ON QUERYING LARGE SCALE INFORMATION NETWORKS

BY

PEIXIANG ZHAO

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Doctoral Committee:
Abel Bliss Professor Jiawei Han, Chair and Director of Research
Associate Professor Kevin Chen-Chuan Chang
Associate Professor Chengxiang Zhai
Research Scientist Charu C. Aggarwal, IBM Research
Abstract

Social and technical information systems usually consist of a large number of interacting physical, conceptual, and human/societal entities. Such individual entities are interconnected to form large and sophisticated networks, which, without loss of generality, are often refereed to as information networks. Examples of information networks include the Web, highway or urban transportation networks, research collaboration and publication networks, biological networks and social networks. Clearly, information networks are ubiquitous and form a critical component of modern information infrastructure.

Theoretically, information networks can be modeled and manipulated as large scale graphs, which have gradually become the first-class citizens in the data management and mining fields. However, it is extremely inefficient to process such graph-structured data in any existing data models or computational frameworks. Real world information networks are massive, whose sheer size may simply overwhelm a direct application of any conventional graph algorithms designed and implemented for small or medium-sized memory-resident graphs. In the mean time, information networks are not static but rapidly changing all the time. The massive and dynamic nature of information networks has posed special challenges to effective query processing especially in scenarios where real-time responses are desirable.

In this thesis, we will consider a series of queries of practical value arising in real world network scenarios, and explore the effective and potentially scalable querying solutions for large scale information networks. All such queries have been found fundamental and critically important at the core of many advanced network applications. First of all, P-Rank is proposed to answer the structural similarity query: “which entities are (structurally) similar to a query...
“entity” in an information network. Second, SPath is proposed as a high performance graph indexing mechanism to address general subgraph queries on large scale information networks.

Third, Graph Cube is designed as a new warehousing model that supports OLAP queries on large multidimensional information networks. Last, but not the least, gSketch is devised as a new sketch method that combines well-studied synopsis structures with a sketch partitioning technique in order to estimate and optimize the responses to basic queries on rapidly changing information networks. Our experimental studies demonstrate that our querying methods are highly efficient and scalable, and have achieved satisfactory performance for the fundamental queries on large scale information networks.

We should admit that the queries examined in the thesis are merely the tip of the iceberg. The marriage of information network analysis and query processing technology will bring many exciting opportunities for future study, which are briefed in the end of the thesis.
To my grandma, Xuemei Zhao, for all her love.
Acknowledgments

This thesis would not have been possible without the consistent support and careful guidance of my thesis advisor, Professor Jiawei Han. I am forever indebted to Jiawei, who is blessed with an astonishing combination of brilliance and patience. I will be grateful for both the research appreciation and the technical skills he has patiently taught me. I owe him my research career, which is one of the most rewarding pursuits in my life.

I would like to thank Dr. Charu C. Aggarwal, Professor Kevin Chen-chuan Chang and Professor Chengxiang Zhai, for taking their valuable time to be on my thesis committee and for providing many thought-provoking comments and helpful suggestions.

I would like to express my sincere gratitude to my (current and former) colleagues in both the DAIS (Data and Information Systems) group and the Data Mining Research group at UIUC: Deng Cai, Chen Chen, Hong Cheng, Marina Danilevsky, Hongbo Deng, Bolin Ding, Quanquan Gu, Jing Gao, Hector Gonzalez, Manish Gupta, Xin Jin, Sangkyum Kim, Xiaolei Li, Zhenhui Li, Xide Lin, Chao Liu, Yizhou Sun, Lu An Tang, Chi Wang, Tim Weninger, Tianyi Wu, Dong Xin, Xifeng Yan, Xiaoxin Yin, Zhijun Yin, Xiao Yu, Yintao Yu, Bo Zhao, Feida Zhu and other group members for their constructive discussions and insightful comments on research. My colleagues gave me enjoyable and unforgettable experiences through the years of study at the University of Illinois at Urbana-Champaign.

Finally, I am deeply indebted to my wife Zhe and my parents for their love, faith and encouragement. For any current or future success that I might have, they deserve more credit than I can possibly express in a few lines of text.
Table of Contents

List of Tables ....................................................... ix

List of Figures ...................................................... x

Chapter 1 Introduction ............................................. 1

Chapter 2 Preliminary Concepts ................................... 9
  2.1 Graphs .............................................. 9
  2.2 Networks .......................................... 12

Chapter 3 Structural Similarity Computation in Information Networks ........................................ 14
  3.1 Introduction ........................................ 14
  3.2 Related Work ....................................... 18
  3.3 P-Rank .................................................. 20
    3.3.1 Preliminaries .................................... 20
    3.3.2 P-Rank Formula ................................ 21
    3.3.3 Derivatives of P-Rank ......................... 25
    3.3.4 Computing P-Rank .............................. 27
  3.4 Experiment ........................................... 31
    3.4.1 A Heterogenous DBLP Information Network ... 31
    3.4.2 A Homogenous DBLP Information Network ... 35
    3.4.3 Synthetic Data Sets: R-MAT .................... 37
  3.5 Conclusion ............................................. 39

Chapter 4 Graph Query Optimization in Information Networks ........................................... 40
  4.1 Introduction ........................................... 40
  4.2 Related Work ......................................... 43
  4.3 Problem Definition .................................... 46
  4.4 The Pattern-based Graph Indexing Framework ............ 47
    4.4.1 The Baseline Algorithmic Framework ......... 47
    4.4.2 Structural Pattern Based Graph Indexing .... 49
    4.4.3 Structural Pattern Evaluation Model ........ 52
  4.5 SPath .................................................. 53
    4.5.1 Neighborhood Signature ...................... 54
    4.5.2 SPath Implementation ........................ 57
List of Tables

4.1 Qualitative Costs for Different Structural Patterns . . . . . . . . . . . . . . . 53
5.1 Major Conferences Chosen For Each Research Area . . . . . . . . . . . . . . 95
5.2 Four Buckets of Publication Numbers for the Productivity Attribute . . . . . 95
6.1 Avg. Relative Error of gSketch and Outlier Sketch in GTGraph Data Set . . 136
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Different Real World Information Networks</td>
<td>2</td>
</tr>
<tr>
<td>3.1</td>
<td>A Heterogenous Information Network and Structural Similarity Scores of Sim-Rank ($C = 0.8$) and P-Rank ($C = 0.8, \lambda = 0.5$)</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>A Homogeneous Information Network and Structural Similarity Scores of Sim-Rank ($C = 0.8$) and P-Rank ($C = 0.8, \lambda = 0.8$)</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>Structural Similarity Matrix for Information Networks</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>The Schema of Heterogeneous DBLP Information Networks. (The Number Represents the Number of Vertices in the Corresponding Category)</td>
<td>32</td>
</tr>
<tr>
<td>3.5</td>
<td>Compactness of P-Rank and of SimRank in Heterogeneous DBLP</td>
<td>33</td>
</tr>
<tr>
<td>3.6</td>
<td>Similarity Measures on Author Pairs in Heterogeneous DBLP</td>
<td>34</td>
</tr>
<tr>
<td>3.7</td>
<td>Top-10 Ranking Results for Author Vertices in DBLP by P-Rank</td>
<td>35</td>
</tr>
<tr>
<td>3.8</td>
<td>Compactness of P-Rank and of SimRank in Homogeneous DBLP</td>
<td>36</td>
</tr>
<tr>
<td>3.9</td>
<td>Vertex Pair Distributions w.r.t. Similarity Scores in Homogeneous DBLP</td>
<td>36</td>
</tr>
<tr>
<td>3.10</td>
<td>Vertex Pair Distributions w.r.t. Similarity Scores in Synthetic Data Set</td>
<td>37</td>
</tr>
<tr>
<td>3.11</td>
<td>P-Rank v.s. Different Parameters</td>
<td>38</td>
</tr>
<tr>
<td>4.1</td>
<td>A Network $G$ and a Query Graph $Q$</td>
<td>46</td>
</tr>
<tr>
<td>4.2</td>
<td>SPath Data Structure for $u_3 \in V(G)$ ($k_0 = 2$)</td>
<td>58</td>
</tr>
<tr>
<td>4.3</td>
<td>Index Construction Cost of SPath</td>
<td>67</td>
</tr>
<tr>
<td>4.4</td>
<td>Query Response Time for Clique Queries</td>
<td>68</td>
</tr>
<tr>
<td>4.5</td>
<td>Query Response Time for Path Queries</td>
<td>69</td>
</tr>
<tr>
<td>4.6</td>
<td>Query Response Time for Subgraph Queries</td>
<td>70</td>
</tr>
<tr>
<td>4.7</td>
<td>Index Construction of SPath</td>
<td>70</td>
</tr>
<tr>
<td>4.8</td>
<td>Query Response Time for Subgraph Queries in the Synthetic Graph</td>
<td>71</td>
</tr>
<tr>
<td>5.1</td>
<td>A Sample Multidimensional Network with a Graph and a Multidimensional Vertex Attribute Table</td>
<td>74</td>
</tr>
<tr>
<td>5.2</td>
<td>Multidimensional Network Aggregation vs. RDB Aggregation (Group by Gender)</td>
<td>75</td>
</tr>
<tr>
<td>5.3</td>
<td>Multidimensional Network Aggregation vs. RDB Aggregation (Group by Gender and Location)</td>
<td>76</td>
</tr>
<tr>
<td>5.4</td>
<td>The Graph Cube Lattice</td>
<td>81</td>
</tr>
<tr>
<td>5.5</td>
<td>The Aggregate Network between a User (ID=3) and Various Locations</td>
<td>86</td>
</tr>
<tr>
<td>5.6</td>
<td>Traditional Cuboid Queries</td>
<td>87</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Social and technical information systems usually consist of a large number of interacting physical, conceptual, and human/societal entities. Such individual entities are interconnected with relationships to form large and sophisticated networks. Without loss of generality, we call these interconnected networks as information networks. Examples of information networks include the Web [72, 96], highway or urban transportation networks [63], research collaboration and publication networks [47], biological networks [98] and social networks [84]. Clearly, information networks are ubiquitous and form a critical component of modern information infrastructure.

An information network can be naturally modeled and represented as a graph [7, 28]. Entities of the information network are depicted as vertices, while relationships among entities are illustrated as edges, thus forming a large scale interconnected graph structure. Recent years have witnessed a rapid proliferation of information networks that keep growing in an astounding speed in various application domains. Some noteworthy examples of information networks are illustrated in Figure 1.1

- Figure 1.1(a) represents an email communication network among 436 employees, depicted as red vertices in the graph, of Hewlett Packard Research Lab [3]. Two individuals are linked, illustrated as grey edges between vertices, if they exchanged at least 6 emails in either direction within more than 3 months between Oct. 2002 and Feb. 2003;

- Figure 1.1(b) represents a political blog network of the 2004 U.S. Presidential election
Figure 1.1: Different Real World Information Networks revealing two natural and well-separated clusters [2]. There were 1,494 blogs in total, 759 liberal, depicted as blue vertices, and 735 conservative, depicted as red vertices. If there is a URL on the page of one blog which references another political blog, an edge will be created accordingly. The colors of edges also reflect political orientation, red for conservative, and blue for liberal. Orange edges go from liberal to conservative, and purple ones from conservative to liberal. The size of each vertex reflects the number of other vertices (blogs) that link to it;

- Figure 1.1(c) represents a disease spread network containing 35 tuberculosis (TB) patients and their 1,039 contacts in southwest Oklahoma in 2002. TB patients are represented by black nodes. Contacts are represented by white boxes. Gray lines
represent the links between TB patients and contacts. Decreasing thicknesses of gray lines represent the strength of the relationship between patients and type of contacts: close, casual, or undetermined, respectively;

- Figure 1.1(d) represents a small social network from Facebook where the friendship connections from one individual, Elliott Bäck, to his friends have been plotted\(^1\). Such online social networks allow granular insights into how individuals behave and how they form and change their connections.

Despite their pervasiveness in the daily life, information networks, to our surprise, have been far not enough to be examined thoroughly and systematically. Past research has merely touched the tip of the iceberg of the studies and analysis of information networks. The significant variety and special characteristics of information networks have posed a series of serious challenges to model, manage and access such graph-structured data in an efficient and cost-effective way:

1. Although relational database theory and XML technology have been mature for decades, there still lack well-established methodologies and industry-strength solutions for newly emerging graph-structured information networks. Information networks, often modeled and manipulated as large scale graphs, have gradually become the first-class citizen in data management and mining fields. However, it is extremely inefficient to process such graph-structured data in any existing data models or frameworks;

2. While information networks are gradually becoming recognized as a rich and powerful repository with vast amounts of implicitly and explicitly embedded knowledge, the corresponding analytical solutions designed in this context have been primitive so far with limited usage. Heterogeneous network contents such as text, images, spatial-temporal information and multidimensional metadata are usually treated independently of the

\(^1\)http://www.digitaltrainingacademy.com/socialmedia/2009/06/social_networking_map.php.
interlinked structure of the networks, therefore resulting in poor query or mining results that are of little interest to users on the one hand, and with low interpretability and usability on the other.

3. Real world information networks are massive and their corresponding underlying graphs typically contain millions of vertices and billions of edges. For example, as of June 2012, the indexed Web graph contains at least 8.94 billion pages\(^2\). Facebook has more than 901 millions active users and 125 billions friendship relations\(^3\). Information networks in other application domains, such as transportation networks and biochemical networks tend to grow massive as well \[^7\]. The sheer size of such information networks may simply overwhelm a direct application of the conventional graph algorithms which were designed and implemented for small or medium-sized memory-resident graphs. Meanwhile, information networks are not static but expanding all the time. As of December 2011, eMarketer estimates just over 1.2 billion people around the world used social networking sites at least once per month. That represented 23.1% growth over 2010, and double-digit growth will continue throughout eMarketer’s forecast period\(^4\). The massive and dynamic nature of information networks has thus posed a special challenge to algorithm design especially in scenarios where real-time responses are desirable.

The burgeoning size and heterogeneity of information networks have inspired extensive interest in supporting effective and efficient querying methods in real applications that are centered on massive graph data. In this thesis, we will be focused on a series of queries of practical value arising in real world information networks, and explore effective and potentially scalable solutions in the case of large scale information networks. The queries picked for exploration are fundamental and critical at the core of many advanced information network operations, and have been found representative and extremely prevalent in a wide variety

\(^2\)http://www.worldwidewebsize.com
\(^4\)http://www.emarketer.com
of information network applications. The primary goal of our research is to explore both principled methodologies and innovative applications for competent and high-performance query processing technology in large scale information networks that can be accessed and retrieved effectively in the context of the massive and ever-changing data conditions and application requirements.

In this thesis, we restrict ourselves to the information networks that can be modeled as large scale connected simple graphs. However, all our query processing methods can be easily generalized toward other disconnected, multi-graphs. Our current research progress is therefore formulated and reported based on the types of both the information networks and the various queries that arise frequently in information networks, as follows:

1. **P-Rank: on structural similarity computation in information networks.** In this work, we are focused on the query “how (structurally) similar are two entities in an information network?”. The structural similarity query is the cornerstone of many advanced operations of information networks, such as proximity query processing [112, 19], outlier detection [48], classification [62] and clustering [103]. We therefore propose a new structural similarity measure, **P-Rank (Penetrating Rank)**, to evaluate structural similarities of entities in real world information networks. **P-Rank** enriches the well-known **SimRank** [61] by jointly encoding both in- and out-link relationships of entities into structural similarity computation. **P-Rank** proves to be a unified structural similarity framework, under which most of the state-of-the-art structural similarity measures, including **CoCitation**, **Coupling**, **Amsler** and **SimRank**, are just its special cases. Based on the recursive nature of **P-Rank**, we further design a fixed-point algorithm to reinforce structural similarity of entity pairs beyond the localized neighborhood scope toward a computation over the entire information network. Our experiments demonstrate the power of **P-Rank** in different information networks. Meanwhile, **P-Rank** outperforms **SimRank** as a more comprehensive and meaningful structural similarity measurement in real world information networks;
2. **SPath: on graph query optimization in information networks.** In this work, we consider a common and critical structural query primitive of information networks: *how to find subgraph structures efficiently in a large information network?* As a key ingredient of many network applications, this graph query has been frequently issued and extensive studied in pattern recognition, computer systems, social networks and bioinformatics. Unfortunately, the subgraph query is hard due to the NP-complete nature of subgraph isomorphism. It becomes even challenging when the information network examined is large and diverse. We thus devise a high performance graph indexing mechanism, **SPath**, to address the subgraph query problem in large scale information networks. **SPath** leverages decomposed shortest path information in vertex neighborhood as basic indexing units, which prove to be both cost-effective in graph search space pruning and highly scalable in index construction and deployment. Via **SPath**, a subgraph query can be further optimized from the traditional vertex-at-a-time querying paradigm to a more efficient *path-at-a-time* way: the graph query is first decomposed to a set of shortest paths, among which a subset of candidates with good selectivity is picked by a query plan optimizer. Candidate paths are joined together afterwards in the information network to help recover the query graph and finalize the graph query processing. We evaluate **SPath** with the state-of-the-art GraphQL [57] on both real and synthetic data sets. Our experimental studies demonstrate the effectiveness and scalability of **SPath**, which proves to be a more practical and efficient indexing method in addressing general subgraph queries on large scale information networks;

3. **Graph Cube: on warehousing and OLAP multidimensional information networks.** In this work, we consider extending decision support facilities by way of resolving OLAP (online analytical processing) queries toward large sophisticated information networks, upon which multidimensional attributes are associated with network entities, thereby forming the so-called *multidimensional information networks*. As important
means of decision support and business intelligence, OLAP queries are advantageous in both network summarization [105,115] and social targeting [12,104]. We therefore introduce **Graph Cube**, a new data warehousing model that supports OLAP queries effectively on large multidimensional information networks. Taking account of both attribute aggregation and structure summarization from the networks, **Graph Cube** goes beyond the traditional data cube model that solely involves numeric value based group-by’s, thus resulting in a more insightful and structure-enriched aggregate network within every possible multidimensional space. Besides traditional cuboid queries, a new class of OLAP queries, *crossboid*, is introduced that is uniquely useful in multidimensional information networks and has not been studied before. We implement **Graph Cube** by combining special characteristics of multidimensional networks with the existing well-studied data cube techniques. We perform extensive experimental studies on a series of real world data sets and **Graph Cube** has proven to be a powerful and efficient tool for decision support on large multidimensional information networks;

4. **gSketch**: **on query estimation in streaming information networks.** Many information networks in real applications are actually *graph streams*, in which edges of the underlying graph are received and updated sequentially in a form of a stream. It is often necessary and important to summarize the behavior of graph streams in order to enable effective query processing on large-scale information networks. In this work, we propose a new graph sketch method, **gSketch**, which combines well studied synopses for traditional data streams with a sketch partitioning technique, to estimate and optimize the responses to basic queries, such as edge queries and aggregate subgraph queries, on streaming information networks. Despite being primitive, both the edge query and the aggregate subgraph query are nontrivial in streaming information networks, and they serve as the building bricks of many advanced querying and mining operations of graph streams [13,13,63]. We consider two different scenarios for query estima-
tion: (1) A graph stream sample is available; (2) Both a graph stream sample and a query workload sample are available. Algorithms for different scenarios are designed respectively by partitioning a global sketch to a group of localized sketches in order to optimize the query estimation accuracy. We perform extensive experimental studies on both real and synthetic data sets and demonstrate the power and robustness of gSketch in comparison with the state-of-the-art global sketch method \cite{31}.

Graph has proven to be a powerful abstraction for interlinked complex and heterogeneous data, and the information network paradigm has posed a wealth of fascinating research problems and high impact applications. It is our strong belief that they will continue to play a key role in many widely encountered scenarios. Meanwhile, modern business and industrial infrastructures are collecting massive amounts of data ("big data") at an unprecedented scale and pace. The ability to perform efficient and data-intensive query processing techniques on such large scale information networks now becomes a key ingredient of success. In the end of the thesis, we also envision some of the future opportunities that go beyond our current explorations.

The remainder of the thesis is organized as follows. In Chapter 2, the preliminary concepts and common definitions are first elaborated to facilitate the subsequent discussion. In Chapter 3, we present P-Rank for supporting structural similarity computation on information networks. In Chapter 4, the subgraph queries are addressed in detail by SPath. Chapter 5 demonstrates Graph Cube for OLAP queries on multidimensional information networks. In Chapter 6, gSketch is elaborated for query estimation in streaming information networks. We will brief the potential future work and conclude our thesis in Chapter 7.
Chapter 2

Preliminary Concepts

In this chapter we develop some of the basic concepts and common definitions behind information networks. This will allow us to formulate both the data of interest and queries to be examined in a unifying language for our thesis. The definitions here are generic enough that they can be applied unambiguously throughout the thesis if the context is not specified otherwise, while other advanced concepts related to complicated information networks and corresponding querying solutions will be defined in subsequent chapters respectively.

2.1 Graphs

The most natural and easiest way to represent an information network is by way of graph. Graph is a basic mathematical construct specifying relationships among a collection of entities. A graph $G = (V, E)$ consists of a set $V$ of entities, called vertices, with a set $E$ of certain pairs of these entities connected by links, called edges. An edge $e$ can be represented as $e = (u, v)$ where $u, v$ are the pair of vertices of $e$. $u$ (or $v$) is said to be adjacent to $v$ (or $u$), and $u, v$ are neighbors. Both vertices and edges can be assigned with unique identifiers for ease of presentation and analysis.

The degree of a vertex $u$ in a graph $G$, denoted $d(u)$, is the number of edges adjacent to $u$. If $G$ contains $m$ edges, it becomes evident $2m = \sum_{u \in V} d(u)$.

In some scenarios, the edges of a graph are used to model symmetric relationships: the edge $e = (u, v)$ simply connects $u$ and $v$ and the relative order of $u$ and $v$ makes no difference in modeling such a symmetric relationship between $u$ and $v$. For example, a friendship
relation connecting Alex and Bob in FaceBook is a symmetric relationship. Hence the graph in these scenarios is often referred to as an *undirected graph*. Likewise, in many other settings, the edges of a graph can be used to express *asymmetric* relationships: the edge \( e = (u, v) \) indicates that \( u \) points to \( v \) but not vice versa. For example, the adviser-advisee relationship between Chelsea (advisor) and David (advisee) in the Mathematics Genealogy Project\(^1\) is an asymmetric relationship. We therefore refer to the graph modeling asymmetric relationships as a *directed graph*.

Besides interlinked relationships, both vertices and edges of a graph may bear attributes. The graph is therefore called *labeled graph*. For example, the city names of a road network are the labels of the vertices in the underlying graph, while the road/highway names are the labels of the edges, accordingly. Theoretically, both vertices and edges may have more complex labels, such as numeric values, strings, multidimensional vectors or even graphs themselves. Likewise, a graph may have no labels for vertices or edges, as the one defined in the beginning of the chapter. Such a graph is often referred to as an *unlabeled graph* indicating that only the interlinked relationships of entities are concerned. When edges of a graph are associated with numeric values, such a graph is called *weighted*. For example, the Internet router graph has physical links between routers as the edges of the graph, and the bandwidth of each link in megabits per second (Mbps) is its weight.

There may exist multiple relationships between entities, *i.e.*, there may be more than one edge between two vertices. For example, Edwin and Fred are friends, colleagues, roommates and members of ACM (Association for Computing Machinery), so there are four edges (bearing different semantic meanings) between the vertices Edwin and Fred. Meanwhile, a vertex can link to itself thus forming a *loop* edge. For example, in an email communication network, Grace may send an email to herself as a reminder. A graph is called *simple* if it contains no loops nor multiple edges between vertices. Otherwise, it is usually referred to as a *multi-graph*.

\(^1\)http://genealogy.math.ndsu.nodak.edu/
Given a graph, a \textit{path} is simply a sequence of vertices with the property that each consecutive pair in the sequence is connected by an edge. We define the \textit{length} of a path to be the number of steps it contains from beginning to end. A path without repetitive vertices is often referred to as a \textit{simple path}. A \textit{cycle} is a special path with at least three edges, in which the first and last vertices are identical, but otherwise all vertices are distinct. Cycles can be frequently found in communication and transportation networks allowing for redundancy. That is, they provide for alternate routings that go the “other way” around the cycle. With this in mind, we say that a graph is \textit{connected} if for every pair of vertices, there is a path between them. Otherwise, the graph is said \textit{disconnected}. If a graph is disconnected, then it breaks apart naturally into a set of \textit{connected components}. A connected component contains a group of vertices each of which has a path to all the others within the group. And such a group of vertices is not part of some larger group with the property that every vertex can reach every other. Dividing a graph into its connected components might be the first global way of describing its structure. Within a given connected component, there may be richer internal structure that is important to one’s interpretation of the network.\textsuperscript{34}

There are a lot of specialized graphs in real world applications. A \textit{bipartite} graph $G$ contains two disjoint independent sets $V_1$ and $V_2$ of vertices with edges between $V_1$ and $V_2$. In a recommender network, $V_1$ is a set of users and $V_2$ is a set of commodities. An edge between $u \in V_1$ and $v \in V_2$ means the user $u$ likes and recommends the commodity $v$. A \textit{tree} is a connected graph with no cycles. For a tree $G$ with $n$ vertices, the following statements are equivalent: (1) $G$ is connected and has $(n - 1)$ edges; (2) $G$ has $(n - 1)$ edges and no cycles; (3) $G$ has no loops and has, for each $u, v \in G$, exactly one path between $u$ and $v$. An XML document can be represented as a tree with attributes attached upon vertices. In bioinformatics, a phylogeny tree is used to represent the evolutionary relationships among various biological species or other entities based upon similarities and differences in their physical and/or genetic characteristics. A \textit{clique}, \textit{a.k.a. complete graph}, is a graph with a set of pairwise adjacent vertices. Clique usually represents a close community in social networks.
2.2 Networks

Graphs are useful mathematical models of information network structures. Although there are many differences between the information network structures, some common patterns or “laws” show up regularly, which together characterize the naturally occurring graphs in real world information networks. Such patterns or “laws”, if leveraged appropriately, can immensely improve the querying effectiveness for large scale information networks, as dictated in the following chapters.

1. Power Laws and Scale-free Networks. It has been witnessed and verified that the degree distribution of an undirect graph obeys the power law distribution. Similarly, the eigenvalues of the adjacency matrix of a graph versus their ranks obey the same law. Power laws also show up in the distribution of “bipartite cores” and the distribution of PageRank values. The significance of power law distributions lies in the fact that they are heavy-tailed, meaning that they decay very slowly. Indeed, power laws appear to be a definitive characteristic of almost all large scale networks, including social networks, protein-to-protein interaction networks, computer networks and the Web.

Definition 1. (POWER LAW) Two variables $x$ and $y$ obey the power law if their scatter plot is linear on a log-log scale: $y(x) = cx^{-\gamma}$, where $c$ and $\gamma$ are positive constants. $\gamma$ is often referred to as the power law exponent.

Networks with power-law degree distributions are often referred to as scale-free networks.

2. Small World Phenomenon. A small-world network is a graph in which most vertices are not neighbors of one another, but most vertices can be reached from every other by a small number of hops or steps. Specifically, a small-world network is defined to be a network where the typical distance $L$ between two randomly chosen vertices grows proportionally to the logarithm of the number of vertices $n$ in the network, i.e., $L \propto \log n$. Many empirical graphs are well-modeled by small-world networks. Social networks, the
connectivity of the Internet, Wikipedia, and gene networks all exhibit small-world network characteristics.

Small-world networks tend to contain cliques, and near-cliques, meaning sub-networks have connections between almost any two vertices within them. Meanwhile, most pairs of vertices will be connected by at least one short path. Several other properties are often associated with small-world networks. Typically there is an over-abundance of hubs — vertices in the network with a high degree. These hubs serve as the common connections mediating the short path lengths between other edges. This property is often analyzed by considering the fraction of vertices in the network that have a particular number of connections going into them (the degree distribution of the network). Networks with a greater than expected number of hubs will have a greater fraction of vertices with high degree, and consequently the degree distribution will be enriched at high degree values. Specifically, if a network has a degree-distribution which can be fit with a power law distribution, it is taken as a sign that the network is small-world. Cohen etc. [27] showed analytically that scale-free networks are ultra-small worlds. In this case, due to hubs, the shortest paths become significantly smaller and scale as \( L \propto \log \log n \).

3. Community Effects. A community is a set of vertices of a graph where each vertex is “closer” to the other vertices within the community than to vertices outside it. The community effect has been found in many real world networks, especially social networks. The closeness of vertices within a community is often characterized by the clustering coefficient:

**Definition 2. (CLUSTERING COEFFICIENT)** For a vertex \( u \) with edges \((u, v)\) and \((u, w)\), the local clustering coefficient of \( u \) measures the probability of existence of the third edge \((v, w)\). The global clustering coefficient of the entire graph is found by averaging over all vertices of the graph.

The networks with the large average clustering coefficient tend to have a modular structure, and they have the small possible average distance among different vertices.
Chapter 3

Structural Similarity Computation in Information Networks

3.1 Introduction

In this chapter, we consider the problem of similarity computation on entities of information networks. Our study is motivated by recent research and applications in proximity query processing [112, 19], outlier detection [48], classification [62] and clustering [103] over different information networks, which usually require an effective and trustworthy evaluation of underlying similarity among entities. It is desirable to propose a comprehensive similarity measure on information networks that can both map human intuition and generalize well under different information network settings.

However, it is nontrivial to systematically compute entity similarity in a general and effective fashion, and it becomes especially challenging when the information networks to be examined are massive and diverse. In the mean time, multiple aspects of entities in information networks can be exploited for similarity computation, and the choices are usually made domain-specifically.

In this chapter, we propose a new structural similarity measure, $P$-Rank (Penetrating Rank), which solely explores the link structure of the underlying information network for similarity computation. Compared with traditional text contents, the link-based structural information is more homogenous and language independent, which is critical for similarity computation [80]. Concretely, within an information network, we compute $P$-Rank that says “two entities are similar if (1) they are referenced by similar entities; and (2) they reference similar entities.” In comparison with the state-of-the-art structural similarity measure,
Figure 3.1: A Heterogenous Information Network and Structural Similarity Scores of SimRank ($C = 0.8$) and P-Rank ($C = 0.8, \lambda = 0.5$)

SimRank \[61\], which considers the first aforementioned factor only, P-Rank encodes both in- and out-link relationships into computation toward a semantically complete and robust similarity measure. Moreover, similarity beliefs of entity pairs are propagated beyond local neighborhood scope to the entire information network, whose global structure is fully utilized in order to reinforce similarity beliefs of entities in a recursive fashion. P-Rank also proves to be a general framework for structural similarity of information networks and can be easily adapted in any information network settings wherever there exist enough interlinked relationships among entities. For practical applicability, P-Rank can be effectively coupled with other non-structural domain-specific similarity measures, for example, textual similarity, toward a unified similarity measure for information networks.

Example 3.1.1: Consider a heterogeneous information network $G$ in Figure 3.1 representing a typical submission, bidding, review and acceptance procedure of a conference. $G$ is regarded as heterogeneous if vertices (entities) of $G$ belong to different mutual exclusive categories, such as Conference = \{c\}, CommitterMember = \{m_1, m_2, m_3\} and Paper = \{p_1, p_2, p_3, p_4\}. Directed edges represent the relationships between vertices in different categories. Two structural similarity measures, SimRank and P-Rank, for different vertex pairs of $G$ are illustrated as well. As shown in Figure 3.1, the conference $c$ is considered similar to itself, and the similarity scores (for both SimRank and P-Rank) are set to be 1. For committee member pairs $\{m_1, m_2\}$, $\{m_1, m_3\}$, and $\{m_2, m_3\}$, as both vertices of each pair are pointed to by $c$ (they both are invited as committee members by the conference,
c), we may infer that they are similar. However, SimRank cannot differentiate these three pairs. (They have the same SimRank score, 0.4). The main reason is that for committee member pairs, SimRank considers their in-link relationships with the vertex c only, while neglecting out-link relationships with paper vertices \{p_1, p_2, p_3, p_4\}. P-Rank, however, takes into account of both in- and out-link relationships for similarity computation. As to \{m_1, m_2\}, for example, because they both point to \(p_2\) (both \(m_1\) and \(m_2\) bid for paper \(p_2\)), the structural similarity between them is further strengthened (P-Rank score is 0.420, which is different from that of \{m_2, m_3\} (0.295), and that of \{m_1, m_3\} (0.380)). We generalize this idea by observing that once we have concluded similarity between \(m_1\) and \(m_2\), \(p_1\) and \(p_3\) are similar as well because they are pointed to by \(m_1\) and \(m_2\), respectively, although such an inference is somehow weakened during similarity propagation. Continuing forth, for every comparable pair of vertices (in the same category) in \(G\), we can infer P-Rank between them.

\[ \square \]

**Example 3.1.2:** Consider a homogeneous information network \(G\) in Figure 3.2, representing a tiny literature graph. \(G\) is homogeneous if vertices of \(G\), which represent scientific publications in this example, belong to one category (“Publication”). Edges between vertices are references/citations from one paper to another. Different from heterogeneous information networks, any pair of vertices in homogenous information networks can be measured by their structural similarity because they all belong to the same category. We present SimRank and P-Rank scores for some of them, as shown in Figure 3.2. SimRank cannot tell the differences between the vertex pair \{P_2, P_3\} and \{P_3, P_4\}, solely because SimRank considers partial in-link relationship information for similarity computation. More severely, SimRank is unavailable for the vertex pairs \{P_4, P_5\} and \{P_2, P_5\}, mainly because these vertex pairs do not have shared in-link similarity factors. However, P-Rank can successfully infer structural similarity for all vertex pairs by considering both in- and out-link relationships into similarity computation, thus outperforms SimRank in homogeneous information networks.

\[ \square \]
As its name dictates, P-Rank encodes both in- and out-link relationships of entities in similarity computation, i.e., P-Rank scores flow from in-link neighbors of entities and penetrate through their out-link ones. Furthermore, this process is recursively propagated beyond the localized neighborhood scope of entities to the entire information network. The major merits of P-Rank are its semantic completeness, generality and robustness. As a comprehensive structural similarity measure, P-Rank can be effectively adapted in information networks with different variety and scale, in which most up-to-date similarity measures, like SimRank, may generate biased answers or simply fail due to the incomplete structural information considered in similarity computation, as illustrated in Example 3.1.1 and Example 3.1.2. In order to compute P-Rank efficiently, we propose an iterative algorithm converging fast to a fixed-point. The correctness of the algorithm is proven that this iterative algorithm always converges to its theoretical upper bound.

The contributions of our work are summarized as follows:

1. We propose a new structural similarity measure, P-Rank, applicable in information networks. We study its mathematical properties, its advantages over other state-of-the-art structural similarity measures, and its derivatives in different network settings.

2. We propose a fixed-point iterative algorithm to effectively compute P-Rank in information networks. We prove the correctness of the algorithm and discuss the optimization techniques to facilitate P-Rank computation in different scenarios.

3. P-Rank is a unified structural similarity framework in information networks, under
which the well-known structural similarity measures, CoCitation, Coupling, Amsler and SimRank, are all its special cases.

4. We conduct extensive experimental studies on both real and synthetic data sets. The evaluation results demonstrate the power of P-Rank as a general structural similarity measure for different information networks.

The rest of the chapter is organized as follows. Section 3.2 discusses related work. In Section 3.3 we present our structural similarity measure, P-Rank, from both mathematical and algorithmic perspectives. We report our experimental studies in Section 3.4. Section 3.5 concludes this section.

3.2 Related Work

As common standards to determine the closeness of different objects, similarity (or proximity) measures are crucial and frequently applied in clustering [103], nearest neighbor classification [54], anomaly detection [48] and similarity query processing [19]. Compared with traditional textual contents, link-based structural context of information networks is of special importance and exploited frequently in similarity computation. In previous studies, SimFusion [111] aimed at “combining relationships from multiple heterogeneous data sources”. [77] proposed a similarity measure based on PageRank score propagation through link paths. [45] explored methods for ranking partial tuples in a database graph. Maguitman et al. did extensive comparative studies for different similarity measures [80], and the results demonstrate that link-based structural similarity measures produce systematically better correlation with human judgements compared to the text-based ones.

In bibliometrics, similarities between scientific publications are commonly inferred from their cross-citations. Most noteworthy from this field are the methods of CoCitation [102], Coupling [70] and Amsler [10]. For CoCitation, the similarity between two papers p and q is based on the number of papers which reference both p and q. For Coupling, the similarity is
based on the number of papers referenced by both p and q. Amsler fuses both CoCitation and Coupling for similarity computation. These methods have been efficiently applied to cluster scientific publications and Web pages [94].

SimRank [61, 41, 78, 11] is an iterative PageRank-like structural similarity measure for information networks. It goes beyond simple CoCitation much as PageRank goes beyond direct linking for computing importance of Web pages. The weakness of SimRank, called the limited information problem, is discussed in [61]. SimRank makes use of in-link relationships only for similarity computation while neglecting similarity beliefs conveyed from out-link directions. Therefore, the structural information of information networks is partially exploited and the similarity computed is inevitably asymmetric and biased. In real information networks, those “unpopular entities”, i.e., entities with very few in-link relationships will be penalized by SimRank. More severely, SimRank can even be unavailable for entities with no in-link similarity flows (shown in Example 3.1.2). However, those entities with few or no in-links dominate information networks in quantity, as expressed by the power law and heavy-tailed in(out)-degree distribution [20]. Meanwhile, these entities are often not negligible because they are new, potentially popular, and interesting to most users. However, they tend to be harder for humans to find. To overcome the limited information problem, we propose P-Rank which refines the structural similarity definition by jointly considering both in- and out-link relationships of entity pairs. Furthermore, the similarity computation goes beyond the localized neighborhood so that the global structural information of information networks can be effectively leveraged to reinforce similarity beliefs of entities. As discussed afterwards, with the same time/space complexity as SimRank, P-Rank can achieve much better similarity results and solve the limited information problem effectively. Heymans et al. [59] proposed similar ideas to model structural similarity of enzymes in metabolic pathway graphs in order for phylogenetic analysis of metabolic pathways. However, their similarity are defined on vertices in different graphs and if the factors of dissimilarity and absence of edges are not considered, their work can be regarded as a special case of P-Rank ($C = 1$ and $\lambda = 0.5$).
Iterative fixed-point algorithms over the Web graph, like HITS [71] and PageRank [15], have been studied and applied to compute “importance” scores for Web pages. Results have shown that the usage of structural information of networks can greatly improve search performance versus text alone.

### 3.3 P-Rank

The basic recursive intuition of P-Rank can be expressed as “two entities of an information network are similar if they are related to similar entities”. More specifically, the two-fold meaning of P-Rank is elaborated as

1. two entities are similar if they are referenced by similar entities;
2. two entities are similar if they reference similar entities.

As the base case, we consider an entity maximally similar to itself, to which we can assign the P-Rank score of 1. (If other entities are known to be similar a-priori, their similarities can be pre-assigned as well.) For each pair of distinct entities, we take into consideration both their in- and out-link relationships for similarity computation. This similarity is then penetrating from in-link neighbors to out-link ones and propagated toward the entire information network.

#### 3.3.1 Preliminaries

We model an information network as a labeled directed graph $G = (V, E, \Sigma; l)$ where vertex $v \in V$ represents an entity of the domain and a directed edge $(u, v) \in E$ represents a relationship from entity $u$ to entity $v$, where $u, v \in V$. $\Sigma$ is an alphabet set and $l : V \rightarrow \Sigma$ is a labeling function. In heterogeneous information networks, $V = \{V_1 \cup V_2 \cdots \cup V_n\}$ can be partitioned into $n$ mutual exclusive vertex subsets, $V_1, V_2, \cdots, V_n$, $V_i \cap V_j = \emptyset$ for $1 \leq i, j \leq n$, which
belong to \( n \) different domain-specific categories. In homogeneous information networks, however, there is no distinction among vertices. Note that our definition of information networks can be naturally extended to undirected graphs or edge-weighted graphs.

For a vertex \( v \) in a graph \( G \), we denote by \( I(v) \) and \( O(v) \) the set of in-link neighbors and out-link neighbors of \( v \), respectively. Note that either \( I(v) \) and \( O(v) \) can be empty.

### 3.3.2 P-Rank Formula

We denote the P-Rank score for vertex \( a \) and \( b \) by \( s(a, b) \in [0, 1] \). Following our aforementioned intuition, P-Rank can be formalized recursively in Equation (3.1), when \( a \neq b \):

\[
s(a, b) = \frac{\lambda C |I(a)||I(b)|}{|I(a)||I(b)|} \sum_{i=1}^{\text{|I(a)|}} \sum_{j=1}^{\text{|I(b)|}} s(I_i(a), I_j(b)) + \frac{(1 - \lambda)C |O(a)||O(b)|}{|O(a)||O(b)|} \sum_{i=1}^{\text{|O(a)|}} \sum_{j=1}^{\text{|O(b)|}} s(O_i(a), O_j(b))
\]

Otherwise, P-Rank is defined as

\[
s(a, b) = 1
\] (3.2)

In Equation (3.1), the relative weight of in- and out-link directions is balanced by parameter \( \lambda \in [0, 1] \). \( C \) is set as a damping factor for in- and out-link directions, \( C \in [0, 1] \). The reason is that \( s(a, b) \) will be attenuated during similarity propagation. When \( I(a) \) (or \( I(b) \)) = \( \emptyset \), the in-link part is invalidated and only the out-link direction takes into effect. Similarly, when \( O(a) \) (or \( O(b) \)) = \( \emptyset \), only the similarity flows from in-link part are considered. If both \( I(a) \) (or \( I(b) \)) = \( \emptyset \) and \( O(a) \) (or \( O(b) \)) = \( \emptyset \), we define \( s(a, b) = 0 \).

Equation (3.1) is written for every pair of vertices \( a, b \in G \), resulting in a set of \( n^2 \) equations for a graph of size \( n \) (\( |V| = n \)). To solve the set of \( n^2 \) equations, we rewrite the recursive P-Rank formula (shown in Equation (3.1)) into the following iterative form.

\(^1\)For a more general form of P-Rank, \( C \) can be replaced by two different parameters \( C_{in} \) and \( C_{out} \) to represent damping factors for in- and out-link directions, respectively. We omit the details as it is fairly easy to extend our work into that form.

21
\[ R_0(a, b) = \begin{cases} 
0 & \text{if } a \neq b \\
1 & \text{if } a = b 
\end{cases} \quad (3.3) \]

and

\[ R_{k+1}(a, b) = \frac{\lambda C}{|I(a)||I(b)|} \sum_{i=1}^{|I(a)|} \sum_{j=1}^{|I(b)|} R_k(I_i(a), I_j(b)) + \frac{(1-\lambda)C}{|O(a)||O(b)|} \sum_{i=1}^{|O(a)|} \sum_{j=1}^{|O(b)|} R_k(O_i(a), O_j(b)) \quad (3.4) \]

where \( R_k(a, b) \) denotes the P-Rank score between \( a \) and \( b \) on iteration \( k \), for \( a \neq b \) and \( R_k(a, b) = 1 \) for \( a = b \). We progressively compute \( R_{k+1}(\ast, \ast) \) based on \( R_k(\ast, \ast) \). That is, on iteration \((k + 1)\), we update \( R_{k+1}(a, b) \) by the P-Rank scores from the previous iteration \( k \). This iterative computation starts with \( R_0(\ast, \ast) \) where \( R_0(a, b) \) is a lower bound of the actual P-Rank score, \( s(a, b) \).

**Theorem 3.3.1:** The iterative P-Rank equations (shown in Equation (3.3) and Equation (3.4)) have the following properties:

1. **(Symmetry)** \( R_k(a, b) = R_k(b, a) \)

2. **(Monotonicity)** \( 0 \leq R_k(a, b) \leq R_{k+1}(a, b) \leq 1 \)

3. **(Existence)** The solution to the iterative P-Rank equations always exists and converges to a fixed point, \( s(\ast, \ast) \), which is the theoretical solution to the recursive P-Rank equations.

4. **(Uniqueness)** the solution to the iterative P-Rank equations is unique when \( C \neq 1 \).

**Proof:**

1. **(Symmetry)** According to Equation (3.3) and Equation (3.4), it is obvious \( R_k(a, b) = R_k(b, a) \) for \( k \geq 0 \)

2. **(Monotonicity)** If \( a = b \), \( R_0(a, b) = R_1(a, b) = \ldots = 1 \), so it is obvious the monotonicity property holds. Let’s consider \( a \neq b \). According to Equation (3.3), \( R_0(a, b) = \ldots = 1 \), for \( k \geq 0 \) and

   \[ R_{k+1}(a, b) = \ldots = 1 \]
0. Based on Equation (3.3.1), \(0 \leq R_1(a, b) \leq 1\). So, \(0 \leq R_0(a, b) \leq R_1(a, b) \leq 1\). Let’s assume that for all \(k\), \(0 \leq R_{k-1}(a, b) \leq R_k(a, b) \leq 1\), then

\[
R_{k+1}(a, b) - R_k(a, b) = \frac{\lambda C}{|I(a)||I(b)|} \sum_{i=1}^{n} \sum_{j=1}^{m} (R_k(I_i(a), I_j(b)) - R_{k-1}(I_i(a), I_j(b))) + \frac{(1 - \lambda)C}{|O(a)||O(b)|} \sum_{i=1}^{n} \sum_{j=1}^{m} (R_k(O_i(a), O_j(b)) - R_{k-1}(O_i(a), O_j(b)))
\]

Based on the assumption, we have \((R_k(a, b) - R_{k-1}(a, b)) \geq 0, \forall a, b \in G\), so the left hand side \(R_{k+1}(a, b) - R_k(a, b) \geq 0\) holds. By induction, we draw the conclusion that for any \(k\), \(R_k \leq R_{k+1}\). And based on the assumption, \(0 \leq R_k(a, b) \leq 1\), so

\[
\frac{\lambda C}{|I(a)||I(b)|} \sum_{i=1}^{n} \sum_{j=1}^{m} R_k(I_i(a), I_j(b)) \leq \frac{\lambda C}{|I(a)||I(b)|} \sum_{i=1}^{n} \sum_{j=1}^{m} 1 = \lambda \times C
\]

and

\[
\frac{(1 - \lambda)C}{|O(a)||O(b)|} \sum_{i=1}^{n} \sum_{j=1}^{m} R_k(O_i(a), O_j(b)) \leq \frac{(1 - \lambda)C}{|O(a)||O(b)|} \sum_{i=1}^{n} \sum_{j=1}^{m} 1 = (1 - \lambda) \times C
\]

So \(R_{k+1}(a, b) \leq \lambda C + (1 - \lambda)C \leq 1\). By induction, we know that for any \(k\), \(0 \leq R_k(a, b) \leq 1\).

3. (Existence) According to Theorem 3.3.1(2), \(\forall a, b \in G, R_k(a, b)\) is bounded and nondecreasing as \(k\) increases. By the Completeness Axiom of calculus, each sequence \(R_k(a, b)\) converges to a limit \(R(a, b) \in [0, 1]\). Note \(\lim_{k \to \infty} R_k(a, b) = \lim_{k \to \infty} R_{k+1}(a, b) = R(a, b)\). So we have

\[
R(a, b) = \lim_{k \to \infty} R_{k+1}(a, b)
\]

\[
= \frac{\lambda C}{|I(a)||I(b)|} \lim_{k \to \infty} \sum_{i=1}^{n} \sum_{j=1}^{m} R_k(I_i(a), I_j(b)) + \frac{(1 - \lambda)C}{|O(a)||O(b)|} \lim_{k \to \infty} \sum_{i=1}^{n} \sum_{j=1}^{m} R_k(O_i(a), O_j(b))
\]

\[
= \frac{\lambda C}{|I(a)||I(b)|} \sum_{i=1}^{n} \sum_{j=1}^{m} \lim_{k \to \infty} R_k(I_i(a), I_j(b)) + \frac{(1 - \lambda)C}{|O(a)||O(b)|} \sum_{i=1}^{n} \sum_{j=1}^{m} \lim_{k \to \infty} R_k(O_i(a), O_j(b))
\]

23
\[ \frac{\lambda C}{|I(a)||I(b)|} \sum_{i=1}^{[I(a)]} \sum_{j=1}^{[I(b)]} R(I_i(a), I_j(b)) + \frac{(1 - \lambda)C}{|O(a)||O(b)|} \sum_{i=1}^{[O(a)]} \sum_{j=1}^{[O(b)]} R(O_i(a), O_j(b)) \]

Note that the limit of \( R_k(\ast, \ast) \) with respect to \( k \) right satisfies the recursive P-Rank equation, shown in Equation (3.1).

4. (Uniqueness) Suppose \( s_1(\ast, \ast) \) and \( s_2(\ast, \ast) \) are two solutions to the \( n^2 \) iterative P-Rank equations. For any entities \( x, y \in G \), let \( \delta(x, y) = s_1(x, y) - s_2(x, y) \) be their difference. Let \( M = \max_{x,y} |\delta(a, b)| \) be the maximum absolute value of any difference. We need to show that \( M = 0 \). Let \( |\delta(x, y)| = M \) for some \( a, b \in G \). It is obvious that \( M = 0 \) if \( a = b \). Otherwise,

\[
\delta(a, b) = s_1(a, b) - s_2(a, b)
\]

\[
= \frac{\lambda C}{|I(a)||I(b)|} \sum_{i=1}^{[I(a)]} \sum_{j=1}^{[I(b)]} (s_1(I_i(a), I_j(b)) - s_2(I_i(a), I_j(b)))
\]

\[
+ \frac{(1 - \lambda)C}{|O(a)||O(b)|} \sum_{i=1}^{[O(a)]} \sum_{j=1}^{[O(b)]} (s_1(O_i(a), O_j(b)) - s_2(O_i(a), O_j(b)))
\]

Thus,

\[
M = |\delta(a, b)|
\]

\[
= \left| \frac{\lambda C}{|I(a)||I(b)|} \sum_{i=1}^{[I(a)]} \sum_{j=1}^{[I(b)]} \delta(I_i(a), I_j(b)) + \frac{(1 - \lambda)C}{|O(a)||O(b)|} \sum_{i=1}^{[O(a)]} \sum_{j=1}^{[O(b)]} \delta(O_i(a), O_j(b)) \right|
\]

\[
\leq \left| \frac{\lambda C}{|I(a)||I(b)|} \sum_{i=1}^{[I(a)]} \sum_{j=1}^{[I(b)]} \delta(I_i(a), I_j(b)) \right| + \left| \frac{(1 - \lambda)C}{|O(a)||O(b)|} \sum_{i=1}^{[O(a)]} \sum_{j=1}^{[O(b)]} \delta(O_i(a), O_j(b)) \right|
\]

\[
\leq \frac{\lambda C}{|I(a)||I(b)|} \sum_{i=1}^{[I(a)]} \sum_{j=1}^{[I(b)]} |\delta(I_i(a), I_j(b))| + \frac{(1 - \lambda)C}{|O(a)||O(b)|} \sum_{i=1}^{[O(a)]} \sum_{j=1}^{[O(b)]} |\delta(O_i(a), O_j(b))|
\]

\[
\leq \frac{\lambda C}{|I(a)||I(b)|} \sum_{i=1}^{[I(a)]} \sum_{j=1}^{[I(b)]} M + \frac{(1 - \lambda)C}{|O(a)||O(b)|} \sum_{i=1}^{[O(a)]} \sum_{j=1}^{[O(b)]} M
\]

\[
= CM
\]

So \( M = 0 \) when \( C \neq 1 \). \( \square \)
Theorem 3.3.1 demonstrates four important properties of P-Rank. For any vertices \( a, b \in G \), the iterative P-Rank between \( a \) and \( b \) is the same as that between \( b \) and \( a \), i.e., P-Rank is a symmetric measure, as mentioned in Property 1 (Symmetry). Property 2 (Monotonicity) shows that the iterative P-Rank is non-decreasing during similarity computation. However, the solution will not go to infinity. Property 3 (Existence) and 4 (Uniqueness) guarantee that there exists a unique solution to \( n^2 \) iterative P-Rank equations, which can be reached by iterative computation to a fixed point, i.e., the solution to iterative P-Rank converges to a limit which satisfies the recursive P-Rank equation, shown in Equation (3.1):

\[
\forall \ a, b \in G, \lim_{k \to \infty} R_k(a, b) = s(a, b) \tag{3.5}
\]

In real applications, iterative P-Rank converges very fast (details are shown in Section 3.4). Empirically, we can choose to fix a small number of iterations \( k \approx 5 \) to derive P-Rank for all pair of vertices in real world information networks.

### 3.3.3 Derivatives of P-Rank

Besides its semantic completeness with a consideration of both in- and out-link relationships in similarity computation, P-Rank outperforms other structural similarity measures by its generality and flexibility. As shown in Figure 3.3, most of the state-of-the-art structural
similarity measures proposed so far for information networks are illustrated in a structural
similarity matrix. Among all measures shown in Figure 3.3 P-Rank enjoys the most general
form, from both the semantic completeness perspective and the structure perspective. All
other measures, such as CoCitation, Coupling, Amsler and SimRank, are just simplified special
cases of P-Rank and can be easily derived from P-Rank. P-Rank therefore provides a unified
framework for structural similarity computation in information networks. By analyzing the
iterative P-Rank shown in Equation (3.4), we can draw the following conclusions:

1. When $k = 1$, $C = 1$ and $\lambda = 1$, P-Rank is reduced to CoCitation.

2. When $k = 1$, $C = 1$ and $\lambda = 0$, P-Rank is reduced to Coupling.

3. When $k = 1$, $C = 1$ and $\lambda = 1/2$, P-Rank is reduced to Amsler, which subsumes
   both CoCitation and Coupling. Amsler can be regarded as a one-step P-Rank without
   similarity propagation.

4. When $k \to \infty$ and $\lambda = 1$, P-Rank boils down to SimRank, which is an iterative form of
   CoCitation with no out-link similarity considered.

5. When $k \to \infty$ and $\lambda = 0$, P-Rank is degenerated to a new structural similarity measure,
   which is an iterative form of Coupling with no in-link similarity involved. Since this new
   measure considers out-link relationships only and is the counterpart of SimRank, we
   name it rvs-SimRank, short for reverse-SimRank. In real world information networks,
   rvs-SimRank is more practical and useful than SimRank, because entities of a massive
   information network are usually widely distributed. It is prohibitive to maintain a
   global view of the whole information network for similarity computation. An entity
   may have a good knowledge of what entities are referenced by it, but it is hard to know
   what entities are referencing it without a full scan of the entire information network.
   For example, a Web page contains hyperlinks to other Web pages for its own sake, but
   it is impossible to know which Web pages are hyperlinking it without examining the

---

26
whole Web beforehand. This becomes even more severe when information networks are dynamically changing over time. So, rvs-SimRank is more robust and adaptive for measuring structural similarity over large yet dynamically changing information networks.

In real applications, P-Rank can be adapted flexibly to different information network settings, as long as there exist enough inter-linked relationships between entities. Even when the information network to be studied has sparse in-link information or biased edge distribution where SimRank may fail, P-Rank still can work well in modeling structural similarities.

Another important issue is to select appropriate values for parameters $C$, $\lambda$ and $k$ in P-Rank computation. $C$ represents the degree of attenuation in similarity propagation, and $\lambda$ expresses the relative weight of similarity computation between in-link and out-link directions. A priori knowledge of the information network usually helps select the values of $C$ and $\lambda$. By sampling a set of subgraphs from the original information network, we can also learn the characteristics of the underlying graph, so that $C$ and $\lambda$ can be set based on the sampled subgraphs as an approximation. The convergence of iterative P-Rank is fast with only several iterations of computation, so $k$ is usually set empirically as a small constant number. In Section 3.4, we will systematically study the effects of different parameters on P-Rank computation.

### 3.3.4 Computing P-Rank

Based on Section 3.3.2, the solution to the recursive P-Rank formula (Equation (3.1)) can be reached by computing its iterative form (Equation (3.4)) to a fixed point. Algorithm 1 illustrates the iterative procedure for computing P-Rank in an information network, $G$. Let $n$ be the number of vertices in $G$ and $k$ be the number of iterations executed until P-Rank converges to its fixed point. For every vertex pair $(a, b)$, an entry $R(a, b)$ maintains the
Algorithm 1 P-Rank \((G, \lambda, C, k)\)

**Input:** The information network \(G\), the relative weight \(\lambda\), the damping factor \(C\), the iteration number \(k\)

**Output:** P-Rank score \(s(a, b), \forall a, b \in G\)

begin

\[
\text{foreach } a \in G \text{ do}
\]
\[
\text{foreach } b \in G \text{ do}
\]
\[
\text{if } (a == b) \text{ then}
\]
\[
R(a, b) = 1
\]
\[
\text{else}
\]
\[
R(a, b) = 0
\]

while \((k > 0)\) do

\[
k \leftarrow k - 1
\]

\[
\text{foreach } a \in G \text{ do}
\]
\[
\text{foreach } b \in G \text{ do}
\]
\[
in \leftarrow 0
\]
\[
\text{foreach } i_a \in I(a) \text{ do}
\]
\[
\text{foreach } i_b \in I(b) \text{ do}
\]
\[
in \leftarrow in + R(i_a, i_b)
\]
\[
R^*(a, b) \leftarrow \lambda \cdot \frac{C_{in}}{|I(a)||I(b)|}
\]
\[
out \leftarrow 0
\]
\[
\text{foreach } o_a \in O(a) \text{ do}
\]
\[
\text{foreach } o_b \in O(b) \text{ do}
\]
\[
out \leftarrow out + R(o_a, o_b)
\]
\[
R^*(a, b) += (1 - \lambda) \cdot \frac{C_{out}}{|O(a)||O(b)|}
\]

return \(R(*, *)\)

end
intermediate P-Rank score of \((a, b)\) during iterative computation. Because the \((k + 1)\)-th iterative P-Rank score is computed based on P-Rank scores in the \(k\)-th iteration, an auxiliary data structure \(R^*(a, b)\) is maintained accordingly. As proven in Theorem 3.3.1, \(R_k(a, b) = R_k(b, a)\), so only one order for each pair is stored explicitly. In real implementation, either sparse matrixes or hash tables can be chosen as core data structures for \(R(\cdot, \cdot)\) and \(R^*(\cdot, \cdot)\).

Because \(G\) can be so large as not to be held in main memory, any advanced data structures that optimize external memory accesses can be accordingly applied.

Algorithm 1 first initializes \(R_0(a, b)\) based on Equation (3.3) (Lines 1 – 7). During iterative computation, P-Rank in \((k + 1)\)-th iteration, \(R^*(\cdot, \cdot)\), is updated by \(R(\cdot, \cdot)\) in the \(k\)-th iteration, based on Equation (3.4) (Lines 10 – 21). Then \(R(\cdot, \cdot)\) is substituted by \(R^*(\cdot, \cdot)\) for further iteration (Lines 22 – 24). This iterative procedure stabilizes rapidly and converges to a fixed point within a small number of iterations. A typical call to the algorithm can be P-Rank \((G, 0.5, 0.8, \lceil \ln(n) \rceil)\), where the relative weight \(\lambda\) is set to be 0.5 and the damping factor \(C\) is set to be 0.8.

The space complexity of Algorithm 1 is \(O(n^2)\), the amount to store intermediate and final P-Rank scores of \(G\), i.e., the sizes of \(R^*(\cdot, \cdot)\) and of \(R(\cdot, \cdot)\). Let \(d_1\) and \(d_2\) be the average in-degree and out-degree over all vertices of \(G\), respectively, the time complexity of the algorithm is \(O(k(d_1^2 + d_2^2)n^2)\), and the worst case time complexity can be \(O(n^4)\). In comparison with SimRank whose space and time complexities are \(O(n^2)\) and \(O(n^4)\), P-Rank has the same space and time complexities.

In [78], the authors improved the time complexity of SimRank from \(O(n^4)\) to \(O(n^3)\). The same memoization based algorithms can be applied in the same way on P-Rank to reduce its time complexity to \(O(n^3)\). In [41], the authors suggested a scalable framework for SimRank computation based on the Monte Carlo method. Essentially their computation is probabilistic and the SimRank scores computed are approximation to the exact answers. In order to make full use of the characteristics of different information networks, we propose different pruning algorithms to efficiently compute P-Rank.
**Homogeneous Information Network:** In homogeneous information networks, all vertices are of the same type. One way to reduce the space/time complexities in this scenario is to prune less similar vertex pairs while not deteriorating the accuracy of similarity computation too much. For \( n^2 \) vertex-pairs of \( G \), only those adjacent to each other (say, vertices within a radius of 3 or 4) are similar, while those whose neighborhood have little or no overlap are far apart and inevitably not similar. Thus *radius-based pruning* [61] can be used to set the similarity between two vertices far apart to be 0, and only those vertex-pairs within a radius of \( r \) from each other in the underlying undirected graph \( G' \) are considered in similarity computation. Given a vertex \( u \in G \), let there be \( d_r \) such neighbors of \( u \) within a radius \( r \) in the underlying undirected graph \( G' \) on average, then there will be \((n \ast d_r)\) vertex-pairs considered. The space and time complexities become \( O(n \ast d_r) \) and \( O(k(d_1^2 + d_2^2)d_r n) \), respectively. Since \( d_r \) is likely to be much less than \( n \), if \( r \) is small w.r.t. \( n \), we can think of this approximate algorithm as being linear with a possibly large constant factor.

**Heterogeneous Information Network:** In heterogeneous information networks, vertices belong to different categories. Given two vertices \( u, v \in G \), it is meaningless to measure structural similarity between \( u \) and \( v \) if they belong to different categories. Thus the pruning technique in this scenario, called *category-based pruning*, is to set the similarity between two vertices belonging to different categories to be 0, and consider only those vertex pairs within the same category. Let there be \( c \) different categories over the vertices of \( G \), and for each category \( i \), there be \( n_i \) vertices included, where \( 1 \leq i \leq c \), then the total number of vertex pairs is \( \sum_{i=1}^{c} n_i^2 \). The space and time complexities then become \( O(\sum_{i=1}^{c} n_i^2) \) and \( O(k(d_1^2 + d_2^2)(\sum_{i=1}^{c} n_i^2)) \). Notice the following inequality holds:

\[
n^2 = \left(\sum_{i=1}^{c} n_i\right)^2 \geq \sum_{i=1}^{c} n_i^2
\]

*Category-based pruning* can eliminate a huge number of vertex pairs belonging to different categories, especially when \( c \) is large. If the number of vertices in a specific category is still
so large that they cannot be held in main memory, radius-based pruning can be further applied within this category to facilitate the computation. [114] [15] presented an advanced index-based algorithm, SimTree, for fast computation of similarity scores in heterogeneous information networks if vertices in every category are hierarchically organized. Our category-based pruning method is actually the specialized one-level SimTree.

3.4 Experiment

In this section, we report our experimental studies for P-Rank as a structural similarity measure over different information networks. We show the power of P-Rank in comparison with the state-of-the-art structural similarity measure, SimRank. In addition, the experiments illustrate the feasibility and efficiency of the P-Rank algorithm with pruning techniques in information networks with different diversity and scale.

We ran our experiments on two different data sets: one is real data from DBLP\footnote{http://www.informatik.uni-trier.de/~ley/db/} and the other is synthetic [21]. For the real data set, we further generate two different information networks: one is a heterogeneous and the other is a homogeneous. All our experiments are performed on an Intel PC with a 2.4GHz CPU, 2GB memory, running Redhat Fedora Core 4. All algorithms including P-Rank and SimRank are implemented in C++ and compiled by gcc 3.2.3. For ease and fairness of comparison, we set the damping factor $C = 0.8$ for both SimRank and P-Rank; The relative weight $\lambda$ is set to be 0.5 for P-Rank, if not specified explicitly. All the values of parameters for SimRank are set in accordance with [61].

3.4.1 A Heterogenous DBLP Information Network

We first build a heterogeneous information network from DBLP. The downloaded DBLP data had its time stamp on March 15th, 2008. The heterogeneous information network, $G$, contains four different kinds of vertices: paper, author, conference and year. If a paper $p$ is
Figure 3.4: The Schema of Heterogeneous DBLP Information Networks. (The Number Represents the Number of Vertices in the Corresponding Category)

written by an author $a$, there exists a directed edge from $p$ to $a$; If an author $a$ participated in a conference $c$, there exists a directed edge from $a$ to $c$; For a specific year $y$, there are bidirectional edges between both $p$ and $y$ and $c$ and $y$, if the paper $p$ was published in conference $c$ in year $y$. Figure 3.4 illustrates the global schema of the heterogenous information network, $G$. The number of vertices in $G$ is 218930 and the number of edges is 818301. More specifically, the number of paper vertices is 211607; the number of author vertices is 4979; the number of conference vertices is 2292 and the number of year vertices is 52.

In order to evaluate the effectiveness of P-Rank, we choose to test how different structural similarity measures perform in clustering authors in $G$. It is worth noting that P-Rank is not confined only in clustering applications. Any data management applications adopting structural similarity as an underlying function can make use of P-Rank as its similarity measure. Meanwhile, P-Rank is orthogonal to the specific clustering algorithms applied, i.e., P-Rank proposes a general structural similarity measure which can be applied in most existing clustering algorithms. We plug P-Rank and SimRank into K-Medoids \cite{69}, respectively. The structural distance between two vertices $u, v \in G$ is defined as

$$d_f(u, v) = 1 - s_f(u, v) \quad (3.6)$$

where $s_f(u, v)$ is the similarity score generated by the similarity function, $f$, (either $p$ for
Figure 3.5: Compactness of \textit{P-Rank} and of \textit{SimRank} in Heterogeneous DBLP

\textit{P-Rank} or \(s\) for \textit{SimRank}). We define \textit{compactness} of the clustering results, \(C_f\), as

\[
C_f = \frac{\sum_{i=1}^{K} \sum_{x \in C_i} d(x, m_i)}{\sum_{1 \leq i < j \leq K} d(m_i, m_j)}
\]

(3.7)

where \(K\) is the number of clusters to be generated\(^3\), \(C_i\) is the \(i\)-th cluster; \(m_i, m_j\) are centers for cluster \(i\) and cluster \(j\), respectively. Intuitively, the numerator of Equation (3.7) describes \textit{intra-cluster} distances and the denominator represents \textit{inter-cluster} distances. Smaller \(C_f\) values demonstrate better clustering performance. In the following experiments, we compare \(C_p\) and \(C_s\) for \textit{P-Rank} and \textit{SimRank}, respectively.

We run both \textit{P-Rank} and \textit{SimRank} over \(G\) until the similarity scores converge. We then cluster author vertices by \(K\)-Medoids algorithm, and \(K = 10\). At the beginning, we randomly choose 10 author vertices (without replacement) as initial centers of clusters and run the \(K\)-Medoids algorithm. We perform \(l = 10\) trials and the clustering results are shown in Figure 3.5. As illustrated, \textit{P-Rank} consistently achieves more compact clustering results than does \textit{SimRank}. The main reasons are as follows: (1) \textit{P-Rank} considers similarity propagation from both in-links (paper vertices) and out-links (conference vertices), and thus generates more comprehensive results than does \textit{SimRank} for clustering authors; (2) By simply considering in-link propagation only, \textit{SimRank} fails to measure quite a few vertex pairs in \(G\). For \textit{SimRank}, only those authors who cooperate (either directly or indirectly) on some papers

\(^3\)Note \(K\) is different from \(k\) in Equation (3.4), which is the number of iterations performed for iterative \textit{P-Rank}.\]
have significant similarity scores, while others are regarded as dissimilar. In comparison, P-Rank is more robust than SimRank. For two author vertices, although they may not cooperate with each other (no in-link propagation), as long as they participate in common conferences (there exists out-link propagation), they are regarded as similar to some extent. Therefore, quite a few vertices which are dissimilar under SimRank’s scheme are now similar in P-Rank, thus leading to an improvement to the compactness of clustering results.

We then test the algorithmic nature and mathematical property of P-Rank. Figure 3.6(a) plots structural similarity scores of author pairs w.r.t. the number of iterations performed. The scores are averaged by the top 10 highest ranked scores of author pairs for P-Rank and SimRank, respectively. We see from the figure that the intermediate similarity scores $R_k(\ast, \ast)$ become more accurate on successive iterations. Iteration 2, which computes $R_2(\ast, \ast)$ from $R_1(\ast, \ast)$, can be thought of as the first iteration taking advantage of the recursive power of algorithms for similarity computation. Subsequent changes become increasingly minor, suggesting a rapid convergence. The figure also manifests that the iterative process stabilizes very fast, when $k$ is greater than 5. Figure 3.6(b) plots the structural similarity scores of P-Rank and SimRank w.r.t. the rank number, $N$. The downward curves for both P-Rank and SimRank present a decrease in structural similarity as $N$ increases, which is expected because highly ranked authors are more similar.

We further examine the ground truth generated by P-Rank on author vertices of $G$ to test
if it really reflects the reality to single out similar authors from the DBLP data set. Although the judgement of similarity might be quite subjective and difficult even for humans, we still find very interesting results by making use of P-Rank. As illustrated in Figure 3.7(a), the top-10 highly ranked author pairs are listed. We may notice that the author pairs with high P-Rank scores share some common characteristics. First, they are usually co-authors or share quite a few authorities as co-authors. And they are purely dedicated in specific research fields. In the mean time, highly ranked authors are inclined to be clustered into a close related community, in which their authorities are further reinforced. That is also another reason why P-Rank outperforms SimRank in entity clustering, as illustrated in the aforementioned experiment. We further issue k-Nearest Neighbor (KNN) queries to retrieve top-k most similar authors in G, given an author vertex q as a query. Figure 3.7(b) shows the ranked results for the query “Philip S. Yu” and Figure 3.7(c) shows the ranked results for the query “Michael Stonebraker”, where k = 10. As illustrated, both results are quite intuitive and conform to our basic understandings. Therefore, P-Rank can be effectively used as an underlying metric for measuring structural similarity in heterogenous information networks, and its results obey our common sense pretty well.

3.4.2 A Homogenous DBLP Information Network

We continue generating a homogeneous information network, G, on the DBLP data set. The vertex set of G is composed of a subset of papers in DBLP and a directed edge exists from paper u to paper v if u cited v. The number of vertices in G is 21740, and the number of
Our first experiment is to study how the different structural similarity measures perform in clustering vertices in homogenous DBLP. We plug P-Rank and SimRank respectively as underlying similarity functions into K-Medoids ($K = 10$). We randomly choose 10 vertices (without replacement) as initial centers of clusters and run the $K$-Medoids algorithm. We perform $l = 10$ trials and the clustering results are shown in Figure 3.8. As illustrated, P-Rank can achieve much better results in clustering vertices in $G$. The improvement can be at least 6 times, and the clustering performance of P-Rank is consistently stable in different experimental trials.

Different from heterogenous information networks, homogeneous information networks have their vertices in one unique category and every vertex pair is eligible for comparison in the P-Rank framework. However, SimRank may fail simply because there might be no
Figure 3.10: Vertex Pair Distributions \textit{w.r.t.} Similarity Scores in Synthetic Data Set

common in-link similarity flows for vertex pairs. The problem becomes even more severe when the information is massive and the interlinked relationships are not evenly distributed. As illustrated in Figure 3.9 vertex pairs are reorganized into different histograms based on their structural similarity scores computed by \textbf{P-Rank} and \textbf{SimRank}, respectively. For example, vertex pairs whose structural similarity scores are between [0.1, 0.2) are put in the third histogram. A special histogram “N/A” represents vertex pairs whose structural similarity can not be measured properly. Because of the very biased information considered during similarity computation, \textbf{SimRank} fails to generate meaningful similarity measures for a majority of vertex pairs in the homogenous DBLP, as shown in the histogram “N/A”. However, \textbf{P-Rank} still works well and is robust enough in structural similarity computation.

### 3.4.3 Synthetic Data Sets: R-MAT

We generate a synthetic homogeneous information network $G$ based on the \textit{Recursive Matrix} (R-MAT) model \cite{21}, which naturally follows power-law (in- and out-)degree distributions for $G$. The homogeneous information network $G$ is a directed graph with $10^5$ vertices and $6 \times 10^5$ edges.

In this homogenous network, we first test how \textbf{P-Rank} and \textbf{SimRank} perform when measuring structural similarity of vertices in $G$. As illustrated in Figure 3.10 vertex pairs are distributed to different histograms with different similarity score intervals. Similar to Fig-
Figure 3.11: P-Rank v.s. Different Parameters

SimRank again fails to deliver meaningful structural similarity for a majority of vertex pairs, as shown in the histogram “N/A”. However, P-Rank can successfully measure structural similarity for every pair of vertices in G.

We are also interested in how different parameters affect P-Rank when computing similarity in the homogeneous information network. We first test how the damping factor C is correlated with P-Rank. Figure 3.11(a) illustrates P-Rank scores in G w.r.t. the number of iterations performed. The structural similarity scores are averaged by the top 10 highest ranked scores of vertex pairs. The damping factor, C, is set to be 0.2, 0.5 and 0.8, and three curves are plotted, respectively. It is obvious that P-Rank grows proportionally with the increase of C. When C = 0.2, P-Rank converges fast when the number of iterations, k, is larger than 2. However, when C = 0.8, P-Rank converges approximately at the 7th iteration of computation. The reason is that when C is set to a small value, the recursive power of P-Rank will be weakened and only vertices nearby can contribute in the structural similarity computation. When C is set high, more vertices in G can participate in the process of recursive computation, so P-Rank scores can be accumulated more easily and the convergence will take more time.

We then test how the relative weight, λ, has an impact on P-Rank. As discussed in Section 3.3.2, λ trades off P-Rank between the in-link and out-link relationships. When λ = 1, P-Rank is equal to SimRank. And when λ = 0, P-Rank is equal to rvs-SimRank.
As shown in Figure 3.11(b), the curve representing $\lambda = 0.5$ lies between curves representing $\lambda = 0$ (rvs-SimRank) and $\lambda = 1$ (SimRank). It means that when $\lambda = 0.5$, P-Rank well balances both in-link and out-link factors for measuring structural similarity. When $\lambda = 0.2$, the out-link relationships are still more important than the in-link ones, and P-Rank is interpolated by similarity scores from both sides. However, the curve representing $\lambda = 0.2$ is quite close to the rvs-SimRank curve ($\lambda = 0$). A similar phenomenon occurs for the curve representing $\lambda = 0.8$, which is quite close to the SimRank curve.

3.5 Conclusion

In this chapter we propose a comprehensive structural similarity measure, P-Rank, for large information networks. We start with the basic philosophy of P-Rank that two entities of an information network are similar if (1) they are referenced by similar entities, and (2) they reference similar entities. In comparison with other structural similarity measures, P-Rank takes into account of both in- and out-link relationships of entity pairs and penetrates the structural similarity computation beyond neighborhood of vertices to the entire network. The advantages of P-Rank are its semantic completeness, robustness and flexibility under different information network settings. P-Rank has shown to be a unified framework for structural similarity measures, under which the state-of-the-art similarity measures as CoCitation, Coupling, Amsler and SimRank are all its special cases. We present a fixed point algorithm for computing P-Rank. Efficient pruning techniques under different network settings are discussed to reduce the space and time complexity of P-Rank. We perform extensive experimental studies on both real data sets and synthetic data sets and the results confirm the applicability and comprehensiveness of P-Rank, as well as its significant improvement over other structural similarity measures.
Chapter 4

Graph Query Optimization in Information Networks

4.1 Introduction

Recent years have witnessed a rapid proliferation of information networks, most of which can be naturally modeled as large graphs [28]. The burgeoning size and heterogeneity of information networks have inspired extensive interest in supporting effective querying methods in real applications that are centered on massive graph data. At the core of many advanced network operations, lies a common and critical graph query primitive: given an information network modeled as a large graph $G$, and a user-specified query graph $Q$, we want to retrieve as output the set of subgraphs of $G$, each of which is isomorphic to $Q$. As a key ingredient of many network applications, the graph query is frequently issued in various domains:

1. In a large protein-protein interaction network, it is desirable to find all protein substructures that contain an $\alpha$-$\beta$-barrel motif, which is specified as a cycle of $\beta$ strands embraced by another cycle of $\alpha$ helices [57];

2. In a large software program which is modeled as large static or dynamic call graphs, software engineers want to locate a suspicious bug which arises as a distortion in the control flow and can be represented as a graph as well [37];

3. In a bibliographic information network, such as DBLP, users are always eager to extract coauthor information in a specified set of conference proceedings [87]. A co-authorship graph is therefore reported as the graph query result.
Unfortunately, the graph query problem is hard in that (1) it requires subgraph isomorphism checking of $Q$ against $G$, which has proven to be NP-complete \[44\]; (2) the heterogeneity and sheer size of information networks hinder a direct application of well-known graph matching methods \[42, 83, 92, 97, 108\], most of which are designed on special graphs with no or limited guarantee on query performance and scalability support. Due to the lack of scalable graph indexing mechanisms and cost-effective graph query optimizers, it becomes hard, if not impossible, to search and analyze any reasonably large information networks. For example, browsing and crosschecking biological networks depicted simultaneously in multiple windows is by no means an inspiring experience for scientists. Therefore, there is a growing need and strong motivation to take advantage of well-studied database indexing and query optimization techniques to address the graph query problem on the large information network scenario.

In this chapter, we present SPath, a new graph indexing technique towards resolving the graph query problem efficiently on large information networks. SPath maintains for each vertex of the network a neighborhood signature, a compact indexing structure comprising decomposed shortest path information within the vertex’s vicinity. As a basic graph indexing unit, neighborhood signature demonstrates considerable merits in that (1) neighborhood signature is very space-efficient (\(O(1)\) for each vertex), which makes it possible to scale SPath up in large scale information networks; (2) neighborhood signature preserves local structural information surrounding vertices, which is especially useful for search space pruning before costly subgraph matching; (3) neighborhood signature based graph indexing, SPath, revolutionizes the way of graph query processing from vertex-at-a-time to path-at-a-time, which proves to be more cost-effective than traditional graph matching methods.

With the aid of SPath, we decompose a query graph into a set of shortest paths, among which a subset of candidate paths with high selectivity is picked by a graph query optimizer. These candidate paths are required to properly cover the original query graph, i.e., for each edge in $Q$, it should belong to at least one candidate path selected. The query is further
processed by joining candidate shortest paths in order to reconstruct the original query graph. Here the graph matching is performed in a path-at-a-time fashion and SPath plays a key role in shortest path reconstruction and orientation in the large network. To the best of our knowledge, SPath is the first scalable graph indexing mechanism which supports effective path-at-a-time graph query processing on large networks, and thus achieves far better query performance, compared with other traditional vertex-at-a-time graph matching methods. Our main contributions can be summarized as follows:

1. We propose a pattern based graph indexing framework to address the graph query problem on large scale information networks. A query cost model is formulated to help evaluate different structural patterns for graph indexing in a qualitative way. As a result, decomposed shortest paths are considered as feasible indexing features in the large information network scenario (Section 4.4);

2. We propose a new graph indexing technique, SPath, which makes use of neighborhood signatures of vertices as the basic indexing structure. SPath has demonstrated an effective search space pruning ability and high scalability in large information networks (Section 4.5);

3. We design a graph query optimizer to help address graph queries in a path-at-a-time manner. With the aid of SPath, the graph query processing is facilitated by joining a set of shortest paths with good selectivity (Section 4.6);

4. We present comprehensive experimental studies on both real and synthetic information networks. Our experimental results demonstrate that SPath outperforms a state-of-the-art graph query method, GraphQL [57]. Moreover, SPath exhibits excellent scalability and practicability in large scale information networks (Section 4.7).
4.2 Related Work

The field of graph data management has seen an explosive spread in recent years because of new applications in bioinformatics, social and technological networks, communication networks, software engineering and the Web. It becomes increasingly important to manage graphs, especially large graphs, in DBMSs. However, existing database models, query languages and access methods, such as the relational model and SQL, lack native support for large graphs. The wave of graph-based applications calls for new models, languages and systems for large graph-structured networks.

Recent research has embraced the challenges of designing special-purpose graph databases. Generally, there are two distinct categories that are often referred to as the graph-transaction setting and the single-graph setting, or network setting. In the graph-transaction setting, a graph database consists of a set of relatively small graphs as transactions. Whereas in the single-graph setting, the data of interest is a single large graph. In both settings lies a common and critical graph query problem, which can be formulated as a selection operator on graph databases and has been studied first in the theoretical literature as the subgraph isomorphism problem \[42, 83, 92, 108\]. Subgraph isomorphism has proven to be NP-complete \[44\].

In a graph-transaction database, the graph query problem is to select all graphs in the database which contain the query graph as subgraph(s). The major challenge in this scenario is to reduce the number of pairwise subgraph isomorphism checkings. A number of graph indexing techniques have been proposed to address this challenge \[101, 113, 56, 116, 118, 24, 100\]. Different structural patterns are examined to help prune the candidate search space at the first step. Costly subgraph isomorphism checking is verified in the second step on the pruned search space, rather than on all the transactions of the graph database.

Although the graph query problem has been studied extensively in the graph-transaction setting, little attention \[106, 57, 117\] has been paid to improve the effectiveness of graph query
processing in the single-graph setting. In this scenario, a graph query retrieves as output the complete set of occurrences of the query graph in a large information network. Note the graph query problem in this setting is more general, in that a set of small graphs can be regarded as a large graph with different disconnected components. So an efficient solution in the single-graph setting will definitely help solve the graph query problem in the transaction setting. The challenge in this scenario is to accelerate the subgraph isomorphism testing itself. To develop effective and scalable techniques that address the graph query problem in the single graph setting is the focus of our work.

A straightforward approach to managing large networks is to store the underlying graph structure in general-purpose relational tables and make use of built-in SQL queries for the graph query. Oracle is currently the only commercial DBMS that provides internal support for graph data [1]. However, the relational model and SQL are fundamentally inadequate to support graph queries on large networks. Queries are translated into a large number of costly join operations and the structural knowledge of graphs is broken down and flattened during the course of database normalization. This allows little opportunity for graph specific optimizations and prevents effective pruning on the search space. The number of intermediate false positives can grow excessively large, especially when the network examined is large and diverse.

In SAGA [106], the authors proposed an approximate graph matching method, which employed a flexible graph distance model to measure similarities between graphs. However, not all exact matchings, but a subset of approximate matchings were returned as answers. In GraphQL [57], the authors made use of neighborhood subgraphs for global pruning and vertex profiles for local pruning. A search order optimizer was designed to jointly reduce the search space for each vertex in the query graph. Their experiments demonstrated that the graph-specific optimizations proposed by GraphQL outperformed an SQL-based implementation by orders of magnitude for graph queries on large information networks. GADDI [117] proposed a distance index-based matching method which were specifically used for biolog-
ical networks and small social networks. The basic indexing unit is the NDS (neighboring discriminating substructure) distance for every pair of vertices in the graph. The costly frequent graph mining algorithm was adopted to help mine the discriminative subgraphs. All of the aforementioned methods have common problems: 1. they all target on pruning the search space of each vertex, such that the whole search space can be jointly reduced as much as possible. However, the query processing is still performed in a vertex-at-a-time way, which is extremely inefficient. 2. the methods proposed can only support graph queries in small networks or networks in specialized areas, whereas they cannot generalize and scale up to real large networks.

Similar graph queries were proposed on large RDF graphs [16]. A RDF database contains millions of RDF tuples \((s, p, v)\) where \(s\) is a subject, \(p\) is a property and \(v\) is a value. Every RDF database has an associated RDF graph where vertices correspond to subjects and values, and the edges linking them are labeled with a property. A graph query expressed in SPARQL language may contain some variable vertices which can be substituted by either subject or value vertices in the RDF graph. Note a RDF graph contains vertices in two categories: subject and value, and each vertex bears a distinct label. So RDF graphs are a special kind of information networks in our study and our indexing method is more general and can be easily extended to answer graph queries on large RDF graphs.

Besides the graph query problem, other kinds of queries were proposed on large graph databases as well. Graph reachability queries [107, 65, 64] examine whether there exist path connections from a vertex \(u\) to another vertex \(v\) in a large directed graph. Keyword search [60, 67, 55] over large graphs explores the graph structure and finds subgraphs that contain all the keywords in the query. Connection-preserving pattern matching queries [120] relax the subgraph isomorphism constraints by allowing two adjacent vertices in the query graph to be mapped to two vertices within distance \(\delta\) in the network. It is believed that more queries of practical use, together with the corresponding query processing techniques, will be proposed and studied toward a better understanding of information networks.
4.3 Problem Definition

An information network can be modeled as a graph $G = \{V, E, \Sigma, l\}$ where $V$ is a set of vertices and $E \subseteq V \times V$ is a set of edges. $\Sigma$ is a vertex label set and $l : V \rightarrow \Sigma$ denotes the vertex labeling function. For ease of notation, the vertex set of $G$ is denoted as $V(G)$ and its edge set is denoted as $E(G)$. The size of $G$ is defined as $|V(G)|$, the size of its vertex set. Analogously, the graph queries posed upon the network can be modeled as graphs as well. In this chapter, we focus our study on the case of connected, undirected simple graphs with no weights assigned on edges. Without loss of generality, our methods can be easily extended to other kinds of graphs.

A graph $G'$ is a subgraph of $G$, denoted as $G' \subseteq G$, if $V(G') \subseteq V(G)$, $E(G') \subseteq E(G)$ and $\forall (u, v) \in E(G'), u, v \in V(G')$. We alternatively say that $G$ is a supergraph of $G'$ and $G$ contains $G'$. Subgraph isomorphism is an injective function $f$ defined from $G'$ to $G$ such that (1) $\forall v \in V(G'), l'(v) = l(f(v))$; and (2) $\forall (u, v) \in E(G'), (f(u), f(v)) \in E(G)$, where $l'$ and $l$ are the labeling functions of $G'$ and $G$, respectively. Under these conditions, $f$ is often referred to as a matching of $G'$ in $G$.

**Definition 3. (GRAPH QUERY)** Given a network $G$ and a query graph $Q$, the graph query problem is to find as output all distinct matchings of $Q$ in $G$.

**Example 4.3.1:** Figure 4.1(a) and Figure 4.1(b) illustrate an information network sample
and a query graph sample $Q$, respectively. Here we use numeric identifiers to distinguish different vertices in a graph. A subgraph $G'$ of $G$ with $V(G') = \{8, 5, 7, 9\}$ colored in grey is isomorphic to $Q$ and hence returned as an answer to the graph query.

Note there may exist multiple matchings of $Q$ in $G$. For example, given a triangle graph $Q$ with $A, B, C$ as the label for each vertex, respectively. All the matchings of $Q$ in $G$, as shown in Figure 4.1(a), are $\{1, 2, 3\}$, $\{4, 2, 3\}$, $\{6, 5, 3\}$, $\{8, 5, 7\}$ and $\{11, 10, 7\}$. Subgraph isomorphism is known to be NP-complete [44].

4.4 The Pattern-based Graph Indexing Framework

In this section, we discuss the feasibility and principle of graph indexing toward addressing the graph query problem on large scale information networks. We first introduce a baseline algorithmic framework with no indexing techniques exploited (Section 4.4.1). In order to improve the query performance, we extend the framework by leveraging structural patterns for graph indexing (Section 4.4.2). A cost-sensitive model is then proposed to help evaluate different structural patterns qualitatively (Section 4.4.3). As a result, path-based graph indexing mechanism is selected as a feasible solution in large networks.

4.4.1 The Baseline Algorithmic Framework

A straightforward approach to answering the graph query $Q$ against a network $G$ is to explore a tree-structured search space considering all possible vertex-to-vertex correspondences from $Q$ to $G$. The search space traversal is halted until the structure of $Q$ implied by the vertex mapping does not correspond in $G$. Reaching a leaf node of the search space means successfully mapping all vertices of $Q$ upon $G$ without violating the structure and label constraints of subgraph isomorphism, and is therefore equivalent to having found a matching $Q$ in $G$. 

47
Algorithm 2 Baseline Algorithm

**Input:** Query graph $Q$, Network $G$

**Output:** All subgraph isomorphism mappings $f$ of $Q$ against $G$

begin
  for $v \in V(Q)$ do
    \[ C(v) \leftarrow \{ u | u \in V(G), l'(v) = l(u) \} \]
    Recursive_Search($v_1$)
  end

Procedure Recursive_Search($v_i$)

begin
  for ($u \in C(v_i))$ and ($u$ is unmatched) do
    if not Matchable($v_i$, $u$) then
      continue
      $f(v_i) \leftarrow u$; $u \leftarrow$ matched
    if $i < |V(Q)|$ then
      Recursive_Search($v_{i+1}$)
    else
      Output a mapping $f$
      $f(v_i) \leftarrow$ NULL; $u \leftarrow$ unmatched
  end

Function boolean Matchable($v_i$, $u$)

begin
  for $\forall$ edge $(v_i, v_j) \in E(Q), j < i$ do
    if edge $(u, f(v_j)) \notin E(G)$ then
      return false
  return true
end
end

**Definition 4.** (MATCHING CANDIDATE) $\forall v \in V(Q)$, the matching candidates of $v$ is a set $C(v)$ of vertices in $G$ sharing the same vertex label with $v$, i.e., $C(v) = \{ u | l(u) = l'(v), u \in V(G) \}$, where $l$ and $l'$ are vertex labeling functions for $G$ and $Q$, respectively.

Algorithm 2 presents the baseline algorithm for graph query processing on large networks [108]. We start with finding matching candidates $C(v)$ for each vertex $v$ in the query graph $Q$ (Lines 2 - 3). The matching candidates $C(v)$ is the set of vertices in the network each of which bears the same label with $v$, and the resulting product $\prod_{i=1}^{|V(Q)|} C(v_i)$ forms the total search space of the algorithm. The core procedure, Recursive_Search, matches $v_i$
over $C(v_i)$ (Line 11) and proceeds step-by-step by recursively matching the subsequent vertex $v_{i+1}$ over $C(v_{i+1})$ (Lines 12 – 13), or outputs a matching $f$ if every vertex of $Q$ has a counterpart in $G$ (Line 15). If $v_i$ exhausts all vertices in $C(v_i)$ and still cannot find a feasible matching, **Recursive Search** backtracks to the previous state for further exploration (Line 16).

Function **Matchable** examines the feasibility of mapping $v_i$ to $u \in V(G)$ by considering the preservation of structural connectivity (Lines 18 – 24). If there exist edges connecting $v_i$ with previously explored vertices of $Q$ but there are no counterpart edges in $G$, the **Matchable** test simply fails.

In the baseline algorithm, for each vertex $v \in V(Q)$, an exhaustive search of possible one-on-one correspondences to $u \in C(v)$ is required. Therefore, the total search space of the algorithm equals $\prod_{i=1}^{N} |C(v_i)|$, where $N = |V(Q)|$. The worst-case time complexity of the algorithm is $O(M^N)$ where $M$ and $N$ are the sizes of $G$ and $Q$, respectively. This is a consequence of subgraph isomorphism that is known to be NP-complete. In practice, the running time of graph query processing depends tightly on the size of the search space, $\prod_{i=1}^{N} |C(v_i)|$.

### 4.4.2 Structural Pattern Based Graph Indexing

It has been shown that answering graph queries is very costly, and it becomes even challenging when the information network examined is large and diverse. In order to alleviate the time-consuming exhaustive search in graph query processing, we consider reducing the search space size, $\prod_{i=1}^{N} |C(v_i)|$, in the following two aspects:

1. Minimize the number of one-on-one correspondence checkings, i.e., $\text{min } N$;

2. Minimize for each vertex of the query graph its matching candidates in $G$, i.e., $\text{min } |C(v_i)|, \forall v_i \in V(Q)$.

The two objectives motivate us to explore the possibility of leveraging structural patterns for graph indexing such that the search space size can be ultimately minimized. For structural
patterns, we mean any kind of substructures of a graph, such as paths, trees, and general subgraphs.

We begin considering the first objective to reduce $N$. Note in the baseline algorithm, $N = |V(Q)|$ because we need to consider one-on-one correspondence for each vertex of the query, i.e., the graph query is performed in a vertex-at-a-time manner. However, if we have indexed a set of structural patterns $p_1, p_2, \ldots, p_k \subseteq Q$ where $\forall e \in E(Q), \exists p_i, s.t., e \in p_i$ ($1 \leq i \leq k$), the graph query can be answered pattern-at-a-time by checking one-on-one correspondence on $p_i$ instead ($1 \leq i \leq k$), such that $N = k$. If $k < |V(Q)|$, we successfully reduce $N$ to achieve our goal. Extremely, if we’ve indexed the query $Q$ in advance, $N$ is minimized to 1 and we can answer the graph query in one shot. Usually we have $1 \leq N \leq |V(Q)|$.

We then examine how to achieve the second objective by reducing $|C(v_i)|$ for all $v_i \in V(Q)$. In the baseline algorithm, every $u \in C(v_i)$ is a potential matching vertex of $v_i$ and therefore needs to be matched temporarily for further inspection. However, a great many vertices in $C(v_i)$ have proven to be false positives eventually if the global structural constraints of subgraph isomorphism are cross-checked. So it is unnecessary to examine every vertex in $C(v_i)$ and it will be desirable if we can make use of structural patterns to help pre-prune false positives in $C(v_i)$, such that $|C(v_i)|$ can be reduced. Given a graph $G$ and $u \in V(G)$, we consider a neighborhood induced subgraph, $G^k_u$, which contains all vertices within $k$ hops away from $u$. This subgraph $G^k_u \subseteq G$ is referred as the $k$-neighborhood subgraph of $u$. We then pick structural patterns in $G^k_u$ based on the following theorem:

**Theorem 4.4.1:** If $Q \subseteq G$ w.r.t. a subgraph isomorphism matching $f$, for any structural pattern $p \subseteq Q^k_v, v_i \in V(Q)$, there must be a matching pattern, denoted as $f(p) \subseteq G$, s.t. $f(p) \subseteq G^k_{f(v)}, f(v_i) \in V(G)$.

**Proof:** Without loss of generality, we consider a subgraph $g$ in the $k$-neighborhood subgraph of $v$, $Q^k_v, v \in V(Q)$. We first prove that for any vertex $w \in V(g)$, its mapping $f(w) \in G^k_{f(v)}, i.e.,$ the vertex $f(w)$ is in the $k$-neighborhood subgraph of $f(v)$. Because
There exists a path \( p = v, \ldots, w \) with length \( k' \leq k \) connecting \( v \) and \( w \). Correspondingly, there exists a mapping path \( f(p) = f(v), \ldots, f(w) \) with length \( k' \) connecting \( f(v) \) and \( f(w) \) in \( G^k_{f(v)} \). So \( f(w) \) is at most \( k' \) hops away from \( f(v) \) (note \( f(w) \) is not necessarily the shortest path between \( f(v) \) and \( f(w) \)), i.e., \( \forall w \in V(g) \), its counterpart \( f(w) \) is in the \( k \)-neighborhood subgraph of \( f(v) \), \( G^k_{f(v)} \).

Then \( \forall e = (w, x) \in E(g) \), there exists a counterpart mapping edge \( e' = (f(w), f(x)) \) in the \( k \)-neighborhood subgraph of \( f(v) \) because \( Q \subseteq G \) w.r.t. \( f \). Therefore, \( f(g) \subseteq G^k_{f(v)} \), i.e., the mapping graph \( f(g) \) is in the \( k \)-neighborhood subgraph of \( f(v) \).

Intuitively, if there exists a structural pattern \( p \) in the \( k \)-neighborhood subgraph \( Q^k_{v_i} \) of \( v_i \in V(Q) \), whereas there is no such \( f(p) \) in the \( k \)-neighborhood subgraph \( G^k_u \) of \( u \in C(v_i) \), we can safely prune the false positive \( u \) from \( C(v_i) \), based on Theorem 4.4.1. It will be advantageous if we can index structural patterns from the \( k \)-neighborhood subgraphs of vertices in the network \( G \) before hand, such that false positives in \( C(v_i) \) can be eliminated before real graph matchings. Therefore we can achieve our second objective to reduce \( |C(v_i)| \).

It is worth mentioning that the baseline algorithm does not consider any structural patterns but vertex labels only for indexing. It is just a special case of our pattern based indexing mechanism if we set \( k = 0 \).

Interestingly, the two objectives are neither independent nor conflicting with each other. By extracting and indexing structural patterns from the \( k \)-neighborhood subgraphs of vertices in the network, can we achieve both objectives effectively during graph query processing. Actually, the indexed patterns capture the local structural information within vertices’ vicinity and it will be extremely useful in search space reduction.

A natural question may arise here: Among different kinds of structural patterns, which one (or ones) are most suitable for graph indexing on large networks? It is evident that by explicitly indexing all structural patterns within the neighborhood scope \( k \) for all vertices is of little practical use due to the exponential number of possible patterns, even when \( k \) is not set high. As a result, we need a careful selection such that our graph indexing solution lies right
between the two extremes of *indexing-nothing* and *indexing-everything*. More importantly, the graph indexing structure should scale well in large information networks and can achieve effective graph query performance, simultaneously.

### 4.4.3 Structural Pattern Evaluation Model

In this section, we propose a cost-sensitive model to help select the best structural patterns specifically used in large networks. Three different patterns are considered, i.e., paths, trees and graphs. For each structural pattern, we focus on two cost-sensitive aspects: 1. feature selection cost, and 2. feature pruning cost.

For a vertex \( u \in V(G) \) (or \( v \in V(Q) \)), the *feature selection cost*, \( C_s \), is to identify a pattern from the \( k \)-neighborhood subgraph of \( u \) (or \( v \)). The number of such patterns is denoted as \( n \) (or \( n' \)). Given a pattern \( p \) in the \( k \)-neighborhood subgraph \( Q^k_v \) of \( v \in V(Q) \), the *feature pruning cost*, \( C_p \), is to check whether there exists a pattern \( p' \) in the \( k \)-neighborhood subgraph \( G^k_u \) of \( u \in C(v) \), such that \( p \subseteq p' \). We further assume the vertex labels of the network \( G \) are evenly distributed, such that \( |C(v)| = |V(G)|/|\Sigma|, v \in V(Q) \). Therefore the *total graph indexing cost*, \( C \), can be formulated as a combination of (1) the total feature selection cost in \( G \); (2) the total feature selection cost in \( Q \); and (3) the feature pruning cost of \( Q \), i.e.,

\[
C = (|V(G)| * n + |V(Q)| * n') * C_s + \frac{|V(Q)| * |V(G)| * n' * C_p}{|\Sigma|}
\]

Given an information network \( G \), both \( |V(G)| \) and \( |\Sigma| \) are constant (\( |V(G)| \) can be very large, though). \( Q \) is always much smaller than \( G \), so \( |V(Q)| \) can be regarded as a small constant as well. The graph indexing cost \( C \) is therefore relevant to \( n, n', C_s \) and \( C_p \). Table 4.1 shows the qualitative costs \( w.r.t. \) these parameters for different structural patterns. First, the number of patterns (\( n \) or \( n' \)) can be exponentially large in the \( k \)-neighborhood subgraphs, even when \( k \) is not set high. However, the number of path patterns is usually

---

Table 4.1: shows the qualitative costs \( w.r.t. \) these parameters for different structural patterns. First, the number of patterns (\( n \) or \( n' \)) can be exponentially large in the \( k \)-neighborhood subgraphs, even when \( k \) is not set high. However, the number of path patterns is usually

---

52
<table>
<thead>
<tr>
<th>Cost</th>
<th>( n(n') )</th>
<th>( C_s )</th>
<th>( C_p )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path</td>
<td>exponential</td>
<td>linear time</td>
<td>linear time</td>
</tr>
<tr>
<td>Tree</td>
<td>exponential</td>
<td>linear time</td>
<td>polynomial time</td>
</tr>
<tr>
<td>Graph</td>
<td>exponential</td>
<td>linear time</td>
<td>NP-complete</td>
</tr>
</tbody>
</table>

Table 4.1: Qualitative Costs for Different Structural Patterns

much less than that for trees and graphs. For the feature selection cost, \( C_s \), we can choose either BFS or DFS traversal method to identify one specific pattern in the \( k \)-neighborhood subgraph. As to the pattern pruning cost, \( C_p \), the path containment testing takes linear time only. While for trees, a costly polynomial algorithm is required \([99]\), and GraphQL \([57]\) took an even more expensive semi-perfect matching method in cubic time. For graphs, though, \( C_p \) is still NP-complete because we are trying to use a set of subgraph isomorphism testings on small graphs to substitute the costly subgraph isomorphism testing on one large graph.

Based on the above analysis, paths excel trees and graphs as good indexing patterns in large networks. Although more structural information can be preserved by trees and graphs, their potentially massive size and expensive pruning cost even outweigh the advantage for search space pruning. Although theoretically the number of paths is still exponentially large in the worst case, in the remainder of this chapter, we selectively use shortest paths for graph indexing. Shortest paths are further decomposed into a distance-wise structure, which makes our graph indexing technique, SPath, highly scalable. During graph query processing, shortest paths can be easily reconstructed and their joint pruning power proves to be very impressive.

### 4.5 SPath

In this section, we present SPath, a path-based graph indexing technique on large information networks. The principle of SPath is to use shortest paths within the \( k \)-neighborhood subgraph of each vertex of the graph to capture the local structural information around the vertex. To tackle a potentially polynomial number of shortest paths within \( k \)-neighborhood subgraphs,
we further decompose shortest paths in a distance-wise structure, *neighborhood signature*, which reduces the space complexity of SPath to be linear *w.r.t.* the size of the network. Therefore SPath lends itself well to large information networks.

### 4.5.1 Neighborhood Signature

**Definition 5.** \(((k, l)-\text{SET})\)** Given a graph \(G\), a vertex \(u \in V(G)\), a nonnegative distance \(k\) and a vertex label \(l \in \Sigma\), the \((k,l)\)-set of \(u\), \(S^l_k(u)\), is defined as

\[
S^l_k(u) = \{v|d(u,v) = k, l(v) = l, v \in V(G)\}
\]

where \(d(u,v)\) is the shortest distance from \(u\) to \(v\) in \(G\).

Namely, \(S^l_k(u)\) is the set of vertices \(k\) hops away from \(u\) and with the vertex label \(l\).

**Definition 6.** \((k-\text{DISTANCE SET})\)** Given \(u \in V(G)\), and a nonnegative distance \(k\), the \(k\)-distance set of \(u\), \(S_k(u)\), is defined as

\[
S_k(u) = \{S^l_k(u)|l \in \Sigma\}\ \setminus \{\emptyset\}
\]

**Definition 7.** \((\text{NEIGHBORHOOD SIGNATURE})\)** Given \(u \in V(G)\), and a nonnegative neighborhood scope \(k_0\), the *neighborhood signature* of \(u\), denoted as \(NS(u)\), is defined as

\[
NS(u) = \{S_k(u)|k \leq k_0\}
\]

\(NS(u)\) maintains all \(k\)-distance sets of \(u\) from \(k = 0\) (a singleton set with element \(u\) only) up to the neighborhood scope \(k = k_0\). Therefore, all shortest path information in the \(k_0\)-neighborhood subgraph \(G^k_{u_0}\) of \(u\) is encoded in the neighborhood signature, \(NS(u)\). Note we do not maintain shortest paths explicitly. Instead all paths are decomposed into the distance-wise neighborhood signature. Although extra costs have to be paid to reconstruct
the exact shortest paths during graph query processing, the time spent is marginal because of the simplicity of path structures.

**Example 4.5.1:** For vertex $u_1$ in the network $G$ shown in Figure 4.1(a), the 0-distance set $S_0(u)$ contains a unique $(0, A)$-set $A : \{1\}$, which contains $u_1$ itself. The 1-distance set $S_1(u)$ is $\{B : \{2\}, C : \{3\}\}$, and the 2-distance set $S_2(u)$ is $\{A : \{4, 6\}, B : \{5\}\}$. If the neighborhood scope $k_0$ is set 2, the neighborhood signature of $u_1$, $NS(u_1) = \{\{A : \{1\}\}, \{B : \{2\}, C : \{3\}\}, \{A : \{4, 6\}, B : \{5\}\}\}$. Similarly, for vertex $v_1$ in the graph query $Q$ shown in Figure 4.1(b), the neighborhood signature of $v_1$, $NS(v_1) = \{\{A : \{1\}\}, \{B : \{2\}, C : \{3\}\}, \{C : \{4\}\}\}$. \hfill \Box

As shortest path information within the $k$-neighborhood subgraph of a vertex is well preserved into its neighborhood signature, it can be used in search space pruning, i.e., the false positives in the matching candidates $C(v)$ can be eliminated before the real graph query processing, where $v \in V(Q)$. We define neighborhood signature containment ($NS$ containment for short), which will be used for search space pruning.

**Definition 8. (NS Containment)** Given $u \in V(G)$ and $v \in V(Q)$, $NS(v)$ is contained in $NS(u)$, denoted as $NS(v) \subseteq NS(u)$, if $\forall k \leq k_0$, $\forall l \in \Sigma$, $|\bigcup_{k \leq k_0} S^l_k(v)| \leq |\bigcup_{k \leq k_0} S^l_k(u)|$.

**Theorem 4.5.1:** Given a network $G$ and a query graph $Q$, if $Q$ is subgraph-isomorphic to $G \ w.r.t. \ f$, i.e., $Q \subseteq G$, then $\forall v \in V(Q), NS(v) \subseteq NS(f(v))$, where $f(v) \in V(G)$.

**Proof:** For $\forall v \in V(Q)$, we consider an arbitrary vertex $v' \in S^l_k(v)$, where $0 \leq k \leq k_0$ and $l \in \Sigma$. If $k_0 = 0$, then $k = 0$ and $l = l'(v)$. However, $l'(v) = l(f(v))$ because $Q \subseteq G \ w.r.t. \ f$, where $l'(\cdot)$ and $l(\cdot)$ are labeling functions of $Q$ and $G$, respectively. So $|S^l_0(v)| = |S^l_0(f(v))| = 1$.

We then consider the situations when $0 < k \leq k_0$. In $Q$, there must be a shortest path $p = v \ldots v'$ of length $k$ with $v$ and $v'$ as its endpoints. because $Q \subseteq G \ w.r.t. \ f$, the counterpart mapping path $f(p) = f(v) \ldots f(v')$, where $f(p) \subseteq G$, can be either (1) the
shortest path between \( f(v) \) and \( f(v') \), such that \( f(v') \in S_k^l(f(v)) \); or (2) a non-shortest path between \( f(v) \) and \( f(v') \), because there must be another path \( p' \), \(|p'| < k \), connecting \( f(v) \) and \( f(v') \) in \( G \). This is true if some edges in \( p' \) cannot be mapped from any edge in \( Q \). If so, \( \exists \tilde{k}, 0 < \tilde{k} < k \) such that \( f(v') \in S_{\tilde{k}}^l(f(v)) \), i.e., \( f(v') \) appears in the \((\tilde{k}, l)\)-set of \( f(v) \). Based on the two aforementioned situations, \( \forall v' \in S_{\tilde{k}}^l(f(v)) \), \( \exists \tilde{k}, 0 < \tilde{k} \leq k \), such that \( f(v') \in S_{\tilde{k}}^l(f(v)) \). So \( |U_{k \leq k_0} S_{\tilde{k}}^l(v)| \leq |U_{k \leq k_0} S_{\tilde{k}}^l(f(v))| \) satisfies. \( \square \)

Based on Theorem 4.5.1, for a vertex \( v \in V(Q) \) and a vertex \( u \in V(G) \), where \( u \in C(v) \), if \( NS(v) \) is not contained in \( NS(u) \), denoted as \( NS(v) \not\sqsubseteq NS(u) \), \( u \) is a false positive and can be safely pruned from \( v \)'s matching candidates \( C(v) \). Therefore, the search space is reduced.

**Example 4.5.2:** For \( u_1 \in V(G) \) shown in Figure 4.1(a) and \( v_1 \in V(Q) \) shown in Figure 4.1(b), their neighborhood signatures are presented in Example 4.5.1. Although \( u_1 \in C(v_1) \) because \( u_1 \) and \( v_1 \) have the same label \( A \), \( NS(v_1) \not\sqsubseteq NS(u_1) \). In particular, when \( l = C \), \( |U_{k \leq 2} S_C(v_1)| = 2 \) as \( v_3 \in S_C^1(v_1) \) and \( v_4 \in S_C^2(v_1) \). However, \( |U_{k \leq 2} S_C^0(u_1)| = 1 \) as \( u_3 \in S_C^0(u_1) \) only, such that \( |U_{k \leq 2} S_C^0(u_1)| < |U_{k \leq 2} S_C^0(v_1)| \). So \( u_1 \) is a false positive and can be safely pruned from \( C(v_1) \). By taking advantage of neighborhood signatures, we can prune the search space for \( C(v_1) \) from \( \{u_1, u_4, u_6, u_8, u_{11}\} \) to \( \{u_6, u_8, u_{11}\} \), for \( C(v_2) \) from \( \{u_2, u_5, u_{10}, u_{12}\} \) to \( \{u_5\} \), for \( C(v_3) \) from \( \{u_3, u_7, u_9\} \) to \( \{u_7\} \), and for \( C(v_4) \) from \( \{u_3, u_7, u_9\} \) to \( \{u_7, u_9\} \). The total search space size, \( \prod_{i=1}^4 |C(v_i)| \), has been reduced from 180 to 6. \( \square \)

Algorithm 3 outlines the neighborhood signature containment algorithm for \( v \in V(Q) \) and \( u \in V(G) \), where \( u \in C(v) \). Note we don’t need to maintain the exact elements in \( S_k^l(u) \) or \( S_k^l(v) \) for NS containment testing. Instead only the cardinality information of the two sets are enough during the computation. In real implementation, we maintain a hash table, \( Count \), to keep track of the value of \( (|\bigcup_{k \leq k_0} C_k^l(u)| - |\bigcup_{k \leq k_0} C_k^l(v)|) \) for all \( l \in \Sigma \). In real implementation, only the labels either in \( k_0 \)-neighborhood subgraph of \( v \) or in \( k_0 \)-neighborhood subgraph of \( u \) (or both) are examined. The time complexity of Algorithm 3
Algorithm 3 Neighborhood Signature Containment

**Input:** \( NS(v), v \in V(Q), NS(u), u \in V(G) \)

**Output:** If \( NS(v) \sqsubseteq NS(u) \), return true Otherwise, return false

```
begin
  for \( l \in \Sigma \) do
    Count[\( l \)] \( \leftarrow 0 \)
  for \( k \leftarrow 1 \) to \( k_0 \) do
    for \( l \in \Sigma \) do
      Count[\( l \)] \( \leftarrow Count[\( l \)] + |S^l_k(u)| \)
      if \( |S^l_k(v)| > Count[\( l \)] \) then
        return false;
      Count[\( l \)] \( \leftarrow Count[\( l \)] - |S^l_k(v)| \)
    return true
end
```

is \( O(k_0|\Sigma|) \), so it is a constant-time algorithm. In practice, the neighborhood signature containment testing can be executed efficiently.

### 4.5.2 SPath Implementation

Our graph indexing structure, SPath, maintains the neighborhood signature of each vertex in the information network \( G \). In practice, we further decompose neighborhood signatures into different components:

1. **Lookup Table:** We separately maintain \( S^l_0(u), u \in V(G) \) as a global lookup table (Note here \( k = 0 \)): \( \mathcal{H} : l^* \rightarrow \{u|l(u) = l^*\}, l^* \in \Sigma \), such that given a vertex \( v \) in the query graph, we can easily figure out its matching candidates, \( C(v) \), which exactly equals \( \mathcal{H}(l(v)) \);

2. **Histogram:** we maintain a succinct distance-wise histogram \( |S^l_k(u)| \) for \( 0 < k \leq k_0 \) in the neighborhood signature. Based on Algorithm 3 we need not maintain the exact elements in the \((k,l)\)-set of \( u \), \( S^l_k(u) \). Instead only the cardinality information, \( |S^l_k(u)| \), is required in search space pruning. A numeric value, *count*, keeps track of \( |S^l_k(u)| \) in the neighborhood signature histogram.
Figure 4.2: SPath Data Structure for \( u_3 \in V(G) \) (\( k_0 = 2 \))

3. **ID-List**: We separately maintain the \((k, l)\)-set of \( u, S_k^l(u), u \in V(G) \), in an auxiliary data structure, \( ID-list \), which keeps track of the exact vertex identifiers in \( S_k^l(u) \).

The principle to decompose neighborhood signatures into a global lookup table, histograms and ID-lists is that both the lookup table and histograms can be maintained as a space-efficient data structure, upon which the NS containment testing can be performed without referring to the exact vertex information stored in ID-lists. Note ID-lists may be very large and only in the graph query processing phase, will ID-lists be visited to reconstruct real paths.

**Example 4.5.3**: Figure 4.2(a) presents the global lookup table of the network \( G \) in Figure 4.1(a). Figure 4.2(b) illustrates the histogram and ID-list structures of the neighborhood signature of \( u_3 \) in \( G \).

To construct our graph indexing structure \( SPath \) for an information network \( G \), we need to build for each vertex \( u \in V(G) \), its neighborhood signature \( NS(u) \). If the neighborhood scope value \( k_0 \) is specified, a BFS traversal from \( u \) up to \( k_0 \) steps is required to collect shortest path information in the \( k_0 \)-neighborhood subgraph of \( u \). Suppose the average degree of vertices in \( G \) is \( d \), the time complexity of building \( NS(u) \) is \( \sum_{i=0}^{k_0} d^i \) and the worst-case time complexity is \( O(|V(G)| + |E(G)|) \). Therefore the worst-case time complexity for index construction is \( O(|V(G)| * |E(G)|) \).

As to the space complexity, the global lookup table \( H \) takes \( O(|V(G)| + |\Sigma|) \) space. Given a vertex \( u \in V(G) \), the space for the histogram structure is \( O(k_0|\Sigma|) \). So the total space
complexity of $\text{SPath}$ is $O(|V(G)| + |\Sigma| + k_0|\Sigma||V(G)|)$, i.e., the size of $\text{SPath}$ grows linearly w.r.t. the network size, $|V(G)|$. Note the ID-List structure is located on the disk in that its size can be very large (the worst space complexity can be $O(|V(G)|^2)$) and it will not be used until the real path reconstruction. In practice, however, if the network is of medium size, e.g., for biological networks, we can maintain both histograms and ID-Lists in main memory to facilitate the graph query processing.

4.6 Graph Query Processing

In this section, we will examine how graph queries are processed and optimized on large information networks with the aid of $\text{SPath}$. Given a query graph $Q$, we first study how $Q$ can be decomposed to a set of shortest paths, among which a subset of paths with good selectivity is then selected as candidates by our query plan optimizer. $Q$ is then reconstructed and instantiated by joining the selected candidate paths until every edge in $Q$ has been examined at least once. The major advantage of our method is its path-at-a-time philosophy in query processing and optimization, which proves to be more cost-effective and efficient than traditional vertex-at-a-time methods.

4.6.1 Query Decomposition

Given a query graph $Q$, we first compute the neighborhood signature $NS(v)$ for each $v \in V(Q)$. We then examine the matching candidates $C(v)$ by calling Algorithm 3 for NS containment testing. For $\forall u \in C(v)$, if $NS(v) \nsubseteq NS(u)$, $u$ is pruned from $C(v)$ as a false positive and the resulting matching candidates after pruning is called the reduced matching candidates of $v$, denoted as $C'(v)$.

During the NS containment testing of $v$ w.r.t. $u \in C'(v)$, the shortest paths originated from $v$ are generated as by-products of the neighborhood signature, $NS(v)$. Note if a path $p$ connecting two vertices is shortest in $Q$, its mapping counterpart $p'$ in the network $G$ is
not necessarily shortest between the mapping vertex-pair. We need to select the shortest paths from $Q$ that are shortest in $G$ as well, because only shortest paths have been indexed properly in $S\text{Path}$.

**Theorem 4.6.1:** For $v \in V(Q)$, a shortest path originated from $v$ with length bounded up by $k^*$ is guaranteed to be shortest as well in the $k_0$-neighborhood subgraph $G^{k_0}_u$ of $u$, where $u \in C'(v)$, if

$$
k^* = \arg \min_k \left\{ | \bigcup_{k \leq k_0} S_l^i(u) | - | \bigcup_{k \leq k_0} S_l^i(v) | > 0 \right\}, \forall l \in \Sigma
$$

**Proof:** Note $k^*$ is the minimum distance with which the $(k, l)$-set of $u$ begins to differ from the $(k, l)$-set of $v$. We prove the theorem by contradiction. Assume there exists a shortest path $p = v \ldots v_k$ in the neighborhood subgraph of $v$, and the length of $p$ equals $k \leq k^*$. Assume the vertex label of $v_k$ is $l$, so $v_k \in C'_k(v)$. However, the counterpart path $p' = u \ldots u_k$ in the neighborhood subgraph of $u$ is no longer a shortest path, $u \in C'(v)$. Equivalently, $\exists k', 0 \leq k' < k$, s.t., $u_k \in C_{k'}^l(u)$. However, for $0 \leq k' < k \leq k^*$, we have $| \bigcup_{i \leq k'} S_l^i(u) | - | \bigcup_{i \leq k'} S_l^i(v) | = 0$. So we must have another vertex $\hat{v}$ in the neighborhood subgraph of $v$, s.t., $\hat{v} \in C_{k'}^l(v)$. It means that both $v_k$ and $\hat{v}$ in the neighborhood subgraph of $v$ have the counterpart vertex $u_k$ in the neighborhood subgraph of $u$. So $NS(v) \not\subseteq NS(u)$. It contracts with the fact that $u \in C'(v)$, i.e., $NS(v) \subseteq NS(u)$.

Based on Theorem 4.6.1, we select the shortest paths originated from $v$ with length no greater than $k^*$. These paths are guaranteed to be shortest in the $k_0$-neighborhood subgraph of $u$, where $u \in C'(v)$. In the extreme case when $k^* = 0$, the shortest path is degenerated to a vertex and our graph query processing algorithm boils down to the baseline vertex-at-a-time algorithm (Algorithm 2).

**Example 4.6.1:** Consider $v_1 \in V(Q)$ in Figure 4.1(b) and $u_8 \in V(G)$ in Figure 4.1(a), because $NS(v_1) \subseteq NS(u_8)$, $u_8 \in C'(v_1)$, the reduced matching candidates of $v_1$. When $k = 1$, $|S^B_l(v_1)| = |S^B_l(u_8)| = 1$, and $|S^C_l(v_1)| = |S^C_l(u_8)| = 1$. However, when $k = 2$, $|S^B_l(v_2)| = |S^B_l(u_8)| = 2$, and $|S^C_l(v_2)| = |S^C_l(u_8)| = 2$. Thus, $k^* = 2$. Theorem 4.6.1 is satisfied.
\[|S_2^G(v_1)| = 1 \text{ but } |S_2^G(u_8)| = 2. \text{ So } k^* = 2, \text{ and the shortest paths originated from } v_1 \text{ w.r.t. } u_8 \text{ are } (v_1, v_2), (v_1, v_3), (v_1, v_2, v_4) \text{ and } (v_1, v_3, v_4). \]

### 4.6.2 Path Selection and Join

After the query graph \(Q\) has been decomposed, for each \(u \in C'(v)\), it is attached with a set of shortest paths, denoted as \(P_u\), which can be easily looked up in \(S\text{Path}\) and will be used jointly to reconstruct \(Q\). However, a natural question may arise: *which shortest paths should we choose in order to reconstruct \(Q\)?* To reconstruct the graph query means for every edge in \(Q\), it should be examined at least once during the subgraph isomorphism testing such that the correctness of the query processing algorithm can be secured. So our selected shortest paths should properly “cover” the query, *i.e.*, \(\forall e \in E(Q),\) there should exist at least one selected shortest path \(p\), such that \(e \in p\). Furthermore, the subset of selected shortest paths should be cost-effective and help reconstruct the query \(Q\) in an efficient way. We consider two objectives in our query plan optimizer to address the path selection problem:

1. We need to choose the smallest set of shortest paths which can cover the query. This problem can be reduced to the *set-cover* problem, if every edge in \(E(Q)\) is regarded as an element and every path is a subset of elements. Set-cover has proven to be NP-complete and a greedy \(\log(n)\)-approximation algorithm was proposed [29];

2. We need to choose shortest paths with good selectivity, such that the total search space can be minimized during real graph matching.

Let’s first assume we have obtained such a subset of shortest paths which suffices for the above-mentioned objectives. Our graph query processing is then performed by joining shortest paths from among the set of selected paths.

**Definition 9. (PATH JOIN)** Given a path \(p = (v_{p_1}, v_{p_2}, \ldots, v_{p_k})\) and a path \(q = (v_{q_1}, v_{q_2}, \ldots, v_{q_{k'}})\), the *join* of \(p\) and \(q\), denoted as \(p \bowtie q\), is defined as an induced graph on the
vertex set \( \{v_{p_1}, v_{p_2}, \ldots, v_{p_k}\} \cup \{v_{q_1}, v_{q_2}, \ldots, v_{q_{k'}}\} \), where \( \{v_{p_1}, v_{p_2}, \ldots, v_{p_k}\} \cap \{v_{q_1}, v_{q_2}, \ldots, v_{q_{k'}}\} \neq \emptyset \). The \textit{join-predicates} are defined on the vertices \( \{v_{p_1}, v_{p_2}, \ldots, v_{p_k}\} \cap \{v_{q_1}, v_{q_2}, \ldots, v_{q_{k'}}\} \). \textit{i.e.}, \( p \) and \( q \) are \textit{joinable} if they share at least one common vertex.

It is reasonable to suppose that the join cost of \( p \bowtie q \) is proportional to \( |C'(p)| \ast |C'(q)| \), where \( |C'(p)| = \prod_{i=1}^{k} |C'(v_{p_i})| \) and \( |C'(q)| = \prod_{j=1}^{k'} |C'(v_{q_j})| \), the multiplicity of sizes of the reduced matching candidates for each vertex in the path. If the number of \textit{join-predicates} (\textit{i.e.}, the number of common vertices shared by both \( p \) and \( q \)) for \( p \bowtie q \) is \( N_{pq} \), and suppose \( N_{pq} \) \textit{join-predicates} are mutually independent, all of which are associated with a selectivity factor \( \theta \), the remaining estimated size of \( p \bowtie q \) will be \( |C'(p)| \ast |C'(q)| \ast \theta^{N_{pq}} \). Given a join path \( \mathcal{J}P = (((p_1 \bowtie p_2) \bowtie p_3) \bowtie \cdots \bowtie p_t) \) which covers the query \( Q \), the total join cost can be formulated as

\[
C(\mathcal{J}P) = |C'(p_1)| \ast |C'(p_2)| + \ldots + |C'(p_1)|\left(\prod_{i=2}^{t-1} |C'(p_i)|\theta^{N_{pq(i-1)p_i}}\right) |C'(p_t)|
\]

In order to minimize the join cost \( C(\mathcal{J}P) \), we can either (1) minimize the number of join operators: \( t - 1 \), which can be achieved by empirically choosing long non-repetitive paths first; Or (2) minimize the estimate size for each join operation, which can be obtained by always choosing the paths with good selectivity; Or minimize both. Note our objectives to minimize \( C(\mathcal{J}P) \) are almost the same as the objectives for path selection, as mentioned above. More interestingly, these objectives share the same philosophy as dictated in Section 4.4.3 for structural feature evaluation.

Keeping the aforementioned objectives in mind, we define \textit{selectivity} of a path \( p \), denoted as \( \text{sel}(p) \), as follows

\[
\text{sel}(p) = \frac{\psi(l)}{\prod_{v \in V(p)} |C'(v)|}
\]

where \( \psi(\cdot) \) is a function of the path length \( l \), \textit{e.g.}, \( \psi(l) = 2^l \). Intuitively, \( \text{sel}(p) \) in Equation (4.2) tries to take both objectives into consideration. The larger the selectivity \( \text{sel}(p) \)
of \( p \), the better chance \( p \) will be chosen from among the subset of shortest paths and be joined first to recover the graph query \( Q \). In practice, our query optimizer takes a greedy approach to always pick the edge-disjoint path with highest selectivity first, and it achieves very effective query performance.

### 4.6.3 Path Instantiation

After a path has been selected, it needs to be instantiated in the information network \( G \), such that its exact matching can be determined and the join predicates can be cross-checked when path joins are executed between selected paths. We again make use of neighborhood signatures for path instantiation. Given a path \( p = (v_1, v_2, \ldots, v_t) \) in the query graph \( Q \), a straightforward way to instantiate \( p \) on \( G \) is an edge-by-edge verification for each edge \((v_i, v_{i+1}) \in p\). Specifically, for each \( v_i \), we examine its matching candidate \( u \in C'(v_i) \). If \( C'(v_{i+1}) \cap S_1(v_{i+1}) \neq \emptyset \), it means there exist counterpart edges for \((v_i, v_{i+1})\) in the network \( G \). It is worth noting that for each verification, we need to retrieve the ID-List of \( S_1(v_{i+1}) \). If the ID-Lists reside on disk, the verification leads to expensive disk accesses, which is the most time-consuming part in graph query processing.

When selected paths are instantiated and joined with no join-predicates violation, we find one matching of \( Q \) against \( G \) successfully. The algorithm will not terminate until all matchings are detected from \( G \).

Our SPath based graph query processing is presented in Algorithm 4. It starts with a preprocessing step by pruning the search space for each vertex \( v \) in the graph query \( Q \) with neighborhood signature containment testings illustrated in Theorem 4.5.1 (Line 3). For each \( u \in C(v) \), if \( NS(u) \nsubseteq NS(v) \), it will be eliminated from \( C(v) \) as a false positive. As a result, we get the reduced matching candidates, \( C'(v) \). For each matching candidate \( u \in C'(v) \), the set of shortest paths, \( P_u \), is generated simultaneously (Lines 4–5). Based on Theorem 4.6.1, all the paths in \( P_u \) with length no greater than \( k^* \) are guaranteed to be shortest as well in the \( k_0 \)-neighborhood subgraph of \( u \). We choose a vertex \( v^* \) with a minimal size of the reduced
Algorithm 4 SPath Based Graph Query Processing

Input: Graph Query \( Q \), Network \( G \)

Output: All matchings \( f \) of \( Q \) against \( G \), s.t. \( Q \subseteq G \)

begin
  \textbf{for} \( v \in V(Q) \) \textbf{do}
    \begin{enumerate}
    \item \( C'(v) \leftarrow \{ u | NS(u) \supseteq NS(v), u \in C(v) \} \)
    \item \textbf{for} \( u \in C'(v) \) \textbf{do}
      \begin{enumerate}
      \item \( \mathcal{P}_u \leftarrow \{ p | |p| \leq k^* \} \) based on Theorem 4.6.1
      \end{enumerate}
    \item \( v^* \leftarrow \arg\min_v |C'(v)| \)
    \item \textbf{for} \( u \in C'(v^*) \) \textbf{do}
      \begin{enumerate}
      \item \( p_u \leftarrow \arg\min_p \text{sel}(p), p \in \mathcal{P}_u \)
      \item \( I \leftarrow \emptyset \)
      \item Recursive\_Search\( (p_u, I) \)
      \end{enumerate}
  \end{enumerate}
end

Procedure Recursive\_Search\( (p_u, I) \)

begin
  \textbf{while} \( i_{p_u} \leftarrow \text{Next\_Instantiation}(p_u) \) \textbf{do}
    \begin{enumerate}
    \item \textbf{if not} Joinable\( (I, (p_u, i_{p_u})) \) \textbf{then}
      \begin{enumerate}
      \item \textbf{continue}
      \item \( I \leftarrow I \cup \{ (p_u, i_{p_u}) \} \)
      \end{enumerate}
    \item \( \forall e \in E(Q) \) has been covered by \( I \) \textbf{then}
      \begin{enumerate}
      \item Output a matching \( f \leftarrow \bigcup_{p_u} \{ i_{p_u} | p_u \in I \} \)
      \item else
        \begin{enumerate}
        \item \( p_{u'} \leftarrow \arg\min_p \text{sel}(p), p \in \mathcal{P}_{u'}, \) where \( u' \) is any vertex covered by the paths in \( I \) so far
        \item Recursive\_Search\( (p_{u'}, I) \)
        \end{enumerate}
      \end{enumerate}
      \item \( I \leftarrow I - \{ (p_u, i_{p_u}) \} \)
    \end{enumerate}
end

Function boolean Joinable\( (I, (p_u, i_{p_u})) \)

begin
  \textbf{for} \( \forall \text{ path } p_i \in I \) \textbf{do}
    \begin{enumerate}
    \item \textbf{if} \( p_i \) and \( p_u \) are joinable w.r.t. the join predicate set \( J \) in \( Q \), but \( i_{p_i} \) and \( i_{p_u} \) fails the corresponding join predicates from \( J \) in \( G \) \textbf{then}
      \begin{enumerate}
      \item \textbf{return} false
      \end{enumerate}
    \item \textbf{return} true
    \end{enumerate}
end
matching candidates as a starting point for graph query processing (Line 6). When all possible matching candidates \( u \in C'(v^*) \) have been explored, our graph query processing algorithm terminates and all matchings of \( Q \) against \( G \) will be figured out. Among the set of shortest paths in \( P_u \), an optimal path is selected based on Equation 4.2 to initiate the recursive search (Lines 7 – 10). \( I \) is a data structure which maintains the pairs of \((\text{path, instantiated\_path})\) discovered so far.

In \textbf{Recursive\_Search}, we first instantiate the path \( p_u \) in the network \( G \) by calling the function \textbf{Next\_Instantiation}(\( p_u \)) (Line 14). In practice, for each vertex in the path \( p_u \) to be instantiated, we maintain an iterator to keep track of the vertex oriented in the network \( G \). Function \textbf{Next\_Instantiation}(\( p \)) is called to manipulate iterators such that a new instantiation of \( p_u \) is enumerated in a pipelining manner. We then test \emph{joinability} between the newly instantiated path with all the previously instantiated paths in \( I \) by calling the \textbf{Joinable} function (Line 15). \textbf{Joinable} function checks the join predicates between the path \( p_u \) and every path \( p_i \in I \) in the query \( Q \) (Lines 25 – 31). If their corresponding matching paths \( i_{p_u} \) and \( i_{p_i} \) fail in any join-predicate verification in the network \( G \), we have to explore the next instantiation of \( p_u \) (Line 16), or backtrack to the previously examined path (Line 23). Otherwise, if \( p_u \) and every path \( p_i \in I \) are joinable, \( p_u \) is coupled with the instantiated path \( i_{p_u} \) in the network \( G \). If every edge in the query graph \( Q \) has been covered by some paths in \( I \), a matching \( f \) is found out as an output (Lines 18 – 19). Otherwise, we proceed by picking another path with the best selectivity for further inspection (Lines 21 – 22). The optimal path selected is from among the set of shortest paths \( P_{u'} \) where \( u' \) is any vertex having been explored so far in \( I \). In practice, a maximum priority queue is maintained to get the path with highest selectivity. If two (or more) paths have the same highest selectivity, ties are broken by always picking the one which overlaps least with previously selected paths. When a new path \( p \) is selected, for any vertex \( u \in p \), all shortest paths \( P_u \) pertaining to \( u \) are added in the priority queue for further selection.
4.7 Experimental Evaluation

In this section, we report our experimental studies to illustrate the effectiveness of SPath in graph query processing on large information networks. We compare SPath with GraphQL and evaluate our algorithm SPath on both real and synthetic data sets. In the real data set, SPath proves to be a high-performance graph indexing scheme, and it achieves up to 4 times speedup in graph query processing, compared with GraphQL. In the synthetic data set which contains a set of disk-resident graphs, SPath demonstrates its scalability and effectiveness in answering graph queries in excessively large networks, whereas other proposed methods may fail in this scenario. All our experiments were tested on an AMD Phenom 8400 Triple-Core 2.10GHz machine with 3GB memory running Ubuntu 9.04. SPath is implemented with C++ and compiled with gcc 4.3.3. We set all parameters of GraphQL as default values specified and recommended in [57]. The only parameter of SPath, i.e., the neighborhood scope $k_0$, is set 4, if not specified explicitly.

4.7.1 A Yeast Protein Interaction Network

We adopt the same real data set used in GraphQL, which is a yeast protein interaction network [38]. The yeast protein interaction network consists of 3,112 vertices and 12,519 edges. Each vertex represents a unique protein and each edge represents an interaction between proteins. It is worth mentioning that the traditional RDBMS based query processing method is extremely inefficient to support graph queries on this biological network [57].

We further add Gene Ontology (GO) information as vertex labels to the proteins. The GO is a hierarchy of categories that describes cellular components, biological processes, and molecular functions of genes and their products (proteins). Each GO term is a node in the hierarchy and has one or more parent GO Terms, and each protein may have one or more GO terms. The original GO terms in the yeast protein interaction network consist of 2,205 distinct labels. We relax these GO terms by using their highest level ancestors. There are
183 such highest level GO terms in total, which constitutes our vertex label set $\Sigma$.

As to the graph queries, GraphQL suggests two extreme kinds of graphs with totally different structures: cliques and paths. For biological networks, the clique structure corresponds to protein complexes, while the path structure corresponds to transcriptional or signaling pathways. We further extract general induced subgraphs by randomly choosing seeds in the network and traversing the network in a DFS fashion. These generated graphs can be thought of as general queries with arbitrary structures lying in the middle of the two extremes: path and clique.

We first consider the index construction cost for SPath on this biological network. Figure 4.3(a) illustrates the memory usage of SPath (in kilobytes) with the variation of the neighborhood scope, $k_0$. With an increase of $k_0$ from 0 to 4, SPath grows linearly and it takes less than 1M memory usage even when $k_0 = 4$. Note when $k_0 = 0$, only the global lookup table is built and it is the only data structure required in the baseline algorithm. When $k_0 > 0$, the histograms of neighborhood signatures in SPath are constructed in main memory as well. Figure 4.3(a) also presents that even the ID-Lists can be loaded in main memory and the total memory cost is less than 6M bytes. SPath proves to be very space-efficient and in the following experiments, we explicitly store ID-Lists into main memory. Figure 4.3(b) illustrates the run-time of index construction for SPath. Even when $k_0 = 4$, SPath can still be constructed within 30 seconds.

Figure 4.3: Index Construction Cost of SPath
We then focus on the graph query processing performance in the information network. We first test clique queries on the biological network. The query set contains 1,000 cliques with sizes spanning from 2 to 7. If a query has too many matchings in the network (more than 1,000), we early terminate the algorithm and show the first 1,000 matchings as output (GraphQL has the same setting to report at most 1,000 matchings of $Q$ against $G$). Figure 4.4(a) shows the average query processing time for both SPath and GraphQL. For queries with small size ($\leq 4$), both methods achieve similar query processing performance. As the query size grows larger, SPath outperforms GraphQL for up to 50% improvement in query response time. Note for clique queries, the neighborhood signature of every vertex contains 1-distance set only which subsumes all other vertices. In the mean time, every clique query is decomposed to a set of edges (length-1 paths) for query processing. So the improvement mainly accounts for the optimal edge selection and join-predicate cross-checking. Figure 4.4(b) shows the average processing time for individual steps by varying clique sizes. The individual steps include query decomposition, abbreviated as decomposition; path selection, abbreviated as selection and path instantiation, abbreviated as instantiation. As shown in the figure, instantiation takes up the majority time during query processing.

We then test path queries on the biological network. Compared with clique queries, paths are at the other extreme of connectivity. Path queries have different sizes ranging from 2 to 10. For each size, we choose 1,000 paths and the average query processing time is examined.
As illustrated in Figure 4.5(a), with the increase of query size, SPath achieves a speedup in graph query processing up to 4 times, compared with GraphQL. In this scenario, the neighborhood signature containment pruning takes into effect beyond the direct neighborhood scope, and the path-at-a-time matching method has proven much more effective than the traditional vertex-at-a-time approach, adopted by GraphQL. Figure 4.5(b) shows the average processing time for individual steps by varying path sizes. Each of the individual steps takes less time than that for clique queries, while selection spends even less because the number of possible paths selected from the path queries is much less than that for the clique queries. Similarly, instantiation still takes up the majority time for query processing because we have to enumerate and instantiate the paths in the network.

We finally test general subgraph queries extracted from the biological network. Subgraphs are generated with sizes ranging from 2 to 10 and for each specific size, 1,000 queries are tested and the average query processing time is measured. As shown in Figure 4.6(a), SPath still outperforms GraphQL with a speedup for almost 4 times, especially when the query size becomes large. Figure 4.6(b) illustrates the individual time spent for subgraph queries and instantiation still dominates the whole graph query process for path instantiation.
4.7.2 Synthetic Disk-resident Graphs

We further evaluate our algorithm, SPath, on a series of disk-resident synthetic graphs based on the Recursive Matrix (R-MAT) model [21]. The graphs generated naturally follows the power-law in- and out-degree distributions. All the parameters of the graph generator are specified with default values suggested by the authors. For SPath, we maintain both the global lookup table and the histogram structures of neighbor signatures in main memory, while keeping all the ID-Lists on disk.

We first examine the index construction cost of SPath on different large networks. We generate four large networks with $|V(G)| = 500,000, 1,000,000, 1,500,000$ and $2,000,000$, and $|E(G)| = 5 \times |V(G)|$, respectively. For each graph generated, the vertex labels are drawn randomly from the label set $\Sigma$, where $|\Sigma| = 1\% \times |V(G)|$. Note in a typical modern PC, the

![Figure 4.6: Query Response Time for Subgraph Queries](image)

![Figure 4.7: Index Construction of SPath](image)
number of potential tree or graph indexing structures can be excessively large, which requires a storage in the tera-byte or even peta-byte order. In this scenario, GraphQL fails simply because it cannot scale up on these large networks. However, as shown in Figure 4.7(a), SPath scales linearly with an increase of the network size, which makes SPath a feasible graph indexing solution applicable in large networks. Figure 4.7(b) illustrates the index construction time for SPath. Note building SPath from the network is a pre-processing step and executes only once before the real graph query processing, so the cost is still affordable for large networks.

We then test the query processing performance of SPath on one synthetic graph $G$ with size $|V(G)| = 1,000,000$ and $|E(G)| = 5,000,000$. We further generate subgraph queries with different sizes 5, 10, 15, and 20 by randomly extracting induced subgraphs from $G$ by DFS traversal. For each specific query size, we generate 1,000 queries and measure the average query response time. As shown in Figure 4.8(a), SPath can achieve satisfactory response time even when the query size is large. However, all previously proposed methods, including GraphQL, cannot answer graph queries on this massive network. Figure 4.8(b) shows the individual time spent by different query processing components of SPath. As both decomposition and selection are performed in main memory, they take up little time during query processing. However, instantiation needs to retrieve ID-lists from disk, so it becomes the leading factor and potential bottleneck for graph queries on large networks.
4.8 Conclusions

In this chapter, we consider the graph query problem on large information networks. Existing data models, query languages and access methods no longer fit well in the large network scenario and we have presented SPath, a new graph indexing method to answer and optimize graph queries effectively on large networks. We evaluated different structural patterns based on our cost-sensitive model and shortest path information were chosen as good indexing features in large networks. Both index construction and query processing issues of SPath were discussed in detail. We performed our experimental evaluations on both real data sets and synthetic ones. The experimental results demonstrated that SPath is a scalable graph indexing technique and it outperforms the state-of-the-art GraphQL in addressing graph queries on large networks.

There are still several interesting problems left for further exploration. First of all, many large scale information networks change rapidly over time, such that incremental update of graph indexing structures becomes important. Second, to accommodate noise and failure in the networks, we need to extend our method to support approximate graph queries as well. These interesting issues will be our research directions in near future.
Chapter 5

Warehousing and OLAP
Multidimensional Information Networks

5.1 Introduction

Recent years have seen an astounding growth of information networks in a wide spectrum of application domains, ranging from sensor and communication networks to biological and social networks. And it becomes especially apparent as far as the great surge of popularity for Web 2.0 applications is concerned, such as Facebook, LinkedIn, Twitter and Foursquare. Typically, these networks can be modeled as large graphs with vertices representing entities and edges depicting relationship between entities [7]. Apart from the topological structures encoded in the underlying graph, multidimensional attributes are often specified and associated with vertices, forming the so-called multidimensional information networks (or multidimensional networks for short). While studies on contemporary networks have been around for decades [91], and a plethora of algorithms and systems have been devised for multidimensional analysis in relational databases [49, 22], none has taken both aspects into account in the multidimensional network scenario. As a result, there exist considerable technology gaps in managing, querying and summarizing such data effectively. And a growing need arises in order to shorten these technology gaps and develop specialized approaches for multidimensional networks.

Example 5.1.1: Figure 5.1 presents a sample social network consisting of several individuals interconnected with friend relationship. There are ten vertices (identified with user ID) and thirteen edges in the underlying graph, as shown in Figure 5.1(a). Each individual of the network contains a set of multidimensional attributes describing her/his properties,
including user ID (as primary key), gender, location (in state), profession and yearly income, which is represented as a tuple in a vertex attribute table, as shown in Figure 5.1(b). The graph structure, together with the vertex-centric multidimensional attributes, forms a multidimensional network.

One possible opportunity of special interest is to support data warehousing and online analytical processing (OLAP) on multidimensional networks. Data warehouses are critical in generating summarized views of a business for proper decision support and future planning [22]. This includes aggregations and group-by’s of enterprise RDB data based on the multidimensional data cube model [49]. For example, in a sales data warehouse, time of sale, sales district, salesperson, and product might be the dimensions of interest, and numeric quantities such as sales, budget and revenue might be the measures to be examined. OLAP operations, such as roll-up, drill-down, slice-and-dice and pivot, are supported to explore different multidimensional views and allow interactive querying and analysis of the underlying data [52]. As important means of decision support and business intelligence, data warehouses and OLAP are advantageous for multidimensional networks as well. For example, a company is investigating how to run a marketing campaign in order to maximize returns. They turn to a large national social network to study the business and preference patterns of interlinked people within different multidimensional spaces, such as genders, locations, professions, hobbies, income levels and possible combinations of these dimensions. This lets
users analyze the underlying network in a summarized manner within multiple multidimensional spaces, which is typical and of great value in most data warehousing applications. In Facebook and Twitter, advertisers and marketers take advantage of their social networks within different multidimensional spaces to better promote their products via social targeting or viral marketing \cite{12,104}. In multidimensional networks, however, much of the valuation and interest lies in the network itself. Simple numeric value based group-by’s in traditional data warehouses are no longer insightful and of limited usage, because the structural information of the networks is simply ignored. As a result, existing data warehousing and OLAP techniques need to be re-examined and revolutionized in order to improve the potential power and core competency of decision support facilities specifically tailored for multidimensional networks.

**Example 5.1.2:** Figure 5.2(a) presents an aggregate network by summarizing the multidimensional network shown in Figure 5.1 on the dimension “Gender”. The vertices with grey color represent condensed vertices “Male” and “Female”, and the weight of each vertex means the number of individuals in the original network that comply with the same values for the dimension(s) represented by the condensed vertex. In this case, there are 5 males and 5 females in the multidimensional network. The edges represent aggregate relationships between condensed vertices while the edge weights present the number of edges in the original network connecting vertices belonging to two condensed vertices, respectively. Self-loops are allowed, as shown for the edges (Male, Male) and (Female, Female). The edge weight that equals 1 is not presented in the diagram by default.
Figure 5.3: Multidimensional Network Aggregation vs. RDB Aggregation (Group by Gender and Location)

In contrast, Figure 5.2(b) presents a traditional group-by along the dimension “Gender” on the vertex attribute table, shown in Figure 5.1(b). In this case, we select \( \text{COUNT}(\cdot) \) as the default aggregate operator.

Figure 5.3(a) presents another aggregate network by summarizing the original multidimensional network on the dimensions “Gender” and “Location”. While Figure 5.3(b) illustrates a traditional group-by on the vertex attribute table along the dimensions “Gender” and “Location”.

As shown in Example 5.1.2, a multidimensional network can be summarized to aggregate networks in coarser levels of granularity within different multidimensional spaces. During the network aggregation, we consider both vertex coalescence and structure summarization simultaneously, thus resulting in much meaningful and structure-enriched aggregate networks, as illustrated in Figure 5.2(a) and Figure 5.3(a). In contrast, the numeric value based aggregation for relational data can be regarded as a special case in our scenario, because the inter-tuple relationships are simply ignored during aggregation, as shown in Figure 5.2(b) and Figure 5.3(b). Therefore, the traditional concepts and techniques of data warehousing and OLAP have been enriched in a more structural way for multidimensional networks. Moreover, a set of new OLAP queries can be addressed on multidimensional networks, such as “What is the network structure as grouped by users’ gender?” The answer is shown in

<table>
<thead>
<tr>
<th>Gender</th>
<th>Location</th>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Male</td>
<td>CA</td>
<td>1</td>
</tr>
<tr>
<td>Female</td>
<td>CA</td>
<td>2</td>
</tr>
<tr>
<td>Female</td>
<td>WA</td>
<td>2</td>
</tr>
<tr>
<td>Male</td>
<td>IL</td>
<td>3</td>
</tr>
<tr>
<td>Male</td>
<td>NY</td>
<td>1</td>
</tr>
<tr>
<td>Female</td>
<td>NY</td>
<td>1</td>
</tr>
</tbody>
</table>
We note that there are a lot of connections between males and females in the network (9 edges as the edge weight), while few connections exist between males (only 1 edge as the edge weight). A closer look at this interesting phenomenon could be expressed as a drill-down query: "What is the network structure as grouped by both gender and location?" The answer is shown in Figure 5.3(a). We notice in the aggregate network that between the 2 females in California and the 3 males in Illinois, there exist 5 connections, taking up 55.6% of the total 9 connections between males and females. While the only connection between males actually exists between two males in Illinois. These queries could reveal interesting structural behaviors and potentially insightful patterns, which are very hard, if not impossible, to detect from the original network, as shown in Figure 5.1.

In this section, we consider extending decision support facilities on multidimensional networks by introducing a new data warehousing model, Graph Cube, for effective network exploration and summarization. Going beyond traditional data cubes which address simple value-based group-by’s on relational data, Graph Cube considers both multidimensional attributes and network structures into one integrated framework for network aggregation. In every potential multidimensional space, the measure of interest now becomes an aggregate network in coarser resolution. In addition, we propose different query models and OLAP solutions for multidimensional networks. Besides traditional cuboid queries with refined structural semantics, a new class of OLAP queries, called crossboid, is introduced, which is uniquely useful in the multidimensional network scenario and has not been studied before. An example crossboid query could be "what is the network structure between users grouped by profession and users grouped by income level?" Despite definitely OLAP in flavor, this query breaks the boundaries established in the traditional OLAP model in that it straddles two different group-by’s simultaneously. We implement Graph Cube by combining special characteristics of multidimensional networks with the existing well-studied data cube techniques. To the best of our knowledge, Graph Cube is the first to systematically address warehousing and OLAP issues on large multidimensional networks, and the solutions proposed in this
paper will help improve decision support and business intelligence in large networks.

The contributions of our work can be summarized as follows:

1. We propose a new data warehousing model, **Graph Cube**, to extend decision support services on multidimensional networks. The multidimensional attributes of the vertices define the dimensions of a graph cube, while the measure turns out to be an aggregate network, which proves to be much more meaningful and comprehensive than numeric statistics examined in traditional data cubes.

2. We formulate different OLAP query models and provide new solutions in the multidimensional network scenario. Besides cuboid queries that explore all potential multidimensional spaces of a graph cube, we introduce a new class of OLAP queries, crossboid, which breaks the boundaries of the traditional OLAP model by straddling multiple different multidimensional spaces simultaneously. Crossboid has shown to be especially useful for network study and analysis.

3. We make use of well-studied partial materialization techniques to implement **Graph Cube**. Specific characteristics of multidimensional networks are leveraged as well for better implementation alternatives.

4. We evaluate our methods on a variety of real multidimensional networks and the experimental results demonstrate the power and effectiveness of **Graph Cube** in warehousing and OLAP large networks. In addition, our query processing and cube implementation algorithms have proven to be efficient even for very large networks.

The reminder of this chapter is organized as follows. Section 5.2 gives preliminary concepts and examines the **Graph Cube** model on multidimensional networks. Section 5.3 formulates different OLAP queries defined upon **Graph Cube**. Section 5.4 focuses on the implementation details of **Graph Cube**. Experimental studies are shown in Section 5.5. After discussing the related work in Section 5.6, we conclude our study in Section 5.7.
### 5.2 The Graph Cube Model

Many information networks in real applications can be abstracted as a *multidimensional network*, which is formally defined as follows,

**Definition 10.** [**MULTIDIMENSIONAL NETWORK**] A multidimensional network, \( \mathcal{N} \), is a graph denoted as \( \mathcal{N} = (V, E, A) \), where \( V \) is a set of vertices, \( E \subseteq V \times V \) is a set of edges and \( A = \{A_1, A_2, \ldots, A_n\} \) is a set of \( n \) vertex-specific attributes, *i.e.*, \( \forall u \in V \), there is a multidimensional tuple \( A(u) \) of \( u \), denoted as \( A(u) = (A_1(u), A_2(u), \ldots, A_n(u)) \), where \( A_i(u) \) is the value of \( u \) on \( i \)-th attribute, \( 1 \leq i \leq n \). \( A \) is called the *dimensions* of the network \( \mathcal{N} \).

As explained in Example 5.1.1, Figure 5.1 presents a sample multidimensional network drawn from a real social network. The dimensions of the network are ID, gender, location, profession, and income. For an individual with \( ID = 1 \) in the network, his corresponding multidimensional tuple is \((1, \text{Male}, \text{CA}, \text{Teacher}, 70,000)\), the first tuple shown in Figure 5.1(b).

Data warehouses and OLAP for traditional RDB data have developed many mature technologies over the years. Here a brief primer of terminologies is listed. Given a relation \( R \) of \( n \) dimensions, an \( n \)-dimensional *data cube* is a set of \( 2^n \) aggregations from all possible group-by’s on \( R \). For any aggregation in a form of \((A_1, A_2, \ldots, A_n)\), some (or all) dimension \( A_i \) could be * (ALL), representing a super-aggregation along \( A_i \) which is equivalent to the removal of \( A_i \) during aggregation. There are *cells* in an aggregation, represented as \( c = (a_1, a_2, \ldots, a_n : m) \), where \( a_i \) is a value of \( c \) on \( i \)-th dimension, \( A_i \) (\( 1 \leq i \leq n \)), and \( m \) is a numeric value, called *measure*, computed by applying a specific aggregate function on \( c \), *e.g.*, \( \text{COUNT}(\cdot) \) or \( \text{AVERAGE}(\cdot) \). In Example 5.1.2, Figure 5.2(b) presents a group-by on \((\text{Gender}, *, *)\), if three dimensions are chosen for aggregation: Gender, Location and Profession, and Figure 5.3(b) presents a group-by on \((\text{Gender}, \text{Location}, *)\). Both group-by’s adopt \( \text{COUNT}(\cdot) \) as the underlying aggregate function.
By analogy, we can define possible aggregations upon multidimensional networks. Based on Definition 10 given a network with $n$ dimensions, there exist $2^n$ multidimensional spaces (aggregations). However, the measure within each possible space is no longer simple numeric values, but an aggregate network, defined as follows,

**Definition 11. [AGGREGATE NETWORK]** Given a multidimensional network $N = (V, E, A)$ and a possible aggregation $A' = (A'_1, A'_2, \ldots, A'_n)$ of $A$, where $A'_i$ equals $A_i$ or $\ast$, the aggregate network w.r.t. $A'$ is a weighted graph $G' = (V', E', W'_V, W'_E)$, where

1. $\forall [v], a$ nonempty equivalence class of $V$, where $[v] = \{v | A'_i(u) = A'_i(v), u, v \in V, i = 1 \ldots n\}$, $\exists v' \in V'$ as a representative of $[v]$. The weight of $v'$, $w(v') = \Gamma_V([v])$, where $\Gamma_V(\cdot)$ is an aggregate function defined upon vertices. $v'$ is therefore called a condensed vertex;

2. $\forall u', v' \in V'$, and a nonempty edge set $E_{(u', v')} = \{(u, v) | u \in [u] \text{ represented as } u', v \in [v] \text{ represented as } v', (u, v) \in E\}$, $\exists e' \in E'$ as a representative of $E_{(u', v')}$. The weight of $e'$, $w(e') = \Gamma_E(E_{(u', v')})$, where $\Gamma_E(\cdot)$ is an aggregate function defined upon edges. $e'$ is therefore called a condensed edge.

As explained in Example 5.1.2, Figure 5.2(a) and Figure 5.3(a) present the aggregate networks for the aggregations (Gender, $\ast$, $\ast$) and (Gender, Location, $\ast$), respectively. We choose COUNT($\cdot$) in the example to derive weights for both vertices and edges, while more complicated aggregate functions can be chosen and the aggregate functions for vertices and edges can be different. For example, AVERAGE($\cdot$) can be used if the edges are weighted in the original network. For the sake of brevity, we will choose COUNT($\cdot$) as the default aggregate function to compute both vertex and edge weights, while our model and algorithms can be easily generalized to accommodate other aggregate functions.

**Definition 12. [GRAPH CUBE]** Given a multidimensional network $N = (V, E, A)$, the graph cube is obtained by restructuring $N$ in all possible aggregations of $A$. For each
aggregation $A'$ of $A$, the measure is an aggregate network $G'$ w.r.t. $A'$, as defined in Definition 11.

Given a multidimensional network $\mathcal{N} = (V, E, A)$, each aggregation $A'$ of $A$ is often called a cuboid. The size of a cuboid $A'$ is $(|V'| + |E'|)$, where $V'$ and $E'$ are the vertex and edge set of the aggregate network corresponding to $A'$, respectively. For a cuboid $A'$, $\dim(A')$ denotes the set of non-* dimensions of $A'$. For example, if $A' = (\text{Gender}, *, *)$, $\dim(A') = \{\text{Gender}\}$.

Consider two cuboids $A'$ and $A''$. $A'$ is an ancestor of $A''$ if $\dim(A') \subseteq \dim(A'')$, and therefore $A''$ is a descendant of $A'$. Specifically, if $|\dim(A')| = |\dim(A')| + 1$, $A'$ is a parent of $A''$, or $A''$ is a child of $A'$. If $|\dim(A')| = |\dim(A'')| = l$, then $A'$ and $A''$ are siblings, both of which are at $l$-th level of the graph cube. A distinguished cuboid $A_b$ with $|\dim(A_b)| = n$ is called base cuboid and it is a descendant of all other cuboids in the graph cube. Another distinguished cuboid $A_{all} = (*, *, \ldots, *)$ where $|\dim(A_{all})| = 0$, is called apex cuboid. The apex cuboid $A_{all}$ is an ancestor of all other cuboids in the graph cube. If we denote the set of all cuboids of a graph cube as $2^A$, i.e., the power set of $A$, a graph cube lattice $L = \langle 2^A, \subseteq \rangle$ can be induced by the partial ordering $\subseteq$ upon $2^A$.

\footnote{Hereafter, we will use equivalently the terms cuboid, aggregation, view and query.}
Example 5.2.1: Figure 5.4 presents a graph cube lattice, each node of which is a cuboid in the graph cube generated from the multidimensional network shown in Figure 5.1. The edges in the lattice depict the parent-child relationship between two cuboids. The size of each cuboid is shown within the lattice node.

Given a multidimensional network $G$ with $n$ dimensions, there are $2^n$ cuboids in the graph cube. For each cuboid in a graph cube, there is a unique aggregate network corresponding to it. Specifically, the original multidimensional network is a special aggregate network corresponding to the base cuboid $A_b$. While the aggregate network for the apex cuboid $A_{all}$ has a singleton vertex with a possible self-loop. An aggregate network corresponding to an ancestor cuboid is more generalized than the aggregate network corresponding to one of its descendant cuboids, which is fine-grained and contains more attribute/structure details. In the graph cube framework, users can explore the original network in different multidimensional spaces by traversing the graph cube lattice. In this way, a set of aggregate networks with different summarized resolution can be examined and analyzed for decision support and business intelligence purposes.

5.3 OLAP on Graph Cube

In traditional OLAP on relational databases, numeric measures can be easily aggregated in the data cube. This naturally leads to queries such as “What is the average income of females?” or “What is the maximum income of software engineers in Washington State?” For multidimensional networks, however, aggregate networks become the measure of a graph cube. Consider some typical OLAP-style queries that might be asked on a multidimensional network:

1. “What is the network structure between the various location and profession combinations?”
2. “What is the network structure between the user with \(ID = 3\) and various locations?”

These queries clearly involve some kind of aggregations upon the original network in different multidimensional spaces. What is atypical here is the answers returned. For the first query, the answer is the aggregate network corresponding to the cuboid (\(*,\text{Location},\text{Profession}\)) in the graph cube. While the second query asks for an aggregate network across two different cuboids, (\(\text{Gender},\text{Location},\text{Profession}\)) and (\(*,\text{Location},\ast\)). In the following sections, we will formulate and address two different queries posed on the graph cube: (1) cuboid queries in a single multidimensional space and (2) crossboid queries across multiple multidimensional spaces.

### 5.3.1 Cuboid Query

An important kind of query on graph cube is to return as output the aggregate network corresponding to a specific aggregation of the multidimensional network. This query is referred to as **cuboid query** because the answer is exactly the aggregate network of the desired cuboid in the graph cube. Algorithm 5 outlines a baseline algorithm to address cuboid queries in detail.

In Algorithm 5, we first create a hash structure, \(\zeta\), which maintains a mapping from each distinct tuple w.r.t. the aggregation \(A'\), to a condensed vertex in the desired aggregate network \(G'\) (Line 2). We then traverse the multidimensional network \(G\). For each vertex \(u\) in \(G\), we create a new condensed vertex \(u'\) corresponding to the tuple \((A'_1(u), A'_1(u), \ldots, A'_n(u))\), if there is no such condensed vertex \(u' \in V'\) before hand (Lines 4 – 7). Otherwise, we simply update the weight for the condensed vertex (Line 8). For each edge \(e(u, v)\) in \(G\), we retrieve the mapped condensed vertices \(u'\) and \(v'\) for the adjacent vertices \(u\) and \(v\), respectively (Lines 10 – 11). If \(u'\) is not adjacent to \(v'\) in the aggregate network \(G'\), we create a new condensed edge \(e'(u', v')\) (Lines 12 – 14). Otherwise, we simply update the weight for the condensed edge \(e'\) (Line 15). The time complexity of Algorithm 5 is \(O(|V| + |E|)\), the time
Algorithm 5 Cuboid Query Processing

Input: A Multidimensional Network $\mathcal{N} = (V, E, A)$, an aggregation $A'$

Output: The aggregate network $G' = (V', E', W_{V'}, W_{E'})$ w.r.t. $A'$

begin
    Initialize a hash structure $\zeta : A' \rightarrow V'$
    for $u \in V$ do
        if $\zeta(A'(u)) = \text{NULL}$ then
            Create a condensed vertex $u' \in V'$, with label $A'(u) = (A'_1(u), \ldots, A'_n(u))$
            $u'.weight \leftarrow 0$
            $\zeta(A'(u)) \leftarrow u'$
        $\zeta(A'(u)).weight \leftarrow \zeta(A'(u)).weight + 1$
    for $e(u, v) \in E$ do
        $u' \leftarrow \zeta(A'(u))$
        $v' \leftarrow \zeta(A'(v))$
        if $e'(u', v') \notin E'$ then
            Create a condensed edge $e'(u', v') \in E'$
            $e'.weight \leftarrow 0$
            $e'.weight \leftarrow e'.weight + 1$
    return $G' = (V', E', W_{V'}, W_{E'})$
end

used for traversing $G$. The space used to maintain the hash structure $\zeta$ is $O(|V'|)$ and we need $O(|V'| + |E'|)$ space to maintain the aggregate network $G'$. So the space complexity of Algorithm 5 is $O(|V'| + |E'|)$.

Based on Algorithm 5, it is straightforward to address all cuboid queries from the multidimensional network, which is exactly the aggregate network corresponding to the base cuboid $A_b$. However, as the original network could be very large and may not be held in memory, query processing can be extremely time-consuming. Consider two cuboids $A'$ and $A''$ in the graph cube, if $A''$ is a descendant of $A'$ ($A'' \neq A_b$) and $A''$ has been precomputed from $A_b$ based on Algorithm 5, can we make use of $A''$ to directly compute $A'$, instead of computing it from $A_b$? The following theorem guarantees a positive answer to this question.

Theorem 5.3.1: Given two cuboids $A'$ and $A''$ in a graph cube, where $\text{dim}(A') \subseteq \text{dim}(A'')$ and $A'' \neq A_b$, the cuboid query $A'$ can be answered directly from $A''$.

Based on Theorem 5.3.1, if the cuboid $A''$ has been precomputed, $A'$ can be answered
directly from $A''$, and not necessarily from the base cuboid $A_b$. The algorithm is the same as Algorithm 5 except that we change the input from the original multidimensional network $G$ corresponding to $A_b$ to the aggregate network $G''$ corresponding to $A''$. In this way, the time complexity of the algorithm becomes $O(|V''| + |E''|)$, way better than $O(|V| + |E|)$ for the baseline algorithm. Theoretically the aggregate network $G''$ is no greater than the original multidimensional network $G$, while in practice, $G''$ can be much smaller than $G$ for some cuboids in the graph cube.

Now if we have a set of precomputed cuboids $A''$, which one should we choose to compute $A'$? We define the descendant set of cuboid $A'$, $des(A')$, as $des(A') = \{ A'' | dim(A') \subseteq dim(A''), A'' \text{ is in the graph cube} \}$. Based on the aforementioned complexity analysis, the following cuboid $A^*$ will be selected:

$$A^* = \arg\min(size(A'')), A'' \in des(A') \quad (5.1)$$

So, we always choose the precomputed cuboid, $A^*$, whose size, $(|V^*| + |E^*|)$, is minimal among all cuboids in $des(A')$ to answer the cuboid query $A'$.

When all the cuboids have been computed, the support of OLAP operations, such as roll-up, drill-down, and slice-and-dice, becomes straightforward in the graph cube framework. Roll-up means going from a cuboid to one of its ancestors, such that we can summarize an aggregate network in finer resolution to another one in coarser resolution. Drill-down, on the contrary, goes from an ancestor cuboid to one of its descendants. As shown in Example 5.1.2, after examining the aggregate network for the cuboid (Gender, *, *), users may be more interested in how males interact with females across different locations. We can drill-down to the cuboid (Gender, Location, *) and more interesting interaction patterns in finer resolution can be discovered. As to slice-and-dice, selections are performed on a cuboid and an induced aggregate network will be generated as a result. For example, users may be interested in the network structure between NY and CA, aggregated by Location. A
slice-and-dice operation can be performed upon the cuboid (*, Location, *) and only the interactions between the vertices NY and CA (including self-loops, if possible) are returned as output. Different OLAP operations can be further combined as advanced OLAP queries on the graph cube and they have formed a powerful tool set and new query mechanism on multidimensional networks.

5.3.2 Crossboid Query

Cuboid query discussed in Section 5.3.1 is the query within a single multidimensional space, which follows the traditional OLAP model proposed on relational data [49]. What is more interesting, however, is that multidimensional networks introduce a new kind of query, which crosses multiple multidimensional spaces of the network, i.e., more than one cuboid is involved in a query. We thus call such queries crossing different cuboids of the graph cube as cross-cuboid queries, or crossboid queries for short.

Example 5.3.1: Consider the query proposed in Section 5.3 “What is the network structure between the user 3 and various locations?” The answer is shown in Figure 5.5. In the network, the vertex with white color represents user 3 in the cuboid (Gender, Location, Profession), while all the other vertices with grey color are different locations in the cuboid (*, Location, *). Edges indicate relationships between user 3 and her friends grouped by different locations. For instance, this user has 3 friends at Illinois state, represented by the edge with weight 3.

Example 5.3.1 shows a crossboid query in the multidimensional network. This query
Figure 5.6: Traditional Cuboid Queries

definitely has an OLAP flavor in that it involves aggregation upon the network. However, it
deviates significantly from traditional OLAP semantics. In traditional OLAP on relational
data, for example, it does not make sense to query the average income of user 3, a numeric
value, with various locations. Although it is natural to compare user 3’s income with the
average income of users at Illinois, this comparison, however, is orthogonal to OLAP. In the
multidimensional network scenario, aggregation involving multiple cuboids becomes possible
within a single query, which is unique to the graph cube.

For a more graphical explanation of this difference, Figure 5.6 shows a 3-dimensional
data cube on traditional relational data. In this model, queries exist wholly within a single
cuboid. Note cuboid queries discussed in Section 5.3.1 follow this query model as well. For
instance, the answer for “What is the aggregate network between two genders?” comes solely
from the 1-dimensional (Gender) cuboid.

In contrast, the crossboid query in Example 5.3.1 breaks the traditional OLAP seman-
tics and straddles two distinct cuboids in different levels of the graph cube. Figure 5.7
shows this query graphically: the dashed lines between cuboids present the regions in which
crossboid queries are interested. For instance, the right region corresponds to the aggregate
network between base cuboid and the (Location) cuboid. Imagine the black dot inside the 3-dimensional base cuboid is the user 3. The aggregate network shown in Figure 5.5 is the exact answer to this crossboid query if we slice-and-dice the right region only for the user 3. Similarly, the dashed region between the (Gender) cuboid and the (Location) cuboid on the left corresponds to a crossboid query: “what is the network structure between users grouped by gender vs. users grouped by location?” Here the crossboid query straddles two distinct cuboids (Gender) and (Location) in the same level of the graph cube, and the query answer is shown in Figure 5.8.

In general, a crossboid query can include any number of cuboids from the graph cube. As shown in Figure 5.7, three different cuboids can be linked together to form a crossboid. In the rest of the chapter, we focus on the crossboid queries straddling two cuboids, while our model can be generalized to address crossboid queries spanning multiple cuboids.

Figure 5.8: Aggregate Network to the Crossboid Query Straddling (Gender) and (Location) Cuboids
Definition 13. [CROSSBOID QUERY] Given two distinct cuboids $S$ and $T$ in the graph cube, the crossboid query, $S \triangleright T$, is a bipartite aggregate network, $G' = (V'_S \cup V'_T, E', W_{V'}, W_{E'})$, where for each vertex $u$ in the multidimensional network $G$, it is aggregated into a condensed vertex $u' \in V'_S$ w.r.t. $S$ and another condensed vertex $u'' \in V'_T$ w.r.t. $T$, respectively. For each edge $e(u, v) \in G$, it is aggregated into two condensed edges $e'(u', v'')$ and $e'(v', u'')$, respectively, where $u'(v')$ and $u''(v'')$ are the condensed vertices for $u(v)$ to be aggregated to w.r.t. $S$ and $T$, respectively. The weights for both condensed vertices and edges, $W_{V'}, W_{E'}$, are determined in the same way as dictated in Definition 11.

In Definition 13, we abuse the join operator, $\triangleright$, to denote a crossboid query between two cuboids, $S$ and $T$. Given an $n$-dimensional network, the graph cube contains $2^{n-1} \times (2^n - 1)$ crossboids if $S \neq T$ (note $S \triangleright T = T \triangleright S$). More specifically, if $S = T$, crossboid queries boil down to cuboid queries, as discussed in Section 5.3.1. That is, cuboid query is just a special case of crossboid query when $S = T$. Therefore, given a graph cube, there are $2^n$ cuboids and $(2^{2n-1} - 2^{n-1})$ crossboids, respectively, resulting in a total of $(2^{2n-1} + 2^{n-1})$ OLAP queries to be addressed.

Algorithm 6 presents a detailed procedure to address the crossboid query, $S \triangleright T$, from the multidimensional network $G$. It is similar to Algorithm 5, while for each vertex $u$ in the network, we need to aggregate it to cuboid $S$ and $T$, respectively (Lines 3 – 13). And for each edge $e(u, v)$ in the network, we need to create or update two condensed edges $e'(u', v'')$ and $e'(v', u'')$ (Lines 14 – 24). The reason is that edge $e(u, v)$ in the original network is undirected and we need to consider the interaction between two condensed vertices from both directions. The time complexity of Algorithm 6 is $O(|V| + |E|)$. And if there are $|V_S|$ and $|V_T|$ condensed vertices in the aggregate networks w.r.t. the cuboid $S$ and $T$, respectively, the space complexity of Algorithm 6 is $O(|V_S| \times |V_T|)$.

Given a multidimensional network $G$ and two cuboids $S, T$ in the graph cube ($S \neq T$), we can answer the crossboid query, $S \triangleright T$, based on Algorithm 6. However, it becomes extremely inefficient to compute every crossboid from the original network $G$. Can we
Algorithm 6 Crossboid Query Processing

Input: A Multidimensional Network $\mathcal{N} = (V, E, A)$, cuboids $S$ and $T$

Output: The aggregate network $S \bowtie T = (V'_S \cup V'_T, E', W_{V'}, W_{E'})$

begin

1. Initialize two hash structures $\zeta_S : S \rightarrow V'_S$ and $\zeta_T : T \rightarrow V'_T$.
2. for $u \in V$ do
3. if $\zeta_S(S(u)) = \text{NULL}$ then
4. Create a condensed vertex $u' \in V'_S$, with label $S(u) = (S_1(u), \ldots, S_n(u))$
5. $u'.weight \leftarrow 0$
6. $\zeta_S(S(u)) \leftarrow u'$
7. $\zeta_S(S(u)).weight \leftarrow \zeta_S(S(u)).weight + 1$
8. if $\zeta_T(T(u)) = \text{NULL}$ then
9. Create a condensed vertex $u'' \in V'_T$, with label $T(u) = (T_1(u), \ldots, T_n(u))$
10. $u''.weight \leftarrow 0$
11. $\zeta_T(T(u)) \leftarrow u''$
12. $\zeta_T(T(u)).weight \leftarrow \zeta_T(T(u)).weight + 1$
13. for $e(u, v) \in E$ do
14. $u' \leftarrow \zeta_S(S(u))$, $v'' \leftarrow \zeta_T(T(v))$
15. if $e'(u', v'') \not\in E'$ then
16. Create a condensed edge $e'(u', v'') \in E'$
17. $e'(u', v'').weight \leftarrow 0$
18. $e'(u', v'').weight \leftarrow e'(u', v'').weight + 1$
19. $v' \leftarrow \zeta_S(S(v))$, $u'' \leftarrow \zeta_T(T(u))$
20. if $e'(v', u'') \not\in E'$ then
21. Create a condensed edge $e'(v', u'') \in E'$
22. $e'(v', u'').weight \leftarrow 0$
23. $e'(v', u'').weight \leftarrow e'(v', u'').weight + 1$
24. return $G' = (V', E', W_{V'}, W_{E'})$
end
address crossboid queries by leveraging precomputed cuboids in the graph cube? Before giving a positive answer to this question, we first define the nearest common descendant, \( \text{ncd}(S, T) \), of two cuboids \( S, T \) in the graph cube, as follows,

**Definition 14.** \( \text{NCD}(S, T) \) The common descendant set of cuboids \( S \) and \( T \) in a graph cube, \( \text{cd}(S, T) \), is defined as \( \text{cd}(S, T) = \text{des}(S) \cap \text{des}(T) \). Then the nearest common descendant of \( S \) and \( T \), \( \text{ncd}(S, T) \), is a cuboid in the graph cube, such that \( \text{ncd}(S, T) \in \text{cd}(S, T) \), and \( \forall U \in \text{cd}(S, T), \; \text{dim}(U) \subseteq \text{dim}(\text{ncd}(S, T)) \).

**Example 5.3.2:** As shown in Figure 5.2.1 for cuboids \((\text{Gender})\) and \((\text{Profession})\), both the base cuboid and the \((\text{Gender, Profession})\) cuboid are their common descendants. However, only the \((\text{Gender, Profession})\) cuboid is the nearest common descendant.

**Theorem 5.3.2:** Given two cuboids \( S \) and \( T \) in the graph cube \((S \neq T)\), the crossboid query \( S \bowtie T \) can be answered directly from the cuboid \( \text{ncd}(S, T) \).

Based on Theorem 5.3.2, we can compute the crossboid query \( S \bowtie T \) from \( \text{ncs}(S, T) \) instead of the original network. Note \( \text{ncs}(S, T) \) can be easily derived because \( \text{dim}(\text{ncs}(S, T)) = \text{dim}(S) \cup \text{dim}(T) \). In this way, the time complexity of the algorithm becomes \( O(|V_{\text{ncs}(S, T)}| + |E_{\text{ncs}(S, T)}|) \), which is way better than Algorithm 6 because the aggregate network \( w.r.t. \) \( \text{ncs}(S, T) \) is always no greater than the original network.

### 5.4 Implementing Graph Cubes

In order to implement a graph cube, we need to compute the aggregate networks of different cuboids grouping on all possible dimension combinations of a multidimensional network. (Note for a crossboid query, it can be indirectly answered by the nearest common descendant, which is a cuboid in the graph cube as well.) Such implementation of a graph cube is critical to improve the response time of OLAP queries and of operators such as roll-up, drill-down and slice-and-dice. The following implementation alternatives are possible:
1. **Full materialization:** We physically materialize the whole graph cube. This approach can definitely achieve the best query response time. However, precomputing and storing every aggregate network is not feasible for large multidimensional networks, in that we have $2^n$ aggregate networks to materialize and the space consumed could become excessively large. Sometimes it is even hard, if not impossible, to explicitly maintain the multidimensional network itself into main memory.

2. **No materialization:** We compute every cuboid query on request from the raw data. Although no extra space is required for materialization, the query response time can be slow because we have to traverse the multidimensional network once for each such query.

3. **Partial materialization:** We selectively materialize a subset of cuboids in the graph cube, such that queries can be addressed by the materialized cuboids, in light of Theorem 5.3.1 and 5.3.2. Empirically, the more cuboids we materialize, the better query performance we can achieve. In practice, due to the space limitation and other constraints, only a small portion of cuboids can be materialized in order to balance the tradeoff between query response time and cube resource requirement.

There have been many algorithms invented for cube implementation in the context of relational data [85], most of which chose to optimize the partial materialization approach that has proven to be NP-complete by a straightforward reduction from the set cover problem [53]. In [68], the authors further proved that partial materialization is inapproximable if P≠NP. Therefore, the ongoing research was mainly motivated to propose heuristics for sub-optimal solutions. Note the partial materialization problem in the graph cube scenario is still NP-complete because the problem in traditional data cubes can be regarded as a special case in our setting. Therefore the main focus here is to select a set $S$ of $k$ cuboids ($k < 2^n$) in the graph cube for partial materialization, such that the average time taken to evaluate OLAP queries can be minimized.
As it turns out, most of the existing algorithms proposed on data cubes can be used to implement the graph cube with minor modification. We adopt a greedy algorithm for partial materialization on the graph cube. We define the cost, $C(v)$, of a cuboid $v$ in the graph cube as the size of $v$, i.e., $C(v) = (|V'| + |E'|)$, where $G'(V', E')$ is the corresponding aggregate network of $v$. Advanced sampling methods can be used for estimating both $|V'|$ and $|E'|$ of the aggregate network. Assume the set $S$ has already contained some materialized cuboids ($|S| < k$), the benefit to further including $v$ into $S$, denoted by $B(S, v)$, is the total reduction cost of the cuboids in the graph cube if $v$ is involved for cuboid computation. Formally,

$$B(S, v) = \sum_{\text{dim}(u) \subseteq \text{dim}(v)} (C(v) - C(w^*))$$

(5.2)

where

$$w^* = \arg\min(C(w)), w \in \text{des}(u) \cap S$$

That is, we compute the benefit introduced by $v$, which indicates how much it can improve the cost for query evaluation in the presence of $S$. To this point, the greedy algorithm becomes straightforward: we initially include the base cuboid in $S$. Then we iterate for $k$ times and for each iteration, we select the cuboid $v$ with the highest benefit $B(S, v)$ into $S$. Note in practice the network corresponding to the base cuboid is usually too large to be materialized, so we actually compute $(k + 1)$ cuboids in $S$. In this way the base cuboid can be excluded while the other $k$ cuboids are materialized. The time complexity of the greedy algorithm is $O((k + 1)N^2)$, where $N$ is the total number of cuboids in the graph cube, or $O((k + 1)2^n)$, where $n$ is the number of dimensions in the multidimensional network.

**Theorem 5.4.1:** Let $B_{\text{greedy}}$ be the benefit of $k$ cuboids chosen by the greedy algorithm and let $B_{\text{opt}}$ be the benefit of any optimal set of $k$ cuboids, then $B_{\text{greedy}} \leq (1 - 1/e) \times B_{\text{opt}}$ and this bound is tight.
We make no assumption in the greedy algorithm about the query workload and distribution, i.e., all the cuboids in the graph cube will be queried with equal probability. However, it has been shown in the data cube scenario that most OLAP queries and operations are performed only on the cuboids with small number of dimensions, e.g., from 1 to 3 [75]. This evidence still holds for graph cubes. The main reason is that the aggregate networks corresponding to the cuboids with large set of dimensions can be massive and with comparable size to the original multidimensional network. So it is still hard to materialize these large aggregate networks explicitly. On the other hand, users will be easily overwhelmed by the large networks and the insights gained can be limited. Instead, they are more likely to query the cuboids with small sets of dimensions and crosscheck afterwards the corresponding aggregate networks with manageable size, for example, with tens of vertices and edges. Drill-downs are selectively performed only on few cuboids of interest toward the cuboids with medium size. To this end, we propose another heuristic algorithm, MinLevel, to materialize the cuboid $c$ where $\dim(c) = l_0$. $l_0$ is an empirical value specified by users, which indicates the level in the cube lattice at which we start materializing cuboids that contain $l_0$ non-$*$ dimensions.

If all the cuboids with $l_0$ dimensions are included in $S$ and $|S| < k$, we continue choosing cuboids with $(l_0 + 1)$ dimensions, until $|S| = k$. The time complexity of MinLevel is $O(k)$, which is irrelevant to the number of dimensions, $n$. In practice, MinLevel has proven to be a more efficient and practical approach for graph cube materialization, compared to the greedy algorithm.

5.5 Experiments

In this section, we present the experimental results of our proposed method, Graph Cube. We examine two real data sets and our evaluation is conducted from both effectiveness and efficiency perspectives. All our algorithms and experimental methods are implemented in C++ and tested on a Windows PC with AMD triple-core processor 2.1GHz and 3G RAM.
### Table 5.1: Major Conferences Chosen For Each Research Area

<table>
<thead>
<tr>
<th>Area</th>
<th>Conferences</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB</td>
<td>SIGMOD, VLDB, ICDE, PODS, EDBT</td>
</tr>
<tr>
<td>DM</td>
<td>KDD, ICDM, SDM, PKDD, PAKDD</td>
</tr>
<tr>
<td>IR</td>
<td>SIGIR, WWW, CIKM, ECIR, WSDM</td>
</tr>
<tr>
<td>AI</td>
<td>IJCAI, AIAI, ICML, CVPR, ECML</td>
</tr>
</tbody>
</table>

### Table 5.2: Four Buckets of Publication Numbers for the Productivity Attribute

<table>
<thead>
<tr>
<th>Productivity</th>
<th>Publication Number $x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Excellent</td>
<td>$50 &lt; x$</td>
</tr>
<tr>
<td>Good</td>
<td>$21 \leq x \leq 50$</td>
</tr>
<tr>
<td>Fair</td>
<td>$6 \leq x \leq 20$</td>
</tr>
<tr>
<td>Poor</td>
<td>$x \leq 5$</td>
</tr>
</tbody>
</table>

### 5.5.1 Data Sets

We perform our experimental studies on two real-world data sets: DBLP[^2] and IMDB[^3]. Specifically, we will focus the effectiveness study on the DBLP data set and the efficiency study on both data sets. The details of the two data sets are given as follows,

**DBLP Data Set.** This data set contains the DBLP Bibliography data downloaded in March, 2008. We further extract a subset of publication information from major conferences in four different research areas: database (DB), data mining (DM), artificial intelligence (AI) and information retrieval (IR). Table 5.1 shows the conferences we choose for each of the four research areas. We build a co-authorship graph with 28,702 authors as vertices and 66,832 co-author relationships as edges. For each author, there are three dimensions of information: Name, Area, and Productivity. Area specifies a research area the author belongs to. Although an author may belong to multiple research areas, we select one only among the four in which she/he publishes most. For Productivity, we discretize the publication number of an author into four different buckets, as shown in Table 5.2.

**IMDB Data Set.** This data set was extracted from the Internet Movies Data Base (IMDB) in September, 2010. It contains movie information including the following dimension:

[^2]: http://www.informatic.uni-trier.de/~ley/db/
[^3]: http://www.imdb.com/interfaces
5.5.2 Effectiveness Evaluation

We first evaluate the effectiveness of Graph Cube as a powerful decision-support tool in the DBLP co-authorship network. We will present some interesting findings by addressing OLAP queries on the network. The summarized aggregate networks demonstrate a new angle to study and explore massive networks. In the experiments, we are interested in the...
co-authorship patterns between researchers from different perspectives. Upon the graph cube built on the DBLP co-authorship network, we first issue a cuboid query (Area) and the resulting aggregate network is shown in Figure 5.9(a). This aggregate network is a complete graph $K_4$ illustrating the co-authorship patterns between researchers grouped by different research areas. Note different research communities exhibit quite different co-authorship patterns. For example, among the four research areas we study, the DB community cooperates a lot with the IR community and the DM community, while the cooperations between DB and AI are not that frequent. More interestingly, the DB community cooperates most with itself (22, 490 coauthor relationships), compared with other communities. The AI community and the DM community cooperate a lot partially because some common algorithms and methods, e.g., SVM or k-means, are frequently shared by both communities.

Figure 5.9(b) presents another aggregate network corresponding to the cuboid query (Productivity) in the graph cube. This aggregate network illustrates the co-authorship patterns between researchers grouped by different productivity. As shown in the figure, the researchers with poor productivity (with publication number no greater than 5) take up around 91.2% of all the researchers we are examining. For this group of researchers, they cooperate a lot with researchers of fair productivity, while they cooperate much less with researchers of good or excellent productivity. If we define $\text{density}$ of a condensed vertex $v$ as $\text{density}(v) = w_E(e(v,v))/w_V(v)$, where the numerator denotes the weight of the self-loop edge of $v$ and the denominator denotes the weight of $v$ itself, then $\text{density}(\text{Excellent}) = 3.02$, which is much larger than the density values of Poor (1.207) and Fair (1.626) vertices. It means that excellent researchers have formed closer and more compact co-authorship patterns and the in-between cooperations are significantly more frequent than other groups.

If users are interested in zooming into a more fine-grained network for further inspection, a drill-down operation can be performed, or equivalently, a cuboid query (Area, Productivity) is addressed. The resulting aggregate network is a complete graph $K_{16}$, shown in Figure 5.9(c). For the sake of clarity, we only illustrate part of the edges (with large edge
Figure 5.10: Crossboid Queries of the Graph Cube on DBLP Data Set

weights) of the network. In this aggregate network, every vertex is in the (Area, Productivity) resolution, and therefore presents more detailed information for research cooperation. For example, for researcher in the DB community with good productivity (represented as the vertex (DB, Good)), they cooperate most with DB researchers of poor productivity, while they cooperate much less with DM or AI researchers. Interestingly, they cooperate frequently with researchers of poor productivity in IR community as well.

After examining the cuboid queries upon the graph cube, we further address different crossboid queries. Figure 5.10(a) present a crossboid query Area △ Productivity straddling two different cuboids (Area) and (Productivity) in the same level of the graph cube. The aggregate network presents a quite different view of co-authorship patterns by cross-checking interactions between research areas and the productivity of researchers. An interesting finding as shown in Figure 5.7(a) is that, although DB is not the largest research community (actually, AI is the largest one), it consistently attracts the highest number of researchers for cooperation across various levels of productivity, compared with the other three communities. From another direction, excellent researchers cooperate with the DB community most, and then the DM community, followed by the IR and AI community. And for the researchers with poor productivity, they cooperate frequently with the DB and AI community, while their cooperation with DM and IR is much less.

Figure 5.7(b) and Figure 5.7(c) present another crossboid query Area △ Base △ Productivity straddling three cuboids in different levels of the graph cube. We further slice-and-dice
the result to show the cooperation patterns for specific researchers “Hector Garcia-Molina” and “Philip S. Yu”, respectively. From Figure 5.7(b), it is pretty clear that Hector cooperated with researchers in the DB community most, and the number of cooperations is much larger than that in other areas. And he cooperated extensively with researchers in different productivity levels. In contrast, Philip cooperated almost equally frequently with both the DB community and the DM community. And he cooperated more with researchers in poor, fair and excellent productivity.

5.5.3 Efficiency Evaluation

In this section, we evaluate the efficiency of our Graph Cube method. We also test different Graph Cube implementation techniques on multidimensional networks.

We first evaluate our algorithms on the DBLP data set. As this data set contains 3 dimensions only, it is fairly easy to hold all cuboids in main memory. We thus focus on the efficiency of full cube materialization on this data set. The raw network data is first stored on disk and we start building the graph cube based on Algorithm 5, which is a baseline method, denoted as Raw Table. Note for each cuboid in the graph cube, Raw Table has to access the disk for cuboid computation, which is inefficient. Graph Cube adopts a bottom-up method to compute cuboids and the intermediate results can be shared to facilitate the computation of ancestor cuboids, as described in Section 5.3.1. As shown in Figure 5.11(a), for different...
numbers of dimensions in the underlying network, the time consumed for two competing methods is significantly different. **Graph Cube** is consistently faster than **Raw Table**. More specifically, when the dimension value is 3, which means we need to materialize the full cube, **Graph Cube** is about 10 times faster than **Raw Table**. Figure 5.11(b) presents the time used for full cube materialization, while in this case, we start varying the size of the underlying network by changing the number of edges. As illustrated, both methods grow linearly w.r.t. the network size. However, **Graph Cube** outperforms **Raw Table** for 8 – 10 times.

We then perform the same experiments on the IMDB data set. The raw network data is first stored on disk. In this experiment, we explicitly drop the **Name** dimension and keep the remaining 6 dimensions. And the pre-computed cuboids by **Graph Cube** are stored on disk as well because of limited space resource. As illustrated in Figure 5.12(a), **Graph Cube** can compute the full graph cube within 10 minutes. Although the cuboids on low levels still need to access the disk for the pre-computed cuboids, the intermediate aggregate networks are much smaller than the original network. In comparison, **Raw Table** spends around 1,000 seconds when the network dimension is 3, and the time spent grows exponentially large w.r.t. the dimension. **Raw Table** therefore becomes extremely inefficient for the networks with high dimensionality. In Figure 5.12(b), we set the network dimension to be 3 and start varying the network size. As shown, **Graph Cube** still outperforms **Raw Table** for up to 4 times.

We then turn to another implementation alternative to partial-materialize the graph cube. In this experiment, we select a set of 20 cuboids in the graph cube with estimated size
no greater than 1,000 and use them as cuboid queries. We further choose arbitrary pairs of these cuboids to compose another set of 200 crossboid queries. The rationale to choose these queries is that, users seldom explore the aggregate networks whose sizes are larger than 1,000.

We compare two different partial materialization algorithms to address both cuboid queries and crossboid queries: the greedy algorithm, denoted as Greedy, and the heuristic algorithm, \textit{MinLevel}, as described in Section 5.4. We set the materialization level $l_0 = 3$ for MinLevel to start materializing cuboids from the level 3 of the graph cube. The average response time of different queries are reported in Figure 5.13. By varying the number $k$ of cuboids to be materialized into main memory, we notice that \textit{MinLevel} outperforms \textit{Greedy} consistently, for both cuboid queries (Figure 5.13(a)) and crossboid queries (Figure 5.13(b)). The main reason is that, it is of little use to materialize a very large cuboid with great benefit, because this query is seldom issued on the graph cube. Instead, most of the commonly issued queries (with manageable size) can be successfully answered by the materialized cuboids chosen by MinLevel.

5.6 Related Work

As key ingredients in decision support systems (DSS), data warehouses and OLAP have demonstrated competitive advantages for business, and kindled considerable research interest
in the study of multidimensional data models and data cubes [49][22]. In recent years, significant advances have been made to extend data warehousing and OLAP technology from the relational databases to new emerging data in different application domains, such as imprecise data [17], sequences [79], taxonomies [95], text [76] and streams [51]. A recent study by Chen et al. [23] aims to provide OLAP functionalities on graphs. However, the problem definition is different from Graph Cube. In [23], the input data is a set of graphs, each of which contains graph-related and vertex-related attributes. The algorithmic focus is to aggregate (overlay) multiple graphs into a summary static graph. In contrast, Graph Cube focuses on OLAP inside a single large graph. Furthermore, a set of aggregated networks with varying size and resolution is examined in the lens of multidimensional analysis.

Graph summarization is a field closely related to our work. Scientific applications such as DNA analysis and protein synthesis often involve large graphs, and effective summarization is crucial to help biologists solve their problems [89]. One path of approach for graph summarization is to compress the input graph [46][88]. Such compressed graphs can be effectively used to summarize the original graph. Graph clustering [119] is another approach that partitions the graph into regions that can be further summarized. GraSS [73] summarizes graph structures based on a random world model and the target of summarization is to help improve the accuracy of common queries, such as adjacency, degree and eigenvector centrality. And finally, graph visualization [110] addresses the problem of summarization as well. In relation to our work, however, most of the aforementioned studies have not had multidimensional attributes assigned on the network vertices. As a result, the summarization techniques are free to choose the groupings and do not have to respect semantic differences between the vertices. In contrast, Graph Cube approaches the problem from a more data cube and OLAP angle, which has to honor the semantic boundaries to match decision support operations. The systematic aggregation in the multidimensional spaces and also the large network analysis aspects are topics not addressed in the above studies.

One interesting recent work by Tian et al. [105] and Zhang et al. [115] brings an OLAP
flavor to graph summarization. It introduces the SNAP operation and a less restrictive $k$-SNAP operation that will aggregate the graph to a summarized version based on user input of attributes and edge relationships. As the authors mentioned, it is similar to OLAP-style aggregations. In contrast to Graph Cube, $k$-SNAP performs roll-up and drill-down by deceasing and increasing the number $k$ of node-groupings in the summary graph, which is like specifying the number of clusters in clustering algorithms. While for Graph Cube, aggregation and OLAP operations are performed along the dimensions defined upon the network. Moreover, Graph Cube proposes a new class of OLAP queries, crossboid, which is new and has not been studied before.

5.7 Conclusions

In this paper, we have addressed the problem of supporting warehousing and OLAP technology for large multidimensional information networks. Due to the recent boom in large-scale networks, businesses and researchers seek to build infrastructures and systems to help enhance decision-support facilities on networks in order to summarize and maximize the potential value around them. This work has studied this exact problem by first proposing a new data warehousing model, Graph Cube, which was designed specifically for efficient aggregation of large networks with multidimensional attributes. We formulated different OLAP query models for Graph Cube and proposed a new class of queries, crossboid, which broke the boundary of the traditional OLAP model by straddling multiple cuboids within one query. We studied the implementation details of Graph Cube and our experimental results have demonstrated the power and efficacy of Graph Cube as the first, to the best of our knowledge, tool for warehousing and OLAP large multidimensional networks. However, this is merely the tip of the iceberg. The marriage of network analysis and warehousing/OLAP technology brings many exciting opportunities for future study.
Chapter 6

Query Estimation in Streaming Information Networks

6.1 Introduction

Recent scientific and technological advances have resulted in a proliferation of graph-structured data, such as E-R schemas, chemical compounds, biological or social networks, work flows and the Web. Accordingly, many data management and mining applications have been extensively studied in the graph domain \[5, 8, 28\]. However, much of the focus of past research has been on a (or a set of) static graph(s) of relatively modest size. In recent years, numerous information network applications have witnessed streams being defined over the massive graph infrastructure \[32, 40, 43, 33, 6\], in which the entire information network of interest is no longer available all the time, but individual edges of the underlying graph are received and updated rapidly over time in a form of a stream. These newly emerging data are referred to as \textit{streaming information networks}, or \textit{graph streams} for short. Some noteworthy examples of graph streams correspond to the activities overlaid on the Web graphs \[96\], social networks and communication networks. In these cases, the vertices of the graph correspond to different web pages, actors, or IP addresses and the edges represent the links or communication activities among them. Such graph streams may change rapidly in the context of a massive domain of potential edges.

A key property of a graph stream is that it is dynamically updated and the speed of incoming edges can be very high. Therefore, the standard stream constraint of being able to process every edge only once applies naturally in this scenario. Besides its dynamic nature, an additional challenge arises due to the massiveness of a graph stream. In theory,
the number of distinct edges of a graph stream is quadratically related to the number of vertices, and thus it becomes prohibitive to manage such a huge number of edges explicitly. For example, in a social network containing $10^7$ users, the number of distinct edges is of the order of $10^{14}$. The number of interactions, such as the instant messages being sent among individuals at the moment, is even prohibitively larger. Without an efficient storage of the underlying graph, it becomes almost impossible to enable effective query processing on graph streams.

In this paper, we consider some typical queries relevant to graph streams:

- **Edge Query**: We estimate the frequency of particular edges in a graph stream;

- **Aggregate Subgraph Query**: We determine the aggregate frequency behavior of constituent edges of a subgraph.

Despite being primitive, both edge query and aggregate subgraph query are nontrivial in graph streams, and they serve as the building bricks of many advanced querying and mining operations of graph streams [13, 43, 33]. Some applications of such queries are as follows:

1. In social networking applications, vertices represent the participants of a social network, and edges correspond to interactions among the participants. For very large and frequently updated social networks, the underlying graph streams have a rapid rate of incoming edges. An edge query is to estimate the communication frequency between two specific friends, while an aggregate subgraph query is to estimate the overall communication frequencies within a community;

2. In network intrusion applications, vertices of the graph stream represent distinct IP addresses, and edges correspond to network attacks between IP pairs. IP attack packages involving different IP pairs are received dynamically in a stream fashion. An edge query is to estimate the attack frequency between a pair of IP addresses, and an aggregate subgraph query is to estimate the overall attack frequencies of a subset of interconnected IP addresses.
It is possible to design a straightforward solution by building a global synopsis structure corresponding to the entire graph stream for query estimation. Note here any well studied sketch method [9, 31, 31, 26] can be leveraged, which, however, is blind to the underlying structural behavior of graph streams. In real applications, the frequencies of edges are often extremely skewed over different regions of the underlying graph. It is therefore inefficient to use a single global sketch for the whole graph stream, which does not consider such structure-related skewness to its advantage.

In this chapter, we propose a new graph sketch method, gSketch, for querying large graph streams. An important contribution of gSketch is that it resolves the challenges of query estimation by making use of typical local and global structural behavior of graph streams. We consider well known properties of graph streams in real applications, such as the global heterogeneity and local similarity, in conjunction with coarse and easy to compute vertex-specific statistics, to create an intelligent partitioning of the virtual global sketch toward optimizing the overall query estimation accuracy. In this way, incoming queries can be answered by the corresponding partitioned local sketches, upon which the query estimation accuracy can be improved. The motivation to use the vertex-specific statistics of the graph stream during sketch partitioning is twofold. First, the locality similarity within the vicinity of vertices can be effectively encoded and leveraged during sketch partitioning. Second, although the number of potential edges may be too large to be characterized, the number of vertices of a graph stream is often much more modest [40], and the vertex-based statistics can be easily quantified during query estimation. We consider two practical scenarios for sketch partitioning: (1) a graph stream sample is available, and (2) both a stream sample and a query workload sample are available. Efficient sketch partitioning algorithms under different scenarios are designed respectively and our experimental results on both real and synthetic graph streams have demonstrated the effectiveness and efficiency of gSketch. In both scenarios, gSketch achieves up to an order of magnitude improvement in terms of the query estimation accuracy, compared with the state-of-the-art global sketch method.
The remainder of this chapter is organized as follows. We discuss related work in Section 6.2. In Section 6.3 we introduce a general framework for querying graph streams. We will have a broad discussion on how sketches can be used for query estimation in graph streams, and the potential problems of a direct application of a global sketch structure. In Section 6.4 we are focused on the issues and algorithms of sketch partitioning under different scenarios, which are determined by the availability of different sample data. Query processing in the presence of a group of partitioned sketches is detailed in Section 6.5. Section 6.6 contains our experimental studies, and Section 6.7 concludes the chapter.

### 6.2 Related Work

The problem of querying and mining data streams has been studied extensively \[86, 4\] in recent years. The earliest work in the graph domain, however, was proposed in \[58\]. The subsequent work explored methods for counting the number of triangles \[13, 18\], determining shortest paths \[40\], estimating PageRank scores \[33\], mining dense structural patterns \[6\], and characterizing the distinct degree counts of the nodes in the multi-graph scenario \[32\]. An excellent survey on mining and querying graph streams can be found as well \[82\]. Surprisingly, none of the previous work has focused on the query estimation issue on large graph streams.

On the other hand, sketch synopses, including but not limited to AMS \[9\], Lossy Counting \[81\], CountMin \[31\] and Bottom-k \[26\], have proven to be effective data structures in the general stream scenario. Such sketches however may not work well for the case of graph data. For example, they do not consider the underlying correlations of the edge frequencies in the graph stream. That is, only partial information available in graph streams is leveraged in these sketch-based structures. As a result, the sketching methods have to be reexamined and designed specifically to accommodate the new challenges and characteristics inherent in graph streams.
Although the sketch partitioning technique has been proposed in the general data stream domain for join size estimation [36], it is in the context of non-structural data. From an algorithmic point of view, this method works with the AMS method as a base, and attempts to minimize the variance of attribute values within each partition. This is based on how the error of join-size estimation is computed, in which the weighted sum of the variances of join attributes needs to be minimized. This approach is quite different from our sketch partitioning method, gSketch, on graph streams. In gSketch, we make use of the structural frequency behavior of vertices in relation to the edges for sketch partitioning. In other words, the structural nature of a graph stream makes it quite different from the former sketch-partitioning problem, which has applied to the multi-dimensional data.

6.3 The Algorithmic Framework

In this section, we will discuss an algorithmic framework for query estimation in graph streams. We first formulate the problem of query estimation for the case of graph streams. A straightforward solution is then proposed to construct a global sketch for the entire graph stream. The main limitation of this global sketch method is that the structural properties of graph streams are totally ignored during query estimation. Such limitation also motivates us to consider leveraging the underlying structural properties of graph streams, thus resulting in our sketch-partitioning based solution, gSketch.

6.3.1 Problem Definition

Given a graph stream, we assume its underlying graph $\mathcal{G}$ can be defined as $\mathcal{G} = (V, E)$, where $V$ is a vertex set of $\mathcal{G}$. For each vertex $x \in V$, there is a string $l(x)$ attached to $x$ as the label of $x$. $E$ is the edge set of $\mathcal{G}$ and every edge $(x, y) \in E$ is a directed edge.\(^1\) The incoming graph

\(^1\)In the event of an undirected graph, lexicographic ordering on vertex labels can be used in order to determine the direction of the edge. Ties are broken arbitrarily.
stream contains elements \((x_1, y_1; t_1), (x_2, y_2; t_2), \ldots (x_i, y_i; t_i) \ldots \), where the edge \((x_i, y_i)\) is encountered at the time-stamp \(t_i\). In some applications, a frequency \(f(x_i, y_i, t_i)\) is associated with the edge \((x_i, y_i)\) at \(t_i\). For example, in a telecommunication application, the frequency \(f(x_i, y_i, t_i)\) may denote the number of seconds in a phone conversation from a person \(x_i\) to another person \(y_i\) starting at the time-stamp \(t_i\). If not specified explicitly, we assume \(f(x_i, y_i, t_i) = 1\) by default.

Hitherto, the graph stream as defined above has been frequently encountered in a number of application domains such as network intrusion detection, telecommunication, and social networks. Some representative queries in such graph streams can be formulated as follows:

- **Edge Query**: To estimate the overall frequency of the edge \((x, y)\): \(\tilde{f}(x, y) = \sum_{t_i \in T} f(x, y, t_i)\), where \(T\) can be the lifetime of the graph stream or a specific time window of interest.

- **Aggregate Subgraph Query**: To estimate the aggregate frequency of the constituent edges of a subgraph \(G = \{(x_1, y_1), \ldots, (x_k, y_k)\}: \tilde{f}(G) = \Gamma(\tilde{f}(x_1, y_1), \ldots, \tilde{f}(x_k, y_k))\), where \(\Gamma(\cdot)\) is an aggregate function of interest, such as \(\text{MIN}(\cdot)\) or \(\text{AVERAGE}(\cdot)\). For example, when \(\Gamma(\cdot) = \text{SUM}(\cdot)\), it can summarize the total frequency of all the edges of the subgraph \(G\), i.e., \(\tilde{f}(G) = \sum_{i=1}^{k} \tilde{f}(x_i, y_i)\).

Aggregate subgraph query is essentially a derivative of edge query in the sense that it is performed on a bag of edges belonging to a subgraph, so it can be naturally resolved by the use of the function \(\Gamma(\cdot)\) on all query results of the constituent edge queries. Therefore, we will mostly be focused on edge query estimation. The generalization from edge query towards aggregate subgraph query is straightforward and will be discussed further in Section 6.6.

### 6.3.2 A Global Sketch Solution

In this section, we will discuss a straightforward solution, denoted as **Global Sketch**, for query estimation in graph streams. **Global Sketch** is an intuitive application of any sketch method. 


for summarizing the entire graph stream, provided that the edges of the graph stream can be represented and accommodated appropriately in the traditional data stream scenario. Our discussion will focus on one specific sketch method, CountMin [31], while our analysis below can be easily generalized toward other sketch methods in an analogous way.

A CountMin sketch consists of a 2-dimensional array with a width of \( w = \lceil e/\epsilon \rceil \) and a depth of \( d = \lceil \ln(1/\delta) \rceil \), and thus there exist \( w \cdot d \) cells in total in the sketch. Here \( e \) is the base of the natural logarithm. \( \epsilon \) and \( \delta \) are user-specified parameters, which imply that the error of answering a query is within a factor of \( (1 + \epsilon) \) of the true value with probability at least \( (1 - \delta) \). In the CountMin sketch, we select \( d \) pairwise independent hash functions, \( h_1, \ldots, h_d \). Each \( h_i \) \( (1 \leq i \leq d) \) uniformly maps onto random integers in the range \([0, w - 1]\) and corresponds to one of \( d \) 1-dimensional arrays with \( w \) cells each. These \( d \) hash functions are used to update the frequency of an element from a data stream on different mapping cells in the 2-dimensional data structure. For example, consider a data stream with elements drawn from a universe of domain values. When a new element \( e_t \) is received at the timestamp \( t \), we apply each of the \( d \) hash functions upon \( e_t \) to map onto a number in \([0 \ldots w - 1]\). The value of each of \( d \) cells, \( h_i(e_t) \), is incremented accordingly by 1. In order to estimate the frequency of the element during query processing, we choose the set of \( d \) cells onto which the \( d \) hash-functions map, and determine the minimum value among all these \( d \) cells as the query estimation result. An example of a CountMin sketch is illustrated in Figure 6.1.

Figure 6.1: A CountMin Sketch
Theoretically, if \( f \) is the true frequency value of the element \( e_t \), the estimated frequency, \( \tilde{f} \), can be lower bounded by \( f \), because we are dealing only with non-negative frequencies, and value collisions during hashing can only cause overestimation. A probabilistic upper-bound of \( \tilde{f} \) can be determined as well \[31\]. Given a data stream with \( N \) arrivals till the time-stamp \( t \), the estimate \( \tilde{f} \) is at most \( f + e \cdot N/w \) with probability at least \( 1 - e^{-d} \), i.e., w.h.p.,

\[
f \leq \tilde{f} \leq f + e \cdot N/w
\] (6.1)

Note that the probability of the error-bound being violated reduces exponentially with \( d \).

The CountMin sketch has proven to be accurate for many practical scenarios in traditional data streams \[31\].

Analogously, the CountMin sketch can be directly applied on graph streams for query estimation by treating each edge as an element with a unique identifier. We note that the edge \((x_i, y_i)\) can be represented as a string \( l(x_i) \oplus l(y_i) \) where \( \oplus \) is the concatenation operator on the vertex labels of \( x_i \) and \( y_i \). This string can be hashed as the key of the edge \((x_i, y_i)\) onto the CountMin sketch to maintain the frequency of \((x_i, y_i)\).

However, such an approach, Global Sketch, has proven to be ineffective in the graph stream domain. This is because when \( N \) edges have been received in a CountMin sketch with hash range \( w \), the (absolute) query estimation error \(|f - \tilde{f}|\) is proportional to \( N/w \), as shown in Equation (6.1). Therefore, the relative estimation error of an edge query with frequency \( f \) is proportional to \( N/(w \cdot f) \), which can be extremely large for small values of \( f \) and large values of \( N \). In practice, the edge frequency distribution of a graph stream can be quite uneven and those low frequency portions of the underlying graph can be very relevant for querying, and may show up repeatedly in the workload. Furthermore, the number of edges \( N \) can be extremely large for graph streams. Therefore Global Sketch may generate inaccurate estimation results. Due to the natural vulnerability and limitations of Global Sketch, we need to design a more effective approach specifically for massive graph streams.
6.3.3 Broad Intuition for a Better Solution

The graph streams such as those found on the Web and various information network applications are typically not random. They often exhibit both local and global structural properties which are potentially useful in sketch construction and query estimation. Some common properties are characterized as follows:

- **Global Heterogeneity and Skewness**: The relative frequency distribution of different edges in a massive graph stream is nonuniform and often observed extremely uneven \cite{39,91};

- **Local Similarity**: Within structurally localized regions of the graph, relative frequencies of the edges are often correlated with one another \cite{39,14,25}. Although this does not mean that the frequency behavior is identical within a locality, the correlations of the edge frequencies in a local region are considerable.

These empirical observations provide us with useful hints for the design of a more effective graph sketch method, denoted as \texttt{gSketch}, for query estimation. The key idea of \texttt{gSketch} is to partition a virtual global sketch corresponding to the entire graph stream to a set of smaller localized sketches, such that edges of different structural localities in the underlying graph can be mapped onto different partitioned local sketches. Therefore edges with similar frequency correlation can be updated and queried by the same local sketch during stream maintenance and query estimation. In this way, the estimation error of each local sketch is much lower compared with the case when a single global sketch is used. This ultimately helps improve the overall query estimation accuracy of \texttt{gSketch} over that of \texttt{Global Sketch}. It is worth noting that most often the data samples of a graph stream are readily available. In some other cases, the query workload samples may also be available. Therefore, it becomes possible for us to fully leverage such sample information with encoded structural properties of graph streams for effective sketch partitioning.
6.4 Sketch Partitioning

In this section, we will introduce our sketch partitioning algorithms, which are performed as a pre-processing step on the sample data before the actual sketch structures are populated with the graph stream. Our goal of sketch partitioning is to maintain the graph streams with sufficient frequency uniformity within each partitioned sketch, such that the query estimation can be optimized over the entire graph stream. Each localized sketch in the partitioning is designed for summarizing the edge frequencies associated with particular source vertices. Therefore, it becomes much easier to maintain the sketch partitioning information in main memory, as the number of vertices is significantly smaller than the number of edges in the underlying graph.

As discussed earlier, we consider two sampling scenarios:

1. In the first scenario, a sample of the original graph stream is available. However, specific information about query workloads is unavailable.

2. In the second scenario, a sample of the original graph stream as well as a sample of the query workload is available. In this case, the sketch partitioning can be further optimized with the additional information of query workloads.

Before discussing the specific algorithms, we will introduce some notations and definitions. We denote the frequency of edge \((i, j)\) by \(f(i, j)\), where \(i, j \in V\). This value is the one to be estimated during query processing, and is therefore not explicitly available. In fact, the edge frequency cannot even be explicitly stored in the case of graph streams because the number of potential edges can be exponentially large. The relative frequency of a vertex \(i\), denoted as \(f_v(i)\), is defined as the sum of the frequencies of the edges emanating from \(i\), i.e.,

\[
 f_v(i) = \sum_j f(i, j) \quad (i, j) \in E
\] (6.2)
The *out degree* of a vertex $i$, denoted as $d(i)$, is defined as follows:

$$d(i) = \sum_j \theta(i, j) \quad (i, j) \in \mathcal{E}$$  \hspace{1cm} (6.3)

where

$$\theta(i, j) = \begin{cases} 
0 & \text{if } (i, j) \text{ is not in the graph stream} \\
1 & \text{otherwise}
\end{cases}$$

### 6.4.1 Sketch Partitioning with Data Sample

In this section, we will discuss the process of sketch partitioning in the presence of a data sample only. In order to construct the sketch partitions, we would like to group together structural regions of the graph stream with similar frequency behavior, which ultimately helps optimize the query estimation accuracy. However, since we are trying to estimate the edge frequencies to begin with, this frequency information for associating the edges with the corresponding sketch partitions is unfortunately not available directly. Therefore, it would seem that there is no practical way to ensure the regions with similar frequencies are assigned to the same partition. However, as discussed in Section 6.3.3, it is possible to exploit the structural properties of graph streams to efficiently approximate the frequency behavior of the edges in different structural localities and create the sketch partitioning accordingly. In order to make the analysis clearer, we first make an oracle assumption that the frequency $f(i, j)$ of the edge $(i, j)$ over the entire graph stream is known in advance. Later, we will relax this assumption by leveraging the structural characteristics of graph streams for frequency estimation.

Let us assume that there is a total of $r$ ($r \geq 1$) partitions of the global CountMin sketch and $S_i$ is the localized sketch corresponding to the $i$th partition ($1 \leq i \leq r$). The total space, which is essentially the available main memory, is allocated equally to each partition by evenly dividing the width of the global CountMin sketch while keeping the depth of each
partitioned sketch to be the same as that of the global CountMin sketch. In other words, the width of $S_i$ is $w_i = w/r$ and the depth of $S_i$ is $d_i = d$, where $w$ and $d$ are the width and depth of the global sketch, respectively. In this way, we can ensure the same probabilistic guarantee of frequency estimation, $(1 - e^d)$, across all partitioned sketches, as indicated in Equation (6.1).

Let $F(S_i)$ be the sum of the edge frequencies in the $i$th sketch, $S_i$, and $(m,n)$ is such an edge that is associated with $S_i$ for frequency maintenance and querying. Then, the expected frequency of $(m,n)$, denoted by $\bar{f}(m,n)$, when hashed into a cell of $S_i$ because of erroneous collisions, is determined by

$$\bar{f}(m,n) = (F(S_i) - f(m,n))/w_i$$

Therefore, the expected relative error of the edge $(m,n)$ is given by

$$\bar{e}(m,n) = \bar{f}(m,n)/f(m,n) = F(S_i)/(f(m,n) \cdot w_i) - 1/w_i$$

for any particular row in the sketch $S_i$. If the depth $d$ of the sketch $S_i$ is 1, the overall relative error, $E_i$, of the sketch $S_i$ is

$$E_i = \sum_{(m,n) \in S_i} \bar{e}(m,n) = \sum_{(m,n) \in S_i} \left(F(S_i)/(f(m,n) \cdot w_i) - 1/w_i\right)$$

(6.4)

Then the optimization problem of sketch partitioning can be formulated as follows:

**Problem 6.4.1:** Partition the global sketch into $r$ localized sketches $S_1, \ldots, S_r$ based on the edge set of the data sample, with the objective to minimize $\sum_{i=1}^r E_i$, where $E_i$ is formulated in Equation (6.4).

Let us consider a simplification of this optimization problem in which we wish to construct $r = 2$ partitions. This is a difficult problem since it can be recast as a 0-1 integer program with a non-linear objective function [90]. There can be an exponential number of solutions
to the problem and it is hard to determine the optimal one. We therefore seek an alternative solution. Our idea is to sort the edges in the global sketch in nondecreasing order of edge frequencies and consider those partitions containing edges in contiguously sorted order. The number of such partitions is equal to the number of edges, since we can choose the partition pivot at each possible position in the sorted order. However, the optimal partition pivot is picked at which the objective function in Equation (6.4) is minimized.

Unfortunately this solution is still not quite implementable, since we do not know the edge frequencies to begin with. Furthermore, the initial data sample is assumed to have a fairly small size compared to the actual graph stream. This means that the data sample cannot be reliably used to estimate the frequency of every edge in the graph stream. However, it can be effectively used to estimate the relative frequencies of vertices, as defined in Equation (6.2). Based on the property of local similarity of graph streams as described in Section 6.3.3, we alternatively use the frequency behavior of the vertices to perform the sketch partitioning. We denote the estimated relative frequency of a vertex \( m \) by \( \tilde{f}_v(m) \), and the estimated out degree of \( m \) by \( \tilde{d}(m) \). Both estimated values are derived from the data sample. Then the average frequency of the edges emanating from the vertex \( m \) is determined by \( \tilde{f}_v(m)/\tilde{d}(m) \). That is, we assume \( \tilde{d}(m) \) edges emanate from the vertex \( m \) with an average frequency of \( \tilde{f}_v(m)/\tilde{d}(m) \). And the total estimated frequencies of the edges in the partitioned sketch \( S_i \) (\( 1 \leq i \leq r \)), denoted as \( \tilde{F}(S_i) \), can be expressed as

\[
\tilde{F}(S_i) = \sum_{m \in S_i, m \in V} \tilde{f}_v(m) \tag{6.5}
\]

As a result, analogous to Equation (6.4), the overall relative error \( E_i \) of the partitioned sketch \( S_i \) with the use of vertex frequency-based statistics can be redefined as follows:

\[
E_i = \sum_{m \in S_i} \frac{\tilde{d}(m) \cdot \tilde{F}(S_i)}{w_i \cdot \tilde{f}_v(m)/\tilde{d}(m)} - \sum_{m \in S_i} \frac{\tilde{d}(m)}{w_i} \tag{6.6}
\]
Note that $\tilde{d}(m)$ in the numerator accounts for the fact that there are $O(\tilde{d}(m))$ edges emanating from the vertex $m$. The optimization problem of sketch partitioning is therefore transformed to the following form:

**Problem 6.4.2:** Partition the global sketch into $r$ localized sketches $S_1, \ldots, S_r$ based on the vertex set of the data sample, with the objective to minimize $\sum_{i=1}^{r} E_i$, where $E_i$ is formulated in Equation (6.6).

As in the previous case, an approximate solution to this problem is to first sort the vertices in the data sample in order of average frequency, $\bar{f}_v(m)/\tilde{d}(m)$, and then pick the partition pivot at which the objective function, as formulated in Problem 6.4.2 can be minimized.

By partitioning the global sketch based on the vertices, rather than the edges, we essentially create a set of localized sketches on different structural portions of the graph stream. The advantages of this vertex-based sketch partitioning approach are as follows. First of all, it intelligently relaxes the oracle assumption of Problem 6.4.1 and successfully transforms this hard optimization problem to a more tractable one, as described in Problem 6.4.2. Second, due to the sparsity of the data sample, the query estimation accuracy can be extremely low if the edge-based sketch partitioning approach is adopted. Instead, the vertex-based partitioning principle takes advantage of the local similarity property of the graph stream, which leads to a much more reliable and robust sketch partitioning method. Last but not least, the vertex-based partitioning information is compact and easy to compute. This enables an efficient storage and maintenance of $gSketch$.

The analysis above suggests a natural way of constructing sketch partitions in a top-down recursive fashion as in decision trees. We call such a partitioning mechanism as a **partitioning tree**. At the first step, we have an initial root node $S$ representing the virtual global **CountMin** sketch with all the available space. The node is then split into two children $S_1$ and $S_2$, and the space allocated to either branch of $S$ is the same. This is done by evenly partitioning the width of the $\text{CountMin}$ sketch corresponding to $S$ between two branches.
rooted with $S_1$ and $S_2$, respectively. In order to optimize such a partitioning of $S$ into $S_1$ and $S_2$, we need to minimize the objective function as expressed in Problem [6.4.2], which corresponds to the summation $E$ below:

\[ E = E_1 + E_2 \]
\[ = \sum_{m \in S_1} \frac{\tilde{d}(m) \cdot \tilde{F}(S_1)}{w_1 \cdot \tilde{f}_v(m) / \tilde{d}(m)} + \sum_{m \in S_2} \frac{\tilde{d}(m) \cdot \tilde{F}(S_2)}{w_1 \cdot \tilde{f}_v(m) / \tilde{d}(m)} - \sum_{m \in S_1 \cup S_2} \frac{\tilde{d}(m)}{w_1} \] (6.7)

Note the sketch widths of $S_1$ and $S_2$ are equal, i.e., $w_1 = w_2$. We therefore use $w_1$ uniformly throughout the expression in Equation (6.7). In order to further simplify the expression, we define an alternative expression $E'$ as

\[ E' = E \cdot w_1 + \sum_{m \in S_1 \cup S_2} \tilde{d}(m) \] (6.8)

It is obvious that $E$ is optimized whenever $E'$ is optimized. This is because $w_1$ is positive, and $\sum_{m \in S_1 \cup S_2} \tilde{d}(m)$ is a constant irrespective of how the partitioning of $S$ into $S_1$ and $S_2$ is performed. We simplify the value of $E'$ as follows:

\[ E' = \sum_{m \in S_1} \frac{\tilde{d}(m) \cdot \tilde{F}(S_1)}{\tilde{f}_v(m) / \tilde{d}(m)} + \sum_{m \in S_2} \frac{\tilde{d}(m) \cdot \tilde{F}(S_2)}{\tilde{f}_v(m) / \tilde{d}(m)} \] (6.9)

To this end, we evaluate the value of $E'$ over all possible partitions of $S$ in sorted order of $\tilde{f}_v(m)/\tilde{d}(m)$. Note there are as many choices of the partition pivot as the number of vertices in $S$, and we pick the one for which the value of $E'$ in Equation (6.9) is minimized. After $S$ is partitioned into $S_1$ and $S_2$, they themselves form the next pair of decision nodes for further partitioning consideration in the partitioning tree. For the internal nodes of the partitioning tree, we do not explicitly construct the corresponding sketches. Instead, we use them to maintain information for further sketch partitioning. This partitioning process is performed recursively until one of the following two termination criteria is met:
1. The width of a sketch $S$ at a given level is less than a particular threshold $w_0$, i.e., $S.width < w_0$;

2. The number of distinct edges being counted within a sketch $S$ is no greater than a constant factor $C$ of the sketch width, i.e., $\sum_{m \in S} \tilde{d}(m) \leq C \cdot S.width$.

In the first case, we do not further partition the sketch but build and materialize it explicitly, because the sketch of this kind is considered small enough and further partitioning will incur more collisions and therefore may hurt the final query estimation accuracy. The second termination case is determined by the following theorem:

**Theorem 6.4.1:** For a given sketch $S$ and a nonnegative constant $C$ ($0 < C < 1$), s.t., $\sum_{m \in S} \tilde{d}(m) \leq C \cdot S.width$, the probability of any collision in a particular cell of $S$ can be bounded above by $C$.

**Proof:** We denote the hash function of a specific row of $S$ as $h(\cdot)$. Given two distinct edges $i$ and $j$, the probability of collision between $i$ and $j$ in a particular cell of $S$ can be determined as

$$\Pr(h(i) = h(j)) \leq 1/S.width$$

There are $\sum_{m \in S} \tilde{d}(m)$ distinct edges associated with the sketch $S$. By pairwise independence of the collision probability of distinct edges, we note the probability of any collision with the edge $i$ is

$$\sum_j \Pr(h(i) = h(j)) \leq \frac{\sum_{m \in S} \tilde{d}(m)}{S.width} \leq C$$

Therefore, the probability of any collision in a particular cell of $S$ is no greater than $C$. \qed

Intuitively, if the number of distinct edges within a sketch $S$ is small enough (bounded up by a constant factor of the width of $S$), the probability of collisions within $S$ will be small, and therefore $S$ can be directly used as a high quality localized sketch for query estimation without further partitioning. In practice, we further set the width of such sketches to the modest value of $\sum_{m \in S} \tilde{d}(m)$. It helps save extra space which can be allocated to other
Algorithm 7 Sketch Partitioning with Data Samples

Input: A data sample \( D \)

Output: A partitioning tree rooted with \( S \)

begin
  Create a root node \( S \) of the partitioning tree as an active node
  \( S.width = w = [e/\epsilon] \)
  \( S.depth = d = [\ln \frac{1}{\delta}] \)
  Create an active list \( L \) containing \( S \) only
  while \( L \neq \emptyset \) do
    Partition an active node \( S \in L \) based on \( D \) into \( S_1 \) and \( S_2 \) by minimizing \( E' \) in Equation (6.9)
    \( S_1.width = S_2.width = S.width/2 \)
    \( L = L \setminus \{S\} \)
    if \( (S_1.width \geq w_0) \) and \( (\sum_{m \in S_1} \tilde{d}(m) > C \cdot S_1.width) \) then
      \( L = L \cup S_1 \)
    else
      Construct the localized sketch \( S_1 \)
    if \( (S_2.width \geq w_0) \) and \( (\sum_{m \in S_2} \tilde{d}(m) > C \cdot S_2.width) \) then
      \( L = L \cup S_2 \)
    else
      Construct the localized sketch \( S_2 \)
  return The partitioning tree rooted with \( S \)
end

The sketch partitioning algorithm (with data sample only) is illustrated in Algorithm 7. We now give a detailed complexity analysis of the algorithm. In the partitioning tree, every internal node is split into two nodes for further inspection (Lines 7–8). In the worst case, the partitioning tree can be a complete binary tree with a height of \( \log(w/w_0) \), and the number of internal nodes in the partitioning tree can be at most \((2^{\log(w/w_0)} - 1)\), i.e., \((w/w_0 - 1)\), which is also the number of active nodes to be processed in \( L \) (Line 6). For each internal node of the partitioning tree, we need to sort corresponding vertices and select the pivot at which the objective function can be minimized. The complexity of such operations is at most
O(|D|\log|D|). Therefore, the overall complexity of the algorithm is O((w/w_0 - 1)|D|\log|D|).

6.4.2 Sketch Partitioning with Data and Workload Samples

In this section, we further assume that a query workload sample is available in addition to the data sample, and discuss how it can be exploited for more effective partitioning. In this scenario, it is possible to estimate the relative weights of different edge queries in the presence of the query workload sample. More specifically, the relative weights of vertices are estimated from the query workload sample and then incorporated into the sketch partitioning process.

The relative weight of a vertex $n$ is the relative frequency of edges emanating from $n$ to be used in the querying process, and can be derived from the query workload sample. Let $\tilde{w}(n)$ be the relative weight of the vertex $n$ in the query workloads. In this scenario, the vertex based relative error, $E_i$, of the $i$th partitioned sketch, $S_i$, can be formulated as follows:

$$E_i = \sum_{n \in S_i} \frac{\tilde{w}(n) \cdot \tilde{F}(S_i)}{\tilde{w}_i \cdot \tilde{f}_v(n)/\tilde{d}(n)} - \sum_{n \in S_i} \frac{\tilde{w}(n) / w_i}{\tilde{w}_i}$$  \hspace{1cm} (6.10)

This condition is similar to that formulated in Equation (6.6) for the data sample scenario. The difference is the term $\tilde{w}(n)$ in the numerator, which has been introduced in order to account for queries emanating from the vertex $n$.

During sketch partitioning, a given node $S$ in the partitioning tree is split into two nodes $S_1$ and $S_2$, such that the overall relative error is minimized. The objective function in this scenario can be formulated as follows:

$$E' = \sum_{n \in S_1} \frac{\tilde{w}(n) \cdot \tilde{F}(S_1)}{\tilde{f}_v(n)/\tilde{d}(n)} + \sum_{n \in S_2} \frac{\tilde{w}(n) \cdot \tilde{F}(S_2)}{\tilde{f}_v(n)/\tilde{d}(n)}$$  \hspace{1cm} (6.11)

We sort the vertices in order of $\tilde{f}_v(n)/\tilde{w}(n)$ and perform the sketch partitioning at the pivot with which the objective function $E'$ in Equation (6.11) is minimized. The sketch partitioning algorithm for this scenario is shown in Figure 8. The major difference here is
Algorithm 8 Sketch Partitioning with Data and Workload Samples

\textbf{Input:} A data sample \( D \), a query workload sample \( W \)

\textbf{Output:} A partitioning tree rooted with \( S \)

\begin{algorithmic}[1]
\State Create a root node \( S \) of the partitioning tree as an active node
\State \( S.\text{width} = w = \lceil e/\epsilon \rceil \)
\State \( S.\text{depth} = d = \lceil \ln \frac{1}{\delta} \rceil \)
\State Create an active list \( L \) containing \( S \) only
\While {\( L \neq \emptyset \)}
\State Partition an active node \( S \in L \) based on \( D \) and \( W \) into \( S_1 \) and \( S_2 \) by minimizing \( E' \) in Equation (6.11)
\State \( S_1.\text{width} = S_2.\text{width} = S.\text{width}/2 \)
\State \( L = L \setminus \{S\} \)
\If {\( (S_1.\text{width} \geq w_0) \) and \( (\sum_{m \in S_1} \tilde{d}(m) > C \cdot S_1.\text{width}) \)} \( L = L \cup S_1 \)
\Else
\State Construct the localized sketch \( S_1 \)
\EndIf
\If {\( (S_2.\text{width} \geq w_0) \) and \( (\sum_{m \in S_2} \tilde{d}(m) > C \cdot S_2.\text{width}) \)} \( L = L \cup S_2 \)
\Else
\State Construct the localized sketch \( S_2 \)
\EndIf
\EndWhile
\State \textbf{return} The partitioning tree rooted with \( S \)
\end{algorithmic}

that we make use of both the data sample and the workload sample for sketch partitioning and the objective function is determined by Equation (6.11). Similarly, the worst-case time complexity of the algorithm in this scenario is \( O((w/w_0 - 1)|D|\log|D|) \).

\section{6.5 Query Processing}

Sketch partitioning is a pre-processing step to determine the association of vertices in the data sample to different partitioned localized sketches. More specifically, we maintain a hash structure \( \mathcal{H} : V \to S_i, \ 1 \leq i \leq r \). For an edge \((m, n)\) in the graph stream, it can be hashed onto the localized sketch \( \mathcal{H}(m) = S_i \) for frequency update and querying. Although this hash structure \( \mathcal{H} \) is an extra overhead that needs to be stored along with the sketches, the cost is marginal compared to the immense advantages of sketch partitioning. Furthermore,
we do not need to explicitly store the hierarchical structure of the partitioning tree. Only the partitioned sketches represented by the leaf nodes in the partitioning tree need to be physically stored for querying purposes.

After the off-line sketch partitioning phase, the resulting partitioned sketches are then populated with the massive graph stream in an online fashion and start supporting the query processing and estimation simultaneously. As the graph stream arrives, we use the hash structure $H$ to associate incoming edges to the corresponding localized sketches, and update the edge frequencies in the corresponding sketch. During the querying phase, we analogously first determine the relevant partitioned sketch to which an edge query is associated. Once the sketch is identified by $H$, the edge query can then be answered specifically by that sketch.

A special case is the one in which a particular vertex occurring in the graph stream does not occur in the original data sample. For edges which contain such vertices, a fixed portion of the original space is allocated as an outlier partition and a separate outlier sketch is constructed accordingly to count the frequencies of these edges. For the purpose of querying, those vertices which do not occur in any partition are resolved by this special outlier sketch. It is important to understand that the real graph streams are often considerably skew in vertex presence in data samples. The vertices involved in those edges with high frequency will typically be present in the sample as well. Therefore, it leaves only a small fraction of the overall frequency to be processed by the outlier sketch. Recall that the estimation error of the sketch-based methods is dependent upon the overall frequency of the items added to the sketch. Since most of the high frequency edges have already been skimmed off, the estimation results from the outlier sketch can still be accurate. Thus, even in the presence of new vertices in the graph stream, gSketch can achieve satisfactory query estimation results because of the removal of most of the high-frequency edges from the outlier sketch.

We note that the confidence intervals of the CountMin sketch method apply within each localized partition of gSketch. Since the number of edges assigned to each of the partitions is known in advance of query processing, it is possible to know the confidence of each partic-
ular query. Therefore, the confidence intervals of different queries are likely to be different depending upon the sketches that they are assigned to. On the average, the confidence intervals of different sketch partitions are likely to be similar to that of the global sketch with the same amount of space. However the exploitation of structural properties in gSketch leads to much better experimental behavior. We will present these experimental advantages in Section 6.6.

Users may sometimes be interested in dynamic queries over specific windows in time. For example, a user may be interested in the frequency behavior of edges in the past one month, one year and so on. In such cases, it makes sense to divide the time line into temporal intervals (or windows), and store the sketch statistics separately for each window. The partitioning in any particular window is performed by using a sample, which is constructed by reservoir sampling from the previous window in time. Queries over a specific time-interval can be resolved approximately by extrapolating from the sketch time windows which overlap most closely with the user-specified time window.

For the case of aggregate subgraph query, we first decompose the query into a bag of constituent edges and then sequentially process each edge as a separate edge query against the graph stream. Each such edge is first mapped to the appropriate sketch, and then is estimated by that sketch. After that, all estimated answers of the constituent edges are summarized by the aggregate function, \( \Gamma(\cdot) \), as the final estimation result of the aggregate subgraph query.

6.6 Experimental Results

In this section, we report our experimental findings on query estimation in graph streams. We compare our gSketch method with Global Sketch, which makes use of a global sketch for the entire graph stream in order for query estimation. Our experiments are evaluated in both scenarios characterized by the availability of data samples and query workload samples.
All our experiments are performed on an Intel PC with a 3.4 GHz CPU and 3.2GB main memory, running Window XP Professional SP3. All algorithms including gSketch and Global Sketch are implemented in C++.

### 6.6.1 Data Sets

We choose two real data sets and one synthetic data set in our experimental studies. Two of the data sets are publicly available, while one real data set is extracted from a large cooperate sensor network. The details of each data set are elaborated as follows.

**DBLP.** The DBLP database\(^2\) contains scientific publications in the computer science domain, and we extract all conference papers ranging from 1956 to March 15th, 2008 for our experimental studies. There are 595,406 authors and 602,684 papers in total. We note that for a given paper, the authors are listed in a particular order as \(a_1, a_2, \ldots, a_k\). An ordered author-pair \((a_i, a_j)\) is then generated if \(1 \leq i < j \leq k\). There are 1,954,776 author-pairs in total, which are considered as streams of the underlying co-authorship graph, and are input to gSketch and Global Sketch in a chronological order.

**IP Attack Network.** Our second real data set is IP attack streaming data extracted from a corporate sensor network. The data set was initially collected from January 1st, 2007 to June 11th, 2007 comprising IP attack packet data from sensors. For each IP attack transaction, the attack type, time-stamp, sensor information, source IP address, target IP address and vulnerability status are recorded. We extract the source IP address and the target IP address of each IP attack packet to compose graph streams and select a time-frame from January 1st, 2007 to January 5th, 2007 as the time window of interest. This data set contains 3,781,471 edges in total.

**GTGraph.** The synthetic data set is generated by the well-known graph generator GTGraph\(^3\). A large network \(G\) with power-law degree distributions and small-world charac-

\(^2\)http://www.informatik.uni-trier.de/~ley/db/
\(^3\)http://www.cc.gatech.edu/~kamesh/GTgraph/index.htm
teristics is generated based on R-MAT model \cite{21}. We choose default values of parameters during network generation, as suggested by the authors. The generated network contains $10^8$ vertices and $10^9$ edges, and the edges of $G$ are used as graph streams for experimental evaluation.

In order to verify the common characteristics of edge frequencies exhibited in graph streams, we denote the \textit{global} variance of edge frequencies of the graph stream by $\sigma_G$. We further define the \textit{average local} variance of edge frequencies on a vertex basis as $\sigma_V$. This is computed by determining the statistical variance of edge frequencies for the edges incident on each source vertex and averaging over different vertices. The \textit{variance ratio}, $\sigma_G/\sigma_V$, for each of the three different data sets, DBLP, IP Attack Network, and GTGraph, is 3.674, 10.107, and 4.156, respectively. It is evident that the edge frequency variance on a vertex basis is consistently much smaller than the edge frequency variance of the whole graph stream. This also shows that there is significant skew in the frequency properties of graph streams, a fact we have considered in our sketch partitioning approach, gSketch.

\subsection{Evaluation Methods}

We evaluate different query estimation algorithms for both edge query and aggregate subgraph query. Edge queries are expressed as a set of edges drawn from the graph stream, whereas aggregate subgraph queries are expressed as a set of subgraphs whose aggregate frequency behavior is examined. Given an edge query set $Q_e = \{q_1, q_2, \ldots, q_k\}$, where $q_i$ is an edge in the graph stream, we consider two different accuracy measures for query estimation:

1. **Average Relative Error.** Given $q \in Q_e$, the relative error, $er(q)$, is defined as

$$er(q) = \frac{\tilde{f}(q) - f(q)}{f(q)} = \frac{\tilde{f}(q)}{f(q)} - 1 \quad (6.12)$$

Here, $\tilde{f}(q)$ and $f(q)$ are the estimated frequency and true frequency of $q$, respectively. The \textit{average relative error of} $Q_e$ is determined by averaging the relative errors over all
queries of $Q_e$:

\[ e(Q_e) = \frac{\sum_{i=1}^{k} \text{er}(q_i)}{k} \]  

(6.13)

2. **Number of “Effective Queries”**. Average relative error may become a biased measure if queries of $Q_e$ have significantly different frequencies. For example, if an edge with low frequency happens to collide with another edge with very high frequency in the sketch, this can lead to an extremely large value of average relative error. That is, a small number of such queries may dominate the overall average relative error of $Q_e$ in query estimation. We therefore propose another more robust measure, *number of effective queries*. The estimation of a query, $q$, is said to be “effective”, if $\text{er}(q) \leq G_0$, where $G_0$ is a user-specified threshold. The idea here is that the estimated relative error of a query larger than $G_0$ may deviate too much from its true frequency, such that it is no longer considered as an effective estimation. We denote the number of effective queries estimated in $Q_e$ as $g(Q_e)$, and

\[ g(Q_e) = |\{q|\text{er}(q) \leq G_0, q \in Q_e\}| \]  

(6.14)

In all our experiments, we set $G_0 = 5$ by default, unless otherwise specified.

In addition to edge queries, we also consider aggregate subgraph queries. Given a subgraph query set $Q_g = \{g_1, g_2, \ldots, g_k\}$, the relative error of $g = \{e_1, \ldots, e_l\} \in Q_g$ is defined as

\[ \text{er}(g) = \frac{\Gamma(\hat{f}(e_1), \ldots, \hat{f}(e_l))}{\Gamma(f(e_1), \ldots, f(e_l))} - 1 \]  

(6.15)

In our experiments, we set $\Gamma(\cdot) = \text{SUM}(\cdot)$, so that the aggregate frequency behavior of a subgraph is summarized by adding up all the frequencies of the constituent edges of this graph. We define the query estimation measures (*average relative error*, $e(Q_g)$, and *number of “effective” queries*, $g(Q_g)$) for aggregate subgraph query in an analogous way as to edge query. As will be shown in the following sections, the query estimation results of aggregate
subgraph query are similar to those of edge query. We therefore present the results of both kinds of queries in the DBLP data set only. For the other two data sets, we present the query estimation results only for edge queries. Besides query estimation accuracy, another important evaluation metric is the efficiency of sketch construction and query processing. The sketch construction time is therefore denoted as $T_c$, and the query processing time is denoted as $T_p$.

### 6.6.3 Query Estimation with Data Samples

We first consider the scenario in which only the data sample is available for query estimation in different graph streams.

In the DBLP data set, a data sample with a size of 100,000 edges is generated by reservoir sampling \[109\] from the underlying co-authorship graph. The edge query set $Q_e$ comprises 10,000 edge queries, which are generated by uniform sampling. The subgraph query set $Q_g$ comprises 10,000 graphs, each of which is generated by first uniformly sampling vertices as seeds from the underlying co-authorship graph and then exploring the seeds' neighborhood by BFS traversal to include more vertices. At any given node during BFS traversal, the next edge to be explored is picked at random and each such subgraph contains 10 edges in total.

In the IP Attack Network data set, we select the IP pair streams from the first day (January 1st, 2007) as the data sample, which contains 445,422 edges. The query set $Q_e$ is generated from the original graph stream by uniform sampling and $|Q_e| = 10,000$.

For the synthetic GTGraph data set, we select 5,000,000 edges, i.e., 5% of the whole graph stream, as the data sample by reservoir sampling. The edge query set, $Q_e$, is selected from the graph stream as well by uniform sampling and $|Q_e| = 10,000$.

We first examine the query estimation accuracy of different algorithms w.r.t. the first evaluation metric, average relative error. The first set of results for the edge query sets, $Q_e$, is illustrated in Figure 6.2 across different data sets. It is evident that $gSketch$ is consistently more accurate than $Global Sketch$ at various memory footprints. This means the
exploitation of underlying structural properties of graph streams is very helpful in boosting query estimation accuracy. When the available memory space is limited (less than 2M bytes in the DBLP and the IP Attack Network cases), the difference of query estimation accuracy of the two algorithms is very large. For example, in the DBLP data set, gSketch can achieve $2^{-8}$ times better estimation results than Global Sketch. In the more interesting space-constrained scenarios, this difference in estimation accuracy becomes very significant, and it suggests that gSketch can be used in extremely space-constrained devices, such as sensors, for effective query estimation. This also suggests that the accuracy difference of two methods will still exist when the size of the underlying graph increases, as shown in the GTGraph case (Figure 6.2(c)). When the graph stream becomes large with $10^9$ edges, gSketch still outperforms Global Sketch even when the available memory becomes large up to 2G bytes. This is an important result, because most typical graph stream applications such as social networks continue to become more and more massive over time. When the available memory becomes large, the difference of estimation accuracy between gSketch and Global Sketch reduces, because theoretically both methods can estimate the queries accurately if given infinitely large memory. However, gSketch still outperforms Global Sketch in such cases.

We then evaluate the query estimation accuracy in terms of the number of effective queries estimated in the edge query set $Q_e$. The estimation results across different data sets are illustrated in Figure 6.3 (log-scaled). Again gSketch outperforms Global Sketch by as much as one or two orders of magnitude. Interestingly, in the synthetic GTGraph data set,
Global Sketch simply fails to estimate any edge queries “effectively” when the available space is less than 512M bytes. However, gSketch can achieve much better estimation results. It suggests that when the graph stream becomes massive and the underlying graph exhibits significant global heterogeneity and local similarity properties, Global Sketch is no longer a feasible solution, while gSketch continues to retain its effectiveness.

Next, we evaluate the estimation accuracy of aggregate subgraph queries in the DBLP graph stream. As illustrated in Figures 6.4, gSketch consistently outperforms Global Sketch in terms of both average relative error and number of effective queries estimated. The experimental results also verify that gSketch is much more effective than Global Sketch for estimating both edge queries and aggregate subgraph queries.

Figure 6.4: Query Estimation Accuracy of Graph Queries $Q_g$ w.r.t. Memory Size (Bytes) in DBLP
Figure 6.5: Average Relative Error of Edge Queries $Q_e$ w.r.t. Memory Size (Bytes) (Zipf Skewness $\alpha = 1.5$)

6.6.4 Query Estimation with Data and Workload Samples

In this scenario, we evaluate the query estimation accuracy of different algorithms when both the data sample and the query workload sample are available from graph streams. We generate the query workload samples from different graph streams as follows. For the DBLP data set, we generate a query workload sample with 400,000 edges by sampling (without replacement) the graph stream which follows the Zipf distribution. The Zipf-based sampling is parameterized by a skewness factor $\alpha$. The larger the value of $\alpha$, the more skewed the query workload sample. It is worth mentioning that a vertex $m$ that exists in the data sample may not necessarily appear in the query workload sample. For such a case, we use the Laplace smoothing \[66\] to avoid $\tilde{w}(m)$, the estimated relative weight of $m$, to be zero.

Edge and subgraph queries are generated in a similar way by Zipf-based sampling and the sizes of both the edge query set and the subgraph query set are 10,000. In the IP Attack Network, we construct a query workload sample with 800,000 edges by Zipf sampling, and the edge query set is generated with $|Q_e| = 10,000$. For the synthetic GTGraph data set, we generate the query workload sample by Zipf sampling from the original graph, which contains 5,000,000 edges, and the edge query set is generated with $|Q_e| = 10,000$.

In our first experiment, we fix $\alpha = 1.5$ for Zipf sampling during the generation of workload samples and queries. We then examine the query estimation accuracy in the presence of both data and query workload samples. The average relative error and number of effective queries...
for edge queries on all data sets are reported in Figure 6.5 and Figure 6.6, respectively. In this scenario, gSketch still outperforms Global Sketch consistently at various memory footprints across different data sets. More interestingly, the estimation accuracy is higher than that for the case when the data samples are available only (as shown in Figure 6.2 and Figure 6.3) because of the involvement of the query workload samples. This accuracy gain is observed under both evaluation metrics, because we are leveraging a greater amount of querying information in sketch partitioning. It may sometimes happen that frequently occurring edges in the query workload sample may not be present as frequently in the data sample. Such edges can be estimated far more accurately in this scenario and further contribute to the improved accuracy.

We then evaluate the query estimation accuracy by varying the value of the skewness factor, $\alpha$, to generate a set of query workloads. The available memory here is fixed with

Figure 6.7: Query Estimation Accuracy of Graph Queries $Q_g$ w.r.t. Memory Size (Bytes) in DBLP Data Set ($\alpha = 1.5$)
Figure 6.8: Average Relative Error of $Q_e$ w.r.t. Zipf Sampling Skewness $\alpha$

2M bytes for the DBLP data set and the IP Attack Network data set, and 1G bytes for the GTgraph data sets, throughout this experiment. The average relative error of query estimation accuracy is illustrated in Figure 6.8 across different data sets. It is evident that with the increase of $\alpha$, gSketch can achieve better estimation accuracy with a decreasing trend of average relative errors, because larger values of $\alpha$ lead to more skewness in the query workload. Such skewness are accounted for in the sketch partitioning process and help improve the final query estimation results. On the contrary, Global Sketch is unable to use the query workload information, so there is no such accuracy improvement. The number of effective queries for the different methods is illustrated in Figure 6.9 across different data sets. Similar to the previous case in which average relative error is adopted as the evaluation metric, gSketch achieves better estimation accuracy with an increasing trend of the number of effective queries estimated, when $\alpha$ varies from 1.2 to 2.0. This accuracy improvement results from a better usage of the workload samples during the sketch partitioning.

Figure 6.9: Number of Effective Queries for Edge Queries $Q_e$ w.r.t. Zipf Sampling Skewness $\alpha$
Figure 6.10: Query Estimation Accuracy of Graph Queries $Q_g$ w.r.t. Zipf Sampling Skewness $\alpha$ in DBLP Data Set

Similar experimental evaluations are performed for aggregate subgraph queries on the DBLP data set. In Figure 6.7, the query estimation accuracy is reported with the sampling factor $\alpha = 1.5$. In Figure 6.10, $\alpha$ varies from 1.2 to 2.0 and the query estimation accuracy is reported if the available memory is 2M bytes. Analogous to edge queries, aggregate subgraph queries can be more effectively estimated with the use of query workload information in gSketch. On the other hand, Global Sketch performs even worse due to the frequency heterogeneity of the constituent edges in subgraph queries.

6.6.5 Efficiency Results

Besides query estimation accuracy, the time used for sketch construction, $T_c$, and the time used for query processing, $T_p$, are evaluated as well. Figure 6.11 illustrates $T_c$ for gSketch across different data sets for the two different scenarios. For the scenario in which both data and query workload sample are available, $T_c$ is slightly larger than that when only the data sample is available. In both scenarios, however, $T_c$ is within tens of seconds even for the largest graph stream, GTGraph, which contains $10^9$ edges. Therefore, gSketch can be rapidly constructed and effectively deployed from a practical point of view.

We further examine the query processing time, $T_p$, for both gSketch and Global Sketch across different data sets, as shown in Figure 6.12 For the DBLP data set, we measure the query processing time for both edge queries and aggregate subgraph queries. For the other two data sets, we present the query processing time only for edge queries. First of
all, $T_p$ for $g$Sketch is insensitive to the allocated memory, as long as the sketch structure can be built and allocated successfully in the sketch construction phase. It is evident that each edge query can be estimated within 0.2 seconds for all different data sets, while the response time for an aggregate subgraph query is within 0.8 seconds for the DBLP data set. Therefore, $g$Sketch can be used in real time on different graphs streams. Compared with Global Sketch, $g$Sketch takes additional time to determine the relevant sketch partition a query belongs to. However, such time is negligible and does not hurt the practicality of $g$Sketch. On the other hand, the enormous advantages of the sketch-partitioning philosophy definitely outweigh such cost.

### 6.6.6 Effect of New Vertices

To this end, we assume the underlying graph model of a graph stream is static. While in reality, such a graph is dynamically changing with new vertices and corresponding edges

Figure 6.12: Query Processing Time $T_p$ (Seconds) $w.r.t.$ Memory Size (Bytes)
### Memory Size

<table>
<thead>
<tr>
<th></th>
<th>Memory</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>128M</td>
<td>256M</td>
</tr>
<tr>
<td>Average</td>
<td>gSketch</td>
<td>58.5968</td>
</tr>
<tr>
<td>relative error</td>
<td>Outlier sketch</td>
<td>58.5971</td>
</tr>
</tbody>
</table>

Table 6.1: Avg. Relative Error of gSketch and Outlier Sketch in GTGraph Data Set

created all the time, thus forming an ever-growing massive network. As these newly added vertices are not in the sample data, their corresponding edges will be uniformly assigned to the outlier sketch for query estimation. A key question is *how these vertices affect the query estimation process*. We will demonstrate that gSketch is robust to the presence of such new vertices upon the underlying graph.

In order to illustrate this point, we present the estimation accuracy only for the edge queries answered by the outlier sketch, and compare it to the accuracy of all the edge queries answered by gSketch. Table 6.1 illustrates such query estimation accuracy in terms of average relative error on the GTGraph data set (For other data sets and scenarios, we have similar results). It is evident that the outlier sketch does not significantly degrade our query estimation accuracy. The main reason is the outlier sketch has already been skimmed of the high frequency edges, which could potentially cause collisions for estimation inaccuracy. The estimation error in the outlier sketch is therefore not significantly higher than other partitioned sketches. This also suggests that to the presence of new vertices and edges on the underlying graph, gSketch is still a robust solution to estimating queries on the graph stream.

#### 6.6.7 Discussion on Experimental Studies

After extensive studies of gSketch and Global Sketch in different experimental settings on various data sets, the following conclusions can be made:

1. When space is considered a critical factor, gSketch consistently achieves better query estimation accuracy on large graph streams in terms of both average relative error
and number of effective queries estimated. Furthermore, the importance of the space limitation increases with the domain size of the underlying graph.

2. By exploiting both data sample and query workload sample, gSketch can achieve better query estimation accuracy than that achieved with only the data sample.

3. When both data samples and query workload samples are available, gSketch will benefit if the samples are skewed. The more skewed the query workload sample, the better query estimation accuracy gSketch may achieve.

4. For gSketch, the time spent for sketch partitioning and construction is marginal. Furthermore, query processing can be performed very fast and the time is relatively invariant to the allocated space.

5. To the presence of new vertices and their corresponding edges on the underlying graph of a graph stream, gSketch is still a robust solution for query estimation, as long as the estimation error of the outlier sketch is not significantly higher than that of gSketch.

6.7 Conclusions

The problem of querying streaming information networks is very challenging because of the high stream speed and the massive universe of distinct edges. In this chapter, we designed an effective sketch-partitioning algorithm, gSketch, for query estimation over massive streaming information networks. We made use of the special structural properties of graph streams to help devise a sketch partitioning solution in order to improve the query estimation accuracy. We tested our approach on a number of real and synthetic data sets and illustrated the advantages of our sketch partitioning algorithm, gSketch, over a global sketching scheme, Global Sketch. gSketch has proven to be significantly more effective, and sometimes provides query estimation accuracy better than Global Sketch by an order of magnitude.
In future work, we will study methods for resolving more complex queries such as those involving the computation of complex functions of edge frequencies in a subgraph query. We will also examine the use of sketch-based methods for resolving structural queries. Finally, we will investigate how such sketch-based methods can be potentially designed for dynamic analysis, which may not require any samples for constructing the underlying synopsis.
Chapter 7

Conclusions and Future Directions

The past decade has seen an explosive growth of information networks in a wide variety of application domains. Information networks, including but not limited to the Internet, social networks, collaboration and recommender networks, public health-care networks, and technological/biological networks, have clearly formed a critical component of modern information infrastructure. The wide availability and popularity of information networks has been asking for efficient and cost-effective querying systems critical to modern network-enabled applications. Our research is primarily motivated to explore both principled methodologies and innovative applications for competent and high-performance querying methods in information networks that can be accessed and studied effectively in the context of the massive and ever-changing data conditions and application requirements.

In this thesis we have made a series of contributions to the study of innovative querying principles and techniques in large scale information networks. New querying methods have been devised and evaluated to address a wide range of fundamental and critical queries prevalent in most existing information networks. Such queries have proven to be hard, if not impossible, to resolve by conventional querying methods designed for static graphs with small or medium sizes. Our querying methods can be effectively leveraged as building blocks for advanced network operations and analytical tools, and can ultimately help advance a better understanding, manipulation and dissemination of large scale information networks in various real-world application domains.
7.1 Summary of Contributions

In this thesis we have presented our solutions to a series of fundamental yet important query processing methods in different large scale information networks. In general, we made the following contributions:

- **Structural-Context Similarity Computation in Information Networks.** Recent research and applications on proximity query processing, outlier detection, classification and clustering in information networks usually require an effective and trustworthy measure of similarity between entities. In order to answer “how similar are two entities in an information network?”, we proposed a new structural-context similarity measure, P-Rank, which exploited the link structure of information networks for effective similarity computation. The merits of P-Rank are its semantic completeness, generality and robustness. P-Rank has proven to be a unified structural-context similarity framework under which other well-known structural similarity measures, such as CoCitation, Coupling, Amsler and SimRank, are just its special cases;

- **Subgraph Query Processing in Information Networks.** At the core of many advanced information network applications lies a fundamental and critical graph query primitive: Given an information network modeled as a graph $G$, and a user-specified query graph $Q$, we want to return as output the set of subgraphs of $G$, each of which is isomorphic to $Q$. Despite NP-hard in nature, graph queries are pervasive in a wide range of application domains, such as bioinformatics, software engineering and pattern recognition. We designed a novel and scalable graph indexing and query processing method, SPath, to address graph queries on large-scale information networks;

- **Warehousing and OLAP Information Networks.** Data warehouses and online analytical processing (OLAP) are essential elements of decision support and business intelligence. Their powerful technologies, such as consolidating, summarizing and
slicing-and-dicing large-scale transactional and multidimensional data, have been successfully deployed in many industries: manufacturing, retail, financial services, utilities and health-care. We considered extending the warehousing and OLAP functionalities on large-scale information networks and proposed **Graph Cube** to effectively summarize an information network into multiple aggregated networks with coarser resolution. In **Graph Cube**, we jointly considered both the multidimensional metadata information together with the graph structure for aggregation and summarization, thus generating a set of structure-enriched, semantically meaningful and concisely summarized networks, which are of special importance to decision making and business intelligence in information networks;

- **Query Estimation in Streaming Information Networks.** Real-world information networks are not static but dynamically changing all the time, so they are often referred to as graph streams, in which the individual edges of the underlying graph model are received and updated rapidly over time in a form of a stream. Some noteworthy examples of graph streams correspond to the activities overlaid on the Web, social networks and communication networks. Due to the dynamic and massive nature of graph streams, we cannot store the fast streaming data explicitly on disk for effective analysis. The standard stream constraint of being able to process every edge only once applies naturally in the graph stream scenario. We proposed a graph sketch method, **gSketch**, for querying large-scale graph streams. **gSketch** has achieved up to an order of magnitude improvement in terms of the query estimation accuracy in comparison with the state-of-the-art **CountMin** sketch method.

## 7.2 Future Work

Graph has proven to be a powerful abstraction for interlinked complex and heterogeneous data, and the information network paradigm has posed a wealth of fascinating research
problems and high-impact applications. It is our strong belief that they will continue to play a key role in many widely encountered scenarios. Meanwhile, modern business and industrial infrastructures are collecting massive amounts of data ("big data") at an unprecedented scale and pace. The ability to perform efficient and data-intensive querying on such large scale information networks in order to extract deep insights now becomes a key ingredient of success. Therefore, besides the aforementioned work we have examined, there are more interesting directions of future work along the line of managing, querying and mining large scale information networks. We believe the following important directions deserve a careful and thorough study to further advance this field in future.

7.2.1 Foundations and Models of Information Networks

Real-world information networks are complex and heterogenous that contain multiple genres of data from different sources, in different formats, and with different types of knowledge at various abstractions. For example, to investigate the mechanism of protein synthesis, a researcher has to conduct a comprehensive study of both unstructured textual information such as biology literature and gene ontology, as well as the interlinked structural information such as protein regulations and interactions. Another example is to support contextual similarity search in FaceBook by integrating the multidimensional relational data (for example, the age, gender, race, education level and salary information of an individual), the interlinked friendship relations, and the user-generated online data, such as blogs, social bookmarks, query logs and customer reviews. As a result, we need a unified model and powerful formalism to understand the characteristics and behavior of such multi-genre information networks. It is fairly easy to see that most of the existing information networks under consideration are actually one-genre networks whose information are collected from a homogeneous source, and thus they can be regarded as a special case of the more general multi-genre heterogeneous information networks. How to extend the existing querying methods toward large multi-genre information networks will become utterly important and
of special interest for such newly emerging and more practical network scenarios.

Real-world information networks are dynamic and resilient that contain volatile, noisy, uncertain and incomplete information. For example, an individual is reluctant to provide a complete profile when registering an online social network due to privacy concerns or personal issues, or she/he may provide with some fake information which might be conflicting with the one stored in the networks of an authoritative agency. Sometimes, we are not 100% sure, but with some confidence of uncertainty, that there exists an edge between two vertices during network construction. For physical or technical reasons, such confidence may even change when the network evolves over time. It is evident past research has made the assumption that the data in the networks are clean, stable, reliable, and complete, while in these new scenarios, this assumption fails and we have to turn to new querying models and methods for a quick and accurate access to the so-called probabilistic information networks. We will reexamine the models and methods developed in previous studies and extend the principles of managing and mining real, noisy, and incomplete information networks to achieve high robustness and quality. This direction will bridge the gap and develop high quality and fault-tolerant methods for noisy and inaccurate information networks.

The data of information networks are no longer limited to cyber-world but more and more elements in physical world have been deeply involved, such as sensors, RFID detectors, GPS’s, satellites and the most important human beings. New models of information networks need to be proposed in order to address the unprecedented challenges met in the networked cyber-physical systems. And we believe people (as accessed via crowdsourcing techniques) can be incorporated as a key role to help analyze information networks from all perspectives.

7.2.2 Efficient and Scalable Querying in Information Networks

Most of the existing querying and access methods for information networks are neither generic nor modular, such that they can only be applied on a confined set of networks, but hard to reuse and generalize in all network applications. There have been a number
of reasons for developing a unified declarative language specific for graph data sets and information networks. Similar to SQL, such a graph-oriented language should be declarative, user-friendly, powerful in expressiveness, and specifically optimized for graph structures in their full generality. Thus, all fundamental graph-based primitives and constraints can be implemented and highly optimized as building blocks behind the scene and the advanced computations can be easily composed by such primitives. In this thesis, we have examined a series of fundamental queries on information networks, each of which can be regarded as a basic operator, or primitive, of the aforementioned graph-oriented language. Analogous to SQL, P-Rank is used to model and address the “likelihood” query (LIKE in SQL) in information networks; SPath is a selection operator (SELECT in SQL) defined with subgraph isomorphism on information networks; GraphQL tries to model the aggregation behavior (GROUP-BY or CUBE-BY in SQL) for multidimensional information networks; and gSketch provides an accurate estimation for basic aggregate queries (COUNT, SUM, AVG in SQL) in streaming information networks. We believe there is plenty of room to explore in order to enrich and complete the current tool set of graph-oriented operations and primitives for a full-fledged graph/network based query language in future.

Nowadays, real-world information networks keep growing in an extraordinary speed and the underlying graphs typically contain millions of vertices and billions of edges. Such data-intensive networks have outpaced our ability in data processing and posed serious challenges to manage them efficiently and scalably. The big success of MapReduce [35] has motivated us to design a distributed graph computational framework in order to address the data-intensive issues in today’s large scale information networks. We believe it is a promising direction to advance graph querying and analytical methods on distributed, parallel or cloud frameworks and a lot of interesting and influential research can be done to support efficient and scalable computations on large scale information networks.
References


[75] Xiaolei Li, Jiawei Han, and Hector Gonzalez. High-dimensional OLAP: a minimal cubing approach. In Proceedings of the thirtieth international conference on Very large data bases (VLDB’04), pages 528–539, 2004.

[76] Cindy Xide Lin, Bolin Ding, Jiawei Han, Feida Zhu, and Bo Zhao. Text cube: Computing IR measures for multidimensional text database analysis. In Proceedings of the 2008 eighth IEEE international conference on data mining (ICDM’08), pages 905–910, 2008.


151


[113] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph indexing: a frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data (SIGMOD’04)*, pages 335–346, 2004.

[114] Xiaoxin Yin, Jiawei Han, and Philip S. Yu. Linkclus: Efficient clustering via heterogeneous semantic links. In *Proceedings of the 32nd international conference on Very large data bases (VLDB’06)*, pages 427–438, 2006.


153