end of this phase, a decision on how the data mining results are to be used should be reached. The following are the tasks in this phase: evaluating results, reviewing process, and determining next steps. Previous evaluation steps dealt with factors such as the accuracy and generality of the model. The results for this stage are disclosed in our evaluation chapters.

Deployment

It is known that the purpose of the model is to enhance knowledge of the data. This knowledge needs to be organized and presented in such a way that the customer can exercise it, applying *live* models within an organization's decision making, for instance. Depending on the initial requirements, the deployment phase can constitute of something as simple and basic as generating a summary report or as complex as building a repeatable data mining process across the enterprise. In most cases, the customer carries out the deployment steps. In rare cases if the analyst carries out the deployment steps, it is necessary for the customer to understand what actions should be carried out in order to use the created models. This phase is not considered in this thesis.

1.3 Structure of the thesis

This thesis is organized as follows: In chapter 2, we discuss the required background to undestand our work. We overview concepts form data management for networks, graph databases; and we report a detailed literature review of graph summarization and relevant aspects from representational learning. In chapter 3, we proceed to discuss the prototypical implementation for both our methods. In chapter 4, we talk about the our results, evaluation and discussion for graph sparsification techniques. In chapter 5, we discuss our observations for chosen graph embedding techniques. In chapter 6, we discuss conclusion and future work.

2. Background

In this chapter we present an overview of the theoretical background required for our research. Since our work is based on examining the contributions and effects of sparsification and embedding techniques on graph data evaluated using graph databases, we do not provide a comprehensive review of data management in graph databases or explanation of graph algorithms, though we cover the basics of these topics. In our discussion we focus on examining various sparsification and embedding techniques available and their effects on a chosen test database only. In order to understand the context of our research, we present in detail the main techniques we used in our work, our research aim, and methodology in subsequent sections.
This chapter is organized as follows:

- In Section 2.1, we discuss the organization of literature review. We briefly talk about how we started our research and what papers/books we referred to carry out our research on various topics like network science, graph databases, sparsification techniques, graph embeddings.
- In Section 2.2, we give the required background for our research. This includes a detailed explanation of graph database storage and querying, graph algorithms, and chosen sparsification and embedding techniques.
- In Section 2.3, we discuss the related work.

2.1 Organization of the literature review

In this section we discuss the overview of the literature we used to get a basic understanding of network science, graph summarization techniques, different types of graph sparsification and embedding methods.

2.1.1 Network science

The study of complex networks is an important research field especially during the last two decades due to rapidly increasing number of applications that use network paradigms to model their systems. To understand its relevance better and more in detail, we used the following literature:

- *Barabási, Albert-László. Network science. Cambridge university press, 2016*[B+16]. This text book gave us a clear understanding of basic graph theory, graph data analysis methods, overview of some graph features, some computational and modeling methods and their applications along with some insightful illustrative examples which allowed us to describe graph-based thinking.

2.1.2 Graph databases

Graph databases follow a graph data model, making graph entities (i.e., edges, nodes, traversals) core abstractions in database management systems. Since our work also involves evaluation of queries on summarized data in a chosen graph database, we made attempts to understand the underlying storage and query processing systems in graph databases. We also tried to get a deeper understanding of graph data models and query languages. For this, we used the following sources:

- *Graph databases (I Robinson, J Webber, E Eifrem)*[RWE13]. This book has details about how graph databases are designed and implemented, an explanation of data models, their applications and query languages.

- *Survey of graph database models (R Angles, C Gutierrez)*[AG08], an extensive survey of graph database models starting from evolution of the different data models from different theoretic aspects.

- *Foundations of modern query languages for graph databases(R Angles et al).*[AAB+17] A survey of foundational features of modern graph query languages along with examples of some query languages, their types and semantics.

- *The ubiquity of large graphs and surprising challenges of graph processing (S Sahu et al.)*[SMS+17] An extensive survey with applications and challenges of large scale graph processing addressing known and unknown issues like scalability and visualization with several participants from across the globe both from industry and academia.

2.1.3 Theoretical foundations of graph summarization

Graph summarization is the process of reducing large graph data into a smaller graph, preserving required properties in such a way that one can draw interesting insights about these graph properties from the summary itself, rather than processing the whole data. To understand the theoretical foundations of graph summarization, we searched by the keywords, “graph stream algorithms”, “graph sketches” in Google scholar and got 447000 and 134000 results, respectively. We found that the following papers were relevant for our research:

- *Graph sketches: sparsification, spanners, and subgraphs(KJ Ahn et al.)*[AGM12b] aims at building a sketch-based sparsifier to preserve properties of a graph like min cuts, distances between nodes and prevalence of dense sub-graphs.

- *Graph sketches (J Abello et al.)*[AFK01] brings to attention, some algorithmic and visualization aids behind the estimation of Graph Sketches.
- *Analyzing graph structure via linear measurements (KJ Ahn et al.)*[AGM12a] focuses on building graph sketches into much smaller dimensional spaces with the help of random projections preserving properties like connectivity, k-connectivity, bipartiteness.
- *Database-friendly random projections- Johnson-Lindenstrauss with binary coins (D Achlioptas)*[Ach03] develops a graph embedding using matrix factorization.
- *Densest Subgraph in Dynamic Graph Streams (McGregor et al.)*[MTVV15] focuses on solving a specific problem in a dynamic graph stream model of computation given only limited working memory.
- *Efficient Online Summarization of Large-Scale Dynamic Networks (Q Qu et al.)*[QLZJ16] a framework for on-line summarization of dynamic networks that aims to produce brief, interestingness-driven synopsis that captures the unfolding of information diffusion processes.
- *Graph Sketching and Streaming - New Approaches for Analyzing Massive Graphs (A McGregor)*[McG17] discusses some novel approaches for construction of sketches for massive dynamic graphs.
- *Labeled Graph Sketches (C Song, T Ge)*[SG18] proposes a labeled graph sketch that stores real-time structural information of graph in sub-linear space and supports various types of queries.
- *Sketch Techniques for Approximate Query Processing (G Cormode)*[Cor11] talks about various types of sketch techniques such as frequency based sketches and sketches for distinct value queries for approximate query processing.

The following are the survey papers we used:

- *Graph stream algorithms: a survey (A McGregor)*[McG14] highlights some simple algorithms that illustrate basic ideas of data structures for dynamic graphs, distributed and parallel computation, and approximation algorithms.
- *Graph Summarization Methods and Applications A Survey (Y Liu et al.)* [LSDK18] provides an extensive survey and a taxonomy of graph summarization algorithms based on the input type and the underlying employed techniques.

2.1.4 Sparsification techniques

From a theorectical computer science perspective, the goal of sparsification is to reduce dense graphs into sparse graphs or subgraphs such that their structural and statistical properties are preserved while reducing storage space requirements and some complexity. We used the key words “graph sparsification”, “sparsification on social networks” to search on google scholar and found the following papers to be relevant for our research:

- *Single pass spectral sparsification in dynamic streams (M Kapralov et al.)*[KLM$^+$17] talks about a technique that reduces dense graphs to sparse graphs/subgraphs with low time and space complexity.
- *Single-and multi-level network sparsification by algebraic distance (E John, I Safro)*[JS16] proposes sparsification in two levels based on an algebraic distance metric.
- *Spanners and sparsifiers in dynamic streams (M Kapralov, D Woodruff)*[KW14] addresses the problem of constructing spanners and sparsifiers of a given graph preserving spectral information.
- *Structure-preserving sparsification of social networks (G Lindner et al.)*[LSH$^+$15] provides various novel sparsification techniques specifically designed for social networks.
- *Networkit: An interactive tool suite for high-performance network analysis (C Staudt et al.)*[SSM14] has information about the implementation of a python library, networkit, which is built specifically for sparsification of social networks.

In the computer science literature, there are numerous sparsification techniques. We went through each sparsification technique used in [LSH$^+$15], in detail to evaluate our results. For this, we gave a thorough reading of the theory and techniques based on which these sparsification methods were designed.

- *Algebraic distance on graphs (J Chen, I Safro)[CS11]* introduces a technique for application of a metric called algebraic distance on graphs.
- *Local graph sparsification for scalable clustering (V Satuluri et al.)* [SPR11] introduces similarity metrics on graphs using Jaccard similarity over adjacency matrices
- *Arboricity and subgraph listing algorithms(N Chiba, T Nishizeki)*[CN85] introduces a strategy for edge-searching and using this strategy, obtains algorithms for listing triangles, quadrangles, complete subgraphs and cliques.
- *Triangle listing algorithms: Back from the diversion*[OB14] suggests a unifying framework that says that different triangle listing algorithms are instantiations of generic procedure and also provides additional variants.

2.1.5 Graph embeddings

Graph embeddings refer to representing nodes of the graph as vectors in a n-dimensional vector space. The main aim of these methods is to make the nodes of a graph accessible to machine learning methods, capturing structural and content features of the nodes, while also reducing memory foot print (since some analysis can work on the embedded data rather than on the actual node data). Therefore, in a way, these methods also serve as summarization methods.

Further, work exists where graph embeddings are purposefully created to preserve some graph properties like communities structures[WCW+17].

To learn more about graph embeddings, we started our research by referring to various survey papers to learn about graph embedding methods already proposed in the literature. We made key word searches on Google scholar using the keywords, “structure preserving graph embeddings”, “graph embeddings”, and “network embedding” for which we got about 17,700, 19,400, and 117,000 results respectively. Later we examined various structure-preserving network embedding methods in detail, and used the following papers to learn about them.

- *Representation Learning on Graphs: Methods and Applications (Hamilton et. al)*[HYL17] reviews both individual nodes and subgraph representations on graphs using various algorithms based on matrix factorization, graph neural networks and random-walk based approaches.

- *struc2vec: Learning node representations from structural identity (LFR Ribeiro et. al)*[RSF17] introduces a novel, flexible approach to learn latent representations of graph to preserve structural identity.

- *Laplacian Eigenmaps for Dimensionality Reduction and Data Representation (Mikhail Belkin, Partha Niyogi)*[BN03] proposes a computationally efficient approach for non-linear dimensionality reduction that preserves the locality and could be further used for clustering, by drawing a near analogy between graph Laplacian and the Laplace Beltrami operator.

- *Nonlinear Dimensionality Reduction by Locally Linear Embedding (Sam T. Roweis, Lawrence K. Sau)*[RS00] proposes an embedding technique to embed high-dimensional inputs that preserve both local and global structures.

- *Asymmetric Transitivity Preserving Graph Embedding(Ou et. al)*[OCP+16] introduces a graph embedding technique that preserves higher order proximities.

- *LINE: Large-scale Information Network Embedding (Tang et. al)*[TQW+15] proposes a scalable network embedding technique using an edge-sampling algorithm that optimizes a carefully designed objective function to preserve both global and local structures.

- *DeepWalk: Online Learning of Social Representations (Perozzi et. al)*[PARS14] introduces a parallelizable, scalable, incremental approach for latent node representational learning using random walk and deep learning approaches.

- *Community Preserving Network Embedding (Wang et. al)*[WCW+17] proposes a modularized non negative matrix factorization algorithms for preserving community structures in network embedding.

- *node2vec: Scalable Feature Learning for Networks(Aditya Grover, Jure Leskovec)*[GL16] proposes a method for rich node representational learning based on a biased random walk algorithm to efficiently explore diverse neighborhoods.

- *Structural Deep Network Embedding (Wang et. al)*[WCZ16]introduces a method to preserve both first order and second order proximities in embeddings. The framework consists of two components for supervised and unsupervised learning. The second order proximities are preserved using auto encoders and first order proximities are further preserved by adapting laplacian eigenmaps.

The following are the survey papers we used for our research on graph embeddings:

- *Graph embedding techniques, applications, and performance: A survey (P Goyal, E Ferrara)* [GF18] provides a comprehensive survey of different graph embedding techniques proposed previously in the literature by dividing the techniques into three categories namely *random-walk based, factorization based*, and *deep learning based*. Also introduces a open source library called GEM which provides an implementation of all presented algorithms.

- *Comprehensive Survey of Graph Embedding: Problems, Techniques, and Applications (Cai et. al)* [CZC18] provides a taxonomy of graph embedding methods along with their benefits and challenges, and how they are overcome by existing methods.

- *A Survey on Network Embedding (Cui et. al)* [CWPZ18] discusses existing graph embedding algorithms and their relationship with network embeddings. Also, covers a wide variety of network embedding methods including embedding methods with side information and the advanced information preserving embedding methods.

2.2 Literature review

2.2.1 The business advantages brought forward by network science

Networks are ubiquitous and have applications to numerous activities in our daily lives ranging from television satellite networks to social networks to brain activity networks. It is extremely hard to group the applications of networks into a set of methodological categories. In fact, in an Australian television documentary in 2008, Duncan J Watts described the nature of network applications in the following way:

"Networks are important because if we don't understand networks, we can't understand how markets function, organizations solve problems, or how societies change."

This statement does not only talk about the potential relevance of networks in wide range of domains, but also about how important and necessary they are to solve certain problems. Thanks to the proliferation of application of network paradigms to model complex systems, networks science has evolved to be a major research area especially during the last two decades. The book Network Science by Albert László Barabási[B$^+$16] categorizes the business impact brought forward by network science into two major categories: *Societal* and *Scientific*.

Economic impact: from web search to social networking:

The most successful companies of the 21st century, Google, Twitter, Linkedin, Cisco and Apple have their business model based on networks. It is not only social networking companies like Facebook, Twitter and Linked which envisioned to map the whole world to a social network, but the search technology built by Google is deeply interlinked with the network characteristics of the Web. Algorithms conceived and built by network scientists foster these sites by aiding them with applications that consider network features to provide functionality from friend recommendation to advertising.

Health: from drug design to metabolic engineering

Increasing awareness on *molecular fragment mining*[BBP05] in *molecular networks* has led to the emergence of research fields like *network biology*[BO04] and *network medicine*[BGL11]. Another application of networks in Drug design, *network pharmacology*[Hop07] is worthy of a special mention as it aims to develop drugs for major diseases without any side effects, by using network analysis. Several companies like GeneGo and Genomatica leverage the predictive power of metabolic networks to identify drug patterns in bacteria and humans. Major companies like Johnson & Johnson have also made significant investments in network medicine seeing its potential.

Epidemics: from forecasting to halting deadly viruses

This area of research focuses on exploiting the potential of transport networks to combat the spreading, and forecasting the flow of epidemics. The emergence of a network-based framework offers a new level of predictability and accountability to this domain.

Today, epidemic prediction is used to foresee the spread of viruses like Ebola. Besides, network paradigms have also been used to predict the spread of other kind of viruses, cyber-viruses, in mobile phones[WGHB09].

Neuroscience: mapping the brain

Analysis of brain networks has a wide variety of applications in the field of Neuroscience. Finding out the batches of neurons which are linked together yields information on the presence of diseases like Alzheimer. The *Connectome* project initiated in 2010 by the National Institutes of Health in the United States could provide accurate neuron-level maps of mammalian brains.

Scientific impact:

In the past two decades, network science has emerged to be a major research area, particularly due to the rise of applications following network paradigms for the World Wide Web. Thereafter, many international workshops, conferences, summer and winter schools have started to focus on network science[B+16]. Figure 2.1 shows the rise in the number of citations to works which focus on complex networks in the past two decades.

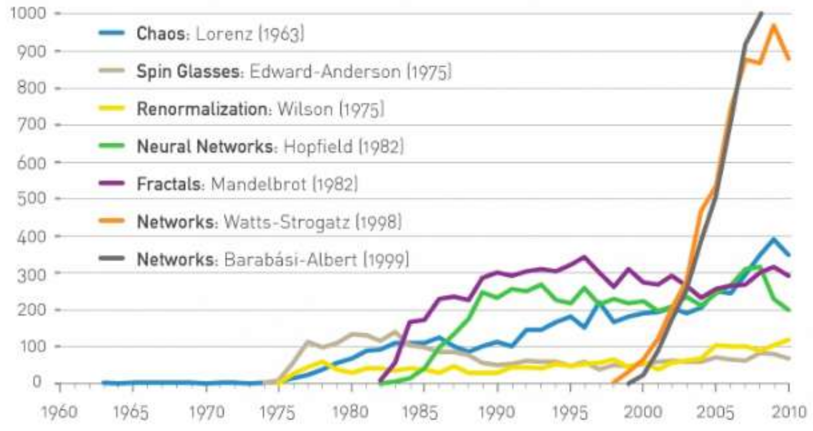


Figure 2.1: Rise in the number of yearly citations to papers related to network science in the past two decades according to Albert László Barabási's book, Network Science [B+16]

2.2.2 Graph processing

In this thesis, we investigate the effects of summarization techniques on a specific graph database and hence, we proceed to introduce some basic concepts about graph databases, encompassing their storage and querying. To this end, we explain details on the specific Neo4j database.

2.2.2.1 Graph database storage

In order to introduce graph databases, it is necessary to begin with the data model adopted. The most common model in graph databases is the *property graph model*. In this model, nodes are connected to each other via named relationships. Both nodes and relationships may have properties. Figure 2.2 shows an example of a property graph model.[1]

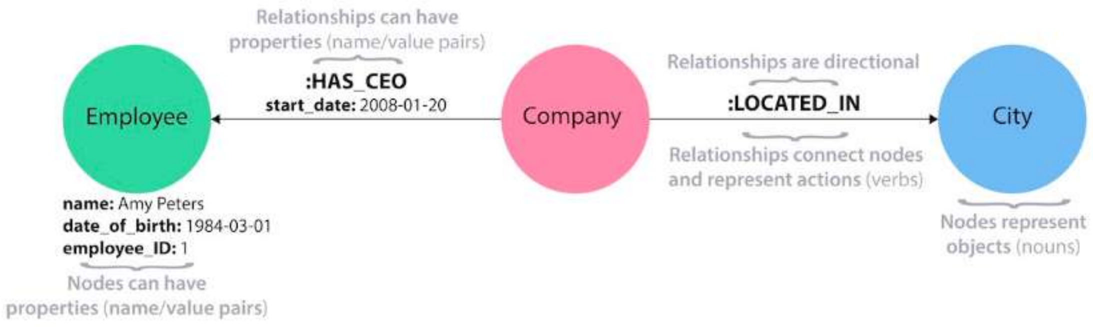

Relationships can have properties (name/value pairs)
:HAS_CEO
start_date: 2008-01-20
Employee
Company
Relationships are directional
:LOCATED_IN
City
Relationships connect nodes and represent actions (verbs)
name: Amy Peters
date_of_birth: 1984-03-01
employee_ID: 1
Nodes can have properties (name/value pairs)
Nodes represent objects (nouns)

Figure 2.2: Example of property graph model

A graph database engine is said to possess *native processing capabilities*, if it exhibits a property known as *index-free adjacency*[IR15]. Index-free adjacency requires for every node in the graph to have a direct reference to its neighboring nodes without the explicit use of indexes. In this way, nodes on the graph act as micro indexes for their neighboring nodes[IR15].
On the other hand, a graph database engine is said to have non-native storage if the indexing is external, that is, it follows another storage approach (e.g. relational or columnar storage).
Native storage capabilities offer the following advantages over non native storage capabilities:

- *storage processing.* Since, index-free adjacency needs every node to maintain a direct reference to all its neighboring nodes, storage processing speed is higher.

- *Query processing.* For the same reason, fast retrievals are also guaranteed without the requirement for indexes.

Evidently, native storage approach is the more suitable for graph data and also has an advantage of low-cost "joins" as joins are pre-computed and stored in the database as relationships[IR15].
In order to describe better one possible mapping between the conceptual native storage model, and the actual physical storage, we describe how a representative commercial graph database (Neo4j) stores its data on disk.

[1]Source: https://neo4j.com/developer/graph-database/

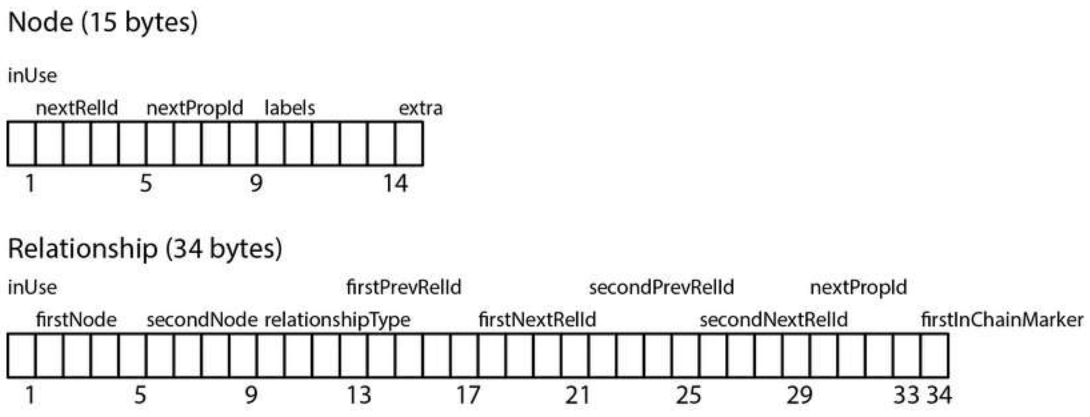


Figure 2.3: Neo4j graph storage on the disk[IR15]

Neo4j stores its graph data in *store files*. Nodes, relationships and properties are stored in three different store files. This separation of storage responsibilities steers fast and high performance traversals, especially due to the division of storage between graph structure and its properties. The node store contains records of fixed size as shown in Figure 2.3. Just like node files, relationship files also have records of fixed length, enabling fast look ups.

The first byte in the node store is a flag that tells the database if the record is being used to store a node or if it is free for a new node. The next four bytes are used for representing ID of the first relationship connected to the node, and the subsequent four bytes are used to represent the ID of the first property of the node. The next five bytes used for labels actually point to the label store of the respective node. The last byte is reserved for flags one of which might be used to identify densely connected components. Relationship files contain a set of records that describe the relationships in the graph. Each relationship record contains IDs of source and target nodes, the pointer to the relationship type stored in *relationship type store*, and pointers corresponding to next and previous relationships. Finally, the *firstInChainMarker* tells if the relationship record is the first in the *relationship chain.*

Through this configuration the Neo4j database is able to match the storage to the basic expected queries in a graph systems. Next, we talk about graph database querying.

2.2.2.2 Graph database querying

Efficient support for querying is the most visible feature of database systems. The types of queries posed to databases implicitly illustrate how complex the supported applications are. Graph database querying is different from traditional database querying in the way that it does not have to deal with explicit joins present in traditional relational databases, as mentioned in the previous section. Schema-less features and complex path-pattern matching can also be expressed in graph query languages.

In terms of the types of queries [KWY12] distinguishes simple queries from complex queries in the following way: Shortest path finding, reachability, pagerank, and graph clustering are some simple queries to graph data. By 'simple', it should be noted that

these queries only require information about structure of the graph. However, they may not be able to capture the rich semantics in complex networks. Graph pattern mining, similarity search, anomaly detection, graph skyline and OLAP, graph aggregation, and keyword search are typical examples of queries that require to consider both the structure and the semantics of the underlying data modeled as a graph.
According to [KWY12], emerging graph queries can be categorized into **(1)** *Mining queries*, **(2)** *Matching queries*, and **(3)** *Selection queries.*

- *Mining queries.* Proximity, frequency, and flexibility are the most important characteristics of proximity patterns that are to be mined by these queries. [KYW10] proposes a method to measure proximities among labels and uses a modified FP-tree algorithm to mine the top-k proximity patterns.

- *Matching queries.* The problem addressed by these queries is typically the subgraph isomorphism[CYD+08, BKS02, TFGER07, Gal06] problem, consisting of finding subgraphs that match the shape of a query subgraph. This problem is NP-complete.

- *Selection queries.* These queries involve identifying top-k nodes close to a specified keyword or a keyword set[BHN+02, LLZW11, BKS02].

Apart from the types of queries, graph query languages can broadly be categorized into two types: *pattern matching based query languages* and *traversal based query languages.*

- *Pattern matching based query languages.* These query languages fundamentally facilitate the use of pattern matching queries in graph databases. Basic graph patterns can be combined with some major relational-like features such as unions, projection and difference to filter out the results and allow only the results which satisfy conditions for the match in the query[AAB+17]. This type of query languages have applications in areas like chemical structure analysis and pattern recognition. *Cypher* and *SPARQL* are two among many popularly used graph pattern query languages. Cypher is a declarative language made for querying property graphs and SPARQL is used for RDF triple stores.

- *Traversal based query languages.* These kind of languages fundamentally support *path queries*, that is, queries which require navigation through topology of the graph. These query languages support queries related to traversal and transitivity closures in directed graphs. All the relevant paths can be returned or resultant paths can be filtered out by imposing restrictions on edge labels. The transitive friend-of-a-friend relation in social networks is one such example where we are interested in paths with edges labeled 'knows' (and not likes or any other label)[AAB+17]. An example of a traversal-based query language is *Gremlin*, which is used in *JanusGraph*, a distributed graph database.

2.2.3 Graph libraries

Not only graph databases can be used for processing graphs, also large scale systems and graph libraries exist. In this section we describe about graph libraries.

Graph libraries are specially built for creation, manipulation, statistical analysis, and study of structure of complex networks. With these libraries in we can load and store networks in different data formats (Eg. JSON). Often, core algorithms associated with these libraries are written in programming languages like C, C++ or FORTRAN and these libraries provide an interface to those existing algorithms; whereas the libraries themselves can also offer interfaces in other languages like Python or Java.
Most libraries, support directed, undirected, weighted and non-weighted, hypergraphs, CSV import and export feature, and implementation of basic graph algorithms such as breadth-first search, depth-first search, Heuristic search (A algorithm), Prim's algorithm for minimum spanning tree, Dijkstra's algorithm for shortest path search, and strongly connected components. Libraries like *Networkx* also allow features such as numerical linear algebra and drawing. Libraries such as Pygraphviz and bokeh are used for visualization of complex networks. SNAP is a general purpose library, written in C++, specifically used for graph mining and is supported by both C++ and Python programming languages.

2.2.4 Graph algorithms

We list in this section some algorithms available for graph querying. In specific, we limit our discussion of algorithms to the official user guide for Neo4j[2]. This guide categorizes graph algorithms into the following categories:

- *Community detection algorithms.* Community detection algorithms are a set of algorithms aimed at identifying communities in a network or a graph. Neo4j facilitates queries in-built for two community detection algorithms, *Louvain algorithm* and *label propagation algorithm.* Louvain uses a modularity-based approach to identify communities. Label propagation, on the other hand, uses labels.

- *Centrality algorithms.* Centrality algorithms focus on finding the most important, or in other words, most influential nodes of a graph. Neo4j offers in-built queries for betweenness centrality, closeness centrality, page rank, and edge centrality. We use betweenness centrality and page rank for our evaluation from this category of graph algorithms.

- *Connected components algorithms.* This class of graph algorithms aims at finding the subgraphs in which each pair of edges are connected by a path and no vertex subgraph is connected to any vertex in the supergraph. Neo4j offers queries for connected components and strongly connected components.

- *Path finding algorithms.* A class of algorithms which focus on finding the shortest path. Neo4j has queires for minimum weight spanning tree algorithm, shortest path algorithm, single source shortest path algorithm, all pairs shortest path algorithm, A* algorithm, Yen's K-shortest paths algorithm, and random walk algorithm in-built for this purpose.

[2]https://neo4j.com/docs/graph-algorithms/current/

In previous sections we have described essentials of graph data systems. In this section we provided a brief listing of graph algorithms, which aims to illustrate the kind of processing that summarization is called upon to improve. In the next section we describe some summarization techniques, with a focus on the techniques that we study in our work.

2.2.5 Summarization techniques for large graph data

There are several approaches to summarizing graph data. These include spanners, sparsifiers and embeddings, among others[LSDK18]. In our work we focus on sparsification and embeddings, which we discuss next.

2.2.5.1 Sparsifiers

Sparsification is a graph summarization technique that aims at lessening the number of edges in a graph using edge sampling techniques, so that the size of the network is reduced while preserving general structural and statistical properties. The edges are sampled depending on the application requirements and connectivity properties. Although there are graph summarization techniques that predominantly preserve specific properties like the diameter[RJH], for instance, in this thesis, we focus on studying and evaluating edge sparsification techniques that preserve the more general structure of the graph.

As mentioned in Section 2.1, the sparsification techniques we use for our study deal with both sparse and dense graphs, and is weakly connected to the theoretic computer science approach in which sparsification refers to reducing dense graphs to sparse graphs with reduced time and space complexity while preserving specific properties like spectral properties[BSST13], for instance.

In addition, real world social networks are already sparse as it is less likely that all the members on a social network are connected to all the other members. However, there would be some typically dense areas which possess large number of nodes and edges which makes it more expensive computationally.

The most important application of these sparsification techniques is *information visualization*. Information visualization in graphs is a quite challenging task. This because of the fact that, even very small real world networks are only seen as “hairballs” when visualized using standard techniques. Although there are numerous techniques for sparsification defined in different contexts, we evaluate the sparsification techniques provided by a Python library called networkit[3][SSM14, LSH+15], due to its ease of use and the fact that it is open source.

Generic framework for sparsification

The core idea of edge sparsification is that, not all edges of a graph are equally important with regards to different properties of a network. This idea is made explicit in some work, as in the case of the work of Lindner et al. [LSH+15], where the importance of edges is quantified by assigning edges of the network with *edge scores*. The idea can

[3]https://github.com/kit-parco/networkit/tree/Dev/networkit

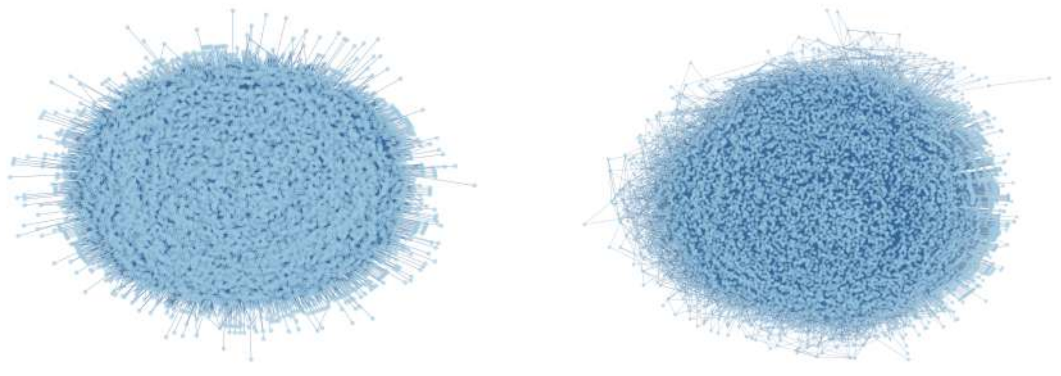


Figure 2.4: Visualization of LDBC SNB SF1 data friendships using the Kamada Kawai layout. Case (a) is formed using the entire graph with 9892 nodes and 180623 edges. Case (b) uses a sparsified version(local similarity sparsification) of the same graph with 9892 nodes and 35022 edges, both, forming “hairballs”.

also be made implicit, being included in the design of a sampling strategy.

The core of most sparsification technique is the sampling strategy it employs. One simple yet fast sampling strategy is assigning a global score based on a metric (similarity between nodes or number of triangles an edge is a part of) and mark as “to-be-removed” all the edges below the specified threshold (e.g. this can mean, for example, to keep them out of memory). This strategy is known as global sparsification. However, this strategy fails to capture local structures. Hence, there is a need for a strategy that avoids relying exclusively on global thresholds, and preserves both local and global structures. In specific cases like algebraic distance sparsification, this general approach can also be used to achieve a balance between both global and local structures preservation.

To achieve this [SPR11] suggests that *edge scores* can be calculated based on some

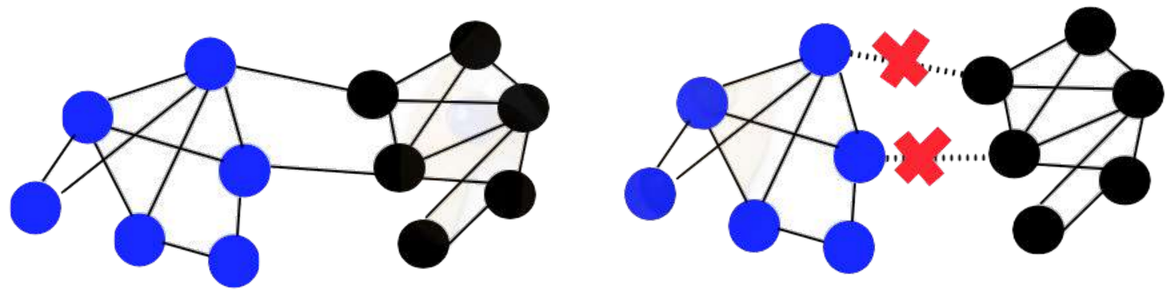


Figure 2.5: An illustration about the importance of edge sampling with awareness of global and local contexts: Keeping intra-cluster edges preserves local structures, removing inter-cluster edges affects the global structures.

metric, and then assigned first to all the edges in the graph. Once this is done, the edge scores are arranged in ascending/descending order. With this strategy, all the edges with edge scores beyond or below the threshold preserve the global structures or local structures.

Next we detail some approaches for sparsification. We focus on approaches that are already available in libraries like NetworkIt.

Random edge sparsification.

This sparsification technique selects edges uniformly at random until a specified sparsification ratio is achieved. It is equivalent to selecting edge scores uniformly at random from a set of given edge scores. Although this approach risks the loss of graph structures, it is beneficial as a baseline for its ease of application. Random edge sparsification has proven to perform quite fast when tested on real time Facebook data networks[LSH$^+$15].

Algebraic distance sparsification

This sparsification method is based on a metric known as *algebraic distance*. The application of this metric on graphs was introduced in [CS11]. Quantifying distances between nodes in graphs had always been an important question in computer science literature. There are numerous ways of quantifying node connectivity (spectral methods, shortest path, probabilistic methods, flow network capacity based approaches, to name a few), in a graph. Algebraic distance is a simple-to-compute metric used to model mostly local connection strengths (though the metric allows for both local and global structure preserving sparsification). To calculate this metric an iterative method is used, where connection weights are propagated to the neighborhood of a vertex, until convergence.

A vector of the weighted adjacency matrix of a graph is updated from random initializations in a given number of iterations. During the execution of the algorithm, these numbers undergo a relaxation process known as Jacobi overrelaxation, forming a mutually influenced environment model on the whole, where the connectivity of vertices is governed by neighborhood vertex connectivity information.

Given below is the formal definition of the algorithm[CS11] to calculate algebraic distance. Let G = (V, E) be a weighted graph, where V is the set of {1, 2, 3, ... , n} vertices and E is the set of edges in the graph. Let W = $[w_{ij}]$ be the weighted adjacency matrix of G, where $[w_{ij}]$ denotes the edge weight between the vertices i and j. If, there is no edge between i and j, $[w_{ij}] = 0$. The following algorithm calculates algebraic distance between two nodes by randomly initializing $x^{(0)}$, where superscript denotes the number of iterations.

In the above algorithm, a vector x is initialized randomly. In the first step, a weighted average of all its neighboring vertices information is taken. This calculated value $\tilde{x}$ along with a relaxation parameter ω and its previous value of x is used to calculate the next value of x again.

Hence, this can described as an iterative process which uses the $(k-1)^{th}$ approximation of $\mathbf{x}$ to calculate the k^{th} value of $\mathbf{x}$ using a weighted average of the previous approximations

Algorithm 1: Computing algebraic distances for graphs

```
Input: Parameter $\omega$, initial vector $x^{(0)}$
1 for $k$ = 1,2,... do
2     $\tilde{x}^{(k)} \leftarrow \sum_j w_{ij} x_j^{(k-1)} / \sum_j w_{ij}\ \forall i$
3     $x^{(k)} \leftarrow (1-\omega) x^{(k-1)} + \omega \tilde{x}^{(k)}$
4 end
```

of neighboring nodes and a relaxation factor ω. The relaxation factor is said to give good convergence for the values when $0<\omega<2$. However, in some systems like Networkit $\omega = 0.5$ and k = 30 are suggested as defaults.

The algebraic distance sparsification in Networkit is done by obtaining the algebraic distances for all pairs of nodes in the graph and filtering out edges based on these distances. For each node, the top d neighboring nodes are sampled based on their algebraic distances. This method can be used for local or global structure preserving or combination of both[JS16].

Local degree sparsification

This sparsification technique, introduced in [LSH$^+$15] is based on the concept of hub nodes. Hub nodes are the nodes having relatively higher degrees in the graph.

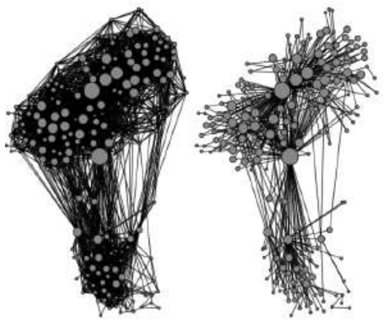


Figure 2.6: Jazz musicians collaboration network and it's Local Degree sparsified version containing 15% of edges[LSH$^+$15].

In this sparsification technique, for each node, degrees of all its neighbouring nodes are calculated. After this, these degrees are arranged in descending order and the top $\lfloor d(u)^\alpha \rfloor$ are kept by each node in the graph while the rest of the nodes are removed (i.e., put out of memory or marked as not necessary for loading). Here, $d(u)$ is the degree of vertex u, and α is a parameter used for local or global edge filtering. This technique prunes the local edges and keeps the global ones.

In order to understand the method better, we give an example.

Example. Let *G = (V, E)*(Figure 2.7) be an weighted undirected graph with node set V = A, B, C, D, E. To illustrate local degree sparsification, we consider calculating degrees of neighboring vertices of node A. Therefore we have, For node A: d(C) = 3, d(B) =

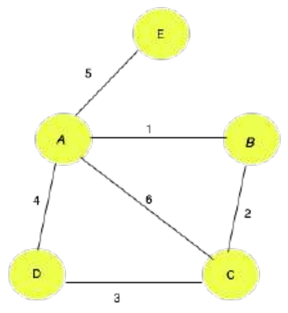


Figure 2.7: Sample weighted undirected graph

2, d(D) = 2, d(E)=1. We keep α=1 for convenience. If we set our threshold to be 3, only the edge to C is preserved because C here is the neighboring node with the highest degree. The same procedure is repeated for all nodes across the graph. Through this sparsification technique, neighboring nodes with highest degrees are preserved.

Triangle count sparsification

This sparsification technique is based on listing triangles in a given graph and assigning edge scores to the edges based on the number of triangles they are present in. This sparsification technique preserves the local structures. In social networks, this filtering is done for counting the number of *triads*. Triads are extremely important by virtue of a fundamental assumption that two persons with a very high number of mutual friends are likely to be friends too. **Chiba and Nishizeki** introduced an algorithm for triangle listing that counts in the triangles by checking the adjacency between two neighboring vertexes[CN85]. A parallelized variant of this algorithm has also been introduced [OB14], being used in NetworkIt.

Algorithm 2: Parallel triangle counting

```
foreach $u \epsilon V$ do
    in parallel
    Mark all $v \epsilon N(u)$
    foreach $v \epsilon N(u)$ do
        foreach $w \epsilon N^+(u)$ do
            if $w$ is marked then
                Count triangle $u, v, w$;
            else

            end
        end
    end
    Un-mark all $v \epsilon N(u)$;
end
```

Chiba and Nishizeki's algorithm starts by sorting the vertices of a given graph in descending order of their degrees. For each vertex, all the vertices adjacent to that vertex are marked. The neighbourhoods of each of the marked vertices are checked and if any vertex in the neighborhood of the current vertex is also checked, then the three vertexes under consideration are declared as a triangle. As mentioned above, the sparsification is proceeded by filtering edges based on the number of triangles they are a part of. This number is assigned as edge score to all the edges of the graph and edges that belong to more triads than an assigned threshold are kept.

Local similarity sparsification

This sparsification technique[SPR11] is based on a Jaccard similarity measure over the set of nodes in a given graph.
Jaccard similarity, in general, is given by:

$$\text{Sim(A, B)} = |\text{A}|\cap|\text{B}|/|\text{A}|\cup|\text{B}|$$

Where A and B are two sets.
In graphs, Jaccard similarity is given by:

$$\text{Similarity(i, j)} = \text{T(i, j)}/(\text{d(i)+d(j)-T(i, j)}),$$

where i and j are two nodes, T(i, j) is the number of triangles the edge joining the nodes i and j is a part of.
Example. Considering the same example we have taken for local degree sparsifcation (Figure 2.7), we have,

Node set N = A, B, C, D, E
Edge set E = 1, 2, 3, 4, 5, 6

The Jaccard similarity between two given nodes A and C is given by:
Similarity(A, C) = 2/4+3-2 = 1/5 = 0.2
Similarities for all pairs of nodes in the graph this similarity is calculated in the same manner. A threshold for similarity is set, beyond which, the edges are kept and the rest are removed (i.e., offloaded from memory). Local similarity sparsification preserves local structures, and for this, the similarities of neighboring nodes are arranged in descending order.

2.2.5.2 Embeddings

Graph representation learning is a technique to represent nodes of a graph as vectors in a n-dimensional vector space, in such a way that structure and other inherent properties of the graph are preserved, and the graph is accessible to common vector/array-based machine learning methods (i.e., where entities are represented as an array of features, for tasks like classification). Figure 2.8 shows an example of a graph embedding of the Karate club network[4]. For this example similarity computations could be done on the

[4]https://iamsiva11.github.io/graph-embeddings-2017-part1/

embedded graph by using simple cosine similarity, instead of using Jaccard Similarity and more hand-crafted similarity on the original graph. Furthermore, the data points in the embedded space might be easy to index and store (they could require less data), since they are composed of uniform data types.

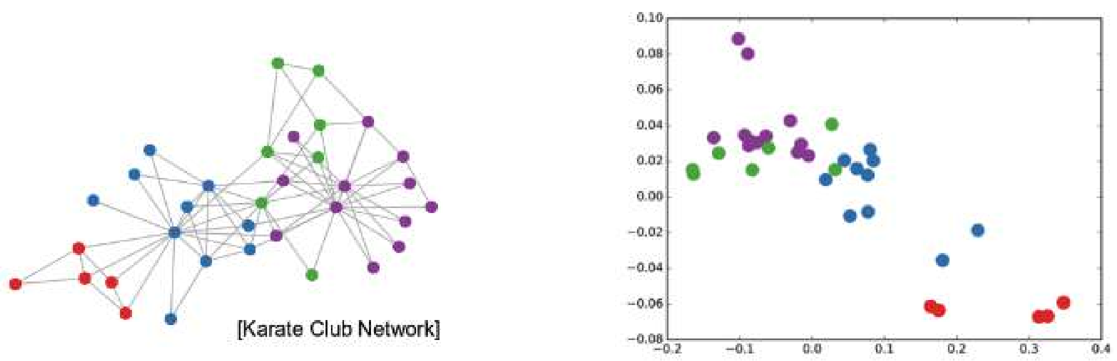


Figure 2.8: Nodes of the graph being represented as vectors in 2-dimensional space

Some applications of this approach include node classification, clustering, link prediction and visualization. More recently, graph embedding approach has also been used for network compression[OCP$^+$16, WCZ16], pattern matching[HPC$^+$18] and pairwise similarity studies[TMKM18].

In order to give a context, graph embeddings can essentially be treated a dimensionality reduction procedure where a similarity graph for a D-dimensional graph is constructed based on the neighborhood of each node. After this process is done nodes are embedded into a d-dimensional space, where d<<D[YXZ$^+$07]. Usually this embedded space is an array of floating point values.

Since graph embedding methods capture the original graph to be embedded in a d-dimensional space (which usually requires less storage per node, in addition to uniform data types) by preserving desired graph properties, they also somehow serve the purpose of graph summarization. Despite its advantages, attaining vector representations for each node in a graph is intrinsically difficult and poses the following challenges:

- *Choice of property.* As some authors[GF18] point out, it is important to know which property of the graph should be preserved after representing the nodes into lower dimensional vector spaces. Given the plethora of properties, such as similarity, page rank, strongly connected components, we need to know what property our embedding should preserve and the performance depends on the application.

- *High non-linearity.* Often, the underlying graph structures are non linear and cannot be accurately approximated by linear manifolds[LNHD11]. Easily stated, to capture detials about the relationships between nodes, complex rather than simple models are required. Therefore, there is a need to design models which capture the non linearity in the underlying structures of the network which is rather difficult.

- *Preserving structure.* Structural properties of graphs often exhibit interesting insights about network data and hence preserving these properties is important. This is particularly challenging because the underlying structure of real world networks is complex[SJ09] and it is important to preserve both local and global properties in order to make accurate analysis.
- *Scalability.* Most real networks are large, therefore embedding techniques need to be scalable to large graphs. Some graph embedding techniques involve deep learning methods to form embeddings as well, since deep learning limits the amount of memory required to create a machine learning model, it can be an approach for scalability. But apart from the storage space of a model, there are a number of ways in which embeddings can be produced (random walks, factorization, and so on). It is important to make sure that our methods are scalable to large graphs[GF18].
- *Sparsity.* Real world networks are often sparse and it is difficult to reach good performance by observing a very limited number of legitimate links[PARS14].
- *Dimensionality of the embedding.* Choosing the optimum dimension can be hard. Higher dimensional representations may increase precision whereas lower dimensional representations could help in link prediction accuracy especially if the chosen model captures local connections[GF18].

[GF18] provides a taxonomy of graph embedding methods and also provides a library called GEM. GEM contains five graph embedding methods namely: locally linear embedding, laplacian eigenmaps, graph factorization, higher order proximity preserving embedding (HOPE), and structure preserving network embedding (SDNE). In this thesis, we use the GEM python library to get embeddings for LDBC social network data and make visualizations of the original data and the embedding. We also examine the training time for each embedding methods for data of two different scale factors.

Laplacian eigenmaps

In the areas of information retrieval, data mining and artificial intelligence, one often faces the problem of low dimensional data residing on a very high dimensional space. Although there is a lot of work in literature dealing with the problem of dimensionality reduction, most approaches do not explicitly consider the structure of the manifold (i.e., a local space that groups nodes next to some of their properties) on which the data is present. Laplacian eigenmaps[BN03] is a geometrically motivated approach for representation of data in lower dimensional spaces in this context.

The crux of the algorithm is a sparse eigenvalue problem, solving this the embedding map is obtained. There are numerous algorithms to solve the eigenvalue problem. However there is no approach till now that is said to solve this problem in one iteration. Power methoda, subspace iterationa, the Jacobi method, and the Arnoldi method are few of

the many methods, which can solve this problem in multiple iterations. We now define the general eigenvalue problem.
Given an $n \times n$ matrix A, the eigenvalue problem requires finding a number λ such that, the following equation is satisfied for some nontrivial vector $\vec{v}$:

$$A\vec{v} = \lambda\vec{v}$$

Here the vector $\vec{v}$ is known as the eigenvector, λ is the corresponding eigenvalue, and A is the transformation matrix. The above expression states that, the matrix vector multiplication on the left hand side essentially yields a scalar times the same vector. For the ease visualization, it can also be said that the vector simply changes its magnitude by expanding or compressing by λ times on its very span without changing it's direction even after a transformation, which is described by the matrix A. The above equation can be rewritten as:

$$(\text{A - }\lambda\text{I})\vec{v} = 0$$

The problem now boils down to looking for values of λ to produce a matrix (A - λI), when multiplied with $\vec{v}$ yields the zero vector. The only possible way for the product of a matrix, when multiplied with a nontrivial vector to yield a zero vector is that the transformation corresponding to that matrix compresses space into a lower dimension, and this process corresponds to a determinant equal to zero to that particular matrix. The matrix (A - λI)$\vec{v}$ has the diagonal elements of a matrix A subtracted with λ. Hence, tweaking of λ values in the diagonal elements of the transformation matrix such that the determinant of the matrix becomes zero results in "squishification" (compression) of space into lower dimension.
Laplacian eigenmap(LAP) is an embedding technique that uses a normalized Laplacian matrix of an input graph and finds its eigenvalues and eigenvectors that can squish the space into a lower dimension. Laplacian eigenmaps aim at keeping the representation of nodes in lower dimensions close to each other when the weight associated with the edge connecting the nodes is higher. The weights are assigned either based on a parameter or are simply assigned the value 1, if two vertices are connected. Optimal embeddings are obtained by optimizing the following objective function:

$$\Phi(\text{y}) = \sum(y_i - y_j)^2 W_{ij}$$

where y_i and y_j are elements of a map $\mathbf{y} = (y_1, y_2, ..., y_n)$ when a connected graph is mapped to a line such that connected points are close to each other. A matrix may possess multiple eigenvectors and eigenvalues. This process is said to yield embeddings that can preserve local structures and first order proximities, with a time complexity, O(|E|d^2), where E is the number of edges and d is a number such that d<<|V|, V being the number of vertices in a given input graph[GF18].

Higher order proximity preserving network embedding

Higher order proximity preserving network embedding(HOPE)[OCP+16] is an embedding technique that focuses on capturing asymmetric transitivities and preserving higher order proximities in directed graphs. There are many studies dealing with embedding directed graphs. However, these methods are not explicitly guaranteed to preserve the asymmetric properties present in real world finite graphs (i.e., large and infinite graphs tend to have symmetries). The asymmetric properties informally speaking indicate local uniqueness in the liking behavior of nodes. Asymmetric transitivities refer to those asymmetric properties that are also visible when looking at the neighborhood of a node. Transitivity properties are used in graph analysis tasks such as calculating similarities between nodes and measuring importance of nodes.
HOPE learns two embedding vectors for each node, to capture the asymmetric transitivity. These vectors are known as source and target vectors. A directed edge from any v_i to v_j in a given graph is represented with similar proximity values in source vector of v_i and target vector of v_j. When there is no reverse link, a very different value is assigned for the source vector of v_j and target vector of v_i. The intuition behind building these vectors is that, the more and shorter are the paths available between v_i and v_j, the more similar will their source and target vectors respectively. This idea is close to metrics that measure higher order proximities in graphs such as Katz index, rooted page rank, common neighbors and Adamic-adar scores. We use the following notations henceforth for elaborating on how HOPE works.

Notation	**Description**
$\mathbf{M}_g$, $\mathbf{M}_l$	Polynomial of matrices
$\mathbf{S}$	higher-order proximity matrix
$\mathbf{U}$	Embedding matrix
$[\mathbf{U}^s, \mathbf{U}^t]$	Source and target vectors in the embedding matrix
$\mathbf{A}$	Adjacency matrix

Table 2.1: Notations used to describe matrices in HOPE

The authors adopt L_2-norm to describe the loss function that is to be minimized.

$$\min||\mathbf{S} \text{ - } \mathbf{U}^s \textbf{ . } \mathbf{U}^{t^T}||_F^2$$

A general formulation that facilitates the approximation of higher order proximities is given by:

$$\mathbf{S} = \mathbf{M}_g^{-1} \textbf{ . } \mathbf{M}_l$$

To be self-contained we very briefly introduce some proximity measures and their transformations into the above formulation.
Katz index. Katz index[] or katz centrality is built on the notion that *"A node is important if it is linked from other important nodes or if it is highly linked."*[5]. It is a

[5]https://www.sci.unich.it/ francesc/teaching/network/katz.html

weighted summation of all the paths passing between two given vertexes in a graph, and is given by:

$$\mathbf{S}^{Katz} = (\mathbf{I} \text{ - } \beta.\mathbf{A})^{-1}.\beta.\mathbf{A}$$

where $\mathbf{I}$ is the identity matrix and β is the decay factor that penalizes connections made with distant neighbors.
Rooted page rank.The similarity matrix is derived from the probabilities that a random walk from any node v_i lands at v_j in steady state, and it is given by:

$$\mathbf{S}^{RPR} = (1 \text{ - } \alpha).(\mathbf{I} \text{ - } \alpha\mathbf{P})^{-1}$$

where $\mathbf{P}$ is the probability transition matrix and α is probability of random walk to a neighbor.
Common neighbors.For directed graphs, common neighbors are the number of vertices that act as a source to a vertex v_i and target to vertex v_j. It is given by:

$$\mathbf{S}^{CN} = \mathbf{A}^2$$

Adamic-adar. Adamic-adar assigns a weight to each neighbor, which is the reciprocal of degree of the neighbor. It is given by:

$$\mathbf{S}^{AA} = \mathbf{A.\ D.\ A}$$

where, $\mathbf{D}$ is a diagonal matrix given by,

$$\mathbf{D}_{ii} = 1/\sum(A_{ij} + A_{ji})$$

In order to obtain an optimal k-rank approximation of the proximity matrix, A generalized singular vector decomposition is performed on the proximity matrix and the largest K singular values and their respective vectors are used to build optimal embedding.

$$\mathbf{S} = \sum_{i=1}^{N} \sigma_i \mathbf{v}_i^s \mathbf{v}_i^{t^T}$$

where $\sigma_1, \sigma_2, ...\sigma_N$ are set of singular values and $\mathbf{v}_i^s$, $\mathbf{v}_i^t$ are corresponding singular vectors associated with σ_i.

Node2vec

Node2vec[GL16] is a way of representing nodes of a graph as vectors in vector spaces, which relies on the skip gram model used by word2Vec. The skip gram model used by word2vec has many different applications in machine learning, specially in natural language tasks. It involves training a neural network with a single hidden layer to perform a specific task. However, in word2vec (or node2vec), it is not used for the task it is trained on. Instead, the aim is to just learn the weights of the hidden layer. To be precise, if given a word from the middle of a sentence and a “near by” word for the given input word is picked at random, the neural networks provides us with the probability of that particular word being the “near by” word to the input word.

An insight into the model of word2vec. The output probabilities tell us how likely it is to find a randomly chosen word being the “near by” for the input word. The input of the neural network are a set of training samples taken from a sentence in the way shown in Figure 2.9[6]. For the sentence “The quick brown fox jumps over the lazy dog”,

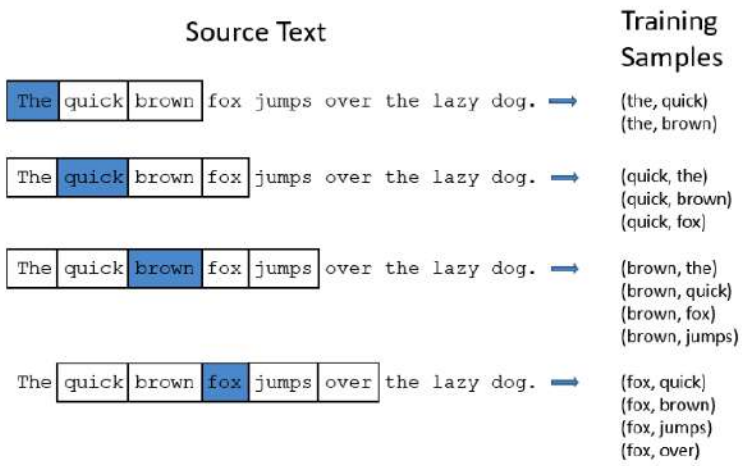


Figure 2.9: Word2Vec Training neural network by feeding it with word pairs from our training documents

if the input word is “quick”, the training samples are picked from the words near by as word pairs. In Figure 2.9, the words highlighted in blue represent the input word. The above example uses a window size of two for each training sample. This window size can be configured depending on the requirements of the user. For example, if the window size is five, five words behind and five words ahead are considered to form the training samples. While training the neural network, the input vector is given a as one-hot vector. If our vocabulary has 5000 unique words, this vector will have 5000 components with “1” at the position of the input word and 0’s at all the other positions.
For example, If we are learning word vectors with 100 features and we have 5000 words in our vocabulary, the hidden layer is represented by 5000×100 weight matrix. Therefore, the matrix vector multiplication of one-hot vector and weight matrix yields the matrix row in the weight matrix, corresponding to “1”.
The output layer(Figure 2.11[7]) is a softmax regression classifier, which gets the 1×100 word vector as input. Each output neuron multiplies its weight vector with the word vector from the hidden layer and applies the activation function ***exp(x)*** to get the results. Finally, the result is divided by the sum of results from all 5000 output nodes, so that the outputs sum up to 1.

[6]http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/
[7]http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

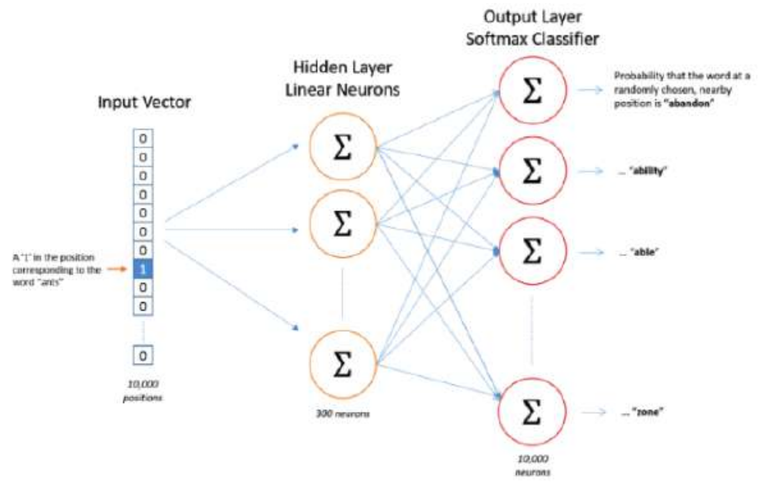


Figure 2.10: Neural network architecture

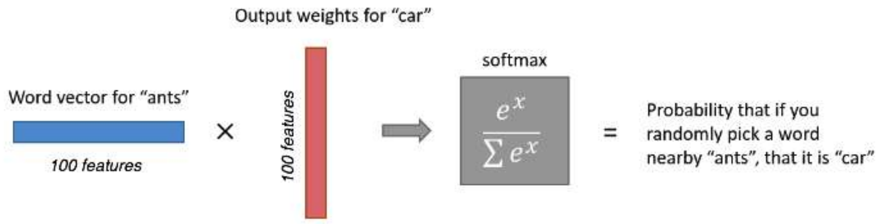


Figure 2.11: Word2Vec Output layer: calculating output of the output neuron for the word "car".

Node2vec. Node2vec follows the exact same approach for calculating node embeddings. However, the way node2vec collects training data is much different compared to word2vec. Node2vec follows second order random walks to obtain training data.
Consider a random walk that just traverses edge (t, v) and now resides at node v. The

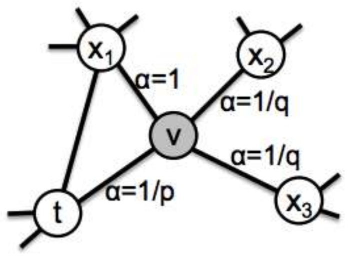


Figure 2.12: Node2vec's second order random walk approach to obtain training data

walk now needs to decide on the next step so it evaluates the transition probabilities π_{vx} on edges (v, x) leading from v (Figure 2.12[GL16]).
Intuitively, the transition probability determines the likelihood of immediately revisiting a node in the walk. If parameter q is relatively low, walk is more inclined to visit nodes

which are further away from the node t. If q is relatively high, the random walk is biased towards nodes close to node t. Figure 2.13[GL16] shows Complementary visualizations of

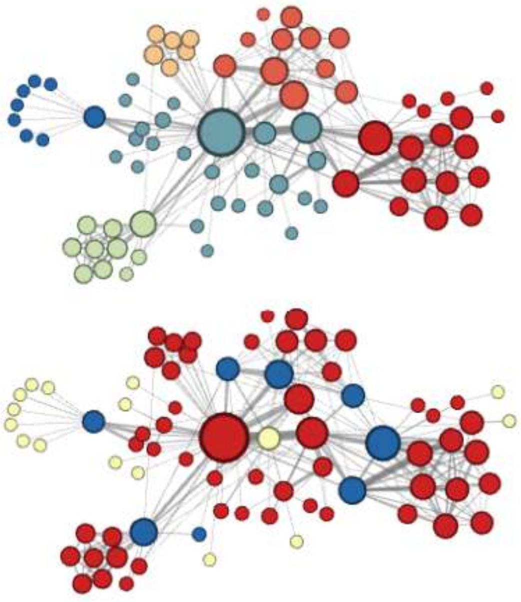


Figure 2.13: Visualizations of Les Misérables coappearance network embedding generated by Node2vec

Les Misérables coappearance network generated by node2vec with label colors reflecting homophily at the top and structural equivalence at the bottom.

Locally linear embedding

Locally linear embedding[RS00] aims at creating embeddings in low dimensional spaces assuming that each node is a linear combination of its neighbours in the embedding space. Therefore, it is defined that,

$$Y_i = \sum_j W_{ij} Y_j, \forall i \epsilon V$$

where,W_{ij} is an element of the weighted adjacency matrix of the graph in consideration which represents weight of edge between node j and node i.
The embedding is obtained by minimizing,

$$\phi(\mathrm{Y}) = \sum_i |Y_i - \sum_j W_{ij} Y_j|^2$$

Just like LAP, the problem of LLE also broils down to finding eigenvectors of the representation matrix of the graph under consideration, whose solution is to take the last d+1 eigen values of the sparse matrix $(I-W)^T(I-W)$. LLE is said to preserve first order proximity with a computational complexity of O(E$|d^2$), where V is the number of nodes and d is the dimensionality.

In this section we described several approaches to embedding graph data. There exist further approaches which consider also content of the nodes. In the next section we review some early work adopting summarization techniques in graph data management.

2.3 Related work in summarization of large graph data in data management

Graph summarization is a relatively new field with a lot of scope for exploration. The existing techniques either process static or dynamic graphs.

Surveys

[YPS$^+$13] presents a survey that mainly focuses on partition-based and compression-based summarization techniques on static graphs. [LY13] delivers a survey on more specific methods, again on static graphs only. [LKF05] , in contrast, deals with de-densification of time evolving graphs. According to this paper, graphs "densify" as they evolve over time with number of edges growing super-linearly in number of nodes. It introduces a method for de-densification of these dynamic graphs (a way of summarization) and also shows that it is related to temporal evolution of degree distribution. [LSDK18] provides a comprehensive survey about summarization techniques based on inputs that these techniques process, the core technique itself and output of these techniques. It provides a taxonomy of approaches to graph summarization.

Graph partitioning, aggregation and compression based summarization

Specifically, the sparsification library we used in this thesis needs input in METIS format. METIS introduced in [KK00] iteratively finds maximal subgraph matchings and merges nodes that incident the edge of the matching. The result on the most "coarsened" is then projected to the original graph. This approach yields compact hierarchical representations of the original graph and almost duplicates the process of summarization. [NG04, YL13] also present approaches which deal with grouping of edges into super nodes also serve the same purpose. [RGSB17] is another method that also generates super nodes and super edges with some guarantee.

Other variants of summarization techniques are the ones applied in compressed sensing settings. These methods essentially reduce dense graphs to sparse graph by preserving some properties of a network[AGM12b, AGM12a, McG14].

[MP10] provides a way to compress social networks and query them efficiently without decompression. [Ahn13] summarizes biological networks via bit compression. [BV04] describes a way to compress web graphs and access the compressed graphs, again, without decompression.

Sparsification techniques

The sparsification techniques we use in this thesis are also structure preserving, designed specifically for social networks. These techniques employ edge filtering method by globally assigning scores to edges based on a metric (algebraic distance, triangle count, for instance). The edges with edge scores higher than a specified threshold are kept.

[MBC+11] introduces a method called SPINE which sparsifies an influence network by observing a log of previous propagations. Another variant of sparsification is spectral sparsification. [SS11] provides a technique to sparsify graphs by keeping edges with a probability proportional to the effective resistance of the edges. [ST11, BSST13] offer methods for sparsification based on spectral similarity and [KLM+17] offers methods for spectral sparsification on dynamic streams.

Graph representation learning

Novel deep learning based network embeddings are becoming increasingly famous in recent times. The idea of graph embeddings is to reduce the graph to a lower dimensional space which could possible form a summary of the original graph.[DKM06, DKM06, MGF11] offer novel factorization approaches on graphs, which reduce a given graph to much lower dimensional spaces by calculating low rank approximations of the adjacency matrix which can be seen as approximate summaries of the original graph. [PARS14, GL16] adopt nodes as low dimensional vectors and, techniques like SDNE[WCZ16] and LINE[TQW+15] take as input the adjacency matrix of a graph to an artificial neural networks to form low dimensional structure preserving network embeddings. More recently, [HPC+18] introduces an approach called PAGE, that makes an attempt to answer latent queries via sub graph matching in the embedding space. [NHH+18] integrates RDF data to vector space to create knowledge graphs. [WWL+18] allows approximate querying in RDF embedding space.

Summarization in database research

The database community has contributed to the field of graph summarization in different ways. [THP08] introduces SNAP and k-SNAP summaries by grouping the nodes which share the same attributes. These groups iteratively split to produce groups which are compatible with relationships to eventually produce maximum attribute and relationship compatibility. k-SNAP further allows roll-up and drill-down operations on these groups enabling control of summary resolution. [HST13] provides a scalable algorithm for graph summarization based on pure relational technology. [STWJ13] extends the above algorithm but enables data security. [FLWW12] proposes a query preserving lossless compression algorithm. [SWL+18] introduces a lossy data summarization scheme in which directed paths up to length 'd' are preserved. This is achieved by grouping nodes with similar entities within d hops to one another. [KS08] reduces the search space of frequent item sets by introducing an algorithm known as R-KRIMP which summarizes multi-relational data. [ČGM15] proposes a query preserving summarization algorithm on RDF graphs.

Though studies propose new summarization methods which are highly tied to data management and real world querying, there is a scarcity of studies to evaluate how these methods can be leveraged by users of commercial or off-the-shelf databases. Therefore, from our background study we identify a research gap which we seek to address in our work.

Summary

Graph data summarization is a relatively new research field that is gaining attention in recent times due to increasing applications of linked data. There is a lot of work in the literature that deals with this area of study. Graph sparsification is the process of putting some edges out of memory based on a selection criteria, preserving specific properties of the graph, and making it easy to process with lower memory foot prints. In this thesis we evaluate random edge sparsification , algebraic distance sparsification, local degree sparsification, local similarity sparsification, and triangle sparsification. The generic framework of any sparsification technique is the sampling strategy it employs. Sparsification can either be global or local depending on various criteria.

Graph embeddings are vector representations of nodes of a graph in lower dimensional vector spaces. In these representations, structure and other inherent properties are preserved. Background properties of the graph are preserved, and the graph is accessible to vector-based machine learning methods. Laplacian eigenmaps, Higher order proximity preserving network embedding, Node2vec, and Locally linear embedding are four embedding techniques we evaluate. Laplacian eigenmaps and locally linear embedding are based on dimensionality reduction, node2vec employs random walk approach to collect training data, and HOPE preserves higher order proximities through a complex mechanism that relies on distinguishing the directions of the edges when embedding the nodes.

In the next chapter we introduce our research questions and experimental design.

3. Prototypical Implementation

In this chapter, we discuss the prototypical implementation details for both our chosen methods. We discuss, in detail, the hardware and software configurations, datasets and libraries used for each of the methods. This chapter is organized as follows:

- In Section 3.1, we discuss the research questions for sparsification and embedding techniques.
- In Section 3.2, we talk about the implementation details, for both the methods, LDBC SNB datasets, METIS, Networkit and GEM libraries, and provide some code snippets of our implementation.

3.1 Research questions

In this thesis we seek to address the practical research gap existing for the adoption of summarization techniques in graph data management. Specifically we propose to study for an off-the-shelf database what are the benefits of data summarization, in terms of runtime for operations that match the data summary. We also would like to consider how efficient the data summary is, in preserving the characteristics that the operation studies. Finally, for practical purposes we also propose to include information regarding the creation of the summaries when using open source libraries, over benchmark data.
We formally propose the following research questions:

3.1.1 Sparsifiers

1. How does the time required to sparsify graph data change with each sparsification technique?
2. How does average pagerank and betweenness centrality and community count get affected if the graph data is sparsified using different techniques?
3. How does query execution time change when different queries are run on data sparsified using different techniques?

4. Are these techniques scalable, that is, do the above findings change for data with a higher scale factor?

3.1.2 Embeddings

1. What is the training time for each graph embedding technique when executed on LDBC data with 500 persons?

2. What is time taken to find the pairwise cosine similarities for the embedded data obtained from node2vec on the Neo4j database when compared to manual implementation using the Python programming language?

3.2 Implementation

3.2.1 Sparsifiers

We have implemented different types of sparsification techniques using the Networkit library, with a Python 3.5 interpreter on an Ubuntu 16.04 operating system for two scale factors, SF1 and SF10. We used data generated by LDBC benchmark and made use of the METIS library for obtaining suitable file formats.

3.2.1.1 Hardware and software configurations

- Machine configuration:
  - Operating System: Ubuntu 16.04 LTS
  - Processor: Intel® Core^{TM} i5 CPU 660 @ 3.33GHz×4, 64-bit
  - disk: 483,9 GB
  - 7.6GB RAM
- python version: 3.5
- networkit version: 3.4
- Neo4j version: 3.4.9

3.2.1.2 Dataset

We have used the LDBC SNB DATAGEN for data generation. DATAGEN generates data in CSV files that mimic real world social networks. Figure 3.1 shows the schema of LDBC SNB data generated by DATAGEN. For sparsification we only use person to form nodes and 'person knows person' to form relationships.

Generally, DATAGEN generates a maximum of 33 CSV files in many different configurable scale factors. For our evaluation, we have used data of scale factors SF1 and SF10. For sparsification, we use person for nodes and 'person knows person' for relationships.

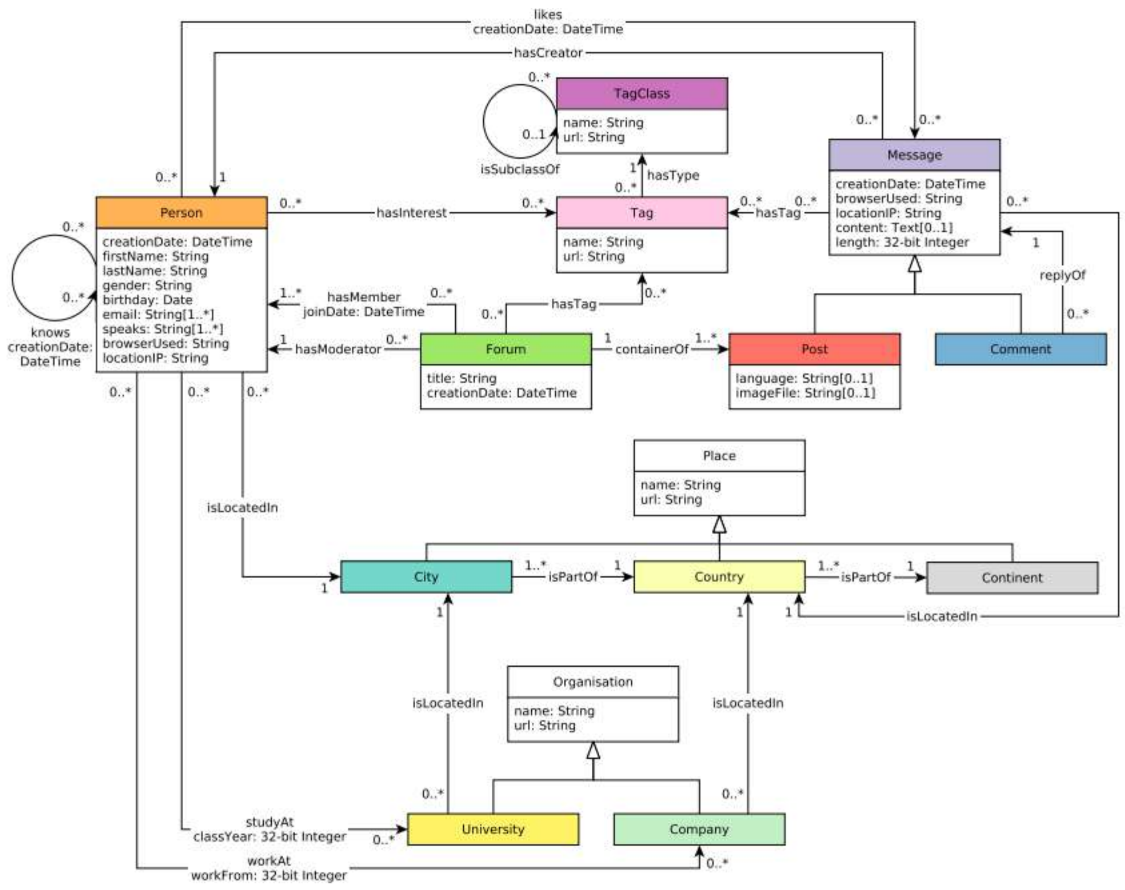


Figure 3.1: LDBC SNB Data Schema

File	Attributes
person_0_0.csv	id, firstName, lastName, gender, birthday, creationDate, locationIP, browserUsed
person_knows_person_0_0.csv	Person.id, Person.id, creationDate

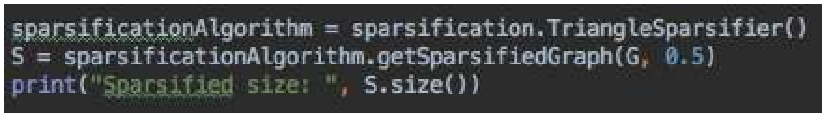
Table 3.1: The list of CSV files we used for implementation.

3.2.1.3 Libraries

Networkit

Networkit[SSM14] is a python graph library which was implemented specifically for sparsification of social networks. It is an open source software package that can be accessed interactively for data analysis using python. A wide variety of sparsification techniques were implemented and evaluated on real time datasets. Figure 3.2 shows our usage of triangle sparsification technique using networkit in python.

Networkit uses *Cython* for writing higher lever Python wrappers. Cython compiles Python source code to machine-code and also, supports a large subset of Python language. It acts as a Python extension that allows type declaration (which is not native



```
sparsificationAlgorithm = sparsification.TriangleSparsifier()
S = sparsificationAlgorithm.getSparsifiedGraph(G, 0.5)
print("Sparsified size: ", S.size())
```

Figure 3.2: Triangle sparsification using Networkit

to Python programming language) and is compiled directly to C/C++ side-stepping the Python interpreter which leads to significant speedups. Moreover Cython can access C++ libraries directly, which allows the programmer to exploit best of both the worlds[BBC+11].

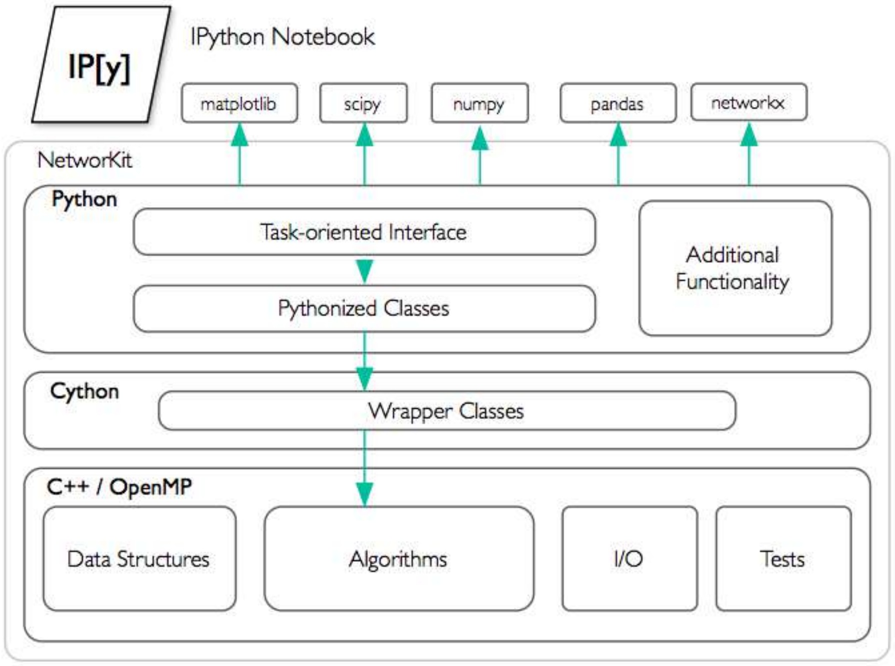


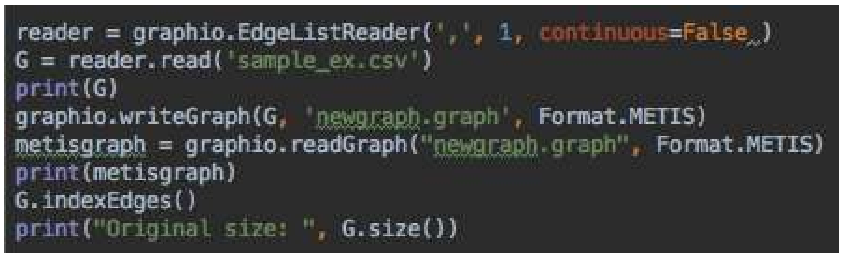
Figure 3.3: An overview of the Networkit architecture

Figure 3.3[SSM14] shows the overview of the architecture of Networkit. Networkit is a combination of high-performance C++ code with additional Python functionality and interface. It allows integration of Python libraries such as pandas, numpy, matplotlib, scipy, and networkx for data analysis and scientific computing. Cython as described above is used for writing wrapper classes. OpenMP provides shared memory parallelism. However we did not use this last feature in our work.

METIS

METIS[KK95] is a framework often used to partition graphs. It takes an input graph and partitions the graph in a balanced way such that each partition has about the same number of nodes and optimally minimized number of cut edges. METIS GRAPH Files have a specific format and certain characteristics, which are suitable as input files for sparsification using the Networkit Python library.

Networkit's *networkit.graphio* has several classes to perform various operations on graphs including read and write. Figure 3.4 shows our usage of METIS file formats that takes



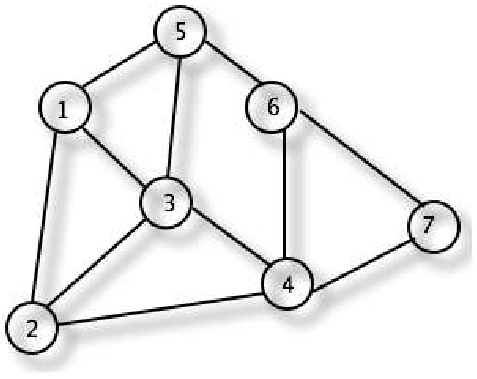
```
reader = graphio.EdgeListReader(',', 1, continuous=False )
G = reader.read('sample_ex.csv')
print(G)
graphio.writeGraph(G, 'newgraph.graph', Format.METIS)
metisgraph = graphio.readGraph("newgraph.graph", Format.METIS)
print(metisgraph)
G.indexEdges()
print("Original size: ", G.size())
```

Figure 3.4: Generation of METIS files

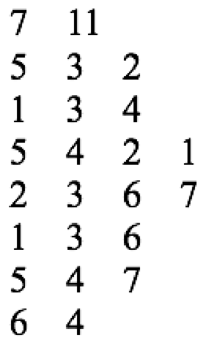
inputs from CSV files, reads them as edge lists and coverts them into METIS format.

METIS graph files characteristics and example:

- A graph of N nodes is stored in a file with N+1 lines.
- The first line of the graph contains the number of nodes and edges of the graph. In addition, it might also contain values that indicate the weights of the graph for weighted graphs.
- Each line in the file after the first line lists the neighbours of the nodes.
- % in the beginning of a line indicates that the line is commented.



(a) Sample graph



```
7  11
5  3  2
1  3  4
5  4  2  1
2  3  6  7
1  3  6
5  4  7
6  4
```

(b) Graph file

Figure 3.5: Example of METIS Graph file format for unweighted graphs

3.5(a) and 3.5(b)[1] shows an example of METIS file. The first line of the file represents number of nodes and edges. Subsequent lines indicate the neighbors of the nodes. For example, neighbors of node 1 can be seen in the first line of the graph, neighbors of node 2 in second line, and neighbors of node 7 can be seen in the last line.

[1]METIS manual - http://glaros.dtc.umn.edu/gkhome/

3.2.2 Embeddings

We have implemented different graph embeddings using the GEM open source python.

3.2.2.1 Hardware and software configurations

- Machine configuration:
  - Operating System: Ubuntu 16.04 LTS
  - Processor: Intel(R) Xeon(R) CPU E5-2609 v2 @ 2.50GHz processors×8, 64-bit
  - 251GB RAM
- python version: 3.5
- networkit version: 3.4
- Neo4j version: 3.4.9

3.2.2.2 Dataset

We have used Data generated by LDBC SNB DATAGEN for evaluation of graph embedding techniques. We used the following entities out of all entities that LDBC offers(Figure 3.1) for graph embedding techniques implementation.

File	**Attributes**
person_0_0.csv	id, firstName, lastName, gender, birthday, creationDate, locationIP, browserUsed
tag_0_0.csv	id, name, url
person_hasinterest_tag_0_0.csv	Person.id, Tag.id

Table 3.2: The list of CSV files we used for implementation.

'Person has interest' represents the relationship between a person and a tag, showing the interest a person has on a topic. Hence, in this graph both person and tag entities form nodes of the graph. LDBC SNB offers users the option to configure the size of the data, in two ways: by setting the scale factor or by configuring the number of persons, starting year and number of years. We chose to set the number of persons to be 500. This yielded us roughly 2697 nodes and 16056 relationships, out of which 2197 was the number of unique tags. We chose this size because the library that we use for evaluation of different graph embedding methods is scalable only up to this scale factor.

3.2.2.3 GEM python library

GEM[GF18] stands for graph embedding methods. It provides open source implementations[2] of various graph embedding techniques such as locally linear embedding, laplacian eigenmaps, graph factorization, HOPE, SDNE and node2vec. For node2vec, authors of [GF18] only created a python interface for the original C++ implementation[3]. GEM also provides an interface to evaluate learned embeddings for link prediction, node classification, graph reconstruction, and visualization. This interface supports multiple metrics like cosine similarity and euclidean distance. GEM uses python libraries such as numpy, scipy, matplotlib, and networkx for graph embedding methods implementation. All the graph embedding methods can be run using a python interpreter. However, Node2vec has to be recompiled from SNAP[4]

3.2.2.4 Neo4j stored procedures

We use Neo4j stored procedures to calculate pairwise cosine similarities obtained from the obtained node2vec embeddings. We chose node2vec only because it is the only implemented embedding technique in GEM that yields embeddings along with labels(tag ids). We chose to implement pairwise cosine similarities of tags based on the users that like them. Hence we imported LDBC SNB data with 500 persons and 2197 unique tags into noe4j database(Section 3.2.2.2).
Neo4j stored procedures are mechanisms through which querying neo4j can be extended by writing custom code in java programming language in such a way that it can be called directly via cypher. The java code has to be compiled into a *jar* file and put in the plugins directory of noe4j root folder. We called the stored procedure we wrote for pairwise cosine similarity using cypher as follows:

$$CALL\ algo.procedure.cosine()$$

Given below is a code snippet of the stored procedure we implemented for calculating pairwise cosine similarities in Neo4j.

```
public class FullTextIndex
{

   private static final Map<String,String> FULL_TEXT =
          stringMap( IndexManager.PROVIDER, "lucene", "type", "fulltext" );


   @Context
   public GraphDatabaseService db;
```

[2]https://github.com/palash1992/GEM
[3]https://github.com/aditya-grover/node2vec
[4]https://github.com/snap-stanford/snap

```
    @Context
    public Log log;


    @Procedure(value = "similarity.procedure")
    @Description("Execute lucene query in the given index, return found
        nodes")
    public Stream<SearchHit> search()
    {
      Stream<SearchHit> s1 = null, s2;
      Boolean s1Empty= true;
      String queryString="";
      List<String> a= new ArrayList<>();

    String[] emb = {
            "0.0797428,0.182545,0.0576887,0.0351693",
            "-0.0777048,0.386052,0.584654,3.87082",
            "-0.0813391,0.0114495,-0.0740742,-0.089435",
            "-0.106586,0.0660901,0.0476956,0.11351",
            "-0.127242,0.333151,-0.0536305,0.245765",
            "-0.015756,0.265684,-0.0116511,0.181644",
            "-0.0444543,0.169539,-0.0595563,0.103512",
            "-0.0888892,0.116316,0.0829568,0.279422",
            "-0.0572816,0.36165,-0.0141653,0.0933123",
            "0.0701616,-0.417067,-0.158711,-0.412007",
                }
     queryString="WITH [";
      for(int i=0;i<emb.length-1;i++){
            queryString+="{item: "+i+", weights: ["+emb[i]+"]}, ";
         }
         queryString+="{item: "+(emb.length-1)+", weights:
             ["+emb[emb.length-1]+"]}] as data CALL
             algo.similarity.cosine.stream(data) YIELD item1, item2,
             similarity RETURN item1, item2, similarity;";
         s1=db.execute(queryString).stream().map(it->new
             SearchHit(it.values().stream().map(it2->it2.toString()).collect(Collectors.joinin
      return s1;
    }
public static class SearchHit
    {
        // This records contain a single field named 'nodeId'
        public String similarity;

        public SearchHit( String similarity )
        {
            this.similarity = similarity;
        }
    }
}
```

Summary

For our research we have selected to study the runtime of using summaries for different kinds of queries that match the summaries. We also propose to evaluate the effectiveness of the data summary in providing results to these queries which are similar to those over the non-summarized graph. Finally we also determined to study the summary creation time. These aspects are condensed in our research questions. The data set used for evaluation of sparsification techniques is generated using LDBC SNB DATAGEN. We generated data of two scale factors. The main libraries used for implementation of sparsification techniques are networkit and METIS. Networkit is a python graph library which was implemented specifically forsparsification of social networks. It is an open source software package that can beaccessed interactively for data analysis using python. METIS frame work is often used to partition graphs. It takes an input graphand partitions the graph in a balanced way such that each partition has about the same number of nodes and optimally minimized number of cut edges. METIS GRAPH Files have a specific format and certain characteristics, which are suitable as input files for sparsification using networkit Python library.
For implementation of graph embeddings, we set the number of persons to be 500. GEM library rovides open source implementa-tions2of various graph embedding techniques such as locally linear embedding, laplacianeigenmaps, graph factorization, HOPE, SDNE and node2vec.
Neo4j stored procedures were used to calculate pairwise cosine similarities obtained from embeddings. In the following chapters we answer to the research questions through evaluation.

4. Sparsifiers

In this chapter, we discuss implementation results and our evaluation details regarding sparsifiers. This chapter is organized as follows:

- In Section 4.1, we recapitulate our research questions for graph sparsification techniques.
- In Section 4.2, we discuss evaluation and discussion of our results.

4.1 Research Questions

The following are our research questions:

1. How does the time required to sparsify graph data change with each sparsification technique?
2. How does average pagerank and betweenness centrality and community count get affected if the graph data is sparsified using different techniques?
3. How does query execution time change when different queries are run on data sparsified using different techniques?
4. Are these techniques scalable, that is, do the above findings change for data with a higher scale factor?

4.2 Evaluation and discussion

We used the sparsification techniques described in Section 3.1.1 for our evaluation. We used betweenness centrality, community detection, connected components, page rank, and partition size algorithms on data sets sparsified using these techniques. To answer our research questions we first ran the sparsification algorithms on LDBC SF1 social networks data generated using LDBC SNB Datagen.

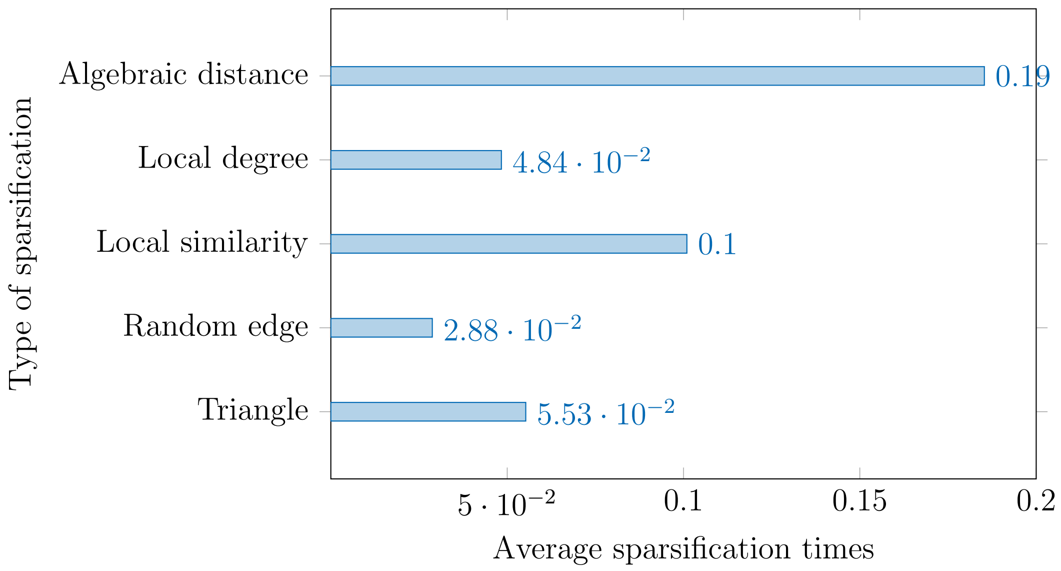


Figure 4.1: Average sparsification times (ms) for different types of sparsification

4.2.1 Sparsification times (SF1) and description of the summaries

Figure 4.1 shows the average sparsification time taken by different sparsification techniques after 50 iterations.

Unsurprisingly, random edge sparsification takes least time compared to other sparsification techniques. Random edge sparsification in networkit filters the edges with a sparsification ratio approximately equal to 0.52. Hence, the time it takes for sparsification is linear in number of edges, and is also parallelized. Since it involves selecting edges randomly to achieve a desires sparsification ratio, we select this as the baseline for our comparisons.

With this consideration, we observe that algebraic distance, with the highest sparsification ratio of approximately 0.98, takes approximately six times more time than that of random edge sparsification. It involves repeated iterations of taking weighted average of randomly selected initial vectors and then, using a parameter ω to obtain desired convergence. l_2-norm of pairs of values obtained for each node gives us algebraic distance between those nodes. Networkit by default performs 30 iterations with ω=0.5(see Section 2.2.5)

Local similarity sparsification is approximately 3.5 times slower than random edge sparsification. However, the sparsification ratio of random edge sparsification is approximately 2.58 times higher than that of local similarity.

Random edge sparsification is approximately 2 times faster than local degree sparsification with sparsification ratio approximately 2 times lower. Therefore, local degree sparsification nearly takes the same time as random edge despite the involved runtime complexity of the algorithm.

Triangle sparsification takes nearly 1.9 times more time than random edge with a sparsification ratio of nearly 0.94, that is, nearly 94% of the edges remain after sparsification. We observe that nearly 45% more edges get sparsified in triangle edge sparsification compared to random edge with 2 times slower running time.

The baseline data set has 9892 nodes and 180623 edges. Since, we use edge filtering sparsification techniques the number of nodes remain the same even after sparsification, however, the number of edges changes. The following is the information of number of nodes and edges left after different types of sparsification.

Sparsification technique	**# of nodes**	**# of edges**
Baseline	9892	180623
Algebraic distance	9893	178277
Local degree	9892	43255
local similarity	9892	35022
random edge	9892	90541
Triangle sparsifier	9892	170049

In the appendix we include visualizations for the sparsifications at the different scale factors. We can observe that all methods preserve, at least visually, the global and local structure of the graph, except for local similarity and local degree, which show differences. Since the layouting algorithms also has an incidence in this, it is difficult to make clear conclusions about the extent of the difference.

4.2.2 Queries over sparsified data (SF1): results

To answer our second research question, we ran the following cypher queries on both sparsified (using all the five techniques we use) and unsparsified LDBC SNB SF1 social network data by importing it to neo4j graph database:

- *betweenness centrality:*
  ```
  CALL algo.betweenness.stream('Person','KNOWS',direction:'out')
  YIELD nodeId, centrality
  MATCH (user:Person) WHERE id(user) = nodeId
  RETURN user.id AS user,centrality
  ORDER BY centrality DESC;
  ```

- *Community detection:*
  ```
  CALL algo.louvain.stream('Person', 'KNOWS', )
  YIELD nodeId, community
  RETURN algo.getNodeById(nodeId).id AS user, community
  ORDER BY community;
  ```

- *Page rank:*
  ```
  CALL algo.pageRank.stream('Person', 'KNOWS', iterations:20, dampingFactor:0.85)
  YIELD nodeId, score
  RETURN algo.getNodeById(nodeId).name AS page,score
  ```

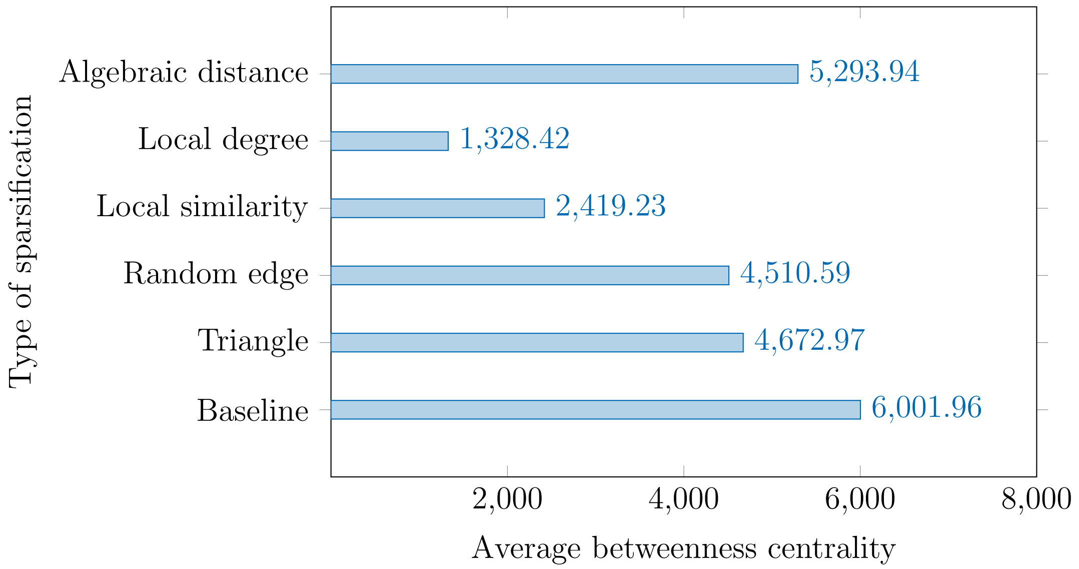


Figure 4.2: Average betweenness centrality for different types of sparsification

ORDER BY score DESC

Betweenness centrality

Figure 4.2 shows the change in average betweenness centrality for different types of sparsification techniques.

In this case, we take the average betweenness centrality of unsparsified data as baseline. Since the unsparsified original dataset has all the edges present, average betweenness centrality for this dataset is the highest. We see from Figure 4.2 that the average betweenness centrality reduces with the number of edges in the graph. Algebraic distance sparsification with an approximate sparsification ratio of 0.98 reduces the average betweenness centrality to nearly 1.13 times the original, being the result closest to it.

Triangle sparsification illustrates an average betweenness centrality which is around 1.28 lesser than the baseline with a sparsification ratio approximately 1.1 times lesser.

On the other hand, Local degree sparsification with relatively lower sparsification ratio possesses an average betweenness centrality which is nearly 4.5 time lesser than the baseline. Random edge sparsification with sparsification ratio nearly 53% lesser than triangle sparsification exhibits nearly the same average betweenness centrality with just a difference of 162 when average centralities of both are compared from the baseline.

Local similarity sparsification has an average betweenness centrality of nearly 2.48 times lesser than the baseline which is the least compared to other sparsification techniques.

Page rank

Since, both betweenness centrality and page rank are node centrality measures, we observe that average page rank also follows nearly the same trend as betweenness centrality in terms of change in page rank with respect to change in number of edges.

The following table shows the sparsification ratio of each sparsification technique and the ratio of change in page rank:

Sparsification technique	**approximate sparsification ratio**	**Decrease in average page rank**
Algebraic distance	0.98	1.3x
Local degree	0.23	2.2x
local similarity	0.19	1.7x
random edge	0.5	1.5x
Triangle sparsifier	0.94	1.33x

Table 4.1: Decrease in average page rank with respect to the baseline for each type of sparsification

We observe that algebraic distance sparsifier and triangle sparsifier exhibit almost the same average page rank, which is 1.3 times lesser than that of the original graph after 20 iterations on LDBC SF1 sparsified data, with approximate sparsification ratios 0.98 and 0.94 respectively.

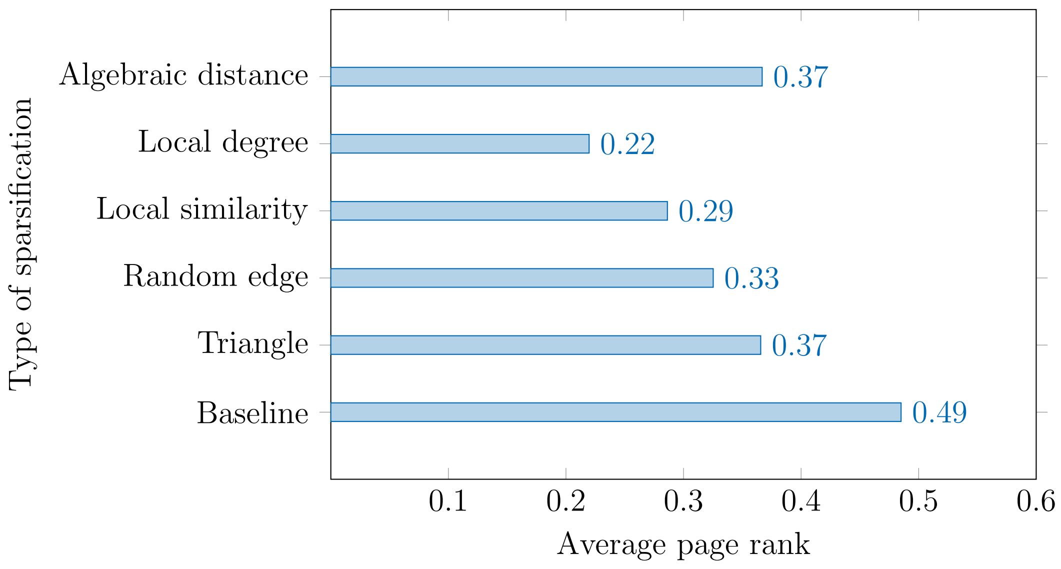


Figure 4.3: Average page rank for different types of sparsification

Local degree sparsification which sparsified the graph relatively more rapidly, shows the average page rank to be 2.2 times lesser than that of the original graph. Random edge that sparsified nearly 50% of the edges from the original graph, exhibits an average page rank nearly 1.5 times lesser than that of the unsparsified graph.

Community count

Neo4j performs community detection using Louvain algorithm. Since the sparsification techniques we employed were based on edge filtering, community counts differ for different sparisification techniques. Simply stated, by removing edges, the algorithms create the impression that there are more communities in the graph than before. In this experiment we also observed that the average triangle count lies in the same range while triangle sparsification. The following table shows the average community counts for data sparsified using different sparsification techniques.

Sparsification technique	**approximate sparsification ratio**	**Community count**
Baseline	N.A	809.16
Algebraic distance	0.98	809.26
Local degree	0.23	1220.16
local similarity	0.19	2041.38
random edge	0.5	1605.56
Triangle sparsifier	0.94	1637.16

Table 4.2: Average community count for different sparsification techniques

We see that with a sparsification ratio of 0.98, ω=0.5, numberSystems=10, and numberIterations=30, Algebraic distance sparsifier does preserve community counts. The average community count for all the other sparsifiers increases irrespective of the sparsification ratios. For local similarity, the average community count increases by nearly 2.5 times compared to the baseline. In case of random edge and triangle sparsification, the average community count increases by about 2 times and local degree sparsifier increases the community count by nearly 1.5 times the baseline.

4.2.3 Queries over sparsified data (SF1): execution times

To answer our third research question we ran the following queries in addition to the community detection, page rank and betweenness centrality in neo4j database on LDBC SF1 data:

- *Connected components*:
  ```
  CALL algo.unionFind('User', 'FRIEND', write:true, partitionProperty:"partition")
  YIELD nodes, setCount, loadMillis, computeMillis, writeMillis;
  ```

- *Strongly connected components*:
  ```
  CALL algo.scc('User','FOLLOW', write:true,partitionProperty:'partition')
  YIELD loadMillis, computeMillis, writeMillis, setCount, maxSetSize, minSetSize;
  ```

Connected components, page rank, and partition size

Figure 4.4 shows the average execution times for connected components, page rank, and partition size queries on LDBC SF1 data in neo4j database: We see that random edge sparsification and local similarity sparsification show significant speed ups compared to the baseline for all the three queries. Random edge sparsifier runs the connected components query nearly 1.28 times faster than the baseline with sparsification ratio as large as 0.5, and local similarity sparsifier runs the same query nearly 1.4 times faster

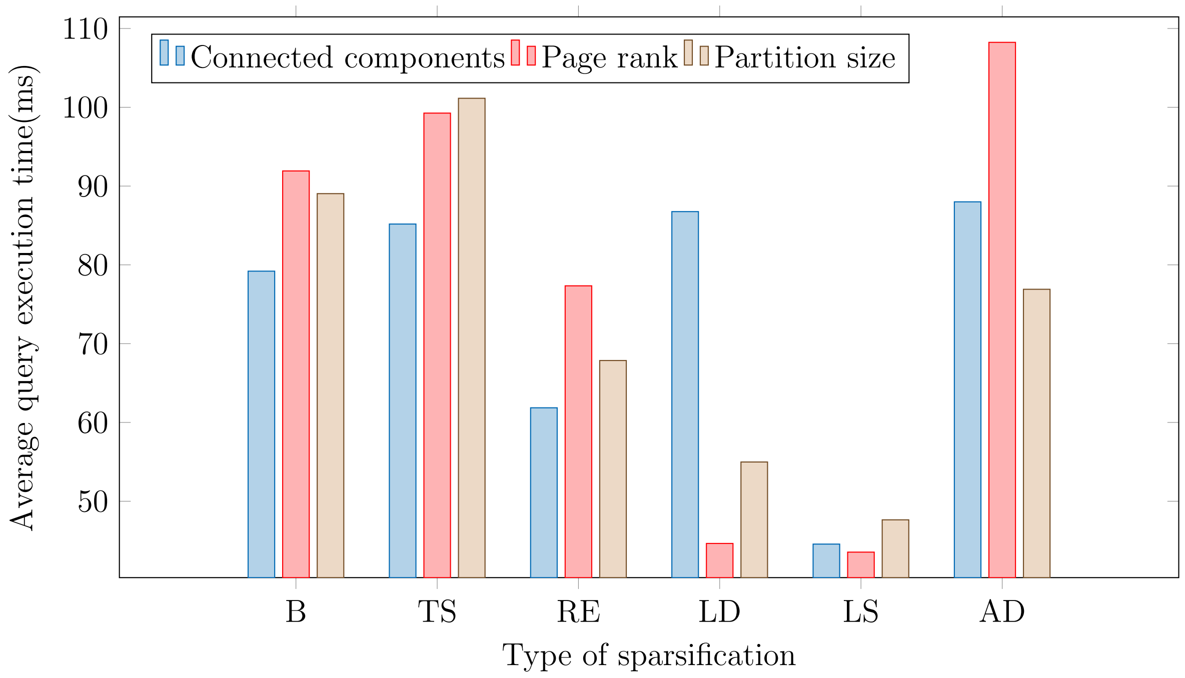


Figure 4.4: Average execution times for different types of sparsification

than Random edge sparsifier with sparsification ratio as small as 0.19.
For page rank, random edge, local degree and local similarity sparsifiers show noticeable speedups. Compared to the baseline, random edge sparsification is nearly 1.2 times faster, and local degree and local similarity sparsifiers are 2 times faster with sparsification ratios 0.5, 0.23 and 0.19 respectively.
In the case of partition size, all the sparsification techniques other than random edge, local degree, local similarity and algebraic distance have significant speed ups compared to the baseline.

Betweenness centrality and community detection
?? shows the average execution time in the betweeenness centrality and community detection tasks, for different kinds of sparsifications. In the case of betweenness centrality, all the sparsification techniques perform faster than the baseline. Triangle sparsification performs 1.6 times faster, random edge sparsification performs 2.2 times faster, local degree sparsification performs 10 times faster, local similarity sparsification performs 4 times faster, and algebraic distance sparsification performs 1.14 times faster. On the other hand, algebraic distance preserves the community count with some specific parameters although it takes nearly 2 times more query execution time than the baseline with a sparsification ratio of 0.98.

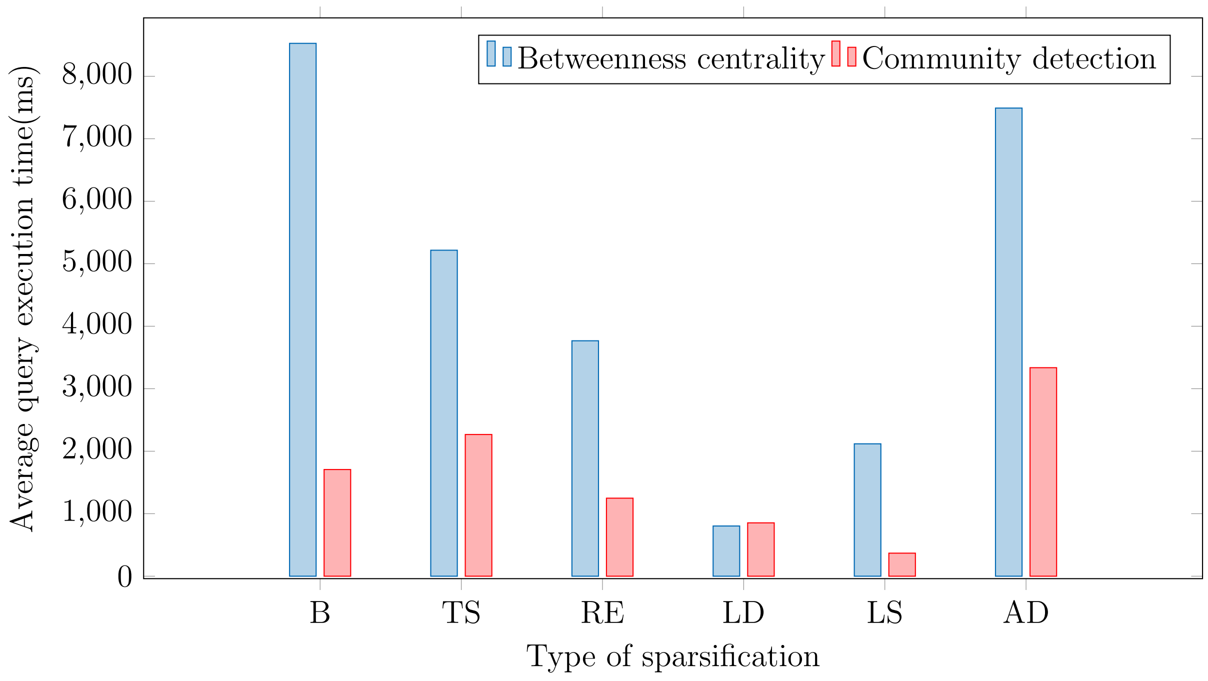


Figure 4.5: Average execution times for different types of sparsification

4.2.4 Sparsification times (SF10) and description of the summaries

To answer our forth research question, we repeated all the tests on LDBC SF10 data with 50 iterations of each query.

Sparsification times

Figure 4.6 shows the change in times, for execution of different sparsification algorithms on LDBC SF10 data. We noticed significant changes in results compared to SF1 data. The below table shows the number of nodes and edges after sparsification of LDBC SF10 data along with the sparsification ratios.

Sparsification technique	**# of nodes**	**# of edges**	**Approximate sparsification ratio**
Baseline	65645	1947294	N.A
Algebraic distance	65645	1928513	0.99
Local degree	65645	376240	0.19
local similarity	65645	308113	0.15
random edge	65645	973645	0.49
Triangle sparsifier	65645	1800432	0.92

Table 4.3: Number of nodes and edges after sparsification of SF10 data

Just as in the case of SF1, we take random edge sparsification to be the baseline. Local similarity sparsification takes the least time on SF10 data, unlike SF1 data, with sparsification ratio of 0.15, that is, approximately 15% of the edges are preserved after sparsification. However, local similarity sparsifier preserves more edges when run on

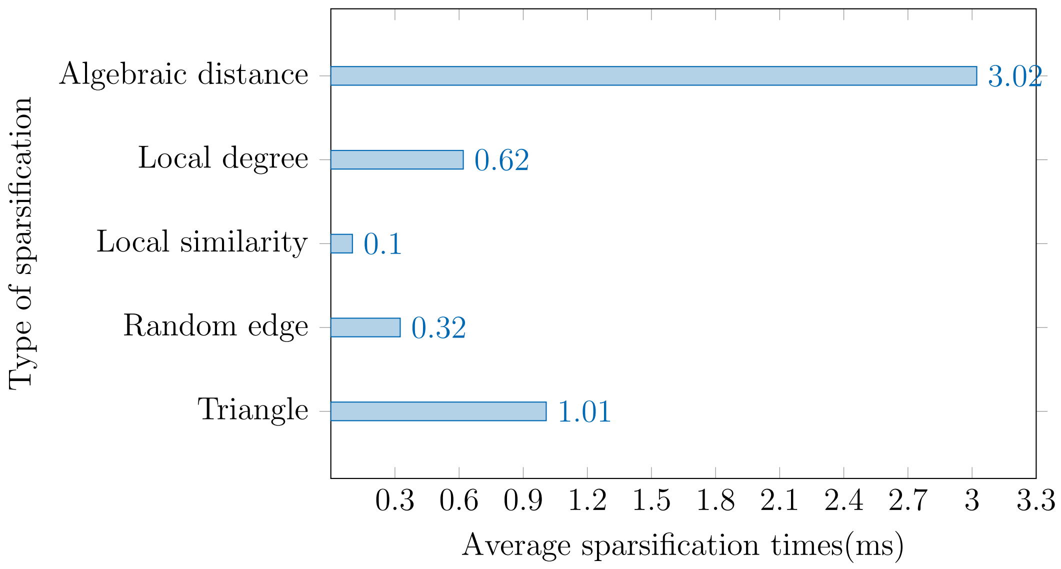


Figure 4.6: Average sparsification times for different types of sparsification on LDBC SF10 data

SF1 data with approximate sparsification ratio of 0.23. It preserves 30% more edges compared to random edge sparsifier with sparsification execution time nearly 3 times lesser.
Local degree sparsifier preserves 19% of the edges after sparsification of SF10 data. For SF1, it stores approximately 23% of the edges. However, compared o when run on the SF1 data, it is nearly 3 times slower. However on SF10 data, random edge sparsification is nearly 2 times faster.
Triangle sparsification is nearly 4 times slower than random edge with sparsification ratio that is 2 times higher. Compared to when run on SF1 data, triangle sparsification preserves nearly the same number of edges with a mild difference of 2%.
Algebraic distance takes the most time for sparsification as in SF1 data. Random edge performs nearly 10 times faster than algebraic distance with a sparsification ratio that is 2 times less. Algebraic distance sparsification scales to SF10 in terms of sparsification ratio with a slight difference of 1%.

4.2.5 Queries over sparsified data (SF10): results

Betweenness centrality

Figure 4.7 shows the change in betweenness centrality with change in sparsification technique compared to the unsparsified data. Like in the case of SF1, algebraic distance sparsification exhibits highest average betweenness centrality, with highest sparsification ratio, compared to other techniques. The average betweenness centrality is about 1.13 times less than the baseline just like in SF1.

Triangle sparsification with an approximate sparsification ratio of 0.92 in the case of SF10 data exhibits an average betweenness centrality that is 1.24 times lesser than the baseline. In the case of SF1 data, triangle sparsification showed a decrease of 33%

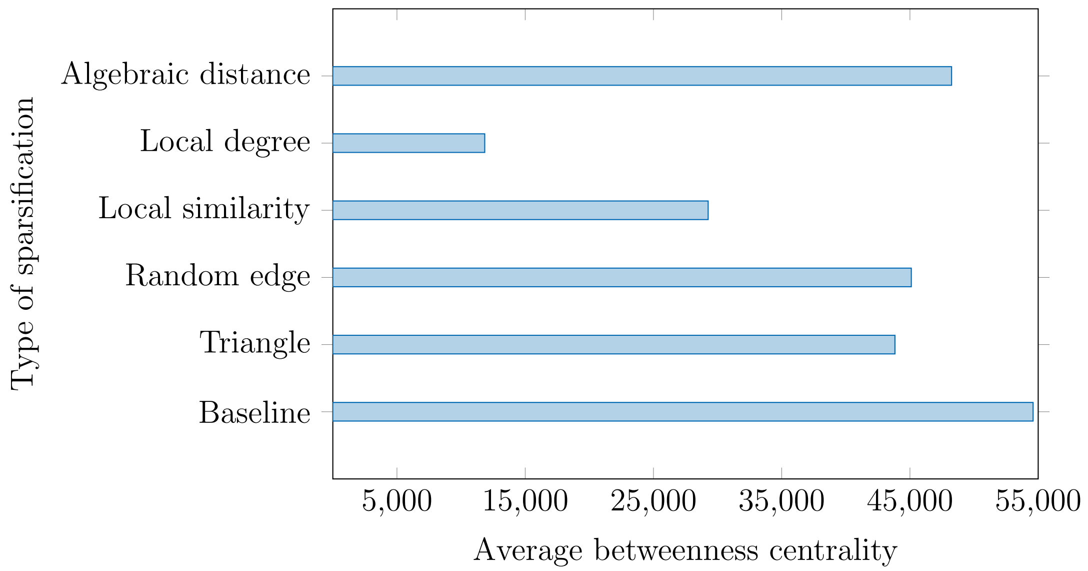


Figure 4.7: Average betweenness centralities for different types of sparsification

in average betweenness centrality with a sparsification ratio nearly equal to that of sparsification ratio of SF10 data with a mild difference of 2%.
Local similarity sparsification exhibits an average betweenness centrality, which is 1.8 times lesser than the baseline by keeping nearly 15% of the edges from the original data. Local degree sparsification keeps nearly 19% of the edges with an average betweenness centrality that is nearly 5 times lesser than that of the baseline.
Random edge sparsification with a sparsification ration of nearly 0.49 displays an average betweenness centrality that is nearly 1.2 times lesser than the baseline. For SF1 data, it shows an average betweenness centrality, which is 1.3 times that of the baseline. This goes to show that, despite the slight difference in sparsification ratios for both SF1 and SF10, the average betweenness centrality is reduced by nearly same multi folds.
Triangle sparsification reduces the average betweenness centrality of the sparsified graph to nearly 1.24 times the baseline with a sparsification ratio of 0.92. In the case of SF1, it reduces the average betweenness centrality to 1.28 times that of the baseline with a sparsification ratio of 0.94.

Page rank

Triangle sparsification reduces the number of edges in SF10 data to 92% of the original unsparsified data with a sparsification ratio that si second highest compared after algebraic distance. With this change in number of edges, it reduces the average page rank by 1.4 times. For SF1 data, it reduces the average page rank of the network by nearly 1.3 times with a sparsification ratio of 0.94.
Random edge sparsification reduces the average page rank by 1.5 times with a sparsification ratio of nearly 0.49. For SF1 data, it reduces the average page rank to the same extent with sparsification ratio of nearly same which is 0.49.
Local similarity sparsification reduces the average page rank of the network by approximately 2.5 times with a sparsification ratio as less as 0.15. Algebraic distance and local

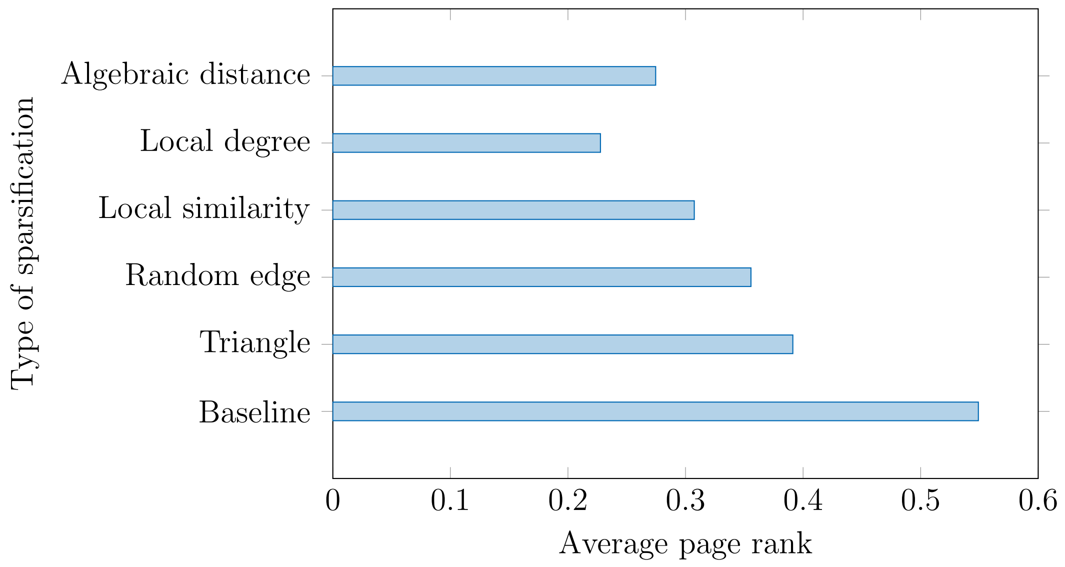


Figure 4.8: Average page ranks for different types of sparsification

degree reduce the average page rank of the network by approximately 2 and 2.4 times respectively. Sparsification ratio of algebraic distance sparsifier was 0.99 and that of local degree was 0.19.

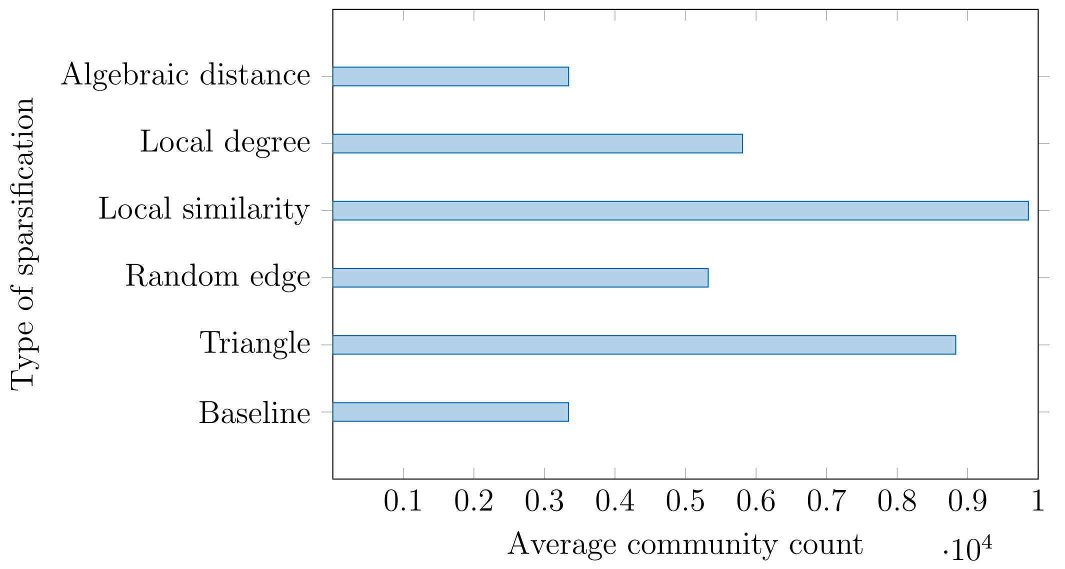


Figure 4.9: Average community count for different types of sparsification

Figure 4.10 and Figure 4.11 show the average execution times for the chosen cypher queries on SF10 data, and Table 4.4 summarizes the change in average betweenness centrality, community count and page rank for both SF1 and SF10 data for all types of sparsification.

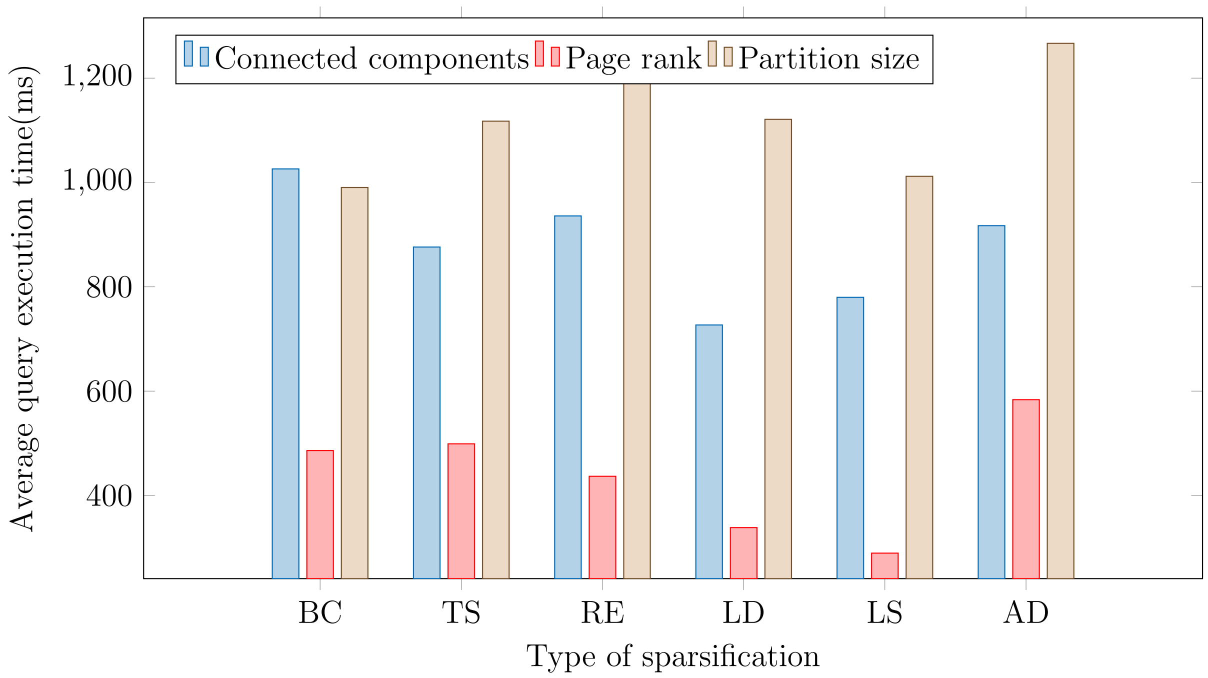


Figure 4.10: Average execution times for different types of sparsification

Summary

For SF1 data, random edge sparsification takes the least time followed by, local degree, triangle, local similarity, and algebraic distance, with respect to the sparsification task. The average betweenness centrality (after 50 iterations) is the highest for unsparsified data followed by algebraic distance, triangle, random edge, local similarity, and local degree.

Random edge sparsification and local similarity sparsification show significant speed ups compared to the baseline for connected components, page rank, and partition size. For page rank, random edge, local degree and local similarity sparsifiers show noticeable speedups. In the case of betweenness centrality, all the sparsification techniques perform faster than the baseline. On the other hand, algebraic distance preserves the community count with some specific parameters. We also observe lossless results for number of partitions for all sparsification techniques except random edge and triangle. In terms of query execution times, we observe benefits only for connected components on local similarity sparsified version.

For SF10 data, Local similarity sparsification takes the least time. It preserves 30% more edges compared to random edge sparsifier with sparsification execution time nearly 3 times lesser. Like in the case of SF1, algebraic distance sparsification exhibits highest average betweenness centrality compared to other techniques. despite the slight difference in sparsification ratios for both SF1 and SF10, the average betweenness centrality is reduced by nearly same multi folds. Algebraic distance sparsification preserves community counts in specifica settings for SF10 data. We particularly observe benefits for connected components in terms of query execution times. The average

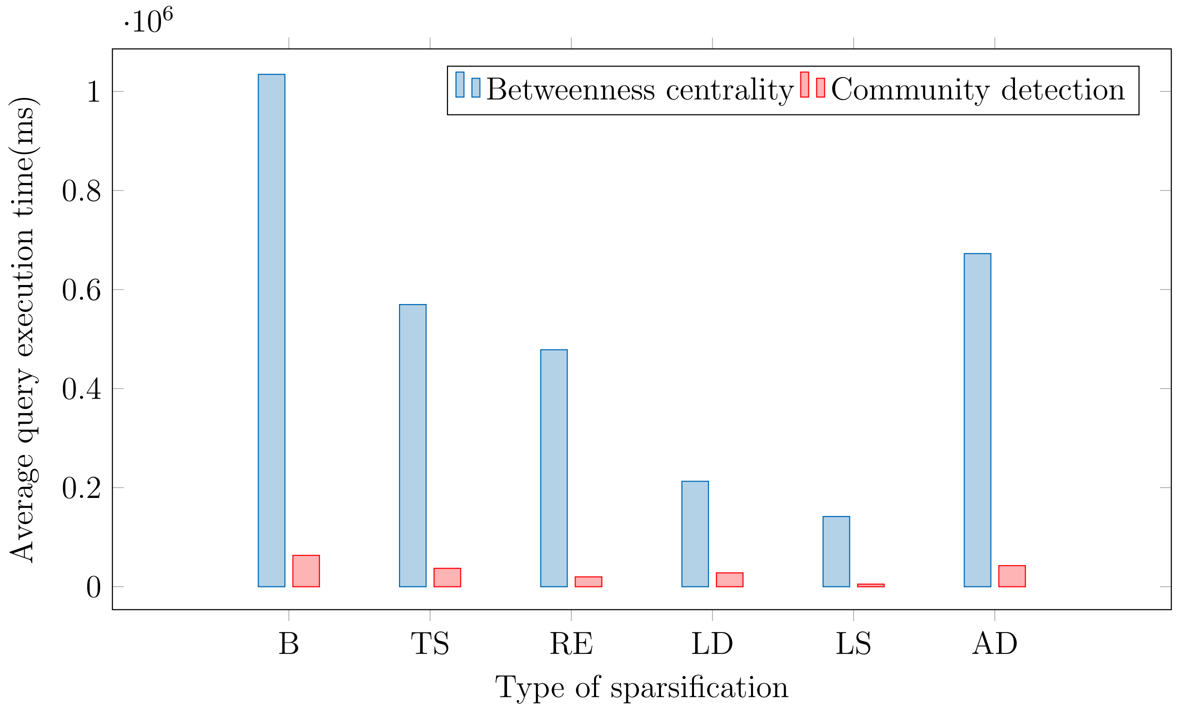


Figure 4.11: Average execution times for different types of sparsification

query execution time after 50 iterations is less compared to the baseline.

Type of sparsification	SF1				SF10			
	SR	↓↑ w.r.t baseline			SR	↓↑ w.r.t baseline		
		↓BC	↑CC	↓PR		↓BC	↑CC	↓PR
Algebraic distance	0.98	1.13	nearly same	1.3	0.99	1.13	nearly same	2
Local degree	0.23	4.1	1.5	2.2	0.19	4.6	1.73	2.41
Local similarity	0.19	2.48	2.5	1.7	0.15	1.8	2.95	1.78
Random edge	0.5	1.3	1.98	1.5	0.49	1.2	1.59	1.5
Triangle	0.94	1.28	2	1.3	0.92	1.24	2.64	1.4

Table 4.4: Average betweenness centrality, average page rank and average community count for both SF1 and SF10 data. The numbers in the table represent the number of times by which there is increase(↑) or decrease(↓) in each feature w.r.t respective baselines. SR, BC, CC, PR represents sparsification ratio, betweenness centrality, community count, and page rank respectively.

5. Graph Embeddings

Structure

- In Section 5.1, we recapitulate our research questions for embeddings.
- In Section 5.2, we talk about the evaluation and discussion of our results.

5.1 Research questions

1. What is the training time for each graph embedding technique when executed on LDBC data with 500 persons?
2. What is time taken to find the pairwise cosine similarities for the embedded data obtained from node2vec on the Neo4j database when compared to manual implementation using the Python programming language?

5.2 Evaluation and Discussion

Out of all the open source implementations provided by GEM, we used locally linear embedding, laplacian eigenmaps, HOPE and node2vec for our evaluation. To answer our first research question, we ran the above mentioned techniques using GEM python library.

5.2.1 Embedding times

We made the following observations in executions times for the chosen embedding techniques after running each of them over 100 iterations. Out of all the chosen embedding techniques, node2vec has the highest running time as seen in Figure 5.1.
We have taken a set of specifications to run node2vec:
specifications.
Number of dimensions: 4,
Length of walk per source: 80,
Number of walks per source: 10,

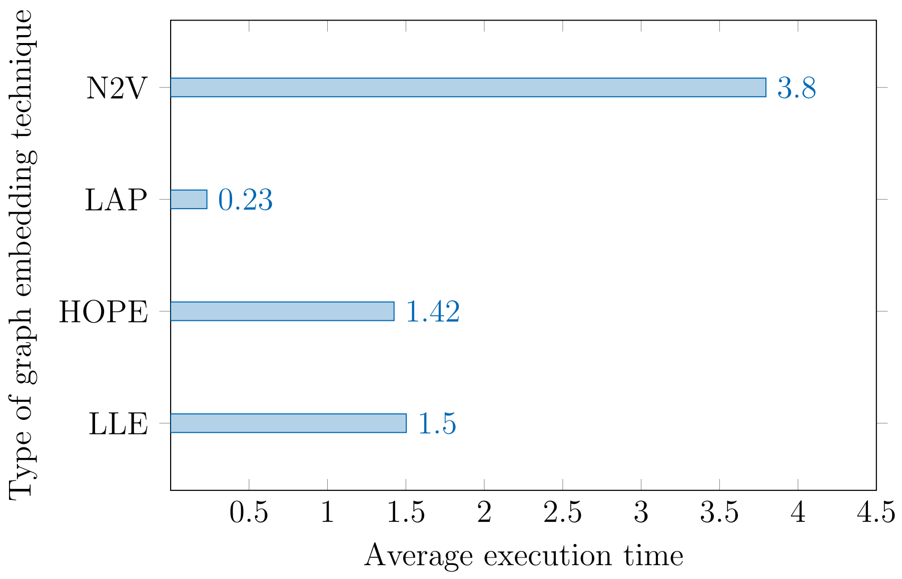


Figure 5.1: Average execution times for each embedding technique(seconds)

Context size for optimization: 10,
Number of epochs in SGD: 1
It is to be noted that when we changed the length of walks per source to 3, we observed that the execution time reduced drastically by nearly 63 times. Similarly, when we changed the number of dimensions to 128 keeping length of walk per source at 80, we observed the average running time increase by nearly 3 times. Below are the execution times for different dimensions and walks per source:
Laplacian eigen maps, which involves calculating eigen vectors of a given Laplacian matrix, takes much less time to calculate embeddings. With time complexity of O(E$|d^2$), where E is the number of edges and d is the number of dimensions, LAP preserves the first order proximity. Node2vec on the other hand, with time complexity of O(V|d), where V is the number of nodes in the graph, preserves 1-k^{th} proximity[GF18]. It is also important to note that, node2vec employs biased random walks to provide trade-off between breadth first search and depth for search approaches[GF18, GL16]. However, node2vec provides more informative embeddings in the sense that, it provides embeddings along with node ids, which no other GEM embedding implementations provide.
HOPE and LLE on the other hand take nearly 2.5 times less time than node2vec. Although both node2vec and HOPE preserve higher order proximity, HOPE has the same time complexity as LAP. The major difference between HOPE and LAP (or LLE) is that, HOPE preserves higher order proximity and LAP(and LLE) preserve only first order proximity. LLE, like LAP employs dimensionality reduction approach to obtain embeddings with a time complexity of O(E$|d^2$).

Number of dimensions	Length of walks per source	Execution times (s)
4	3	5.17967832088
4	80	5.34998829365
128	80	71.7708213568

Table 5.1: Execution times to calculate pairwise cosine similarities of node2vec embedded data in python (average of 10 repetitions each)

5.2.2 Queries over embedded data: execution times

To answer our second research question, we made use of stored procedures in Neo4j (Chapter 3). We calculated node2vec embeddings through GEM library, compiling via SNAP, and ran a stored procedure in Neo4j for calculating cosine similarities. Time taken to calculate pairwise cosine similarities of node2vec embeddings (no. of dimensions = 4, length of walks per source = 80) is *14455ms*. We observe that time taken to calculate pairwise cosine similarities for same specifications via a self-written python program is much less (in avwerage 5349 ms). We also checked the execution time for calculating pairwise cosine similiaties for different configurations in node2vec. We observed that, as the number of dimensions increased to 128, there is a drastic increase in execution time to calculate pairwise cosine similarity (Table 5.1).

Pairwise similarities of unembedded data

After, calculating pairwise similarities of node2vec embeeded data, we ran the following cypher query in neo4j database on the original nodes to check if the similarities matched:

```
MATCH (p:Person), (c:Tag)
OPTIONAL MATCH (p)<-[likes:HAS_INTEREST]-(c)
WITH item:id(c), weights: collect(coalesce(likes.score, 0)) as userData WITH
collect(userData) as data
CALL algo.similarity.cosine.stream(data)
YIELD item1, item2, count1, count2, similarity
RETURN algo.getNodeById(item1).id AS from, algo.getNodeById(item2).id AS to,
similarity
ORDER BY similarity DESC
```

We observed that the average time taken to run this query on unembedded data is 32985.6 on average over 10 repetitions.

Semantic equivalence

After calculating the pairwise cosine similarities of node2vec embedded data for different dimensions and lengths of walks per source, we tried to analyze their semantic equivalences. Since, since the meaning and goodness of the query results are not the purpose of this thesis, we add our inferences in the appendix.

Cosine similarities of node2vec embeddings explain the structural equivalence of nodes since node2vec preserves structures. If two tags have a high similarity, it is highly likely that they are liked by same set of users. Therefore, these results have applications to reccommender systems. Moreover, graph-based reccommender systems do not face the problem of data sparsity that a normal reccommender system would face. For semantic equivalence of our results, please refer to Appendix.

Summary

The execution time for node2vec changes as the length of walks per source changes. Time taken to calculate pairwise cosine similarities increases with increase in number of dimensions. The average time taken to calculate pairwise cosine similarities is significantly less when compared to the time taken to calculate via stored procedure. We also observe that the time taken to calculate pairwise similarities on unembedded data is nearly 6 times higher compared to execution on embedded data using self written python code.

6. Conclusion and Future Work

Recapitulating our research question, we examined the performance of graph data summaries on some specific graph database tasks. We loaded a social network data sets of two different scale factors that mimicked real world data using LDBC SNB DATAGEN. We ran five different sparsification algorithms on these datasets to get their structure preserving sparsified versions, using a python library, networkit.
We observed that the average community count is preserved for a specific setting in algebraic distance sparsification. We also observed significant speedups in query execution times for most of these techniques for specific queries. Particularly, although the sparsification ratio of local similarity sparsification is not the least, we see significant speedup in running all the queries for all the sparsification techniques. Specifically, Random edge sparsification and local similarity sparsification show notable speed ups compared to the baseline for connected components, page rank, and partition size. For page rank, random edge, local degree and local similarity sparsifiers show noticeable speedups. In the case of betweenness centrality, all the sparsification techniques perform faster than the baseline. For SF10 data, Local similarity sparsification takes the least time. Like in the case of SF1, algebraic distance sparsification exhibits highest average betweenness centrality, with highest sparsification ratio, compared to other techniques. In the case of graph embeddings, the average time taken to calculate pairwise cosine similarities is significantly less when compared to the time taken to calculate via stored procedures. We also observe that the time taken to calculate pairwise similarities on unembedded data is much higher higher compared to execution using self written python code.

Future work
Neo4j offers APOCS, a library to write custom algorithms and user defined functions, while using efficient primitives supported by the graph database. APOCS for different graph summarization techniques can be written and added. Existing graph summarization techniques do not deal well for graphs with properties. Hence, developing algorithms to deal with properties could be a valuable contribution. Current representa-

tional learning techniques only deal with static graphs. Graph stream embedding could be an interesting area to explore. Handling diverse input types for summarization is also an unexplored area of research. As of now, each sparsification technique only deals with very specific queries or approximate queries. Developing a generic framework that accommodates more types of queries for different types of sparsifications could be an area of improvement.

7. Appendix

In this appendix we provide further information, which might be relevant to understand better our study. We begin by disclosing the top page ranks, which apart from the average page rank reported in our study, gives more insights into the dynamics of the ranking. We found that a mismatch between identifiers prevents us from understanding in a comparable manner the results of the baseline against that of sparsified approaches. Nonetheless, it is possible to see that the top items are similar between the sparsified approaches, except for local similarity. Similar findings occur at other scale factors and for other aspects, such as betweeness centrality.

Top 10 page ranks for LDBC SNB SF1 data:

- *Baseline version:*

Id	**Rank**
32985348841922	54.9478595
30786325585162	32.4143255
32985348842270	32.4124895
32985348834375	26.659851
32985348840984	25.0260235
32985348843944	24.153915
32985348843769	20.33206
30786325583918	20.1957625
32985348842280	19.659693
30786325581208	19.0356655

- *Random edge sparsified version*

Id	**Rank**
3412	14.661047
3627	12.21745
2194	10.8081755
10995116279283	10.567073
870	8.6449255
609	8.1232635
2608	7.21299
1564	6.874061
1753	6.3664495
2199023255688	6.2369775

- *Algebraic distance sparsified version*

Id	**Rank**
3412	18.854386
2194	16.6818965
3627	15.226892
10995116279283	13.122139
870	13.001201
5113	12.907769
10026	12.3314265
6690	12.161622
609	11.08406
4848	10.199295

- *Local degree sparsified version*

Id	**Rank**
3412	51.179954
10995116281261	24.6105095
4517	18.184365
10104	13.024168
13194139539713	10.804648
4398046516395	9.343396
2608	8.978457
2194	8.4959375
8599	8.1956155
4398046514585	7.875123

- *Triangle sparsified version*

Id	Rank
3412	18.2684385
2194	17.9099595
3627	15.9220305
6690	14.1451565
5113	13.8293305
870	13.7231485
10995116279283	13.712498
10026	12.957613
609	12.415653
150	11.5862485

- *Local similarity sparsified version*

Id	Rank
6061	4.041946
4398046511185	3.978689
143	3.3842075
6717	3.2550585
2519	3.2407445
8796093022938	3.2160095
2698	3.199885
150	2.9555185
3825	2.871632
6597069769386	2.8106445

Top 10 betweenness centralities for LDBC SNB SF10 data:

- *Baseline version:*

Id	Centrality
17592186083813	9877615.796
17592186112916	9183340.82
13194139587125	7873588.086
15393162855740	7799457.614
24189255822332	7776284.116
17592186115216	7756088.929
24189255818480	7354880.78
24189255878619	7018813.933
10995116326262	7004635.847
13194139581724	6982680.058

- *Random edge sparsified version*

Id	**Centrality**
36226	10121131.14
29011	9731099.952
9116	7559465.624
55828	7497495.602
2199023306776	6148352.919
18879	4844960.302
6597069780295	4808066.216
22140	4759070.713
2199023282670	4702890.376
38518	4682517.363

- *Algebraic distance sparsified version*

Id	**Centrality**
36226	14398169.12
9116	13926489.49
29011	12094094.29
2199023306776	11123489.66
55828	10011858.21
2194	7059864.842
22140	6737954.415
2783	6579571.963
4534	6471255.633
18879	6369084.556

- *Local degree sparsified version*

Id	**Centrality**
36226	8818051.088
18879	8740315.152
29011	8438802.537
2199023308999	7258244.829
9116	6909976.18
6597069811451	6834822.973
4398046578617	6026489.061
2199023269673	5859624.788
4398046534165	5554393.51
6597069780295	5466626.357

- *Triangle sparsified version*

Id	Centrality
36226	12517524.27
9116	11942286.58
29011	11142226.38
2199023306776	10018533.75
55828	8871860.527
4398046526403	5939827.604
18879	5812819.165
2199023269673	5592670.028
2783	5533730.888
2194	5362183.108

- *Local similarity sparsified version*

Id	Centrality
39715	1876040.706
24189255824932	1806431.81
6597069828476	1683650.96
8796093077626	1442415.493
13194139555194	1366794.036
7375	1338085.015
15393162842631	1171896.245
8796093031057	1171603.247
21990232617201	1156878.089
13194139580971	1121364.785

Top 10 page ranks for LDBC SNB SF10 data:

- *Baseline version:*

Id	Rank
32985348902205	133.817481
32985348904217	88.104022
32985348904798	84.6787855
32985348905770	84.56758
35184372118193	72.019217
32985348905111	66.990991
32985348901067	66.8596405
32985348905164	62.089823
32985348905743	60.7240175
32985348905625	59.688726

- *Random edge sparsified version:*

Id	**Rank**
2783	46.7846465
2194	38.2352445
7725	32.7192375
2317	27.920996
4534	27.118562
4607	22.484651
4555	22.221236
3627	20.263397
10024	19.7005355
8061	19.6000485

- *Algebraic sparsified version:*

Id	**Rank**
2783	73.651982
2194	54.606967
7725	49.8837635
2317	49.169721
4534	39.6113095
4607	33.6236205
1564	30.909222
4273	30.191703
4555	30.0098455
6006	29.976568

- *Local degree sparsified version:*

Id	**Rank**
2783	135.1561375
2194	107.854333
7725	84.549186
4534	72.3473
9116	69.994041
55828	63.883272
4555	61.8168795
29011	56.377075
2199023306776	52.499256
36226	49.4173175

- *Triangle sparsified version:*

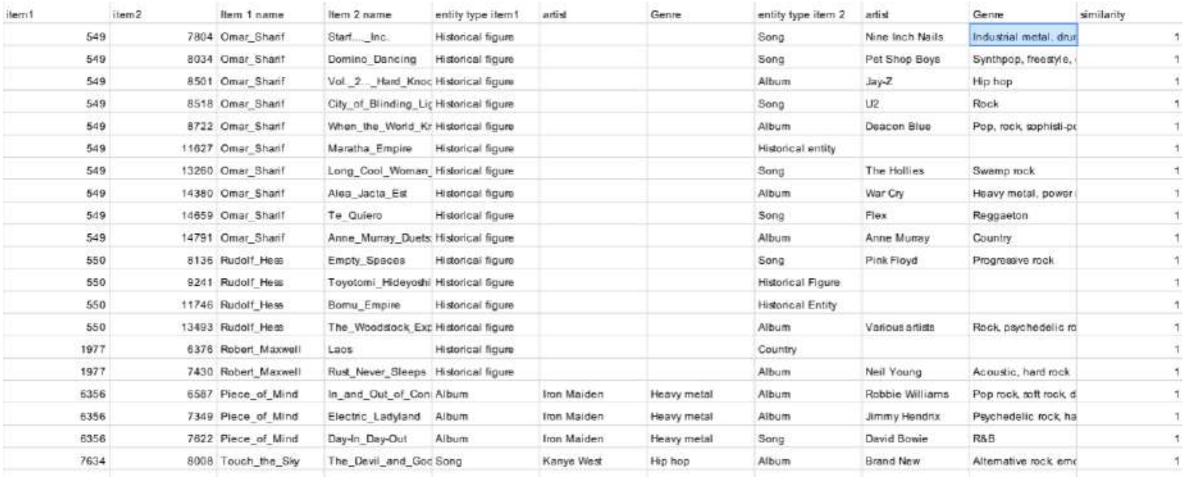
Id	**Rank**
2783	73.0690435
7725	50.3346375
2194	48.7567315
2317	47.671256
4534	36.985073
4607	31.7500505
10024	30.482845
3627	30.0686825
1564	29.883646
6006	29.3935275

- *Local similarity sparsified version:*

Id	**Rank**
870	12.3303895
4607	9.4795915
2310	8.475529
974	8.209445
14330	6.9434635
10024	6.8453905
1490	6.8334055
8911	6.1900915
2214	6.1227715
71774	6.0810195

Top 20 most similar nodes(tags) after embedding them using Node2vec technique and calculating similarities using stored procedure in Neo4j for different dimensions(d) and lengths of walks per source.
In our observation on the nearest neighbours using pairwise cosine similarity, we find that the top 20 do not seem to be reasonably well related for most cases (e.g tags that refer to historical figures are considered to be similar to tags related to songs); however this improves with a larger extent for the random walk, and for larger dimensions. d=128 and l=80 seems to give better results.

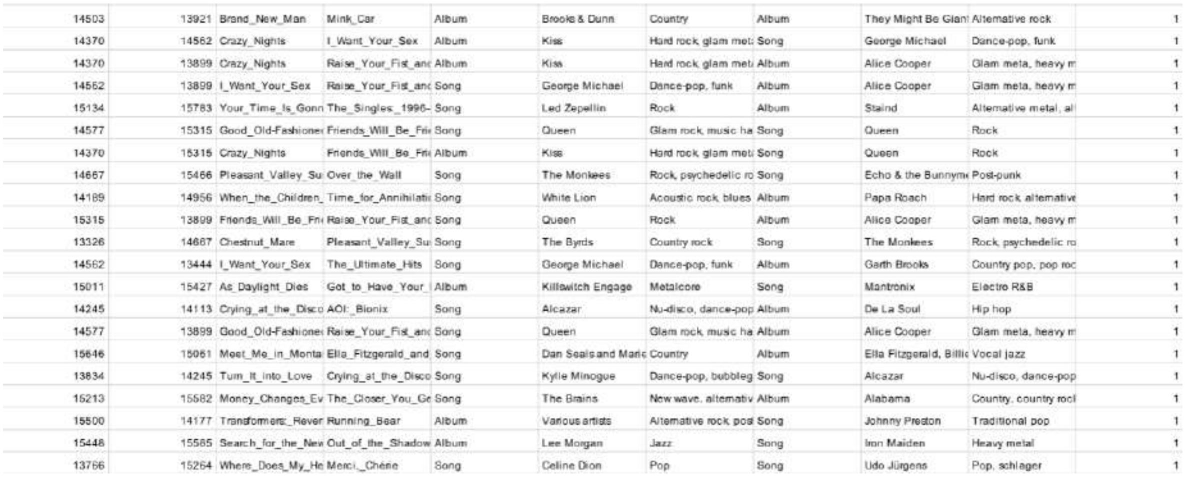
1. *Top 20 most similar nodes (Node2Vec d=4, l=3)*



item1	item2	Item 1 name	Item 2 name	entity type item1	artist	Genre	entity type item 2	artist	Genre	similarity
549	7804	Omar_Sharif	Starf..._Inc.	Historical figure			Song	Nine Inch Nails	Industrial metal, dru	1
549	8034	Omar_Sharif	Domino_Dancing	Historical figure			Song	Pet Shop Boys	Synthpop, freestyle,	1
549	8501	Omar_Sharif	Vol._2..._Hard_Knoc	Historical figure			Album	Jay-Z	Hip hop	1
549	8518	Omar_Sharif	City_of_Blinding_Lig	Historical figure			Song	U2	Rock	1
549	8722	Omar_Sharif	When_the_World_Kr	Historical figure			Album	Deacon Blue	Pop, rock, sophisti-pc	1
549	11627	Omar_Sharif	Maratha_Empire	Historical figure			Historical entity			1
549	13260	Omar_Sharif	Long_Cool_Woman_	Historical figure			Song	The Hollies	Swamp rock	1
549	14380	Omar_Sharif	Alea_Jacta_Est	Historical figure			Album	War Cry	Heavy metal, power	1
549	14659	Omar_Sharif	Te_Quiero	Historical figure			Song	Flex	Reggaeton	1
549	14791	Omar_Sharif	Anne_Murray_Duets:	Historical figure			Album	Anne Murray	Country	1
550	8136	Rudolf_Hess	Empty_Spaces	Historical figure			Song	Pink Floyd	Progressive rock	1
550	9241	Rudolf_Hess	Toyotomi_Hideyoshi	Historical figure			Historical Figure			1
550	11746	Rudolf_Hess	Bornu_Empire	Historical figure			Historical Entity			1
550	13493	Rudolf_Hess	The_Woodstock_Exp	Historical figure			Album	Various artists	Rock, psychedelic ro	1
1977	6376	Robert_Maxwell	Laos	Historical figure			Country			1
1977	7430	Robert_Maxwell	Rust_Never_Sleeps	Historical figure			Album	Neil Young	Acoustic, hard rock	1
6356	6587	Piece_of_Mind	In_and_Out_of_Con	Album	Iron Maiden	Heavy metal	Album	Robbie Williams	Pop rock, soft rock, d	1
6356	7349	Piece_of_Mind	Electric_Ladyland	Album	Iron Maiden	Heavy metal	Album	Jimmy Hendrix	Psychedelic rock, ha	1
6356	7622	Piece_of_Mind	Day-In_Day-Out	Album	Iron Maiden	Heavy metal	Song	David Bowie	R&B	1
7634	8008	Touch_the_Sky	The_Devil_and_Goc	Song	Kanye West	Hip hop	Album	Brand New	Alternative rock, emo	1

Figure 7.1: Top 20 most similar nodes (Node2Vec d=10, l=80)

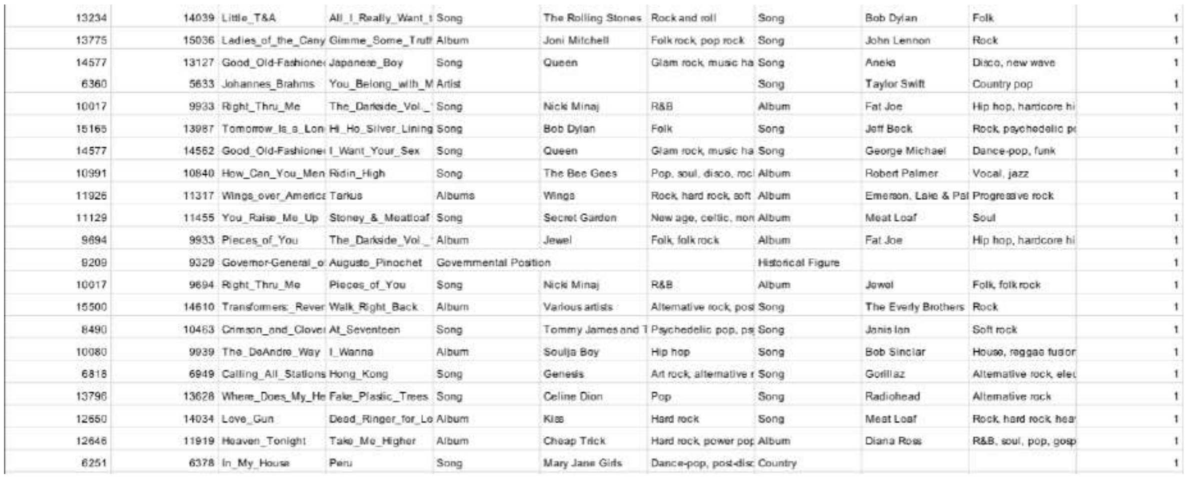
2. *Top 20 most similar nodes (Node2Vec d=128, l=80)*



14503	13921	Brand_New_Man	Mink_Car	Album	Brooks & Dunn	Country	Album	They Might Be Giant	Alternative rock	1
14370	14562	Crazy_Nights	I_Want_Your_Sex	Album	Kiss	Hard rock, glam meta	Song	George Michael	Dance-pop, funk	1
14370	13899	Crazy_Nights	Raise_Your_Fist_anc	Album	Kiss	Hard rock, glam meta	Album	Alice Cooper	Glam meta, heavy m	1
14562	13899	I_Want_Your_Sex	Raise_Your_Fist_anc	Song	George Michael	Dance-pop, funk	Album	Alice Cooper	Glam meta, heavy m	1
15134	15783	Your_Time_Is_Gonn	The_Singles_1996-	Song	Led Zepellin	Rock	Album	Staind	Alternative metal, al	1
14577	15315	Good_Old-Fashione	Friends_Will_Be_Fri	Song	Queen	Glam rock, music ha	Song	Queen	Rock	1
14370	15315	Crazy_Nights	Friends_Will_Be_Fri	Album	Kiss	Hard rock, glam meta	Song	Queen	Rock	1
14667	15466	Pleasant_Valley_Su	Over_the_Wall	Song	The Monkees	Rock, psychedelic ro	Song	Echo & the Bunnym	Post-punk	1
14189	14956	When_the_Children_	Time_for_Annihilatic	Song	White Lion	Acoustic rock, blues	Album	Papa Roach	Hard rock, alternative	1
15315	13899	Friends_Will_Be_Fri	Raise_Your_Fist_anc	Song	Queen	Rock	Album	Alice Cooper	Glam meta, heavy m	1
13326	14667	Chestnut_Mare	Pleasant_Valley_Su	Song	The Byrds	Country rock	Song	The Monkees	Rock, psychedelic ro	1
14562	13444	I_Want_Your_Sex	The_Ultimate_Hits	Song	George Michael	Dance-pop, funk	Album	Garth Brooks	Country pop, pop roc	1
15011	15427	As_Daylight_Dies	Got_to_Have_Your_	Album	Killswitch Engage	Metalcore	Song	Mantronix	Electro R&B	1
14245	14113	Crying_at_the_Disco	AOI:_Bionix	Song	Alcazar	Nu-disco, dance-pop	Album	De La Soul	Hip hop	1
14577	13899	Good_Old-Fashione	Raise_Your_Fist_anc	Song	Queen	Glam rock, music ha	Album	Alice Cooper	Glam meta, heavy m	1
15646	15061	Meet_Me_in_Monta	Ella_Fitzgerald_and	Song	Dan Seals and Maric	Country	Album	Ella Fitzgerald, Billic	Vocal jazz	1
13834	14245	Turn_It_into_Love	Crying_at_the_Disco	Song	Kylie Minogue	Dance-pop, bubbleg	Song	Alcazar	Nu-disco, dance-pop	1
15213	15582	Money_Changes_Ev	The_Closer_You_Ge	Song	The Brains	New wave, alternativ	Album	Alabama	Country, country rocl	1
15500	14177	Transformers:_Rever	Running_Bear	Album	Various artists	Alternative rock, post	Song	Johnny Preston	Traditional pop	1
15448	15585	Search_for_the_New	Out_of_the_Shadow	Album	Lee Morgan	Jazz	Song	Iron Maiden	Heavy metal	1
13766	15264	Where_Does_My_He	Merci,_Chérie	Song	Celine Dion	Pop	Song	Udo Jürgens	Pop, schlager	1

Figure 7.2: Top 20 most similar nodes (Node2Vec d=128, l=80)

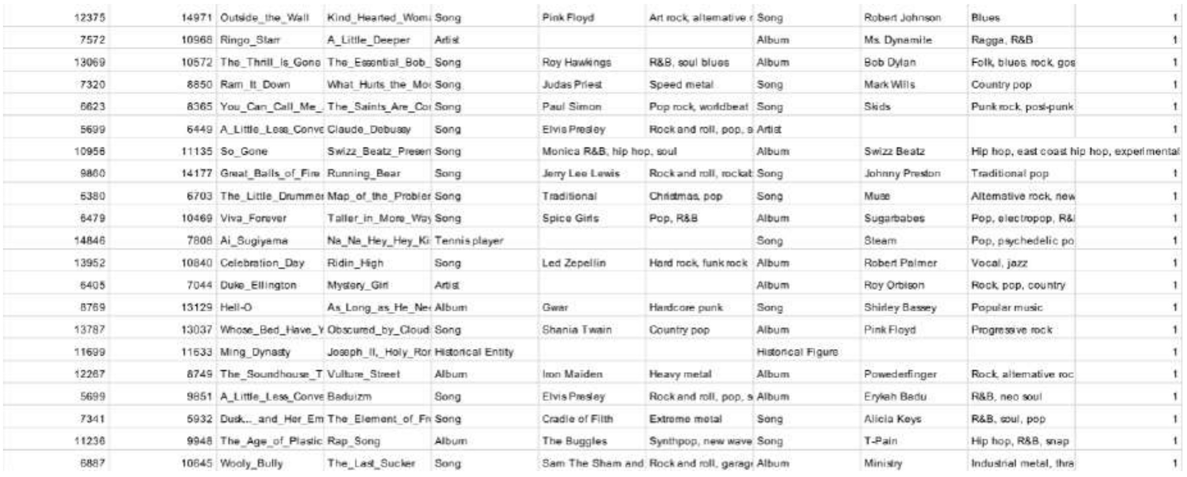
3. *Top 20 most similar nodes (Node2Vec d=4, l=80*



13234	14039	Little_T&A	All_I_Really_Want_t	Song	The Rolling Stones	Rock and roll	Song	Bob Dylan	Folk	1
13775	15036	Ladies_of_the_Cany	Gimme_Some_Truth	Album	Joni Mitchell	Folk rock, pop rock	Song	John Lennon	Rock	1
14577	13127	Good_Old-Fashione	Japanese_Boy	Song	Queen	Glam rock, music ha	Song	Aneka	Disco, new wave	1
6360	5633	Johannes_Brahms	You_Belong_with_M	Artist			Song	Taylor Swift	Country pop	1
10017	9933	Right_Thru_Me	The_Darkside_Vol._	Song	Nicki Minaj	R&B	Album	Fat Joe	Hip hop, hardcore hi	1
15165	13987	Tomorrow_Is_a_Lon	Hi_Ho_Silver_Lining	Song	Bob Dylan	Folk	Song	Jeff Beck	Rock, psychedelic p	1
14577	14562	Good_Old-Fashione	I_Want_Your_Sex	Song	Queen	Glam rock, music ha	Song	George Michael	Dance-pop, funk	1
10991	10840	How_Can_You_Men	Ridin_High	Song	The Bee Gees	Pop, soul, disco, roc	Album	Robert Palmer	Vocal, jazz	1
11926	11317	Wings_over_America	Tarkus	Albums	Wings	Rock, hard rock, soft	Album	Emerson, Lake & Pal	Progressive rock	1
11129	11455	You_Raise_Me_Up	Stoney_&_Meatloaf	Song	Secret Garden	New age, celtic, non	Album	Meat Loaf	Soul	1
9694	9933	Pieces_of_You	The_Darkside_Vol_	Album	Jewel	Folk, folk rock	Album	Fat Joe	Hip hop, hardcore hi	1
9209	9329	Governor-General_o	Augusto_Pinochet	Governmental Position			Historical Figure			1
10017	9694	Right_Thru_Me	Pieces_of_You	Song	Nicki Minaj	R&B	Album	Jewel	Folk, folk rock	1
15500	14610	Transformers:_Rever	Walk_Right_Back	Album	Various artists	Alternative rock, post	Song	The Everly Brothers	Rock	1
8490	10463	Crimson_and_Clover	At_Seventeen	Song	Tommy James and T	Psychedelic pop, ps	Song	Janis Ian	Soft rock	1
10080	9939	The_DeAndre_Way	I_Wanna	Album	Soulja Boy	Hip hop	Song	Bob Sinclar	House, reggae fusion	1
6818	6949	Calling_All_Stations	Hong_Kong	Song	Genesis	Art rock, alternative r	Song	Gorillaz	Alternative rock, ele	1
13796	13628	Where_Does_My_He	Fake_Plastic_Trees	Song	Celine Dion	Pop	Song	Radiohead	Alternative rock	1
12650	14034	Love_Gun	Dead_Ringer_for_Lo	Album	Kiss	Hard rock	Song	Meat Loaf	Rock, hard rock, hea	1
12646	11919	Heaven_Tonight	Take_Me_Higher	Album	Cheap Trick	Hard rock, power pop	Album	Diana Ross	R&B, soul, pop, gosp	1
6251	6378	In_My_House	Peru	Song	Mary Jane Girls	Dance-pop, post-disc	Country			1

Figure 7.3: Top 20 most similar nodes (Node2Vec d=4, l=80)

4. *Top 20 most similar nodes (Node2Vec d=4, l=3)*



12375	14971	Outside_the_Wall	Kind_Hearted_Wom	Song	Pink Floyd	Art rock, alternative r	Song	Robert Johnson	Blues	1
7572	10968	Ringo_Starr	A_Little_Deeper	Artist			Album	Ms. Dynamite	Ragga, R&B	1
13069	10572	The_Thrill_Is_Gone	The_Essential_Bob_	Song	Roy Hawkings	R&B, soul blues	Album	Bob Dylan	Folk, blues, rock, gos	1
7320	8850	Ram_It_Down	What_Hurts_the_Mo	Song	Judas Priest	Speed metal	Song	Mark Wills	Country pop	1
6623	8365	You_Can_Call_Me_	The_Saints_Are_Co	Song	Paul Simon	Pop rock, worldbeat	Song	Skids	Punk rock, post-punk	1
5699	6449	A_Little_Less_Conve	Claude_Debussy	Song	Elvis Presley	Rock and roll, pop, s	Artist			1
10956	11135	So_Gone	Swizz_Beatz_Presen	Song	Monica R&B, hip hop, soul		Album	Swizz Beatz	Hip hop, east coast hip hop, experimental	
9860	14177	Great_Balls_of_Fire	Running_Bear	Song	Jerry Lee Lewis	Rock and roll, rockab	Song	Johnny Preston	Traditional pop	1
6380	6703	The_Little_Drummer	Map_of_the_Probler	Song	Traditional	Christmas, pop	Song	Muse	Alternative rock, new	1
6479	10469	Viva_Forever	Taller_in_More_Way	Song	Spice Girls	Pop, R&B	Album	Sugarbabes	Pop, electropop, R&	1
14846	7808	Ai_Sugiyama	Na_Na_Hey_Hey_Ki	Tennis player			Song	Steam	Pop, psychedelic po	1
13952	10840	Celebration_Day	Ridin_High	Song	Led Zepellin	Hard rock, funk rock	Album	Robert Palmer	Vocal, jazz	1
6405	7044	Duke_Ellington	Mystery_Girl	Artist			Album	Roy Orbison	Rock, pop, country	1
8769	13129	Hell-O	As_Long_as_He_Ne	Album	Gwar	Hardcore punk	Song	Shirley Bassey	Popular music	1
13787	13037	Whose_Bed_Have_Y	Obscured_by_Cloud	Song	Shania Twain	Country pop	Album	Pink Floyd	Progressive rock	1
11699	11633	Ming_Dynasty	Joseph_II,_Holy_Ror	Historical Entity			Historical Figure			1
12267	8749	The_Soundhouse_T	Vulture_Street	Album	Iron Maiden	Heavy metal	Album	Powederfinger	Rock, alternative roc	1
5699	9851	A_Little_Less_Conve	Baduizm	Song	Elvis Presley	Rock and roll, pop, s	Album	Erykah Badu	R&B, neo soul	1
7341	5932	Dusk..._and_Her_Em	The_Element_of_Fr	Song	Cradle of Filth	Extreme metal	Song	Alicia Keys	R&B, soul, pop	1
11236	9948	The_Age_of_Plastic	Rap_Song	Album	The Buggles	Synthpop, new wave	Song	T-Pain	Hip hop, R&B, snap	1
6887	10645	Wooly_Bully	The_Last_Sucker	Song	Sam The Sham and	Rock and roll, garag	Album	Ministry	Industrial metal, thra	1

Figure 7.4: Top 20 most similar nodes (Node2Vec d=4, l=3)

Visualizations:

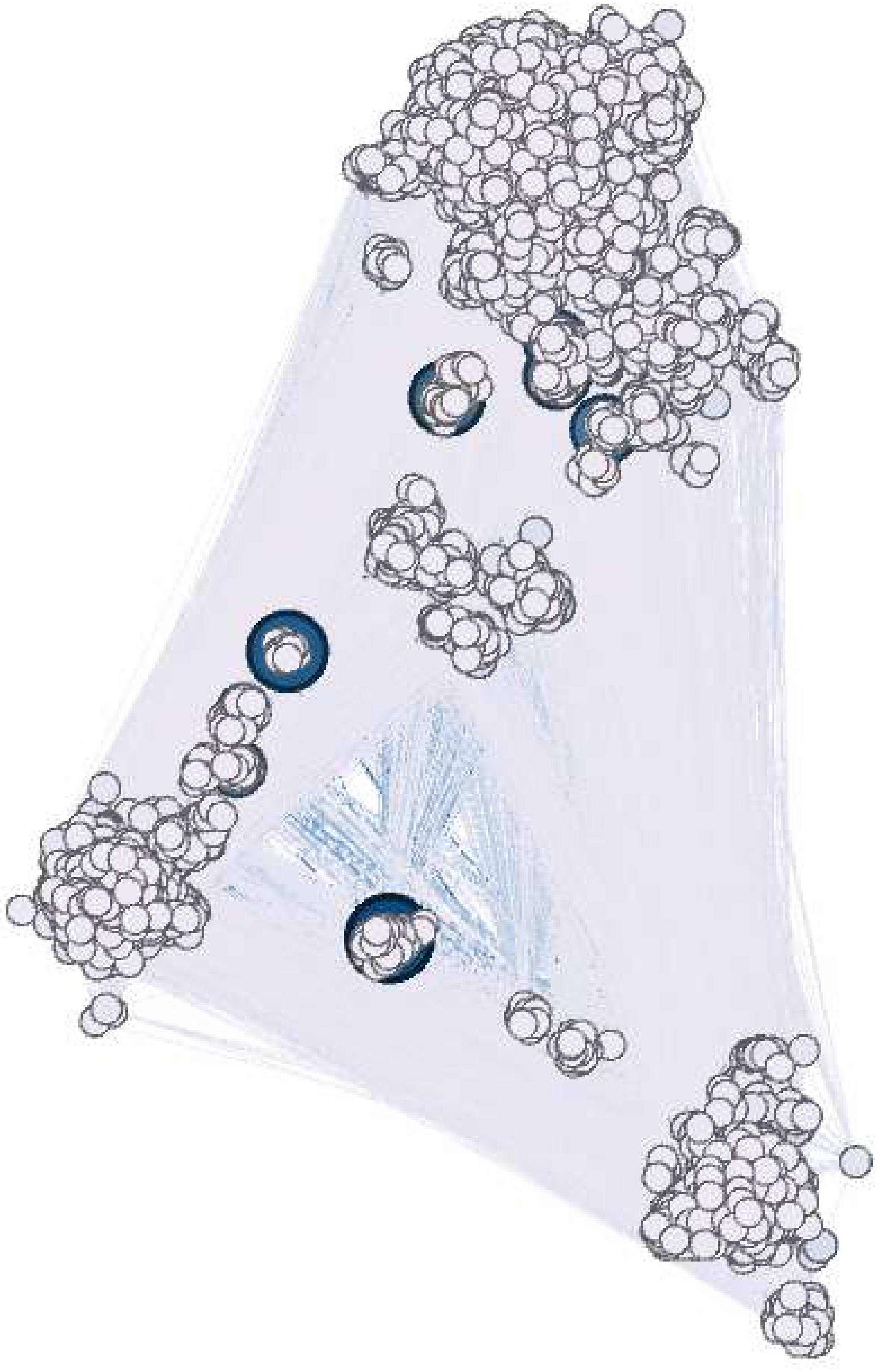


Figure 7.5: Visualization of SF1 LDBC dataset

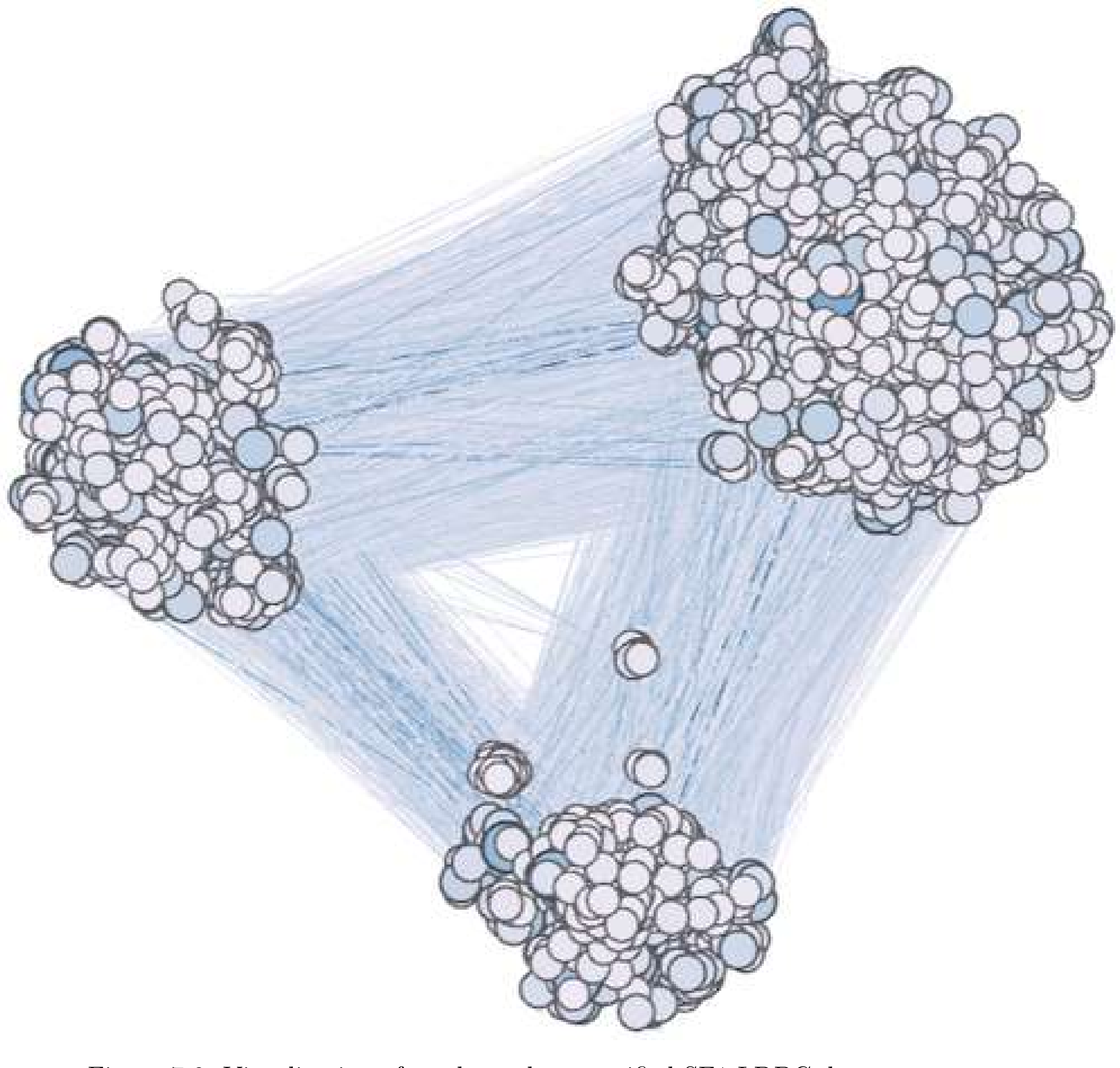


Figure 7.6: Visualization of random edge sparsified SF1 LDBC dataset

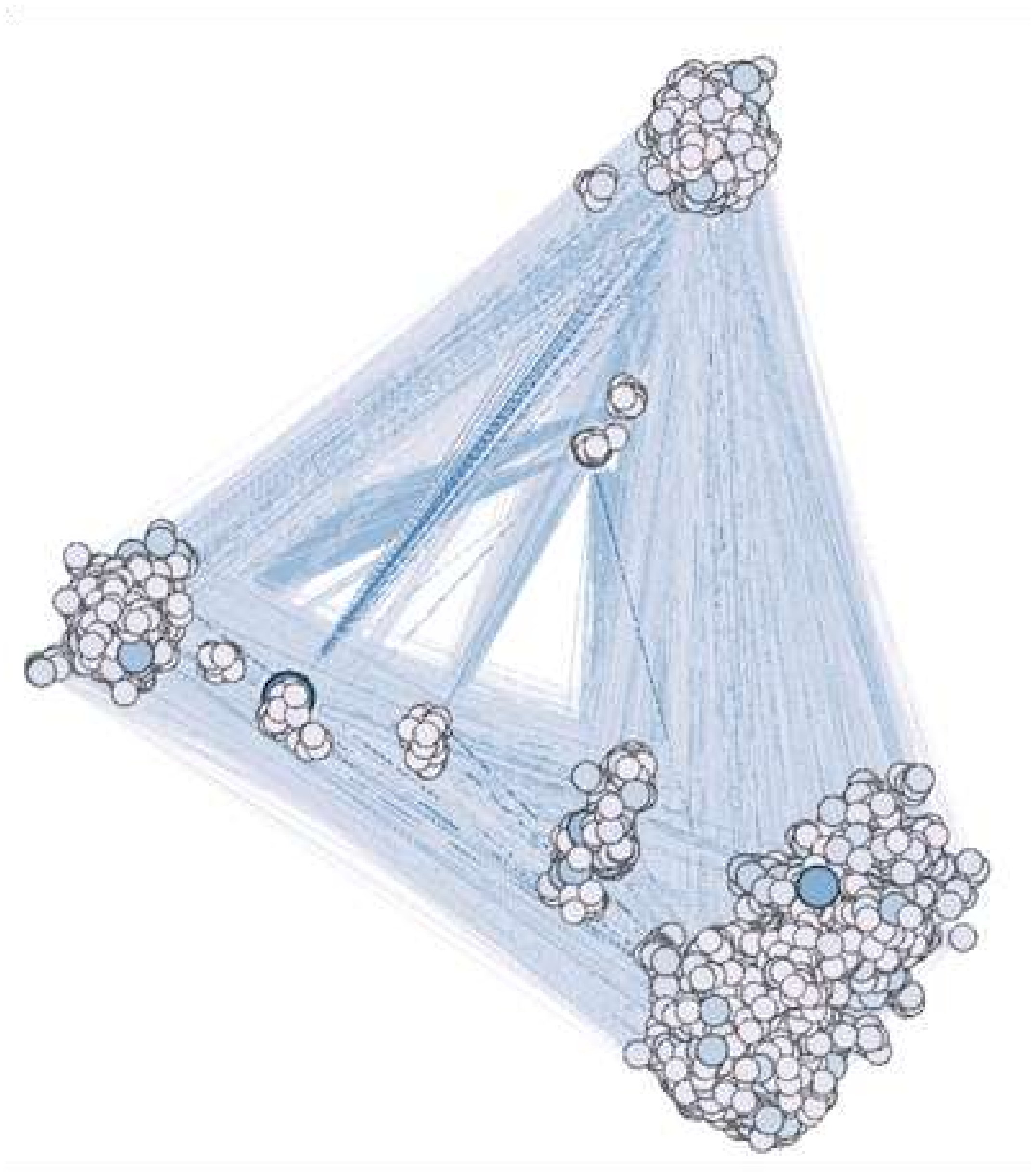


Figure 7.7: Visualization of SF1 algebraic distance sparsified LDBC dataset

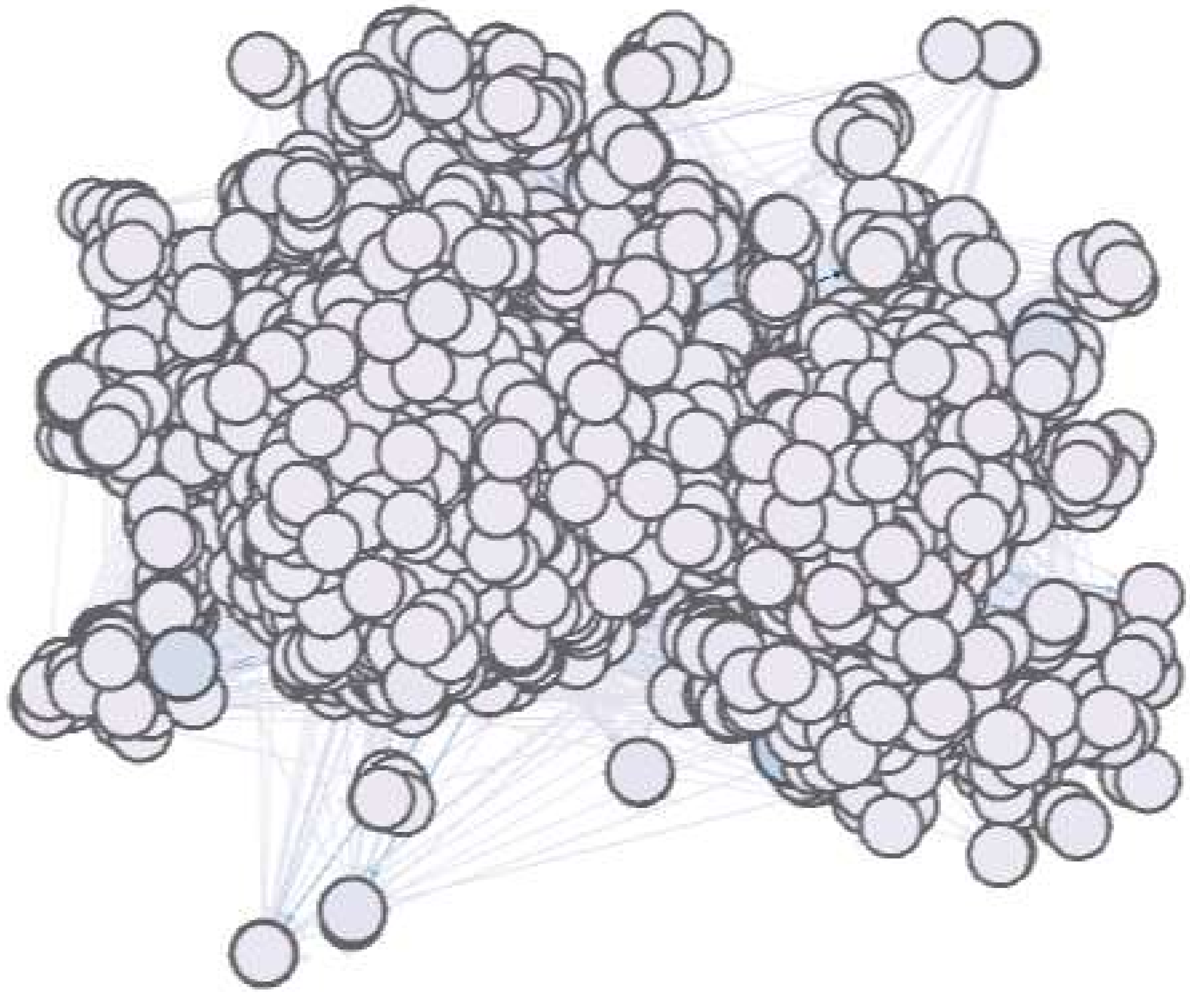


Figure 7.8: Visualization of SF1 local degree sparsified LDBC dataset

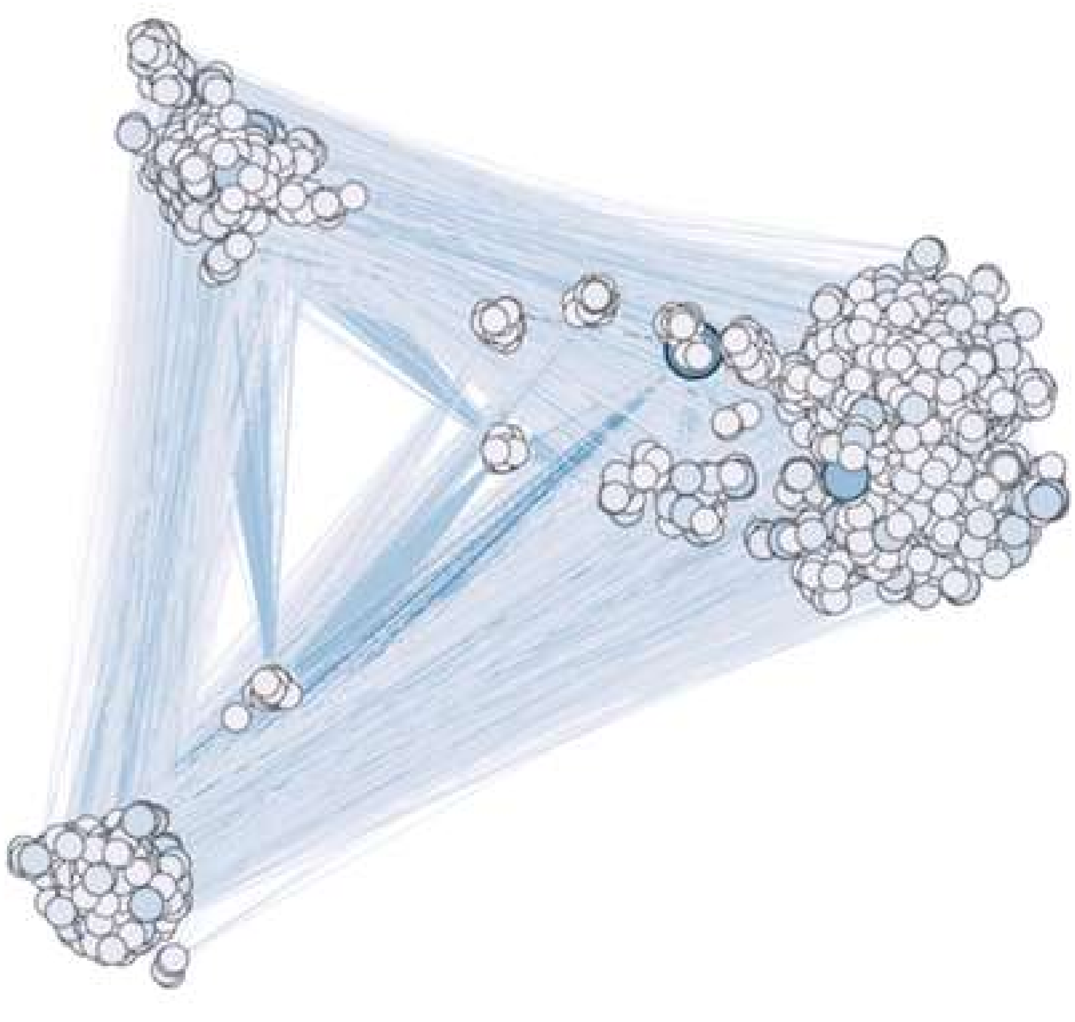


Figure 7.9: Visualization of SF1 triangle sparsified LDBC dataset

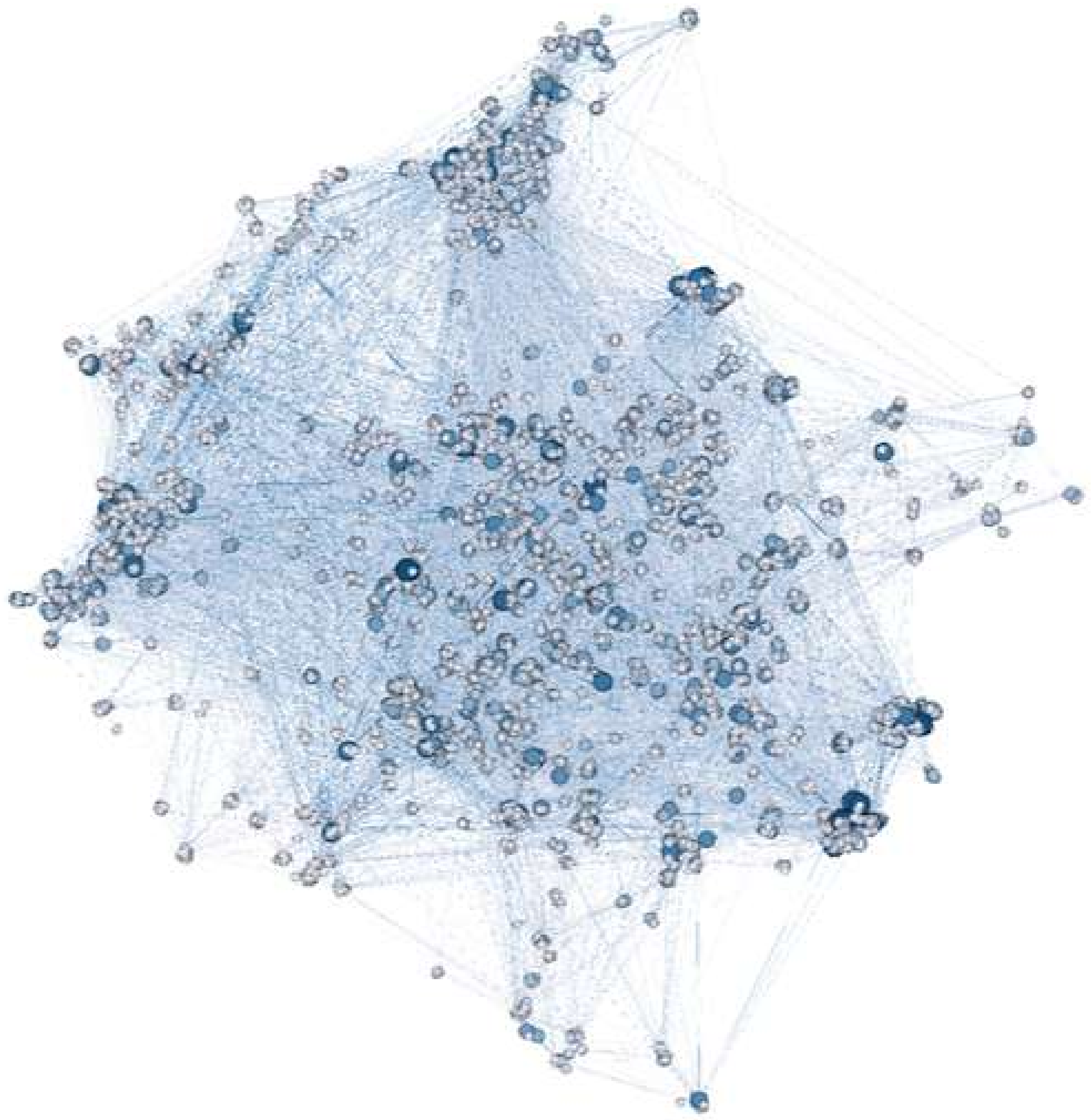


Figure 7.10: Visualization of SF1 local similarity sparsified LDBC dataset

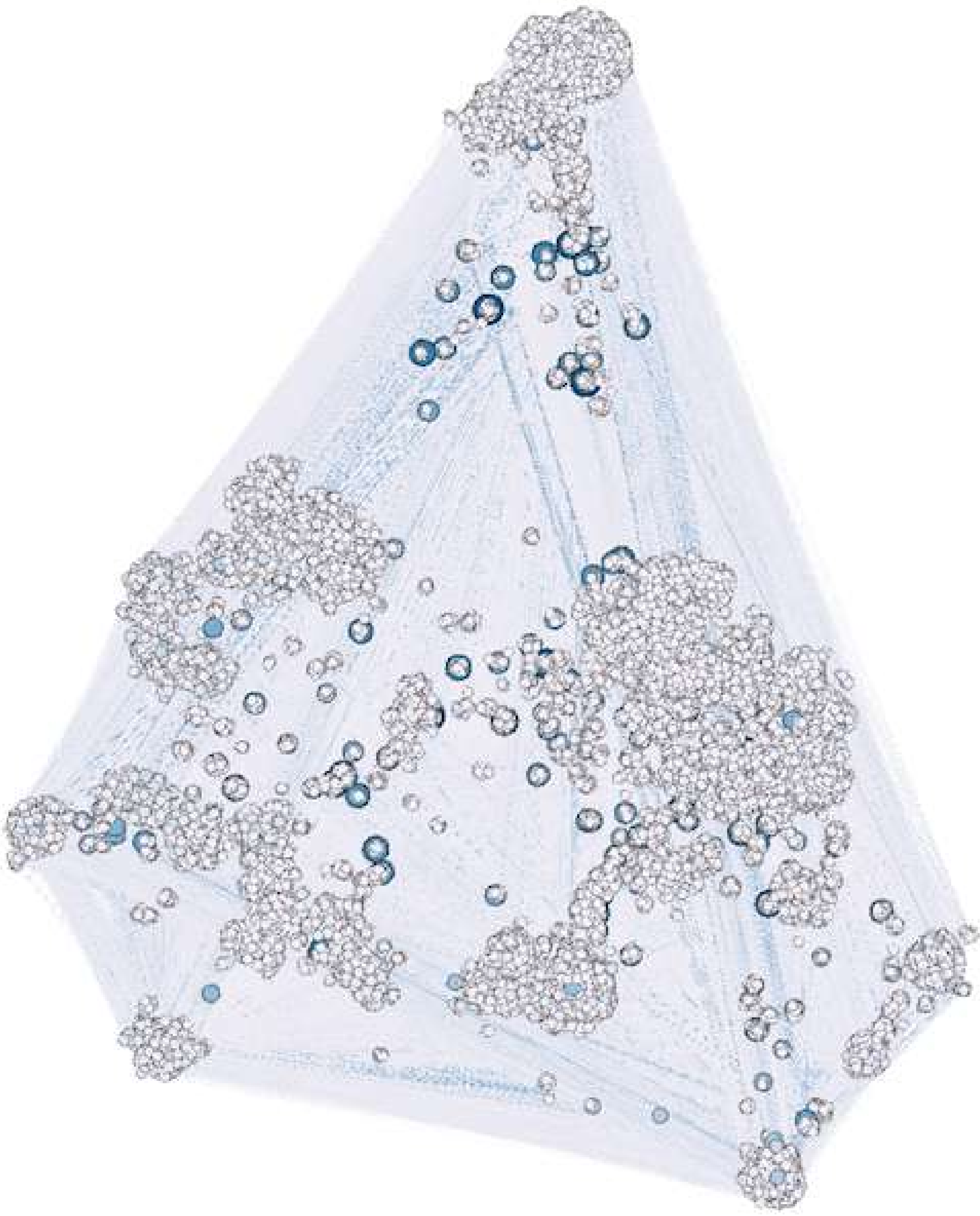


Figure 7.11: Visualization of SF10 LDBC dataset

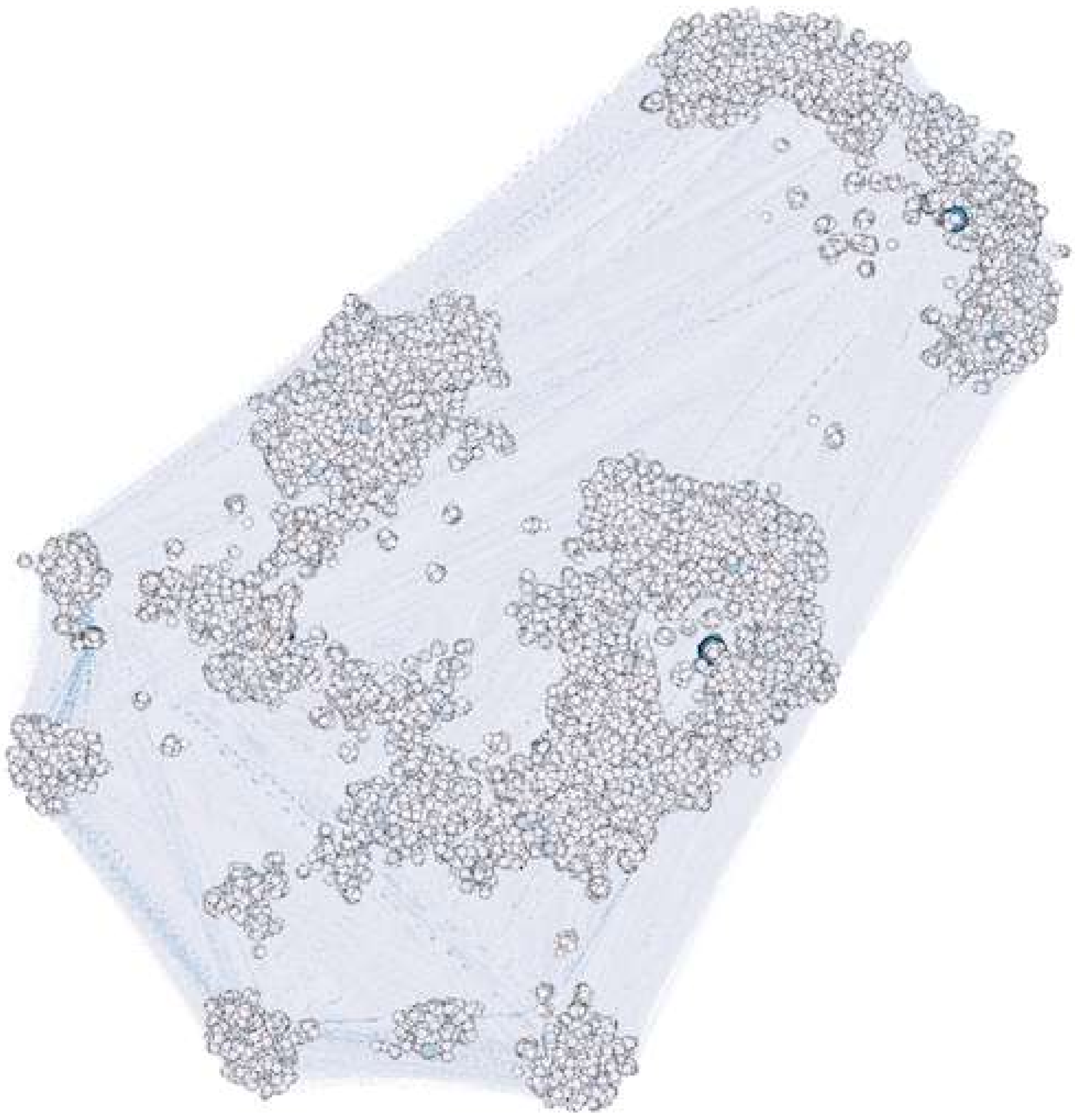


Figure 7.12: Visualization of SF10 random edge sparsified LDBC dataset

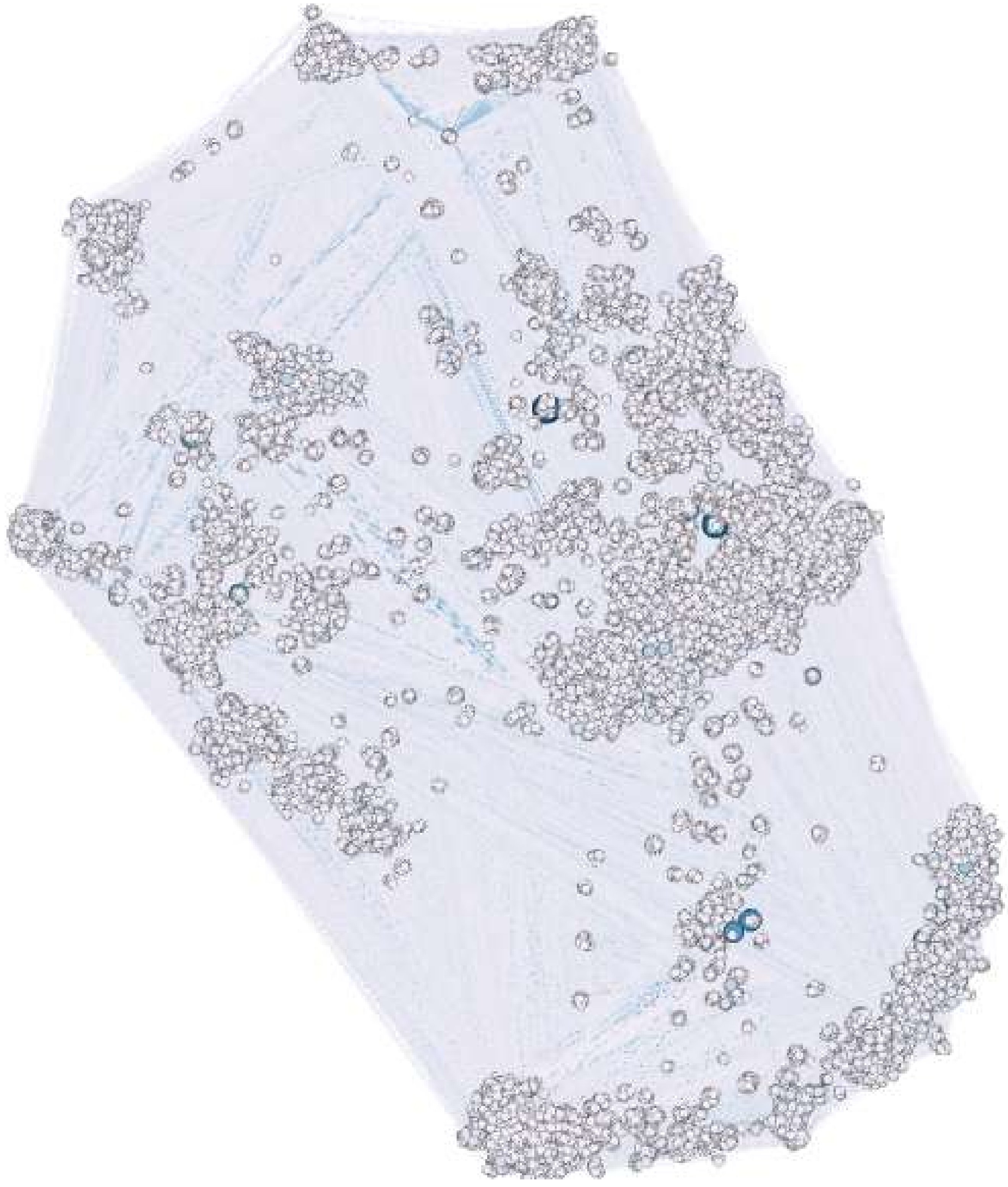


Figure 7.13: Visualization of SF10 algebraic distance sparsified LDBC dataset

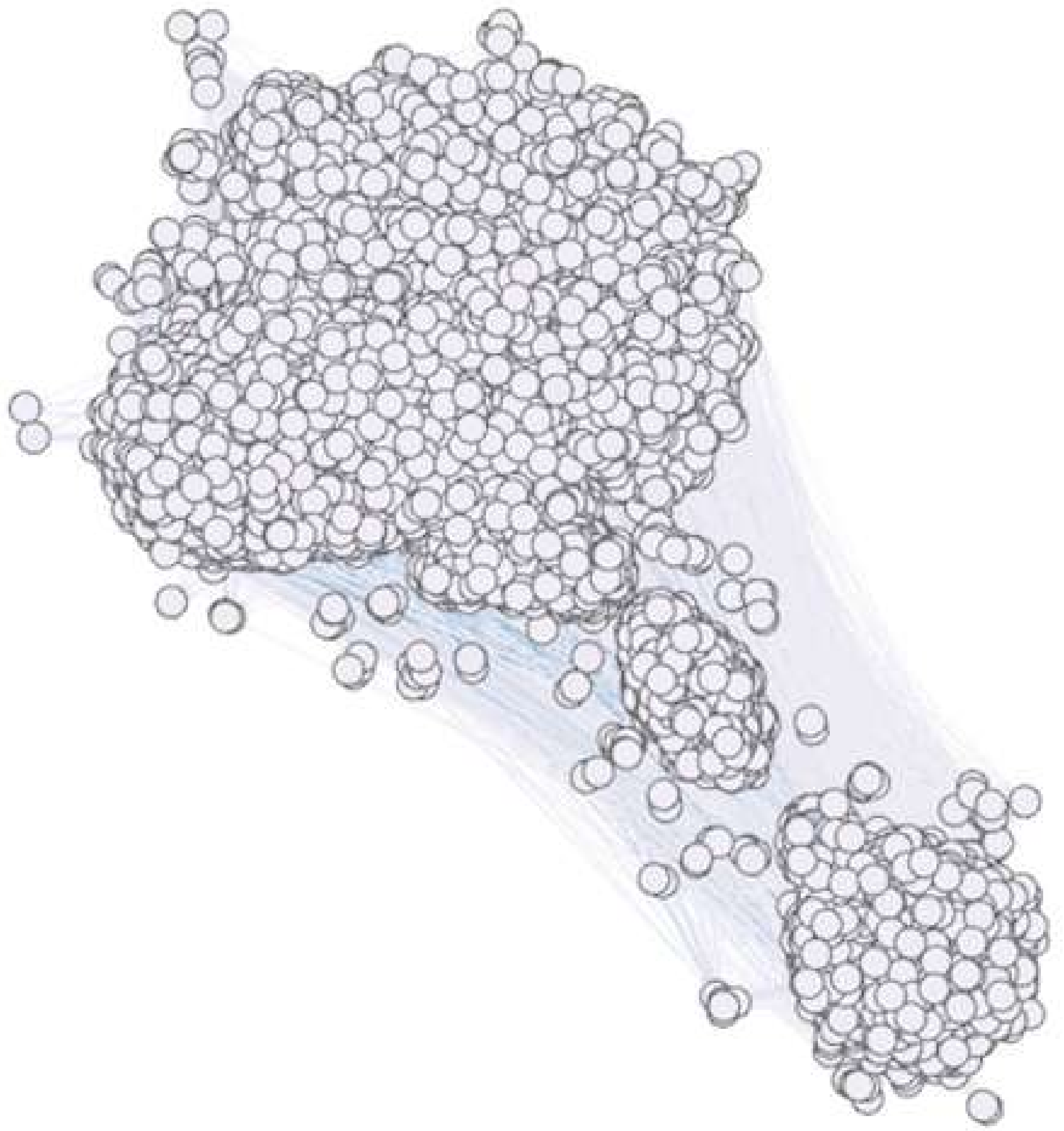


Figure 7.14: Visualization of SF10 local degree sparsified LDBC dataset

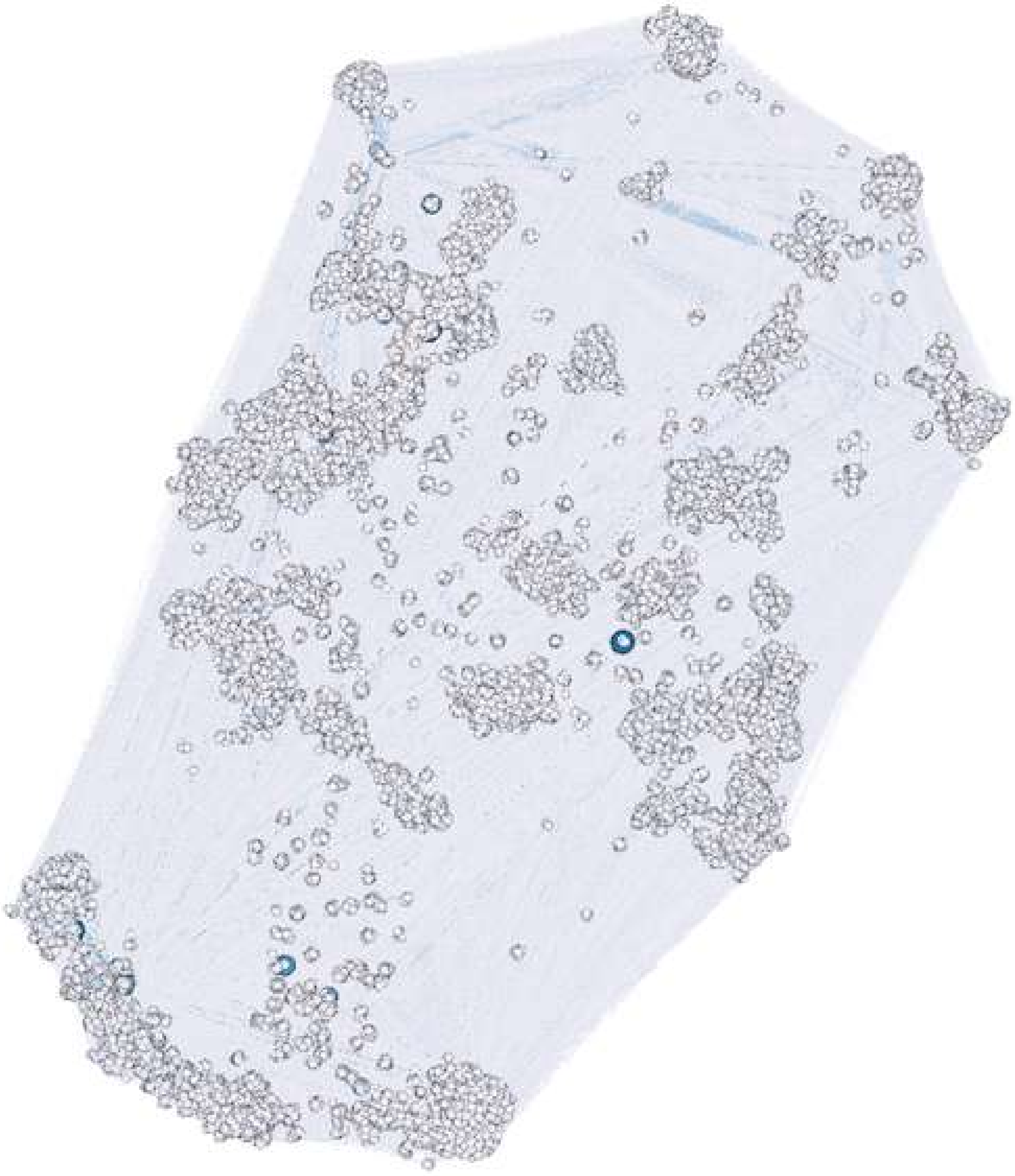


Figure 7.15: Visualization of SF10 triangle sparsified LDBC dataset

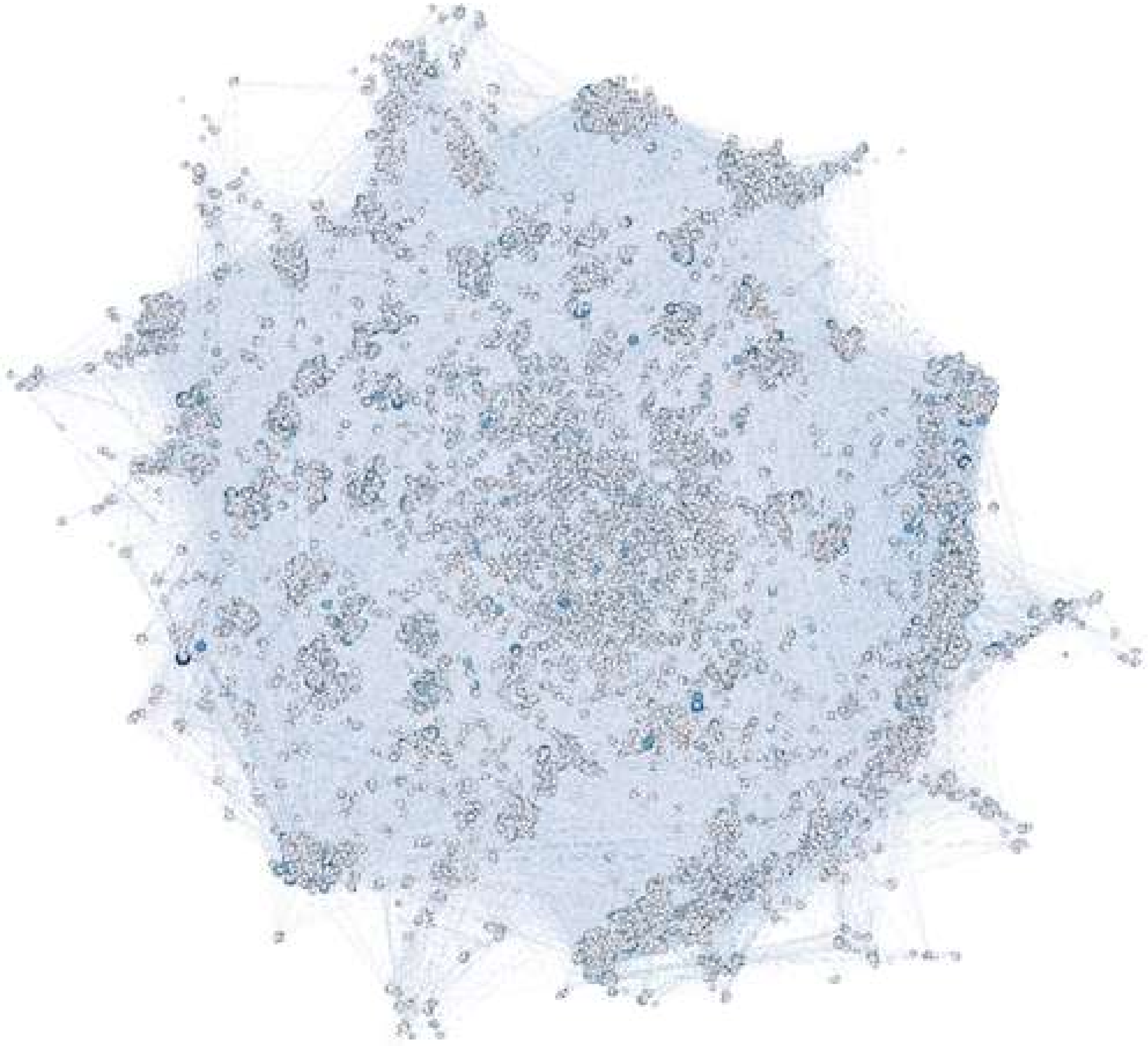


Figure 7.16: Visualization of SF10 local similarity sparsified LDBC dataset

Bibliography

[AAB$^+$17] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)*, page 68, 2017. (cited on Page 14 and 23)

[Ach03] Dimitris Achlioptas. Database-friendly random projections: Johnson-lindenstrauss with binary coins. *Journal of computer and System Sciences*, pages 671–687, 2003. (cited on Page 15)

[AFK01] James Abello, Irene Finocchi, and Jeffrey Korn. Graph sketches. *Proceedings of the IEEE Symposiom on Information Visualization, San Diego, CA*, 2001. (cited on Page 15)

[AG08] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, page 1, 2008. (cited on Page 14)

[AGM12a] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 459–467, 2012. (cited on Page 15 and 39)

[AGM12b] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, pages 5–14, 2012. (cited on Page 14 and 39)

[Ahn13] Sebastian E Ahnert. Power graph compression reveals dominant relationships in genetic transcription networks. *Molecular BioSystems*, pages 2681–2685, 2013. (cited on Page 39)

[B$^+$16] Albert-László Barabási et al. *Network science*. Cambridge university press, 2016. (cited on Page 7, 14, 19, and 20)

[BBC$^+$11] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering, IEEE*, pages 31–39, 2011. (cited on Page 46)

[BBP05] Christian Borgelt, Michael R Berthold, and David E Patterson. Molecular fragment mining for drug discovery. In *European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 1002–1013, 2005. (cited on Page 19)

[BGL11] Albert-László Barabási, Natali Gulbahce, and Joseph Loscalzo. Network medicine: a network-based approach to human disease. *Nature reviews genetics*, page 56, 2011. (cited on Page 19)

[BHN$^+$02] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and Shashank Sudarshan. Keyword searching and browsing in databases using banks. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 431–440, 2002. (cited on Page 23)

[BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal xml pattern matching. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 310–321, 2002. (cited on Page 23)

[BN03] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural computation*, pages 1373–1396, 2003. (cited on Page 17 and 32)

[BO04] Albert-Laszlo Barabasi and Zoltan N Oltvai. Network biology: understanding the cell's functional organization. *Nature reviews genetics*, page 101, 2004. (cited on Page 19)

[BSST13] Joshua Batson, Daniel A Spielman, Nikhil Srivastava, and Shang-Hua Teng. Spectral sparsification of graphs: theory and algorithms. *Communications of the ACM*, pages 87–94, 2013. (cited on Page 8, 25, and 40)

[BV04] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602, 2004. (cited on Page 39)

[ČGM15] Šejla Čebirić, François Goasdoué, and Ioana Manolescu. Query-oriented summarization of rdf graphs. *Proceedings of the VLDB Endowment*, pages 2012–2015, 2015. (cited on Page 40)

[CN85] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, pages 210–223, 1985. (cited on Page 16 and 29)

[Cor11] Graham Cormode. Sketch techniques for approximate query processing. *Foundations and Trends in Databases. NOW publishers*, 2011. (cited on Page 15)

[CS11] Jie Chen and Ilya Safro. Algebraic distance on graphs. *SIAM Journal on Scientific Computing*, pages 3468–3490, 2011. (cited on Page 16 and 27)

[CWPZ18] Peng Cui, Xiao Wang, Jian Pei, and Wenwu Zhu. A survey on network embedding. *IEEE Transactions on Knowledge and Data Engineering*, 2018. (cited on Page 18)

[CYD$^+$08] Jiefeng Cheng, Jeffrey Xu Yu, Bolin Ding, S Yu Philip, and Haixun Wang. Fast graph pattern matching. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 913–922, 2008. (cited on Page 23)

[CZC18] Hongyun Cai, Vincent W Zheng, and Kevin Chang. A comprehensive survey of graph embedding: problems, techniques and applications. *IEEE Transactions on Knowledge and Data Engineering*, 2018. (cited on Page 18)

[DKM06] Petros Drineas, Ravi Kannan, and Michael W Mahoney. Fast monte carlo algorithms for matrices iii: Computing a compressed approximate matrix decomposition. *SIAM Journal on Computing*, pages 184–206, 2006. (cited on Page 40)

[FLWW12] Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. Query preserving graph compression. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 157–168, 2012. (cited on Page 40)

[Gal06] Brian Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. *AAAI FS*, pages 45–53, 2006. (cited on Page 23)

[GF18] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, pages 78–94, 2018. (cited on Page 18, 31, 32, 33, 49, and 68)

[GL16] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016. (cited on Page 17, 35, 37, 38, 40, and 68)

[Hop07] Andrew L Hopkins. Network pharmacology. *Nature biotechnology*, page 1110, 2007. (cited on Page 19)

[HPC$^+$18] Sanghyun Hong, Noseong Park, Tanmoy Chakraborty, Hyunjoong Kang, and Soonhyun Kwon. Page: Answering graph pattern queries via knowledge graph embedding. In *International Conference on Big Data*, pages 87–99, 2018. (cited on Page 31 and 40)

[HST13] Nasrin Hassanlou, Maryam Shoaran, and Alex Thomo. Probabilistic graph summarization. In *International Conference on Web-Age Information Management*, pages 545–556, 2013. (cited on Page 40)

[HYL17] William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *arXiv preprint*, 2017. (cited on Page 17)

[IR15] Emil Eifrem Ian Robinson, Jim Webber. Graph databases, 2nd edition new opportunities for connected data. chapter 6. O'Reilly Media, 2015. (cited on Page v, 21, and 22)

[JS16] Emmanuel John and Ilya Safro. Single-and multi-level network sparsification by algebraic distance. *Journal of Complex Networks*, pages 352–388, 2016. (cited on Page 16 and 28)

[KK95] George Karypis and Vipin Kumar. Metis–unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995. (cited on Page 46)

[KK00] George Karypis and Vipin Kumar. Multilevel k-way hypergraph partitioning. *VLSI design*, pages 285–300, 2000. (cited on Page 39)

[KLM+17] Michael Kapralov, Yin Tat Lee, CN Musco, CP Musco, and Aaron Sidford. Single pass spectral sparsification in dynamic streams. *SIAM Journal on Computing*, pages 456–477, 2017. (cited on Page 16 and 40)

[KS08] Arne Koopman and Arno Siebes. Discovering relational item sets efficiently. In *Proceedings of the 2008 SIAM International Conference on Data Mining*, pages 108–119, 2008. (cited on Page 40)

[KW14] Michael Kapralov and David Woodruff. Spanners and sparsifiers in dynamic streams. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 272–281, 2014. (cited on Page 16)

[KWY12] Arijit Khan, Yinghui Wu, and Xifeng Yan. Emerging graph queries in linked data. In *2012 IEEE 28th International Conference on Data Engineering*, pages 1218–1221, 2012. (cited on Page 22 and 23)

[KYW10] Arijit Khan, Xifeng Yan, and Kun-Lung Wu. Towards proximity pattern mining in large graphs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 867–878, 2010. (cited on Page 23)

[LKF05] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 177–187, 2005. (cited on Page 39)

[LLZW11] Jianxin Li, Chengfei Liu, Rui Zhou, and Wei Wang. Top-k keyword search over probabilistic xml data. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 673–684, 2011. (cited on Page 23)

[LNHD11] Dijun Luo, Feiping Nie, Heng Huang, and Chris H Ding. Cauchy graph embedding. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 553–560, 2011. (cited on Page 31)

[LSDK18] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. Graph summarization methods and applications: A survey. *ACM Computing Surveys (CSUR)*, page 62, 2018. (cited on Page 8, 15, 25, and 39)

[LSH+15] Gerd Lindner, Christian L Staudt, Michael Hamann, Henning Meyerhenke, and Dorothea Wagner. Structure-preserving sparsification of social networks. In *Advances in Social Networks Analysis and Mining (ASONAM), 2015 IEEE/ACM International Conference on*, pages 448–454, 2015. (cited on Page v, 16, 25, 27, and 28)

[LY13] Shou-De Lin and Mi-Yen Yeh. Cheng-te li. 2013. sampling and summarization for social networks. In *Proceedings of the 17th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'13)*, 2013. (cited on Page 39)

[MBC+11] Michael Mathioudakis, Francesco Bonchi, Carlos Castillo, Aristides Gionis, and Antti Ukkonen. Sparsification of influence networks. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 529–537, 2011. (cited on Page 40)

[McG14] Andrew McGregor. Graph stream algorithms: a survey. *ACM SIGMOD Record*, pages 9–20, 2014. (cited on Page 15 and 39)

[McG17] Andrew McGregor. Graph sketching and streaming: New approaches for analyzing massive graphs. In *International Computer Science Symposium in Russia*, pages 20–24, 2017. (cited on Page 15)

[MGF11] Koji Maruhashi, Fan Guo, and Christos Faloutsos. Multiaspectforensics: Pattern mining on large-scale heterogeneous networks with tensor analysis. In *Advances in Social Networks Analysis and Mining (ASONAM), 2011 International Conference on*, pages 203–210, 2011. (cited on Page 40)

[MP10] Hossein Maserrat and Jian Pei. Neighbor query friendly compression of social networks. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 533–542, 2010. (cited on Page 39)

[MTVV15] Andrew McGregor, David Tench, Sofya Vorotnikova, and Hoa T Vu. Densest subgraph in dynamic graph streams. In *International Symposium on*

Mathematical Foundations of Computer Science, pages 472–482, 2015. (cited on Page 15)

[NG04] Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review E*, page 026113, 2004. (cited on Page 39)

[NHH$^+$18] Andriy Nikolov, Peter Haase, Daniel M Herzig, Johannes Trame, and Artem Kozlov. Combining rdf graph data and embedding models for an augmented knowledge graph. In *Companion of the The Web Conference 2018 on The Web Conference 2018*, pages 977–980, 2018. (cited on Page 40)

[OB14] Mark Ortmann and Ulrik Brandes. Triangle listing algorithms: Back from the diversion. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 1–8, 2014. (cited on Page 16 and 29)

[OCP$^+$16] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. Asymmetric transitivity preserving graph embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1105–1114, 2016. (cited on Page 17, 31, and 34)

[PARS14] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014. (cited on Page 17, 32, and 40)

[QLZJ16] Qiang Qu, Siyuan Liu, Feida Zhu, and Christian S Jensen. Efficient online summarization of large-scale dynamic networks. *IEEE Transactions on Knowledge and Data Engineering*, pages 3231–3245, 2016. (cited on Page 15)

[RGSB17] Matteo Riondato, David García-Soriano, and Francesco Bonchi. Graph summarization with quality guarantees. *Data mining and knowledge discovery*, pages 314–349, 2017. (cited on Page 39)

[RJH] Ning Ruan, Ruoming Jin, and Yan Huang. Distance preserving graph simplification. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 1200–1205. (cited on Page 25)

[RS00] Sam T Roweis and Lawrence K Saul. Nonlinear dimensionality reduction by locally linear embedding. *science*, pages 2323–2326, 2000. (cited on Page 17 and 38)

[RSF17] Leonardo FR Ribeiro, Pedro HP Saverese, and Daniel R Figueiredo. struc2vec: Learning node representations from structural identity. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 385–394, 2017. (cited on Page 17)

[RWE13] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases.* " O'Reilly Media, Inc.", 2013. (cited on Page 14)

[SG18] Chunyao Song and Tingjian Ge. Labeled graph sketches. 2018. (cited on Page 15)

[SJ09] Blake Shaw and Tony Jebara. Structure preserving embedding. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 937–944, 2009. (cited on Page 32)

[SMS+17] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proceedings of the VLDB Endowment*, pages 420–431, 2017. (cited on Page ix, 9, 10, and 14)

[SPR11] Venu Satuluri, Srinivasan Parthasarathy, and Yiye Ruan. Local graph sparsification for scalable clustering. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 721–732, 2011. (cited on Page 16, 26, and 30)

[SS11] Daniel A Spielman and Nikhil Srivastava. Graph sparsification by effective resistances. *SIAM Journal on Computing*, pages 1913–1926, 2011. (cited on Page 40)

[SSM14] Christian Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: An interactive tool suite for high-performance network analysis. *CoRR*, 2014. (cited on Page 16, 25, 45, and 46)

[ST11] Daniel A Spielman and Shang-Hua Teng. Spectral sparsification of graphs. *SIAM Journal on Computing*, pages 981–1025, 2011. (cited on Page 40)

[STWJ13] Maryam Shoaran, Alex Thomo, and Jens H Weber-Jahnke. Zero-knowledge private graph summarization. In *BigData Conference*, pages 597–605, 2013. (cited on Page 40)

[SWL+18] Qi Song, Yinghui Wu, Peng Lin, Luna Xin Dong, and Hui Sun. Mining summaries for knowledge graph search. *IEEE Transactions on Knowledge and Data Engineering*, pages 1887–1900, 2018. (cited on Page 40)

[TFGER07] Hanghang Tong, Christos Faloutsos, Brian Gallagher, and Tina Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 737–746, 2007. (cited on Page 23)

[THP08] Yuanyuan Tian, Richard A Hankins, and Jignesh M Patel. Efficient aggregation for graph summarization. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 567–580, 2008. (cited on Page 40)

[TMKM18] Anton Tsitsulin, Davide Mottin, Panagiotis Karras, and Emmanuel Müller. Verse: Versatile graph embeddings from similarity measures. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 539–548, 2018. (cited on Page 31)

[TQW^{+}15] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1067–1077, 2015. (cited on Page 17 and 40)

[WCW^{+}17] Xiao Wang, Peng Cui, Jing Wang, Jian Pei, Wenwu Zhu, and Shiqiang Yang. Community preserving network embedding. In *AAAI*, pages 203–209, 2017. (cited on Page 17)

[WCZ16] Daixin Wang, Peng Cui, and Wenwu Zhu. Structural deep network embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1225–1234, 2016. (cited on Page 18, 31, and 40)

[WGHB09] Pu Wang, Marta C González, César A Hidalgo, and Albert-László Barabási. Understanding the spreading patterns of mobile phone viruses. *Science*, pages 1071–1076, 2009. (cited on Page 20)

[WWL^{+}18] Meng Wang, Ruijie Wang, Jun Liu, Yihe Chen, Lei Zhang, and Guilin Qi. Towards empty answers in sparql: Approximating querying with rdf embedding. In *International Semantic Web Conference*, pages 513–529, 2018. (cited on Page 40)

[YL13] Jaewon Yang and Jure Leskovec. Overlapping community detection at scale: a nonnegative matrix factorization approach. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 587–596, 2013. (cited on Page 39)

[YPS^{+}13] Jinguo You, Qiuping Pan, Wei Shi, Zhipeng Zhang, and Jianhua Hu. Towards graph summary and aggregation: A survey. In *Social Media Retrieval and Mining*, pages 3–12. 2013. (cited on Page 39)

[YXZ^{+}07] Shuicheng Yan, Dong Xu, Benyu Zhang, Hong-Jiang Zhang, Qiang Yang, and Stephen Lin. Graph embedding and extensions: A general framework for dimensionality reduction. *IEEE transactions on pattern analysis and machine intelligence*, pages 40–51, 2007. (cited on Page 31)