QUERYING GRAPH DATABASES

by

Yuanyuan Tian

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2008

Doctoral Committee:
Associate Professor Jignesh M. Patel, Chair
Professor Hosagrahar V. Jagadish
Professor Farnam Jahanian
Professor David J. States

© Yuanyuan Tian 2008
All Rights Reserved

To my family.

ACKNOWLEDGEMENTS

This dissertation could not have been completed without the support and encouragement of many people. First, I would like to thank my advisor, Professor Jignesh M. Patel. Jignesh is a great advisor. I have learned a great deal about research, academic writing and presentation skills from him. He is extremely honest, and always encourages me to think independently and argue with him about research ideas. I am very lucky to have worked with him as a student, teaching assistant and research assistant for the past 5 years.

I would also like to thank my dissertation committee, Professor Hosagrahar V. Jagadish, Professor Farnam Jahanian and Professor David J. States, for their time and effort to help improve and refine my thesis. In particular, Professor States provided me with the BioNLP dataset for the approximate graph matching experiments and helped me develop the statistical significance evaluation method for this dataset.

I appreciate all of Dr. Richard A. Hankins' mentorship and support throughout my internship at Nokia Research Center. He advised the early stage of the graph summarization work.

The National Center for Integrative Biomedical Informatics (NCIBI) provided me a golden opportunity to apply my dissertation work to real life science applications.

Professor David J. States, Professor Matthias Kretzler, Dr. Richard C. McEachin, Dr. Carlos Santos, Viji Nair, Sebastian Martini, Terry Weymouth, Glenn Tarcea and Vasudeva Mahavishnu all deserve special thanks in helping me with the application.

Spending five years in graduate school could very well have been unbearable without many colleagues and friends to make life fun. I would like to thank all the database slaves who made my time in the office so enjoyable: You Jung Kim, Adriane Chapman, Magesh Jayapandian, Willis Lang, Arnab Nandi, Bin Liu, Anna Shaverdian, Neamat el Tazi, Jing Zhang, Dr. Yunyao Li, Dr. Cong Yu, Dr. Sandeep Tata, Dr. Mike Morse, Dr. Yun Chen, Dr. Nuwee Wiwatwattana and Dr. Stelios Paparizos. Especially, thank Magesh Jayapandian for his wonderful home-made cakes which made the database lab feel like a home. Thank You Jung Kim, Adriane Chapman, Neamat el Tazi and Dr. Yunyao Li for being my long-time "lunch buddies". I would also like to thank the many other friends who have supported me throughout my graduate school life. Special thank to Ying Zhang for being my "water buddy" and best listener.

Finally, my thanks goes to my parents, who have always been extremely supportive of my study and work in other areas of life. I consider myself very lucky to have so many wonderful people always behind me.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

CHAPTER I

Introduction

Graphs provide a natural way to model data in a variety of applications. Nodes in graphs usually represent real world objects and edges indicate relationships between objects. Examples of data modeled as graphs include social networks, biological networks, and computer networks. Many graph databases are large and growing rapidly in size. For example, the number of pathways (a pathway is a graph of cellular entities and their interactions) in the well-known KEGG pathway database [34] has increased from 2,706 in 1999 to 29,921 in 2005, then to 66,407 in 2007. The social networking site Facebook contains a large network of registered users and their friendships. The number of Facebook users has grown from less than 5 million in September 2005 to close to 10 million in September 2006, then to 50 million in September 2007. There is a critical need for efficient and effective graph querying systems to query and mine these growing graph databases.

Previous graph querying systems [21, 46] have largely focused on relatively *simple* graph operations, such as retrieving nodes, edges and paths. However, none of these

systems support sophisticated query operations like approximate graph matching or graph summarization (see Table 2.1 for the descriptions of these operations). On the other hand, tools for individual query operations, such as GraphGrep [20], GIndex [58] and Grafil [59], have been developed. These tools are useful, but the power of individual query operations is limited. Complex analysis on graphs usually requires more than one query operation. Users have to combine these individual tools together, going through the complication of resolving the differences in execution platforms and data formats. Therefore, it is crucial to develop graph querying systems that include sophisticated graph operations as well as simple ones.

This thesis describes the efforts in developing an effective and efficient graph querying system that support sophisticated graph query operations.

1.1 Contributions

To address the need of complex analysis on graph data, this thesis develops a graph querying toolkit, called Periscope/GQ. This toolkit is built on top of a traditional RDBMS. It provides a uniform schema for storing graphs in the database and supports various simple and sophisticated graph query operations. Users can easily combine several operations to perform complex analysis on graphs. To speed up query operations, Periscope/GQ employs novel indexing techniques that make use of the existing robust index structures in a typical RDBMS, which makes adoption and implementation easy.

The key feature of Periscope/GQ is the support of various sophisticated graph

query operations as well as simple ones. In particular, this thesis focuses on graph matching and graph summarization queries.

Graph matching queries allow a user to discover, in the database, graphs or subgraphs similar to the query graph. It is analogous to the keyword search in a text database. The previous studies on graph matching methods [22, 28, 45, 51, 55, 58, 59, 60, 61], have mostly been carried out within the context of precise graph data, and have focused on exact graph or subgraph matching queries (i.e. graph or subgraph isomorphism). However, many real graph datasets are noisy and incomplete in nature. For example, it is well known that protein interaction networks produced by high-throughput methods contain many false positives [47]. As a result, exact graph or subgraph matching often fails to produce useful results for real graph data.

In contrast, approximate graph or subgraph matching plays a critical role in these applications. Approximate matching allows node/edge insertions and deletions, and node/edge mismatches. Furthermore, many new graph applications prefer approximate matching results rather than exact ones as they can provide more information such as what might be missing or spurious in a query or a database graph.

This thesis presents a novel approximate graph matching technique called SAGA. This technique employs a flexible model for computing graph similarity, which allows for node gaps, node mismatches, and graph structural differences. SAGA employs an index-based matching technique that allows it to efficiently evaluate queries even against large graph datasets. In addition, SAGA can produce meaningful matches on actual examples, whereas existing tools fail.

Most graph matching methods, including SAGA, are applicable to query graphs

that are small in size (tens of nodes and edges). However, in many new applications, both the query and database graphs are "large". Each graph can contain hundreds to thousands of nodes and edges. For example, in life sciences applications, protein interaction networks for individual species are often matched to determine similarities and differences across species. Each protein interaction network typically contains hundreds to thousands of nodes and edges.

To address the need for approximate matching of large query graphs, a novel technique, called TALE, is developed. TALE employs a novel indexing method that incorporates graph structural information in a hybrid index structure. This indexing technique achieves high pruning power and the index size scales linearly with the database size. In addition, an innovative matching paradigm is proposed to query large graphs. This technique distinguishes nodes by their importance in the graph structure. The matching algorithm first matches the important nodes of a query and then progressively extends these matches. Through experiments, TALE has been shown to be effective for real applications, and scalable for large queries and databases.

Graph summarization techniques are very useful for understanding underlying characteristics of graphs. In many applications, graphs contain thousands or even millions of nodes and edges. As a result, it is almost impossible to understand the information encoded in large graphs by mere visual inspection. Therefore, effective graph summarization methods are required to help users extract and understand the underlying information. However, existing graph summarization methods are mostly statistical [11, 12, 37] (studying statistics such as degree distributions, hop-plots

and clustering coefficients). While these methods are useful, the summaries contain limited information and can be difficult to interpret and manipulate.

This thesis introduces two graph aggregation operations to summarize graphs. Like the OLAP-style aggregation methods, that allow users to drill-down or roll-up to control the resolution of summarization, the proposed methods provide an analogous functionality for large graph datasets. The first operation, called *SNAP*, produces a summary graph by grouping nodes based on user-selected node attributes and relationships. The second operation, called *k-SNAP*, further allows users to control the resolutions of summaries and provides the "drill-down" and "roll-up" abilities to navigate through summaries with different resolutions. The effectiveness and efficiency of the two aggregation operations are demonstrated by experiments on both real and synthetic datasets.

1.2 Outline

This dissertation is structured as follows. Chapter II describes the design and architecture of the Periscope/GQ toolkit. Chapter III presents the approximate graph matching method, SAGA. To support approximate matching for large graphs, the TALE technique is introduced in Chapter IV. The *SNAP* and *k-SNAP* graph summarization methods are presented in Chapter V. Chapter VI describes the related work. Finally, Chapter VII concludes this thesis.

CHAPTER II

Periscope/GQ: A Graph Querying Toolkit

2.1 Introduction

Efficient and effective graph querying systems are critical to query and mine the ever growing graph datasets in various applications. To address this need, we design and develope a graph querying toolkit, called Periscope/GQ. This toolkit is built on top of a traditional RDBMS. It provides a uniform schema for storing graphs in the database. The key feature of Periscope/GQ is that it supports various simple and complex graph query operations. Users can easily combine several operations to perform complex analysis on graphs. To speed up query operations, Periscope/GQ employs novel indexing techniques that make use of the existing robust index structures in a typical RDBMS, which makes adoption and implementation easy. By applying Periscope/GQ to life science and social networking applications, we demonstrate the power of Periscope/GQ in performing complex analysis on graph databases.

2.2 System Architecture

Periscope/GQ is built on top of the RDBMS PostgreSQL (`http://www.postgresql.org/`). Graph data are stored and indexed in the RDBMS, while graph query algorithms are implemented as applications on top of the RDBMS. This design allows us to easily port the implementation to other RDBMSs. Figure 2.1 shows the architecture of Periscope/GQ.

2.2.1 Data Model and Data Storage

Periscope/GQ supports a general graph model. Under this model, graphs can be directed or undirected. Nodes and edges are allowed to have arbitrary labels and attributes. In fact, node and edge labels can be viewed as special attributes. Furthermore, attributes can be of arbitrary types.

Graphs are stored in a graph table, a node table and an edge table using the following schema:

```
Graph(graphID, attrName, attrType, attrValue)
Node(graphID, nodeID, attrName, attrType, attrValue)
Edge(graphID, node1ID, node2ID, attrName, attrType, attrValue)
```

Each graph is uniquely identified by a `graphID` in the graph table. A graph can have attributes associated with it. For example, in Figure 2.2, the graph with `graphID=1` has a string attribute called `name`. The value of this attribute is `wnt pathway`. This graph has another string attribute describing the `source` of the graph data. Within each graph, nodes are uniquely identified by their `nodeIDs`. Simi-

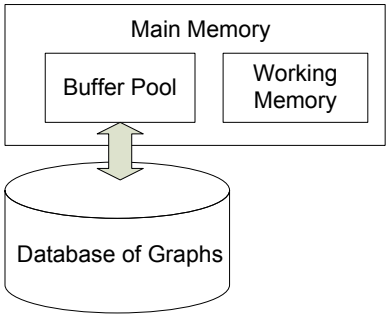

Figure 2.1: Periscope/GQ architecture

larly, nodes can have attributes associated with them. In Figure 2.2, the node with `nodeID=1` in the graph with `graphID=1` has two attributes `label` and `familyID`. Edges within a graph are identified by the IDs of the end nodes. Again, edges can have attributes associated with them. In Figure 2.2, the edge in graph 1 with end nodes 1 and 2 is undirected, which is indicated by setting the `directed` attribute to the value `false`. A directed edge is represented by setting this `directed` attribute to the value `true` and the direction is from `node1ID` to `node2ID`. This edge has another attribute indicating its `link type`. Graphs, nodes or edges with multiple attributes have multiple entries in the corresponding tables. In the current implementation, all graphs in the system must have a `name` attribute with non-null values; all nodes must have a `label` attribute with non-null values; and all edges must have a `directed` attribute with non-null values.

2.2.2 Graph Query Operations

The graph query operations supported in Periscope/GQ are listed in Table 2.1. The first six operations in this table are relatively "simple" and have been extensively

Graph Table

graphID	attrName	attrType	attrValue
1	name	string	wnt pathway
1	source	string	KEGG Database
.....	.....	.....	.....

Node Table

graphID	nodeID	attrName	attrType	attrValue
1	1	label	string	wnt
1	1	familyID	string	K00182
.....	.....	.....	.....	.....

Edge Table

graphID	node1ID	node2ID	attrName	attrType	attrValue
1	1	2	directed	boolean	false
1	1	2	link type	string	protein interaction
.....	.....	.....	.....	.....	.....

Figure 2.2: Examples of the graph table, the node table and the edge table

studied. The last three operations are more complex and play crucial roles in complex analyses, hence are described below.

Approximate Graph Matching: Analogous to the keyword search in a sequence/text database, graph matching finds graphs or subgraphs in the database similar to the query graph. It is an important operation to analyze graphs in complex ways. Due to the noisy and incomplete nature of most real graph datasets, *approximate* matching plays a more critical role than exact matching in practice. Approximate matching allows node/edge insertions and deletions, and node/edge mismatches.

Periscope/GQ incorporates the novel approximate graph matching method SAGA (see Chapter III). SAGA employs a flexible model for computing graph similarity and utilizes an index-based matching technique that allows it to efficiently evaluate queries

Operation	Description
Graph Selection	Select graphs based on conditions of graph attributes.
Node Selection	Select nodes based on conditions of node attributes and/or graphID.
Edge Selection	Select edges based on conditions of edge attributes and/or graphID, and/or nodeIDs.
Node Similarity	Given a node, find nodes that have similar attribute values and similar neighbors.
Path Existence	Decide whether two given nodes are connected by a path.
Shortest Path	Find the shortest path between two given nodes.
Approximate Graph Matching	Find graphs or subgraphs in the database that are similar to the query graph.
Large Graph Alignment	Align two or more large graphs to find conserved subgraphs.
Graph Summarization	Produce summaries capturing the characteristics of the original graphs.

Table 2.1: Graph operations supported in Periscope/GQ

even against large graph datasets.

Large Graph Alignment: Most graph matching methods, including SAGA, are designed to query graphs that are small in size (tens of nodes and edges). However, some applications require matching *large* graphs. One such example is to align protein interaction networks (graphs with thousands of nodes and edges usually) of two species to study evolutionary conservation.

To address the need for approximate matching of large graphs, Periscope/GQ incorporates the novel technique TALE (see Chapter IV). TALE employs an indexing technique, which achieves high pruning power and scales linearly with database sizes. The innovative matching algorithm utilizing this index is orders of magnitude faster than the state-of-the-art graph alignment methods.

Graph Summarization: As graphs in many applications, especially large-scale social networking applications, grow larger and larger, it becomes almost impossible for users to understand the information encoded in large graphs by mere visual inspection. Therefore, graph summarization methods are required to help users understand the underlying characteristics of large graphs.

Periscope/GQ employs the k-SNAP method (see Chapter V) to summarize graphs. k-SNAP allows users to freely choose the attributes and relationships that are of interest, and then makes use of these features to produce small and informative summary graphs. Furthermore, users can control the resolution of the resulting summaries and "drill down" or "roll up" the information, just like the OLAP-style aggregation methods in traditional database systems.

2.2.3 Efficient Query Evaluation using Indices

To efficiently evaluate queries, Periscope/GQ employs a variety of indexing techniques. For simple operations, such as graph selection, node selection and edge selection (see Table 2.1), traditional indexing methods are sufficient. However, designing indexing mechanisms for the more complex operations, such as approximate graph matching and large graph alignment, is more challenging. Rather than designing new index structures, which makes adoption and implementation hard, Periscope/GQ makes use of existing index structures already provided inside the RDBMS in interesting ways.

In Chapter III, we proposed the *Fragment Index* to speed up the approximate

graph matching in SAGA. The indexing units of the *Fragment Index* are small subgraphs in the database. We used a B+-tree to implement the *Fragment Index* (see Chapter III for details). The large graph alignment method TALE in Chapter IV employs the *Neighborhood Index* to expedite the query processing. The indexing units of the *Neighborhood Index* are the neighborhoods of all the nodes in the database. A *neighborhood* is defined as the induced subgraph of a node and its neighbors (adjacent nodes). This *Neighborhood Index* is implemented as a hybrid index structure, which has two levels. The first level of this index structure is a B+-tree. Each leaf entry of this B+-tree points to a second-level bitmap index (see Chapter IV for details). Both the *Fragment Index* and the *Neighborhood Index* are easily implemented inside a typical RDBMS, and result in orders of magnitude speedup for the corresponding query operations in most cases.

2.3 Case Studies

In this section, we use two real example applications: one life science application and one social networking application, to demonstrate the power of Periscope/GQ in performing complex analysis on graphs.

2.3.1 Example 1: Gene Regulatory Networks

Life science is experiencing a transition from focusing on the function of a single molecule to analyzing biological systems and their behavior as regulatory networks. Genome wide microarray analysis with pathway mappings and scientific literature

searches can generate gene regulatory networks of different species under different biomedical conditions. These gene regulatory networks can be naturally modeled as graphs, where nodes represent genes and edges indicate their interactions. The size of an individual gene regulatory network can be as large as several thousands of nodes and tens of thousands of edges. These networks serve as a rich source of information to be analyzed for discoveries that can lead to the cure of human diseases. Graph querying systems plays a critical role in helping life scientists analyze large gene regulatory network datasets.

In this section, we show an example of how different graph query operations in Periscope/GQ can be combined to help a group of life scientists find the key drugable pathways to validate therapeutic targets for Type 1 Diabetic Nephropathy (DN). Figure 2.3 shows the workflow of the analysis.

Through genome wide microarray analysis on human and mouse DN samples, large gene regulatory networks of the two species are generated. Each network contains hundreds to thousands of nodes and edges. By issuing graph summarization queries using k-SNAP, summaries based on features of interest are generated to help life scientists understand the underlying characteristics of individual networks.

In addition, cross-species network comparison is an effective way to identify which subnetwork produce the disease in both systems. This operation can be achieved by aligning the networks of the two species using TALE. This conserved subnetwork is a good candidate for therapeutic target validation. This operation can also be pipelined with further queries, such as querying the conserved subnetwork against a database of pathways to find out which biological processes might be involved or

affected by the conserved mechanism. A pathway consists of a set of cellular entities interacting to carry out some biological process. The query against a database of pathways can be achieved by an approximate graph matching operation using SAGA. Alternatively, the conserved subnetwork can also be used to query a database of parsed literature graphs to search for papers that may have already studied the conserved mechanism. In Chapter III, we described a way to perform document comparison using the graph matching method SAGA. Through natural language analysis, each biomedical document is represented by a graph in which a node indicates a gene studied in the document and a link is drawn between two genes if they are discussed in the same sentence (indicating a potential association between the two). Matching the conserved subnetwork against these parsed literature graphs can help life scientists find out previous studies with similar interests.

Through the above analysis, the life scientists actually identified several good candidates that they are validating for therapeutic targets.

2.3.2 Example 2: DBLP Coauthorship Networks

In this section, we demonstrate how Periscope/GQ can be used to analyze coauthorship networks from the DBLP Bibliography data [32]. In a coauthorship network, each node represents an author, and the edge between two authors indicate their coauthorship.

In this example, we are first interested in studying how researchers in the database area coauthor with each other. However, the coauthorship patterns are hidden inside

the large DB coauthorship network, which contains over 7 thousand nodes and around 20 thousand edges. Graph summarization operation (using the k-SNAP method) is very critical in understanding the characteristics of this large graph (see Figure 2.4). k-SNAP generates summaries with the resolutions that the users can understand, and also provides "drill-down" and "roll-up" abilities to navigate summaries with different resolutions. Detailed analysis on coauthorship networks using the k-SNAP method can be found in Chapter V. The k-SNAP operation can also be used to examine the similarities and differences in the coauthorship relations across communities, such as the DB community and the AI community. Our demonstration will include these examples.

As shown in Figure 2.4, further analysis can be performed on subnetworks of the DB coauthorship network. Subnetworks can be easily generated by node selection and edge selection operations. For example, one can construct a coauthorship network only about authors who publish in SIGMOD conference from 2001 (when double-blind review was first adopted) to 2007. This SIGMOD coauthorship network can be constructed by selecting authors (nodes) who have at least one SIGMOD paper in year ≥ 2001 and ≤ 2007 and coauthorships (edges) that appear in these publications. Similarly, one can also construct a VLDB coauthorship network. Further analysis can be performed by aligning the SIGMOD and VLDB coauthorship networks to compute the conserved coauthorships across the two communities.

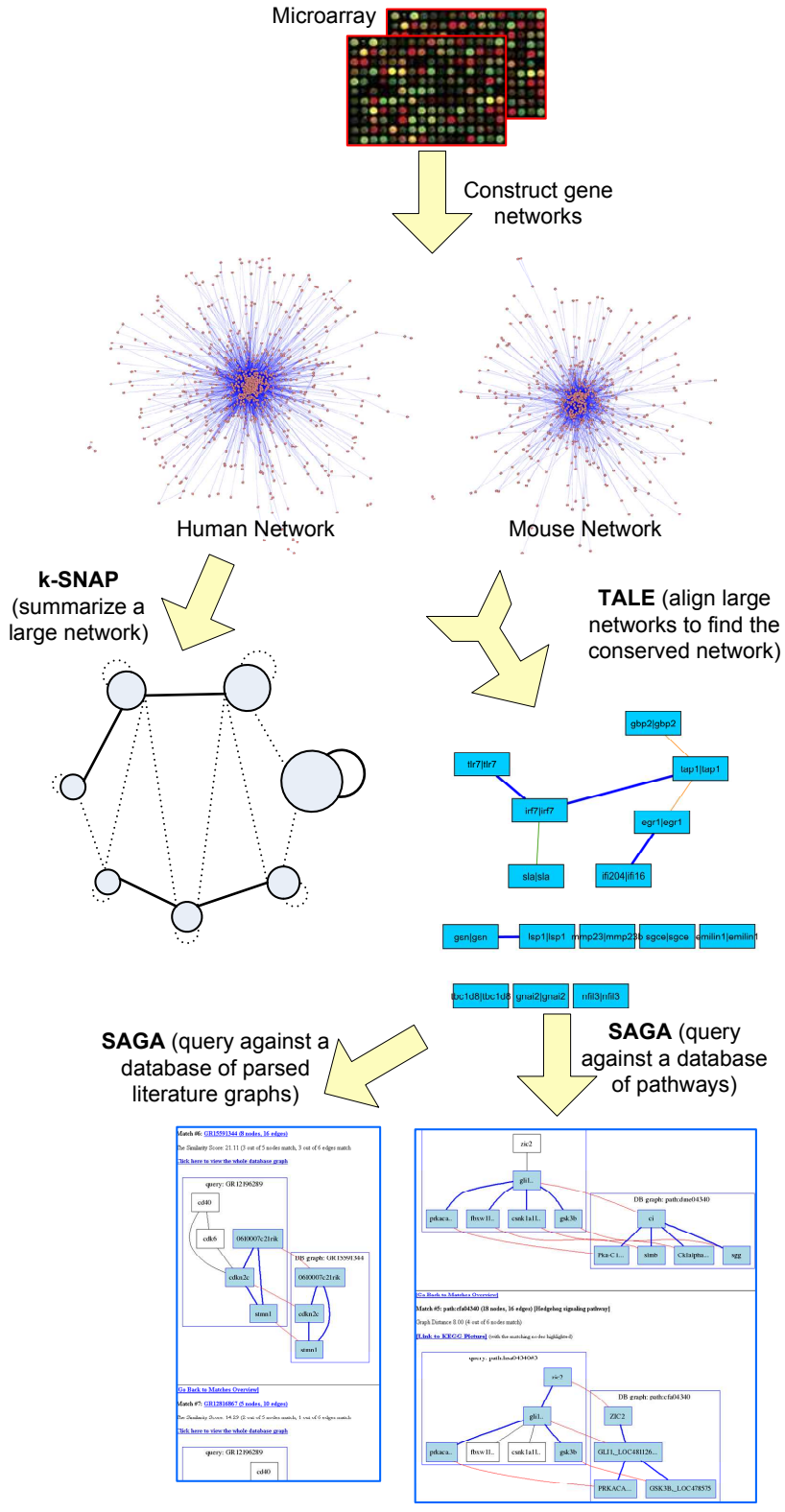

Figure 2.3: Example application of Periscope/GQ for gene regulatory network analysis

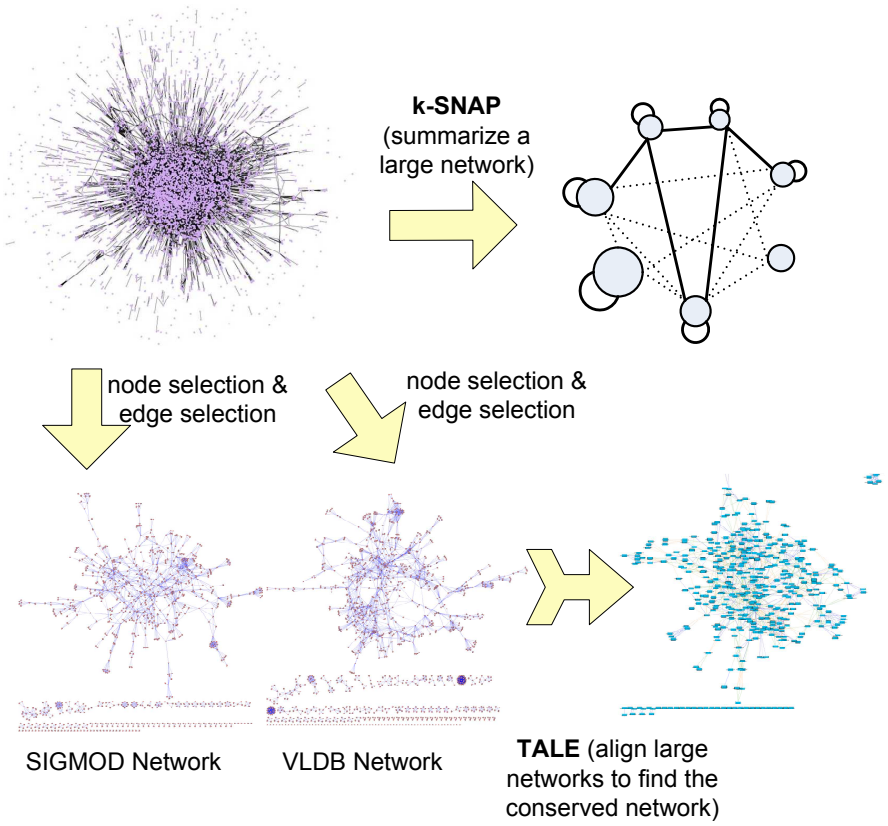

Figure 2.4: Example application of Periscope/GQ to analyze coauthorship networks

CHAPTER III

Approximate Graph Matching

3.1 Introduction

Analogous to the keyword search in a sequence/text database, graph matching finds graphs or subgraphs in the database similar to the query graph. It is an important operation to analyze graphs in complex ways. This chapter presents SAGA (**S**ubstructure Index-based **A**pproximate **G**raph **A**lignment), for approximate graph matching.

More formally, the problem that we address is *approximate subgraph* matching: Given a query graph and a database of graphs, we want to find *subgraphs* in the database that are similar to the query, allowing for node mismatches, node gaps (node insertions or deletions), as well as graph structural differences. Node mismatches model the behavior that two nodes representing different cellular entities can exhibit similar functionality. For example, two different proteins may be in the same protein orthologous group, which indicates similar functionality. Node gaps

represent the situation where a certain node in one graph cannot be mapped to any node in the other graph. Graph structural differences allow for differences in node connectivity relationships. For example, two nodes may be directly connected in one graph, whereas the corresponding matching nodes in the other graphs may be indirectly related through one or more additional nodes.

As a motivating example for the approximate subgraph operation, consider the following scenario: A scientist working on a certain disease has constructed a small portion of a pathway based on analysis of various experimental data. This pathway fragment, which is modeled as a graph, contains nodes that represent cellular entities (proteins, genes, mRNA, etc.) and edges that represent interactions. The scientist is interested in finding the biological processes that may be affected by the disease. This task can be expressed as a query that searches a database of known pathways using the query graph. Furthermore, the search can identify similar subcomponents shared between the query and graphs in the database, which may reveal clues about what information might be missing or spurious in the query graph, and provide a way of generating additional hypotheses.

While there is a long history of research on graph matching, most of this work has focused on exact subgraph matching, i.e., the subgraph isomorphism problem, which is known to be NP-complete. GraphGrep [45] and GIndex [58] are index-based filtering methods for exact subgraph matching. Grafil [59], PIS [60] and C-Tree [22] introduce some approximation for subgraph isomorphism. However, these approximate models are very limited. None of these tools allow node gaps in their models. PathAligner [14] is a tool for aligning pathways. However, it assumes that all path-

ways are linear paths. The tools most closely related to our work are PathBlast [30] and the successive NetworkBlast [40], which are designed for aligning protein interaction networks. Their graph similarity model allows node mismatches and node gaps, but graph structural differences are largely confined to short paths. As shown in Section 3.3.5, our graph similarity model tolerates more general structural differences, and can find biologically relevant matches, when both PathBlast and NetworkBlast fail. Another related method [31] has been proposed for aligning protein interaction networks. However, the match technique used in this method largely focuses on capturing the penalty associated with gene duplication. Finally, PathBlast, NetworkBlast, and the method proposed in [31] can only perform one graph comparison at a time. To match a query against a *database* of graphs, the matching algorithms must be run for each graph in the database. As a result, these methods are not computationally efficient when querying large graph databases.

In this chapter, we present a novel approximate subgraph matching technique called SAGA. At the heart of SAGA is a flexible model for computing graph similarity, which permits node gaps, node mismatches, and graph structural differences. To speed up the execution of queries with this powerful matching model, we employ an indexing method for efficient query evaluation. Through experimental evaluation, we demonstrate that SAGA is more flexible and powerful than existing models. SAGA allows additional information derived from the relationships between entities in pathways to be incorporated into comparative analysis. Our experimental results show that SAGA finds expected associations, like Insulin signaling in Type 2 Diabetes Mellitus. SAGA also finds less well studied associations, like the Toll-like receptor, T-cell

receptor, and Apoptosis pathways in H. pylori infection, as well as Calcium, Wnt and Hedgehog signaling in Bipolar Disorder. In addition, SAGA provides a powerful tool for biomedical text comparison.

3.2 System and Methods

3.2.1 Graph model

In our model, a graph, G, is a 3-tuple $G = (V, E, \phi)$. V is the set of nodes and $E \subseteq V \times V$ is the set of (directed or undirected) edges. Nodes in the graphs have labels specified by the mapping $\phi : V \rightarrow L$, where L is the set of node labels. This model captures the features that are commonly present in most biological graph datasets, in which nodes represent molecules/complexes, labels denote molecule/complex names, and edges indicate relationships between nodes. We assume that each node in the graph has a unique ID. This ID is used to establish a total order among the nodes.

In the example graph in Figure 3.1(a), v_i is used to represent the unique node ID and L_k is the node label. Note that two different nodes in a graph can have the same label.

Our distance model and matching algorithm (discussed below) support both directed and undirected graphs. We present our method using undirected graphs; adaptations of the distance measure and the matching algorithm for directed graphs are straightforward and omitted here.

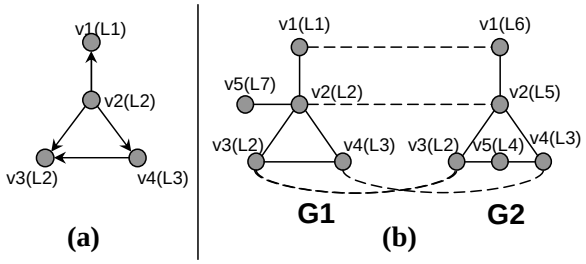

Figure 3.1: (a) An example graph. (b) An example subgraph match.

3.2.2 Distance measure for subgraph matching

Our model measures similarity by a distance value, so graphs that are more similar have a smaller distance. Formally, the subgraph matching is defined as follows: Let $G_1 = (V_1, E_1, \phi_1)$ and $G_2 = (V_2, E_2, \phi_2)$ be two graphs. An approximate matching from G_1 (the query) to G_2 (the target) is a bijection mapping function $\lambda : \hat{V_1} \leftrightarrow \hat{V_2}$, where $\hat{V_1} \subseteq V_1$ and $\hat{V_2} \subseteq V_2$.

An example match is shown in Figure 3.1. The dashed lines indicate the matched nodes in the two graphs. Note that nodes can be mapped even if they have different labels. Also, note that not all nodes are required to be mapped, e.g., v_5 in G_1 has no mapping in G_2, and is a *gap node*.

The subgraph distance (SGD), with respect to λ, is defined as:

$$\begin{aligned} SGD_\lambda(G_1, G_2) &= w_e \times StructDist_\lambda \\ &+ w_n \times NodeMismatches_\lambda \\ &+ w_g \times NodeGaps_\lambda \end{aligned} \tag{III.1}$$

where

$$StructDist_\lambda = \sum_{u,v \in \hat{V_1}, u<v} |d_{G_1}(u,v) - d_{G_2}(\lambda u, \lambda v)| \quad \text{(III.2)}$$

$$NodeMismatches_\lambda = \sum_{u \in \hat{V_1}} mismatch(\phi_1(u), \phi_2(\lambda u)) \quad \text{(III.3)}$$

$$NodeGaps_\lambda = \sum_{u \in V_1 - \hat{V_1}} gap_{G_1}(u)) \quad \text{(III.4)}$$

The distance model contains three components. The *StructDist* component measures the structural differences of the match, the $NodeMismatches$ component is the penalty associated with matching two nodes with different labels, and the *NodeGaps* component is used to measure the penalty for the gap nodes. (Gap nodes are nodes in the query that cannot be mapped to any nodes in the target graph.) Each of these components is described in more detail in subsections 3.2.2.1 through 3.2.2.3.

In Equation III.1, w_e, w_n, and w_g are the weights for each component in this matching model, and can be used to change the emphasis on the different parts of the similarity model. While Equation III.1 computes the subgraph distance for a specific matching λ, the actual subgraph distance from a query to its target is the minimum distance over all possible matchings, namely:

$$SGD(G_1, G_2) = \min_\lambda SGD_\lambda(G_1, G_2) \quad \text{(III.5)}$$

3.2.2.1 The StructDist component

The *StructDist* component measures the structural differences for the matching node pairs in the two graphs. In Equation III.2, the $d_{G_i}(u,v)$ function measures the

"distance" between node u and node v in graph G_i, and is defined as the length of the shortest path between u and v. The *StructDist* component compares the distance between each pair of matched nodes in one graph to the distance between the corresponding nodes in the other graph, and accumulates the differences.

3.2.2.2 The NodeMismatches component

The $NodeMismatches$ component in Equation III.3 is the sum of the penalties (quantified by the $mismatch$ function) associated with matching nodes with different labels.

A common and biologically intuitive mismatch penalty model is to *implicitly* group node labels based on similarity, allowing for a node label to be associated with more than one group. Nodes can then be compared based on the group labels. This model of node comparison is quite general and practical for many biological applications. For example, the functional similarity between two enzymes can be determined based on the length of the common prefix of the corresponding Enzyme Commission (EC) numbers. For general proteins, one can use databases like KEGG [34] and COG [50] which organize proteins into *orthologous* groups, and consider two proteins to be functionally similar only if they are in the same group. This mismatch model can also be generalized to other settings, such as comparing nodes belonging to different classes based on the positions of the two classes in a classification hierarchy, such as Gene Ontology (`http://www.geneontology.org`).

We utilize the concept of *orthologous groups* for our node mismatch model. The mapping from a node label to a set of orthologous groups, allowing a node to belong

to more than one orthologous group, is defined as $\varrho : L \rightarrow P(GL)$, where L is the set of node labels, GL is the set of group labels, and $P(GL)$ is the power set of GL. Under this model, $mismatch(L_i, L_j) = \infty$ if $\varrho(L_i) \cap \varrho(L_j) = \varnothing$, and $mismatch(L_i, L_j) < \infty$, otherwise.

3.2.2.3 The NodeGaps component

The *NodeGaps* component in Equation III.4 measures the penalties associated with the gap nodes in the query graph, thereby favoring matches that have fewer gap nodes. In our model, different nodes in the query graph can have different penalty values, and nodes with the same label can have different penalties as well.

The model also gives users the freedom to choose between gapped matches (matches that allow gap nodes) and ungapped matches. If $gap_G(u)$ is set to ∞ for every node, then the model only supports ungapped matches, otherwise it allows gapped matches.

For simplicity, for the rest of the discussion, we will assume that all nodes have the same gap penalty value denoted as *SingleGapCost*.

3.2.2.4 Characteristics of the subgraph distance model

Our subgraph matching model is very flexible and allows for incorporation of domain knowledge into the scoring criteria. The only restriction is that the gap penalty must be positive and the mismatch penalty must be non-negative. These restrictions ensure that the subgraph distance is a non-negative value. With these restrictions, if the query graph is subgraph-isomorphic to the target graph, the subgraph distance is 0, and vice versa.

3.2.3 The index-based matching algorithm

A naïve technique for evaluating subgraph matching queries is to compare the query with every graph in the database and report the matches, which is prohibitively expensive. We propose a novel index-based heuristic algorithm that allows for a much faster evaluation of the approximate subgraph matching operation.

First, an index is built on small substructures of graphs in the database. This index is then used to match fragments of the query with fragments in the database. Finally, the matching fragments are assembled into larger matches. The actual method is described in detail below.

3.2.3.1 The index structures

The index on small substructures of graphs in the database is called the *FragmentIndex*. It is probed by the matching algorithm to produce hits for substructures in the query.

The indexing unit is a set of k nodes from the graphs in the database. We call each such set a *fragment*. Here k is a user specified parameter, and is usually a small number like 2, 3 or 4. However, simply enumerating all possible k-node sets is expensive in terms of both time and space. At the same time, if any pair of nodes in a fragment is too far apart by the pairwise distance measure (refer to Section 3.2.2.1), this fragment does not correspond to a meaningful substructure, thus is not worth indexing. Therefore, a parameter d_{max} is specified to control whether a fragment is to be indexed. For a given k-node set $v_1, v_2, ..., v_k$, if any two nodes v_i and v_j satisfy

$d(v_i, v_j) \leq d_{max}$, we connect the two nodes by a pseudo edge. Then, we index this fragment only if the k nodes form a connected graph by the pseudo edges. Using this heuristic, we can dramatically reduce the size of the *FragmentIndex*.

Note that in contrast to existing methods, which index connected subgraphs, the fragments in SAGA do not always correspond to connected subgraphs. The reason for using the more general definition of fragments is to allow node gaps in the match model. For example, in Figure 3.1(b), nodes v_3 and v_4 in G_1 can be matched to nodes v_3 and v_4 in G_2, respectively. Although v_3 and v_4 do not form a connected subgraph in G_2, they correspond to a fragment that needs to be indexed so that this match can be detected.

An entry in the *FragmentIndex* has the following format: {*nodeSeq*, *groupSeq*, *distSeq*, *sumDist* *gid*}, where *nodeSeq* is the sequence of node IDs for the nodes in the fragment, *groupSeq* is the sequence of group labels associated with the nodes, *distSeq* is the sequence of pairwise distances between the nodes in the fragment, *sumDist* is the sum of these pairwise distances, and *gid* is a unique graph ID. Recall that a node label can be associated with multiple group labels. In this case, we generate all possible group label sequences for a fragment, and index each one.

A sample *FragmentIndex*, with $k = 3$ and $d_{max} = 2$ for the database shown in Figure 3.2, is presented in Figure 3.3. In this index, the *groupSeq*'s are ordered by the group IDs, and the *nodeSeq*'s are ordered according to the *groupSeq*'s. If u, v, w is the *nodeSeq*, then the corresponding *distSeq* is $d(u, v)$, $d(u, w)$, $d(v, w)$. Note that node v_8 with the label L_8 in G_1 belongs to two groups B and D, thus for this node set $\{v_1, v_3, v_8\}$, there are two index entries $\{(B, E, E), G_1, (v_8, v_1, v_3), (1, 2, 2), 5\}$ and

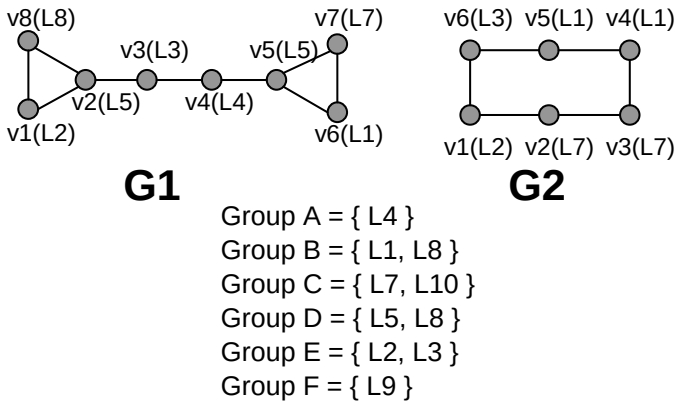

Figure 3.2: Example database graphs

$\{(D, E, E), G_1, (v_8, v_1, v_3), (1, 2, 2), 5\}$ in the index.

To efficiently evaluate the subgraph distance between a query graph and a database graph, an additional index called *DistanceIndex* is also maintained. This index is used to look up the precomputed distance between any pair of nodes in a graph (Section 3.2.2.1).

3.2.3.2 The matching algorithm

The matching algorithm proceeds as follows: First, the query is broken into small fragments and the *FragmentIndex* is probed. Then, the hits from the index probes are combined to produce larger candidate matches. Finally, each candidate is examined to produce the actual results. Each of these three steps is described in detail below.

Step 1: Finding small hits. In this step, the query is broken into small fragments and the *FragmentIndex* is probed to find database fragments that are similar to the query fragments.

Given the query, fragments (k-node sets) are enumerated in the same way as we

Fragment Index				
Group Seq	Graph ID	Node Seq	Distance Seq	DistSum
A,B,C	G1	v4,v6,v7	2,2,1	5
.....	.....	.....	.....	.....
A,C,D	G1	v4,v7,v2	2,2,4	8
		v4,v7,v5	2,1,1	4
.....	.....	.....	.....	.....
B,E,E	G1	v8,v1,v3	1,2,2	5
	G2	v4,v1,v6	3,2,1	6
		v5,v1,v6	2,1,1	4
.....	.....	.....	.....	.....
D,E,E	G1	v2,v1,v3	1,1,2	4
		v5,v1,v3	4,2,2	8
		v8,v1,v3	1,2,2	5

Figure 3.3: The *FragmentIndex* for the example database

did for the database graphs. Next, for each query fragment, the *groupSeq*, *nodeSeq*, *sumDist*, and *distSeq* values are computed. Then, the *FragmentIndex* is probed with each of these *query fragments*.

The actual index probe uses the following multi-level filtering strategy: First, the *groupSeq* and *sumDist* values are used to filter out fragments that cannot match. Next, additional false positives are removed using the *distSeq* values.

In the first level of filtering, database fragments are fetched only if they have the same *groupSeq* as the query fragment. We also develop safe bounds for the *sumDist* attribute as follows: Suppose that q is the query, p is a database graph, f_q is a query fragment, and k is the fragment size. We introduce a user-controllable parameter $MaxPairDist$ to restrict the weighted pairwise distance difference between the query and the database fragments as $w_e \times |d_{G_1}(u,v) - d_{G_2}(\lambda u, \lambda v)| \leqslant$

$MaxPairDist$. From this structure similarity restriction , we get the following inequality: $\sum_{u,v\in\hat{V}_q,u<v}|d_q(u,v)-d_p(\lambda u,\lambda v)| \leqslant \frac{k(k-1)}{2}\times\frac{MaxPairDist}{w_e}$, where k is the fragment size. In addition, we have the following trivial inequality:

$$|\sum_{u,v\in\hat{V}_q,u<v} d_q(u,v) - \sum_{u,v\in\hat{V}_q,u<v} d_p(\lambda u,\lambda v)| \leqslant \sum_{u,v\in\hat{V}_q,u<v} |d_q(u,v)-d_p(\lambda u,\lambda v)|$$

Using the two inequalities above, we can conclude that a database fragment f_d cannot match the query fragment f_q, if $|f_d.sumDist - f_q.sumDist| > \frac{k(k-1)}{2}\times\frac{MaxPairDist}{w_e}$. In other words, when probing the *FragmentIndex* in the first level of filtering, we only fetch the database fragments $\{t \mid t \in FragmentIndex,\ t.groupSeq = f_q.groupSeq,\ f_q.sumDist - \frac{k(k-1)}{2}\times\frac{MaxPairDist}{w_e} \leqslant t.sumDist \leqslant f_q.sumDist + \frac{k(k-1)}{2}\times\frac{MaxPairDist}{w_e}\}$.

The probing condition above includes an equality search and a range search. It imposes several optimization opportunities. First, to reduce the IO costs, we can group all the probes by the *groupSeq*. A good way of implementing the *FragmentIndex* is to order the physical layout of the index by *groupSeq* and *sumDist* attributes. Then, probes with the same *groupSeq* have very high spatial locality, which reduces the number of random IOs that are incurred during the index probes. In addition, for each group of probes with the same *groupSeq* value, we can optimize the range query scans on the *sumDist* attribute. Essentially, if query ranges overlap, a query can be issued with the union of the ranges rather than several overlapping individual queries, which further reduces the IO cost.

It is possible that more than one node in a fragment has the same group label. To correctly handle this case, we simply expand the query probe set to include a probe

set for every possible node sequence for the same group sequence.

After the first level of filtering, we get a list of candidate database fragments for every query fragment. This list can be further refined by using the *distSeq* information (which contains the pairwise distances) to check that all pairwise distances satisfy the $MaxPairDist$ criterion defined above.

Step 2: Assembling small hits. Step 1 produces a set of small fragment hits. These smaller hits are assembled into bigger matches as follows: First, the hits are grouped by the database graph IDs. Then, a *hit-compatible graph* is built for each matching graph. Each node in a hit-compatible graph corresponds to a pair of matching query and database fragments. An edge is drawn between two nodes in the hit-compatible graph if and only if two query fragments share 0 or more nodes, and the corresponding database fragments in the hit-compatible graph also share the same corresponding nodes. An edge between two nodes tells us that the corresponding two hits can be merged to form a larger match, since they have no conflicts in the union. Therefore, a clique in the hit-compatible graph represents a set of hits that can be merged without any conflicts.

After forming the hit-compatible graph, the hits assembling problem reduces to the maximal clique detection problem, which can be solved using existing efficient implementations, such as [8], or approximate methods such as [24]. The set of hits in each maximal clique is a candidate match.

As an example of the second step of the SAGA matching algorithm, Figure 3.4(b) shows the hit-compatible graph for the database graph G_1 in Figure 3.2 when querying Q in Figure 3.4(a), with $MaxPairDist = 1$. The nodes in the hit-compatible graph

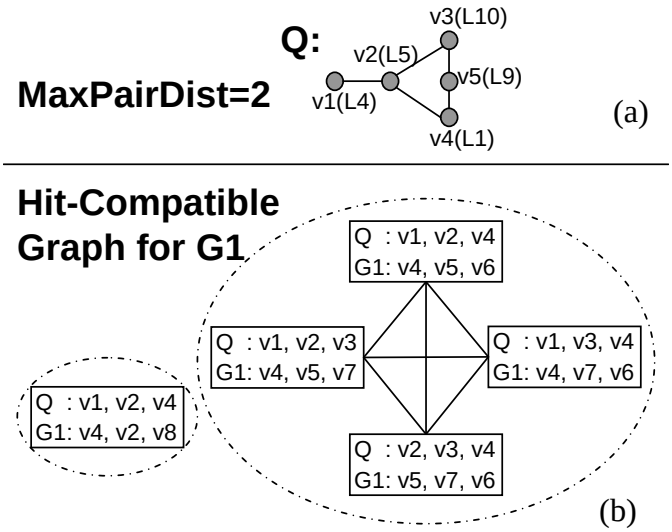

Figure 3.4: (a) An example query Q (b)The hit-compatible graph for G_1 when querying Q.

are denoted by rectangles. Two maximal cliques (shown as dotted circles) are detected in this hit-compatible graph. Therefore, this step produces two candidate matches in G_1 for query Q, namely $Q(v_1, v_2, v_3, v_4) \leftrightarrow G_1(v_4, v_5, v_7, v_6)$, and $Q(v_1, v_2, v_4) \leftrightarrow G_1(v_4, v_2, v_8)$.

Step 3: Examining candidates. This step examines each candidate match and produces a set of real matches. Here, we allow users to specify a threshold P_g to control the percentage of gap nodes in the subgraph match. With a given P_g value, the desired matches are those with at most P_g percentage of gap nodes in the query.

For each candidate match obtained from Step 2, we first check whether the percentage of the gap nodes exceeds the threshold P_g. If so, we ignore the candidate. Otherwise, we probe the *DistanceIndex* and calculate the real subgraph matching distance as defined in Section 3.2.2. Recall that the required subgraph matching is the one that minimizes the matching distance (cf. Equation III.5). We also further

examine the submatches of the candidate. A submatch can be obtained by removing one or more node mappings from the original match. This introduces more gap nodes to the query, and thus increases the subgraph distance by additional gap penalties. However, at the same time, the $StructDist$ and $NodeMismatches$ may be reduced according to its definition in Equations III.2 and III.3. Therefore, if the decreased amount exceeds the increased amount, the overall matching distance will be lower than the original one, which also means that a better match is found for the query. If two matches have the same matching distance and one is a submatch of the other, only the supermatch is considered.

3.2.4 Fragment size parameter

The fragment size parameter (k in Section 3.2.3.1) controls the size of fragments in the *FragmentIndex*. This parameter affects the size of the index, query performance, and sensitivity of search results. A larger fragment size results in a larger *FragmentIndex*, which increases the index probe cost. However, a large fragment size may also results in fewer false positives in the hit detection phase (and lower query sensitivity), which reduces the cost of the remaining steps. A practical way of picking a fragment size is based on the selectivity of the queries. If queries are expected to have many matches in the database, then a smaller fragment size is preferred as it may not introduce many false positives, and also potentially lead to smaller sizes of hit-compatible graphs. However, when queries tend to have very few matches, a large fragment size may be favored to prune false positives in the early stages of the

matching algorithm.

3.2.5 Statistical significance of matching results

The Monte Carlo simulation approach is employed to assess the statistical significance of the matches. A p-value is computed for each match based on the frequency of obtaining such a match, or a better match, when applying SAGA with randomized data. Random graphs are generated by random shuffling of edges of the graphs preserving the node degrees, and randomizing the orthologous groups of each node preserving the number of orthologous groups that each node belongs to. For a given query, in addition to querying the real database, we run SAGA on a large number of random graphs, and estimate the p-value of a match from the real database as the fraction of matches from the random graphs with the same or a larger size (in number of nodes) and the same or a smaller distance value.

Appendix A contains the detailed description of the statistical evaluation methods for approximate graph matching results in Periscope/GQ.

3.3 Implementation and Results

In this section, we describe the implementation of SAGA and present results demonstrating its effectiveness and efficiency. The well-known KEGG pathway database [34] is used for the experiments. In addition, we use a dataset, called bioNLP, which contains parsed PubMed documents represented as graphs. In these graphs, nodes represent genes and edges denote that two genes were discussed in the same sentence

Query	Match	#nodes matched	p-value	# refs.
T2DM (hsa04930)	Insulin (hsa04910)	8	0.0009	21,326
	Adipocytokine (hsa04920)	5	0.0009	37
H.pylori (hsa05120)	Toll-like receptor (hsa04620)	7	0.001	12
	T-cell receptor (hsa04660)	4	0.001	2
	Apoptosis (hsa04210)	4	0.006	130

Table 3.1: Significant matches for the T2DM and H.pylori disease associated KEGG pathways. The number of PubMed references is simply produced by querying PubMed with the keywords in the pathway names.

somewhere in the document. With bioNLP, graph similarity can be used to identify related documents.

3.3.1 Implementation

We have implemented SAGA using C++ on top of PostgreSQL (`http://www.postgresql.org`). For detecting maximal cliques, we use the version 2 algorithm described in [8]. The *DistanceIndex* and *FragmentIndex* are implemented as clustered B+-tree indices. The fragment size was set to 3. The execution times reported correspond to the running time of the C++ program (which includes reading the query specifications and issuing SQL queries to the DBMS to fetch index entries and related database tuples). All experiments were run on a 2.8GHz Pentium 4, Fedora 2 machine equipped with a 250GB SATA disk. We used PostgreSQL version 8.1.3 and set the buffer pool size to 512MB.

For all the experiments with KEGG, the values for the SAGA parameters are: $w_e = w_g = w_n = 1$, $SingleGapCost = 3$, $d_{max} = 3$, and $MaxPairDist = 3$. The P_g value is set for every query so that each match contains at least four node mappings.

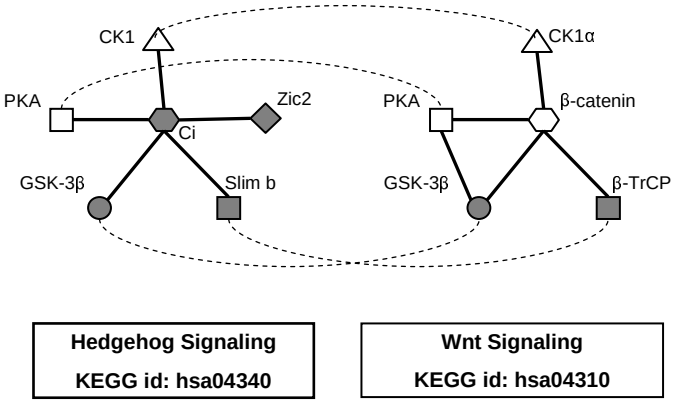

Figure 3.5: Hedgehog pathway matched the Wnt pathway.

For the node mismatch penalty, we use a simple model: if two nodes belong to the same KEGG orthologous group or they have the same EC number, then the mismatch penalty is 0, and ∞ otherwise. For the significance test, we generate 100 random graphs for each graph in the database, so there are totally $n \times 100$ random graphs, if n is the number of graphs in the database. We only retain matches with 0.01 significance level or better. When a query graph is also included in the database, we always exclude the self-match (the query graph matching itself) from the results. For the experiment with the bioNLP dataset, the SAGA parameter settings are: $w_e = w_g = w_n = 1$, $SingleGapCost = 0.5$, $d_{max} = 3$, and $MaxPairDist = 3$. For the node mismatch model, nodes with the same label have 0 penalty, otherwise the mismatch penalty is ∞.

3.3.2 Finding conserved components across pathways

Two experiments are used to investigate components that are shared across different pathways.

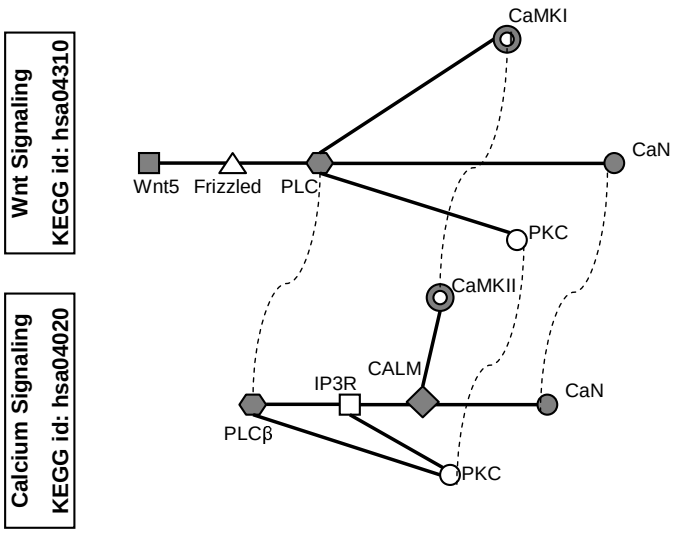

Figure 3.6: Wnt pathway matched the Calcium pathway.

3.3.2.1 Querying disease-associated pathways

This experiment is an exploratory analysis to find biological processes that are involved in, or are affected by, a particular disease. We use all 162 KEGG human pathways (downloaded on July 4, 2006) as the database and chose the 10 disease-associated human pathways as queries (see Table 3.2). This query set is a subset of the 162 human pathways and it includes three metabolic disorder pathways, six neuro-degenerative disorder pathways, and one infectious disease pathway. Of these pathways, only two query pathways produced significant hits (p-value ≤ 0.01): the "Type 2 Diabetes Mellitus" (T2DM) pathway (hsa04930) and the "Epithelial cell signaling in Helicobacter pylori infection" (H. pylori) pathway (hsa05120). Results for these two pathways are presented in Table 3.1.

Table 3.1 shows both the p-values and the number of PubMed references for the matches, as a measure of how well the disease association has been studied in previous

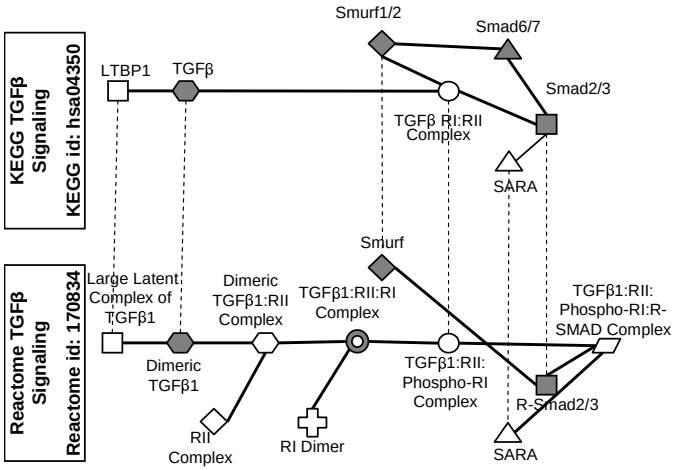

Figure 3.7: The shared components between KEGG and Reactome TGF-β pathways.

literature. We are particularly interested in disease-associated pathway matches that are significant but are not yet well studied.

As can be seen in Table 3.1, SAGA finds that the T2DM pathway (hsa04930) is significantly associated with both Insulin signaling (hsa04910) and Adipocytokine signaling (hsa04920). In the case of Insulin signaling, we find a match of eight nodes of Insulin signaling in the T2DM pathway. The number of PubMed references for "Type II diabetes mellitus AND Insulin" is 21,326, consistent with the well-studied nature of Insulin signaling in T2DM. This result demonstrates that SAGA finds pathway matches that would be expected by researchers experienced in disease-related pathways research. In the case of Adipocytokine signaling in T2DM, we find a match of five nodes and the number of references is 37, in agreement with the less well-studied nature of Adipocytokine signaling in T2DM.

The H. pylori pathway (hsa05120) demonstrated significant matches to the Toll-like receptor, T-cell receptor, and Apoptosis pathways. The association between H.

Category	KEGG ID	Pathway	#nodes	#edges
Metabolic Disorders	hsa04930	Type II diabetes mellitus	33	36
	hsa04940	Type I diabetes mellitus	22	2
	hsa04950	Maturity onset diabetes of the young	34	33
Neuro-degenerative Disorders	hsa05010	Alzheimer's disease	23	17
	hsa05020	Parkinson's disease	19	10
	hsa05030	Amyotrophic lateral disease	24	13
	hsa05040	Huntington's disease	24	28
	hsa05050	Dentatorubropallidoluysian atrophy	8	10
	hsa05060	Prion disease	11	15
Infectious Disease	hsa05120	Epithelial cell signaling in Helicobacter pylori infection	57	26

Table 3.2: The ten disease associated human pathways in KEGG

pylori infection and Apoptosis is relatively well studied (130 PubMed references), while the association with Toll-like receptor signaling is less well studied (12 references) and the association with T-cell receptor signaling shows only two references. This result suggests that T-cell receptor signaling is potentially a significant but relatively unstudied avenue for research into the etiology of H. pylori infection.

3.3.2.2 Querying signal transduction pathways

In this experiment, we use the same database of pathways as in section 3.3.2.1 (162 KEGG human pathways) but we choose all the 12 signal transduction pathways (KEGG IDs: hsa04010, hsa04020, hsa04070, hsa04150, hsa04310, hsa04330, hsa04340, hsa04350, hsa04370, hsa04630, hsa04910, and hsa04920) as the query set to demonstrate additional benefits to be derived from identifying pathways matches. Many of the matches are intuitive for researchers familiar with specific cellular, tissue, or disease phenomena (as expected). However, pairs of pathways between which the

similarities are not intuitive can be useful in both pathway annotation and disease association research. In the following discussion, we present two examples of such matches.

In the first example, Figure 3.5 shows components that are shared by the Hedgehog (hsa04340) and Wnt (hsa04310) signaling pathways (p-value 0.005). Note that nodes are matched based on functionality. For example, Slimb is matched with B-TrCp as both are SCF complex F-box proteins (KEGG Orthology, KO:K03362). While SAGA can find this orthologous match, the difference in terminology seen in the KEGG pathways database might make it difficult for many researchers to find the match. These similarities between Hedgehog and Wnt signaling are consistent with `http://www.stanford.edu/~rnusse/pathways/WntHH.html`, as well as [29] and [39].

In the second example, the Wnt and Calcium signaling pathways share four enzymes (Figure 3.6, p-value 0.007). However, the Calcium signaling pathway has two additional components (CALM and IP3R) that arguably belong to the Wnt pathway. By identifying the common components, we can provide information to improve the annotation of the Wnt pathway.

Based on the significant similarities between the Wnt/Hedgehog and Wnt/Calcium pathways, we hypothesize that the three pathways (Wnt, Calcium, and Hedgehog signaling) could share disease associations. Calcium signaling has been investigated in relation to Bipolar Disorder (BD) for more than 40 years [15]. After examining the Wnt/Calcium and Wnt/Hedgehog matches, we conducted a literature search and found 335 PubMed references investigating Calcium signaling in BD, as well as 15 PubMed references for Wnt signaling in BD, consistent with our hypothesis.

However, when looking for BD association with Hedgehog signaling, we found zero PubMed reference, which suggests that the Hedgehog signaling pathway has been largely overlooked in BD research, although it uses BD-associated components. This result poses new hypotheses for exploring the relationship between BD and Hedgehog signaling, and shows how SAGA can be useful in disease research.

3.3.3 Reactome pathways vs. KEGG pathways

SAGA can also be used to compare pathways in different databases (e.g., as a precursor to integrating data from different pathway databases). In this experiment, we compare two well-known pathway databases: Reactome [18] and KEGG.

We use the same 162 KEGG human pathways as the database. The queries are the eight newly updated pathways in Reactome version 17. The query set includes TGF-β (Reactome ID: 170834), RIG-I (168928), Toll-like receptors 3 (168164) and 4 (166016), the conjugation phase of xenobiotic metabolism (156580), aspects of the metabolism of lipoproteins(174824), cell cycle regulation by the anaphase-promoting complex (APC) (174143), and ATR activation in response to replication stress (176187).

Naturally, the TGF-β pathway (with 23 nodes and 25 edges) in Reactome matches the TGF-β (hsa04350) pathway (with 65 nodes and 45 edges) in KEGG. However, pathways in the two databases are not perfectly matched (graph distance > 0). Each of the pathways contains some details missing in the other. Also, as shown in Figure 3.7, there are some differences even in the shared similar components between the

Dataset	Pathways	# graphs	avg. # nodes	avg. # edges	*FragmentIndex* Size (# entries)
d1	human	162	86.0	35.3	1.38×10^7
d2	d1 + mouse	320	86.3	34.8	2.94×10^7
d3	d2 + rat	470	86.6	31.7	4.07×10^7
d4	d3 + worm	567	89.0	28.5	5.34×10^7
d5	d4 + yeast	654	91.3	27.3	6.08×10^7

Table 3.3: Characteristics of various databases used for the scalability experiment. This table shows the number of graphs in each database, the average number of nodes and edges per graph in the databases, and the number of entries in the FragmentIndex.

two pathways. By identifying the similar subcomponents using SAGA, researchers can combine the two databases and produce more complete data.

The two databases also organize pathways in different ways. Reactome represents pathways in a hierarchy (i.e. a pathway consists of several subpathways and subpathways again can be made of subpathways). On the contrary, KEGG stores pathways in a flat fashion. As examples of the organizational difference, the Toll-like receptor 3 and 4 pathways in Reactome match the Toll-like receptor (hsa04620) pathways in KEGG, and both cell cycle regulation by the Anaphase-promoting complex (APC) and ATR activation in response to replication stress pathways in Reactome hit the cell cycle pathway in KEGG. Thus, SAGA can be used for graph data integration even if databases organize the same information in different ways.

3.3.4 SAGA for querying parsed literature graphs

This experiment examines how SAGA can be applied within an information retrieval setting. While traditional IR methods employ term-based comparisons and the cosine similarity measure [44] for comparing documents, we look at the docu-

ment comparison problem specifically in the biomedical domain and address it using a graph matching method. Each PubMed document is represented by a graph in which a node indicates a gene studied in that document. A link is drawn between two genes if they are discussed in the same sentence (indicating there is potentially association between the two genes). The graph presentation summarizes the genes and gene associations derived from a document. By querying the graph representation of a document against those of other documents, documents that address the same topics as the query document can be identified, even if they are published in different areas of research. For example, we queried the publication [36] (5 nodes and 6 edges) against 48,444 PubMed documents using the cut-off value $P_g = 50\%$. (This dataset has an average of 5.0 nodes and 18.8 edges per graph, and the list of documents in this set can be accessed at `http://enigma.eecs.umich.edu/doc.txt`.) Among the 11 matches found by SAGA, the top hit is [49], which does not have a citation to [36]. The shared components between the two graphs are three genes: CDK inhibitor p18(INK4c), 0610007C21Rik and Stmn1, as well as their 3 pairwise associations. The query publication [36] explored p18(INK4c) in the generation of functional plasma cells, while [49] investigated the role of this gene in the regenerating liver. Thus, SAGA can be used to connect related studies even in different sub-areas of biomedical research.

Query	# nodes	# edges	d1	d2	d3	d4	d5
hsa05050	8	10	25.6	28.6	37.1	37.3	37.4
hsa05060	11	15	45.4	53.6	62.0	62.1	62.1
hsa05020	19	10	26.8	36.9	53.7	53.7	53.7
hsa04940	22	2	0.2	0.2	0.2	0.2	0.2
hsa05010	23	17	45.2	58.0	61.9	62.1	62.1
hsa05030	24	13	42.3	42.4	52.9	52.9	53.1
hsa05040	24	28	347.2	431.3	457.4	459.1	462.4
hsa04930	33	36	243.6	411.7	540.6	541.4	546.2
hsa04950	34	33	29.6	29.6	29.6	29.7	29.7
hsa05120	57	26	116.5	160.6	182.8	183.1	183.7

Table 3.4: Execution time (in milliseconds) for the 10 disease-associated pathways in KEGG when querying the databases listed in Table 3.3.

3.3.5 Comparison with existing tools

GraphGrep [45] and Gindex [58] are designed to match one graph against a collection of graphs. However, they only support exact subgraph isomorphism. Given the noisy and incomplete characteristics of biological graphs, exact matching cannot help much in our target applications. Grafil [59], PIS [60], and Closure-Tree [22] disallow gap nodes in their match models, which prohibits them from getting results that SAGA can find. For example, none of the 12 signal transduction pathways queries produce any matches (excluding self-matches) in the KEGG human pathway database using these three tools.

As discussed in Section 3.1, NetworkBlast is a tool for aligning large protein interaction networks. On the other hand, SAGA is designed for matching relatively small graph queries (sparse graphs with less than 100 nodes) against a large set of (large or small) graphs. Although NetworkBlast and SAGA have different characteristics, it is interesting to consider applying NetworkBlast to pathway matching. To query the set of pathways in KEGG (cf. Section 3.3.2.2), we have to run NetworkBlast once for

each pathway in the database. In other words, for the experiment in Section 3.3.2.2, for each query, we need to invoke 162 calls to NetworkBlast. For the Wnt signaling pathway (hsa04310) with 73 nodes and 92 edges, the 162 runs of NetworkBlast takes more than 20 hours, while SAGA only takes about eight minutes! Besides the more than two orders of magnitude speedup, SAGA produces results with higher quality. First, SAGA never misses any matching pathways that NetworkBlast can find. Secondly, SAGA can find matches that NetworkBlast cannot find. The reason is that graph structural differences in NetworkBlast are largely confined to short paths, while SAGA tolerates more general structure differences. For example, neither of the two matches shown in Figures 3.5 and 3.6 can be found by NetworkBlast.

3.3.6 Efficiency evaluation

This experiment evaluates the efficiency of SAGA. To measure the raw performance, we only measure the time it takes for SAGA to produce matches, and do not include the time for generating the p-value statistics.

We choose as queries the 10 disease associated KEGG pathways (mirroring the experiment in Section 3.3.2.1). To vary the database sizes, we add pathways for other species to the database. The details of the databases are described in Table 3.3.

The query execution times for the 10 queries with increasing database sizes are shown in Table 3.4. Even for the largest database, the query execution times using SAGA are less than one second.

Besides the database sizes, the query execution times also depend on the number

of nodes and edges in the query, the actual query graph structure, and the number of hits in the database. Almost all the 10 human disease pathways have matches in the human, mouse and rat pathways, but no matches exist for them in the worm and yeast pathways. This explains why the execution times for the queries on the databases d4 and d5 are similar to the execution times against the database d3. For the databases d1 through d3, even though the database sizes roughly doubles at each step, the query execution times grow at a slower rate, since the index matching components grow at at rate that is slower than the database growth rate.

Another observation is that a larger query does not necessarily result in a larger execution time. For example, hsa05040 is a single connected graph with more matches in the databases than hsa04950, which is a graph with several connected components. The execution times with hsa05040 are more than hsa04950, although hsa04950 has more nodes and edges than hsa05040.

3.4 Discussion

This chapter discusses SAGA, a powerful method for approximate subgraph matching. SAGA employs a match model that can be used to accurately incorporate domain knowledge for capturing the domain-specific notion of graph similarity. An index-based algorithm makes approximate subgraph matching queries very efficient. Our evaluations using a number of actual biomedical applications show that SAGA can produce biologically relevant matches on actual examples, whereas existing tools fail. In addition, we have demonstrated the efficiency of the SAGA approach.

SAGA is very effective and efficient for querying relatively small graphs (ideally sparse graphs with less than 100 nodes) against very large databases, and there are many compelling applications in this setting (cf. Section 3.3). However, we do not recommend using the existing tool when the query graph is very dense and/or has a large number of nodes. For such large query graphs, the performance of the existing SAGA method degrades since potentially a large number of small hits can be produced by Step 1 of the matching algorithm (cf. Section 3.2.3.2). Assembling these hits is computationally expensive with the existing SAGA algorithm. In Chapter IV, we introduce another graph matching method, called TALE, to handle the case of very large query graphs.

CHAPTER IV

Approximate Large Graph Matching

4.1 Introduction

In the previous chapter, we have introduced an efficient subgraph matching method, SAGA. As discussed in Section 3.4 of Chapter III, SAGA works efficiently for querying relatively small graphs (ideally sparse graphs with less than 100 nodes), but its performance degrades rapidly for large query graphs (with hundreds to thousands of nodes and edges). Most existing tools [22, 28, 55, 59, 60, 61] are also only applicable to small queries. However, in many new applications, both the query and database graphs are "large". For example, in life sciences applications, protein interaction networks for individual species are often matched to determine similarities and differences across species. Each protein interaction network is large, and typically contains hundreds to thousands of nodes and edges in each graph.

To handle large query graphs, we present an index-based method for approximate subgraph matching of large queries, called TALE (a **T**ool for **A**pproximate Sub-

graph Matching of **L**arge Queries **E**fficiently). TALE employs a novel graph indexing method, called NH-Index (Neighborhood Index). Most existing graph indexing methods only index subgraphs (paths, trees or general subgraphs), which can lead to index sizes that are exponential in the database size. The indexing unit of NH-Index is the neighborhood of each database node. The neighborhood concept captures the local graph structure around each node, and results in an index with a high pruning power. At the same time, the number of indexing units is equal to the number of nodes in the database, which allows the index to grow linearly with the database size. Furthermore, NH-Index is a disk-based index, which allows it to handle graph databases that do not fit in memory. It employs a hybrid index that uses existing common disk-based index structures, which makes implementation in existing DBMSs straightforward.

We also propose an innovative matching paradigm for querying large graphs. Unlike most previous graph matching tools which treat every node in a graph equally, this matching technique distinguishes nodes by their importance in the graph structure. The algorithm first probes the NH-Index to match the important nodes in a query graph, and then progressively extends the matches by enclosing satisfiable nearby nodes of already matched nodes.

We have applied TALE to three real biological datasets. Our experiments demonstrate that TALE is able to produce useful and meaningful results in all the three cases. In addition, our experimental evaluation shows that TALE is very efficient for large queries, and that the execution time grows gracefully with increasing number of graphs in the database. Through comparisons with other existing tools, we also show that TALE is significantly faster than existing methods.

The main contributions of this chapter are as follows:

(1) We propose TALE – a general tool for *approximate* subgraph matching of *large* graph queries. TALE uses a novel disk-based indexing method, which indexes the neighborhood of each database node. It achieves high pruning power and its size scales linearly with the database size. We introduce an innovative graph matching paradigm, which distinguishes nodes by their importance in the graph structure, and accordingly treats them differently in the matching process.

(2) By applying TALE to real applications, we show its effectiveness, significant performance improvements over existing methods, and ability to gracefully handle large graph queries and databases.

The remainder of this chapter is organized as follows: Section 4.2 defines the preliminary concepts. Section 4.3 describes our indexing mechanism, and Section 4.4 introduces the TALE algorithm. Experimental results are presented in Section 4.5, and Section 4.6 contains our conclusions and directions for future work.

4.2 Preliminaries

A graph G is denoted as (V, E), where V is the set of nodes and $E \subseteq V \times V$ is the set of (directed or undirected) edges. Nodes and edges can have labels specified by mappings $\phi : V \rightarrow \Sigma_v$ and $\psi : E \rightarrow \Sigma_e$ respectively, where Σ_v is the set of node labels and Σ_e is the set of edge labels. In order to uniquely identify a node, we assign an unique id to each node in a graph. We also impose an order on the ids. Our indexing method and matching algorithm support both directed and undirected graphs with

labeled nodes and/or labeled edges. For ease of presentation, we present our method using undirected graphs with labeled nodes. Adaptations of our method to other graph types are fairly straightforward unless discussed. The simple adaptations are omitted in the interest of space.

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs. An exact graph match (graph isomorphism) is a bijection mapping function $\lambda : V_1 \leftrightarrow V_2$, in which for every $v \in V_1$, $\phi(v) = \phi(\lambda v)$, and $(u, v) \in E_1$ if and only if $(\lambda u, \lambda v) \in E_2$. An exact subgraph match (subgraph isomorphism) from G_1 (the query) to G_2 (the target) is defined as $\exists G_2' \subseteq G_2$, and G_2' is an exact graph match for G_1.

Approximate graph matching allows node mismatches (i.e. $\phi(v) \neq \phi(\lambda v)$), and node/edge insertions and deletions. We define an approximate graph match as a bijection mapping $\lambda : V_1' \leftrightarrow V_2'$, where $V_1' \subseteq V_1$ and $V_2' \subseteq V_2$. Similarly, an approximate subgraph match from G_1 (the query) to G_2 (the target) is defined as $\exists G_2' \subseteq G_2$, and G_2' is an approximate graph match for G_1.

An approximate subgraph matching tool often returns a large number of matches for a query. Often the user is only interested in the top-K results. To return the top-K results, TALE has to sort the matches based on their similarities to the query. We do not want to limit the generality of TALE by tailoring it to a particular similarity model. Instead, we let the users customize the similarity method that best models their application, thereby allowing TALE to serve as flexible graph matching tool that can be used in a variety of graph matching applications. Section 4.5 shows examples of how this similarity model can be customized in practice.

4.3 The NH-Index

In this section, we introduce the novel indexing technique, Neighborhood Index (NH-Index).

4.3.1 Indexing Unit

The first question that arises with a graph indexing method is the graph entities, e.g. nodes, edges, subgraphs, etc., that should be indexed. The NH-Index is used by the matching algorithm to match the important nodes in the query graph. These initial matches for the important nodes are then extended to produce the final matching results. A naive indexing method is to index all the nodes in the database. This method has the benefit that the index size grows linearly with the number of nodes in the database, but suffers from low pruning power, as each query node can have many false positive matches (matches that cannot be extended later). Our NH-Index size is linear in the number of nodes in the database and also has a high pruning power. NH-Index achieves this by incorporating neighborhood information into the naive node indexing method. When matching a query node, instead of looking at the node in isolation, NH-Index also considers its neighborhood. A database node matches the query node, only if the two nodes match *and* their neighborhoods also match. Using this technique, a large fraction of false positives can be eliminated.

A *neighborhood* is defined as the induced subgraph of a node and its neighbors (adjacent nodes). There are three main properties that characterize the neighborhood of a node: the number of neighbors, how the neighbors connect to each other, and the

labels of the actual neighbors. The number of neighbors is simply the degree of the node. To quantify the "connectedness" amongst the neighbors, we define *neighbor connection* as the number of edges between the neighbors. For example, the neighbor connection of the black node in Figure 4.1 is 5.

To capture the neighbors of a node, a naive method is to simply enumerate the labels of the neighbors. However, this naive approach results in variable-length index entries as well as large index size (in the worst case of a clique, the storage cost is $O(n^2)$, where n is the number of nodes in the database). An alternative to the naive approach is to use a compact bit array to capture the neighbors set. In the simple case when the total number of different labels in the problem domain is small (i.e. $|\Sigma_v|$ is small), we can use a deterministic bit array to store the neighbors. The size of the bit array is equal to $|\Sigma_v|$, and each bit in the array indicates whether a neighbor with a specific label exists (set to 1) or not (set to 0). We call this bit array *neighbor array*. When $|\Sigma_v|$ is a large number, using a deterministic bit array is very expensive. To handle this situation, we employ the Bloom filter approach [5]. We fix the size of the bit array to be S_{bit}, where S_{bit} is a user-controllable parameter. A hash function is utilized to map a node label to a bit array position. To improve precision, multiple bit arrays and hash functions can be used to characterize the neighbors of a node. For simplicity, we only use one bit array to store the neighbor information in this work.

In summary, the indexing unit of the NH-Index contains the following information: *(label, degree, nbConnection, nbArray)*, where *nbConnection* is the neighbor connection of the node, and *nbArray* is the neighbor array.

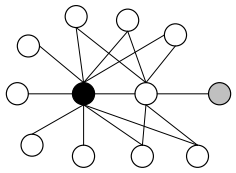

Figure 4.1: An example graph

4.3.2 Matching a Query Node

In the previous section, we discussed the indexing unit of the NH-Index. Next, given a query node, we examine how our method finds the matching database nodes. For ease of presentation, we first investigate the matching conditions for exact subgraph matching, and then extend it to approximate subgraph matching.

For exact subgraph matching, in order to match a query node to a database node, the two nodes must have the same label. The degree of the query node should be no more than that of the database node. The same condition holds for neighbor connections. Besides, the neighbors of the query node should have corresponding matching nodes in the neighborhood of the database node.

For approximate matching, we want to tolerate some misses in the match. We introduce a single user-defined parameter ρ, which is used to control the degree of approximation. Intuitively, ρ is the percentage of neighbors of a query node that can have no corresponding matches in the neighborhood of a database node. In other words, $nb_{miss} = (\rho \times N_q.degree)$ neighbors of the query node can be missing in the match to a database node. If nb_{miss} nodes are allowed to be missing, then at most $nbc_{miss} = nb_{miss} \times (nb_{miss} - 1)/2 + (N_q.degree - nb_{miss}) \times nb_{miss}$ neighbor connections are allowed to be missing in the match, i.e. in the worst case, the nb_{miss} nodes all

connect to each other, and also connect to all of the remaining $(N_q.degree - nb_{miss})$ nodes.

Note that we also support node mismatches (nodes with different labels are matched) in TALE. For ease of presentation, we delay the discussion of node mismatches to Section 4.3.5.1, and for now assume that matching nodes are required to have the same label.

Formally, the conditions for matching a query node to an NH-index entry for approximate subgraph matching is:

$$N_{db}.label = N_q.label \tag{IV.1}$$

$$N_{db}.degree \geq N_q.degree - nb_{miss} \tag{IV.2}$$

$$\sum_{i=1}^{S_{bit}} Miss(N_{db}.nbArray[i], N_q.nbArray[i]) \leq nb_{miss} \tag{IV.3}$$

$$N_{db}.nbConnection \geq N_q.nbConnection - nbc_{miss} \tag{IV.4}$$

The $Miss$ function in Equation IV.3 is defined as follows:

$$Miss(x, y) = \begin{cases} 1 & \text{if } x = 0 \text{ and } y = 1 \\ 0 & \text{otherwise} \end{cases}$$

In fact, exact subgraph matching can be viewed as a special case of approximate subgraph matching when $\rho = 0$.

Note that the conditions expressed in Equations IV.1 to IV.4 can result in pro-

ducing some false positives. Our index serves as a filtering mechanism to prune the search space. These matches are then refined in the matching algorithm (Section 4.4).

4.3.2.1 Node Match Quality

Given a query node, there can be more than one database node that satisfies the conditions specified in Equations IV.1 to IV.4. Each of these matches can have a different match quality. Therefore, we need to measure the quality of the node matches. This quality metric will then be used at a later step (see Section 4.4.2) following the index probe. In this section, we describe the match quality computation.

Let $\widetilde{nb}_{miss}$ be the actual number of missing neighbors in the node match, and $\widetilde{nbc}_{miss}$ be the actual number of missing neighbor connections. Then the fraction of missing neighbors of the query node can be defined as $f_{nb} = \frac{\widetilde{nb}_{miss}}{N_q.degree}$. And the fraction of missing neighbor connections can be defined as $f_{nbc} = \frac{\widetilde{nbc}_{miss}}{N_q.nbConnection}$. Then, we define the *quality* of a node match, w, as:

$$w = \begin{cases} 2 - f_{nbc} & \text{if } \widetilde{nb}_{miss} = 0 \\ 2 - (f_{nb} + \frac{f_{nbc}}{\widetilde{nb}_{miss}}) & \text{otherwise} \end{cases} \tag{IV.5}$$

Note that f_{nbc} is correlated with f_{nb}, as more missing neighbors is likely to result in more missing neighbor connections in the match. Therefore, we amortize f_{nbc} by the number of missing neighbors $\widetilde{nb}_{miss}$ in Equation IV.5. The value of $(f_{nb} + \frac{f_{nbc}}{\widetilde{nb}_{miss}})$ falls between 0 and 2. We subtract this value from 2, so that higher w value means a better node match.

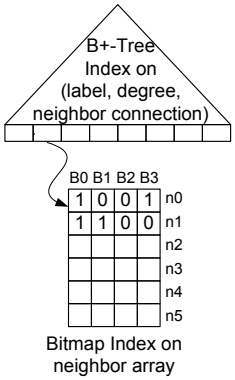

Figure 4.2: The hybrid index structure

4.3.3 Index Structure

Next, we consider the index structure to implement the NH-index. Rather than designing a new index structure, which makes adoption and implementation hard, it is desirable to consider using existing index structures that can implement the NH-index efficiently. A suitable index structure needs to support the conditions specified in Equations IV.1 through IV.4. We propose a simple hybrid index structure (see Figure 4.2) for the NH-Index.

This hybrid index structure has two levels. The highest level of the index structure is a B+-tree index on node label, degree and neighbor connection. This part of the index is used for fast evaluation of the equality search on node labels (Equation IV.1), range search on node degrees (Equation IV.2) and neighbor connections (Equation IV.4). Each leaf entry in the B+-tree index points to a second-level index. This second-level index has two components. The first is a list of database node ids that are represented by the B+-tree leaf index entry. (Recall from Section 4.2 that every database graph node has a unique node id.) These nodes have the same unique

label, degree and neighbor connection. The second component is a bitmap index for the neighbor arrays of these database nodes. Each node has one corresponding bit array in the bitmap. Figure 4.2 shows an example bitmap index for a B+-tree leaf entry that is mapped to six distinct database nodes with the same label, degree and neighbor connection. The bitmap index is used to expedite the evaluation of Equation IV.3 using Algorithm 1 (discussed in detail below).

Note that our hybrid index structure is easily implemented in existing relational systems. The second level indices can be implemented simply as a relation with two attributes: one that stores the list of database nodes, and the other that stores a bitmap (using an extensible data type). The first level index is simply a B+-tree built on this table. This simple implementation is robust and allows us to easily realize the NH-Index.

4.3.4 Index Probing

Given a query node, we first utilize the label, degree and neighbor connection information to probe the B+-Tree index. Then, we obtain a list of bitmaps that must be further examined using the conditions specified in Equation IV.3. An efficient algorithm for this evaluation is shown in Algorithm 1. This algorithm contains two steps. The first step (line 1 to 17) counts the number of missing neighbors of the query node in the match to each database node in a bitmap. The second step (line 18 to 30) prunes all the database nodes with the number of missing neighbors higher than the user threshold. We discuss these two steps in detail below.

Algorithm 1 Bitmap Probe for Approximate Subgraph Matching (N_q, $Bitmap$, ρ)

Input: N_q is the query node, $Bitmap$ is the bitmap index to be probed, assuming that there are n nodes in the bitmap index and the size of neighbor array is S_{bit}, ρ is the percentage of neighbors of a query node that can be missing in the match to a database node

Output: $Result_{le}$ is the bit vector indicating which nodes satisfy the query

1: // [Step 1]: count the number of missing neighbors
2: $nb_{miss} = \lfloor \rho \times N_q.degree \rfloor$ // the threshold for the number of missing neighbors
3: $countSize = \lfloor \log_2(nb_{miss}) \rfloor + 1$
4: **for** i from 0 to $countSize$ **do**
5: $Count[i] = (0, 0, ..., 0)$ // $Count[i]$ is a bit vector of size n
6: **end for**
7: **for** j from 0 to $S_{bit} - 1$ **do**
8: **if** $N_q.nbArray[j] = 1$ **then**
9: $Carries =$ NOT $Bitmap.B_j$
10: **for** k from 0 to $countSize - 1$ **do**
11: $Temp = Count[k]$ AND $Carries$
12: $Count[k] = Count[k]$ XOR $Carries$
13: $Carries = Temp$
14: **end for**
15: $Count[countSize] = Count[countSize]$ OR $Carries$
16: **end if**
17: **end for**
18: // [Step 2]: only return nodes with no more than nb_{miss} missing neighbors
19: $Result_{lt} = (0, 0, ..., 0)$ // $Result_{lt}$ is a bit vector of size n
20: $Result_{eq} = (1, 1, ..., 1)$ // $Result_{eq}$ is a bit vector of size n
21: **for** k from $countSize$ to 0 **do**
22: **if** bit k of nb_{miss}'s binary format is 1 **then**
23: $Result_{lt} = Result_{lt}$ OR ($Result_{eq}$ AND (NOT $Count[k]$))
24: $Result_{eq} = Result_{eq}$ AND $Count[k]$
25: **else**
26: $Result_{eq} = Result_{eq}$ AND (NOT $Count[k]$)
27: **end if**
28: **end for**
29: $Result_{le} = Result_{lt}$ OR $Result_{eq}$
30: **return** $Result_{le}$

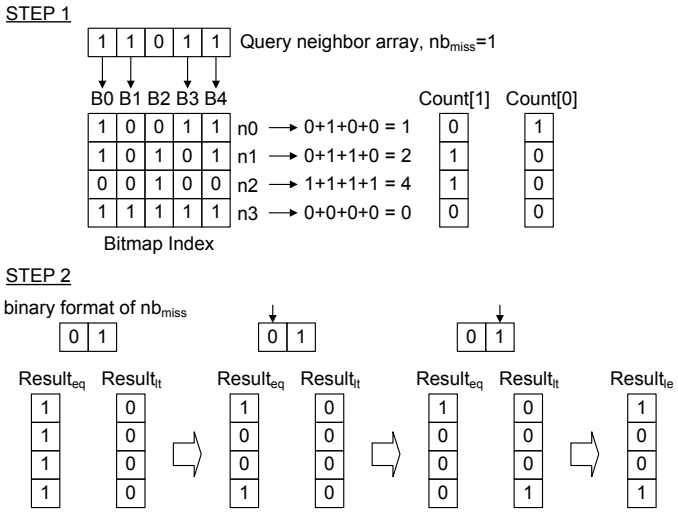

Figure 4.3: Example demonstrating Algorithm 1

If a position in the query neighbor array is set to 1, but the corresponding position in a database neighbor array is 0, we count it as one miss. Step 1 of Algorithm 1 simulates the binary addition operation to count the total number of misses. We keep a counter of $countSize + 1$ bits ($countSize = \lfloor \log_2(nb_{miss}) \rfloor + 1$) for each database node to record the number of misses. These counters are stored in the $countSize + 1$ bit vectors $Count[0]$ to $Count[countSize]$, i.e. vector $Count[0]$ stores the bit position 0 for all the counters, and so on. The algorithm scans through the query neighbor array from the lowest bit (position 0) to the highest bit (position $S_{bit} - 1$). If the current bit is 1, then the algorithm negates the bits in the corresponding column of the bitmap index and adds all the bit values to the counters of the database nodes. To avoid overflow, the highest bit $Count[countSize]$ for a database node is set to 1 when the number of misses exceeds $countSize$ bits. An example of the first step is shown in Step 1 of Figure 4.3.

The second step of Algorithm 1 prunes all the database nodes with more than

nb_{miss} misses. We use two bit vectors $Result_{eq}$ and $Result_{lt}$ to record the nodes with nb_{miss} misses and less than nb_{miss} misses, respectively. As the algorithm scans the binary format of nb_{miss} from the highest bit (position $countSize$) to the lowest bit (position 0), it updates $Result_{eq}$ and $Result_{lt}$. Finally, the bitwise OR of the two vectors gives us the right answer. Each position in the result vector indicates whether the corresponding database node is in the query result or not. Figure 4.3 also shows an example of this step.

Next, we analyze the complexity of Algorithm 1. This algorithm takes $O(S_{bit} \times \log(\rho \times d))$ bitwise operations in step 1, where d is the degree of the query node. And step 2 takes $O(S_{bit})$ bitwise operations. Therefore, the complexity of Algorithm 1 is $O(S_{bit} \times \log(\rho \times d))$ bitwise operations on bit vectors. Usually, $\rho \times d$ is very small value, thus $\log(\rho \times d)$ is even smaller, and often negligible.

We have also compared Algorithm 1 with a naive bitmap index probing method, which scans through every neighbor array in the bitmap index, and decides whether the neighbor array satisfies the condition specified in Equation IV.3. We set up a simulation to test the efficiency of Algorithm 1 against this naive method. We randomly generated 12 bitmap indexes with increasing sizes. The smallest bitmap index contains neighbor arrays for 16 nodes, while the largest one contains neighbor arrays for 32768 nodes. Each neighbor array in the bitmap has 32 bits. We use 50 randomly generated query neighbor arrays to probe these bitmap indexes. Algorithm 1 shows significant performance advantage over the naive method – the speedup ranges from 2X for the smallest index to more than 12X for the largest index.

4.3.5 Extensions to the Basic Approach

Next we introduce several extensions to the basic indexing technique to improve the basic approach and handle more general cases.

4.3.5.1 Node Mismatches

In the above indexing method, TALE requires two matching nodes to have the same label. However, real applications often need to allow matchings between nodes with different labels. We adopt the SAGA node mismatch model discussed in Section 3.2.2.2 of Chapter III, which implicitly groups nodes based on a specific notion of similarity. In this model, the grouping of nodes is defined based on the application domain, and two nodes are allowed to match only if they belong to the same group. For example, if a node represents a gene, then its group membership is defined by the orthologous group that it belongs to (orthologous groups are organized based on similar gene functionalities), and two nodes match if they belong to the same orthologous group. To accommodate this model, we extend the basic indexing approach by replacing the node labels with their corresponding group labels and hashing the group labels for the bit arrays. The remaining indexing method remains unchanged. In Section 4.5, we show how TALE can be applied to real applications using this node mismatch model.

4.3.5.2 Directed Graphs

The above indexing method works for undirected graphs. However, it is fairly easy to extend it to handle directed graphs. In a directed graph, every edge has

direction. Given a node, an adjacent edge either goes towards the node or away from the node. Therefore, the indexing unit becomes *(label, in-degree, out-degree, in-nbConnection, out-nbConnection, in-nbArray, out-nbArray)*. For the index structure, we can build one B+-Tree index on label, in-degree, out-degree, in-nbConnection and out-nbConnection. And each leaf entry in the B+-Tree points to one bitmap index for the in-nbArray and another bitmap index for the out-nbArray. Other candidate index structures are also possible.

4.3.5.3 Edge Labels

A simple extension can be made to the basic indexing method proposed above to handle graphs with labeled edges. In the basic method, we hash the labels of neighbors to get the neighbor array. To handle labeled edges, we hash (node label, edge label) pairs to produce the neighbor arrays. The remaining index method is unchanged.

Any of the above extensions can be combined together to meet the requirement of different applications.

4.4 The Matching Algorithm

In this section, we introduce the approximate subgraph matching algorithm. We first start with an overview of this algorithm in Section 4.4.1, and then describe the algorithm in detail in Sections 4.4.2 and 4.4.3.

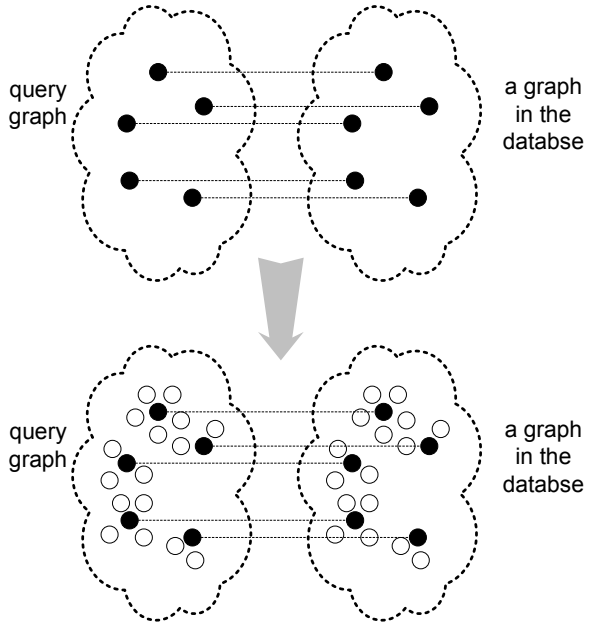

Figure 4.4: Overview of the matching algorithm

4.4.1 Algorithm Overview

Our approximate subgraph matching algorithm is based on the following two observations.

Observation 1: Some nodes in a graph play more importance roles in the graph structure than others. As shown in Figure 4.1, some nodes (e.g. the black node) connect to many other nodes. If these nodes are absent, then the graph structure quickly gets fragmented. In contrast, some nodes (e.g. the gray node) sit on the periphery of the graph and only connect to few other nodes. The overall graph structure will not be dramatically affected by removing these nodes. There are various ways of measuring the importance of a node in a graph. For simplicity, we use the degree centrality measure in this work. In this measure, nodes with high degrees are considered more important than nodes with low degrees. In Section 4.5.5, we

will evaluate the effectiveness of this importance measure. Note that the definition of "importance" is flexible in TALE and customizable for specific application needs. TALE can be easily extended to use other measures of node importance, such as closeness, betweenness, and eigenvector centralities.

Algorithm 2 GrowMatch (G_q, G_{db}, M_{imp})

Input: G_q is the query graph, G_{db} is the database graph, M_{imp} contains the matches for the important nodes in G_q
Output: M contains the node matches for the resulting graph match
1: put all node matches from M_{imp} to a priority queue Q sorted by their qualities
2: **while** Q is not empty **do**
3: pop up the best node match (N_q, N_{db}) from Q
4: put (N_q, N_{db}) into M
5: **ExamineNodesNearBy**(G_q, G_{db}, N_q, N_{db}, M, Q) // finding new matches for nodes nearby N_q
6: **end while**
7: **return** M

Observation 2: A good approximate match should be more tolerant towards missing unimportant nodes in the query than missing important nodes. In other words, most of the important nodes in the query should be present in the match, while missing unimportant nodes is more tolerated. In addition, the number of matched important nodes, and the qualities of these node matches can be used to estimate the quality of an approximate subgraph match.

Based on these two observations, we introduce a new approximate subgraph matching algorithm. The overview of this algorithm is as follows: First, the algorithm selects a number of important nodes from the query based on the specified importance measure (degree centrality in this work), and then probes the NH-Index to find matching nodes for these important query nodes. These matching node pairs

Algorithm 3 ExamineNodesNearBy (G_q, G_{db}, N_q, N_{db}, M_c, Q_c)

Input: G_q is the query graph, G_{db} is the database graph, N_q is a node in G_q, N_{db} is the node in G_{db} matched to N_q, M_c contains all the current node matches found so far, Q_c contains all the candidate node matches to be examined

1: $NB1_q$ = immediate neighbors of N_q that have no matches in M_c
2: $NB2_q$ = nodes two hops away from N_q that have no matches in M_c
3: $NB1_{db}$ = immediate neighbors of N_{db} that have no matches in either M_c or Q_c
4: $NB2_{db}$ = nodes two hops away from N_{db} that have no matches in either M_c or Q_c
5: **MatchNodes**(G_q, G_{db}, $NB1_q$, $NB1_{db}$, M_c, Q_c)
6: **MatchNodes**(G_q, G_{db}, $NB1_q$, $NB2_{db}$, M_c, Q_c)
7: **MatchNodes**(G_q, G_{db}, $NB2_q$, $NB1_{db}$, M_c, Q_c)

serve as anchor points for producing graph matches. In the second step, for each matching database graph, the algorithm extends the graph match from the anchor points by progressively adding satisfiable nearby nodes of already matched nodes. The entire matching process is outlined in Figure 4.4.

4.4.2 Step 1: Match the Important Nodes

In this step, the algorithm selects a number of important nodes from the query and probes the NH-Index to match these important nodes.

The algorithm first needs to decide how many nodes count as important nodes. We introduce a parameter P_{imp}, defined as the fraction of important nodes in the query. Given P_{imp}, we sort the nodes in the query by their importance (degree centrality in this work) and select the top P_{imp} percent as the important nodes. In Section 4.5.2, we show how to choose the P_{imp} value based on graph properties of specific applications.

After selecting the important nodes, the algorithm probes the NH-Index for each important node as discussed in Section 4.3.4. After the index probe, we obtain a

list of database graphs that have matches for some or all of the important nodes in the query. A match score is also calculated for each matching node pair using Equation IV.5. In the results produced by the index probes, a single important query node can be mapped to multiple database nodes and vice versa. Since the main purpose of this first step is to find the anchor points that can be expanded in the next step, we need to find one-to-one node mappings between the query and database nodes. For this part, we use a maximum weighted bipartite graph matching algorithm (using node match scores as weights) from the LEDA-R 3.2 library (`http://www.algorithmic-solutions.com/index.htm`).

4.4.3 Step 2: Extend the Match

Step 1 of the matching algorithm produces a list of candidate database graphs. For each candidate graph, Step 2 of the algorithm utilizes the node matches produced by Step 1 as anchor points to match the remaining nodes in the database and query graphs, and produces the final graph match.

The overall idea of this step is as follows. For each node that is already matched, we try to match its "nearby" nodes (as described below these includes not just the adjacent nodes, but also nodes that are two hops away). We perform this extension progressively until no more nodes can be added to the match. The detailed algorithm is shown in Algorithm 2, 3 and 4.

Algorithm 2 is the main procedure for step 2. It first puts all the important node matches (the anchor points) into a priority queue sorted by the qualities of the node

matches (cf. Section 4.3.2.1). In each iteration of the loop, we pop up the best node match (with the highest quality) from the queue and put it into the final graph match. In addition, we examine the nearby nodes of the query node, as well as the nearby nodes of the database node, to see whether any of them can be matched. If so, we add these new node matches to the priority queue. This process ends when the priority queue is empty.

Algorithm 3 implements the ExamineNodesNearBy function called by Algorithm 2. Based on a pair of already matched nodes, this function tries to match their nearby nodes. In order to allow more flexibility in the approximate matching, we do not limit the matching extensions to just adjacent nodes of the query node and the database node. Instead, this algorithm examines nodes at most two-hops away from the query node and the database node. Note that this algorithm is generic. It can be easily extended to match nodes more than two-hops away to allow more approximation (at the expense of an increased computational cost).

Algorithm 4 shows the details of the MatchNodes function called by Algorithm 3. For each node from the given set of query nodes, this algorithm finds the best matching node from the set of database nodes. If the new node match does not conflict with any existing ones in the priority queue, it is simply put into the priority queue. However, if this node match is better than an existing match in the queue, the existing one is replaced with the new one.

Note that our algorithm only produces one match for each database graph. In some applications, users may want more than one match for each database graph. In this case, we can extend our matching algorithm to retain more than one set of

anchor points (in step 1, instead of only retaining the maximum weighted bipartite matching, also retain other high weighted maximal bipartite matchings) and then extend each of them to produce a match.

Algorithm 4 MatchNodes(G_q, G_{db}, S_q, S_{db}, M_c, Q_c)

Input: G_q is the query graph, G_{db} is the database graph, S_q is a set of nodes in G_q, S_{db} is a set of nodes in G_{db}, M_c contains all the current matches found so far, Q_c contains all the candidate matches to be examined

```
1:  for every node N_q in S_q do
2:     N_db=the best mapping of N_q in S_db
3:     if N_db=null then
4:        continue
5:     end if
6:     if N_q has no matches in Q_c then
7:        put (N_q, N_db) into Q_c
8:        remove N_db from S_db
9:     else if (N_q, N_db) is a better node match then
10:       remove the existing match of N_q from Q_c
11:       put (N_q, N_db) into Q_c
12:       remove N_db from S_db
13:    end if
14: end for
```

4.5 Evaluation

In this section, we apply TALE to three real biological applications, and present results evaluating TALE with three measures: effectiveness (whether the results produced by the tool are useful and meaningful in real life applications), efficiency and scalability.

Note that while the applications discussed in this paper are from life sciences, TALE can be applied to any area in which there is a need for approximate subgraph matching. Other such areas include comparing RDF graphs in semantic web applica-

tions, and comparing parse trees produced by natural language parsers for literature mining. We have chosen to focus on life sciences applications since we have actual collaborators who have ready applications for our tool.

TALE is implemented in C++ on top of PostgreSQL (`http://www.postgresql.org`). The execution times reported in this section correspond to the running times of this C++ program including the DBMS access times. All experiments were run on a 2.8GHz Pentium 4 Fedora Core 2 machine, with 2GB memory, and a 250GB SATA disk. We use PostgreSQL version 8.1.3 and set the buffer pool size to 512MB.

4.5.1 Experimental Datasets

BIND Dataset: We use the BIND [10] dataset (version May 25, 2006) to demonstrate the application of TALE for comparing Protein Interaction Networks (PINs). A PIN is a large graph, in which nodes represent proteins and edges indicate protein-protein interactions. Comparing PINs of different species allows a biologist to discover the evolutionary conserved functional units across species. However, due to the high error rate of detection methods, PINs are noisy in nature [47]. Therefore, approximate subgraph matching is useful for comparing PINs.

KEGG Dataset: This dataset consists of biological pathways from the well-known KEGG database [34] (downloaded on Feb 28, 2007). We use this dataset to demonstrate the application of TALE for biological pathways analysis. A pathway is a directed graph with nodes representing cellular entities such as proteins and regulatory elements, and edges representing their interactions. The graph shows the sequence of

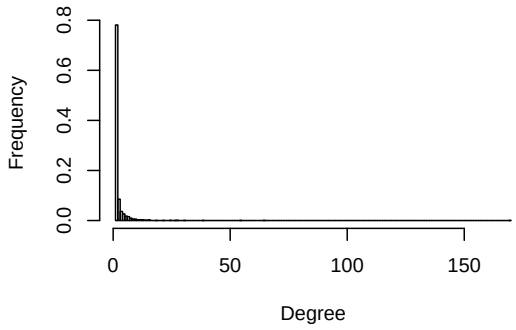

Figure 4.5: Degree distribution for the BIND dataset

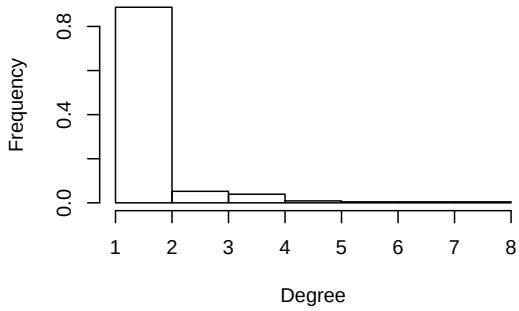

Figure 4.6: Degree distribution for the KEGG dataset

actions that lead to different cellular entities interacting to carry out some function. Similar to PINs, each interaction in a pathway is based on an experimentally observed phenomenon. Therefore, pathway data are often noisy and incomplete. Nature is very effective at modularizing complex actions and reusing subcomponents. There are a lot of common building blocks in the cellular machinery that often get "reused" in different pathways. By investigating the similarities shared by different pathways, a biologist can generate various hypotheses that can help refine the understanding of a

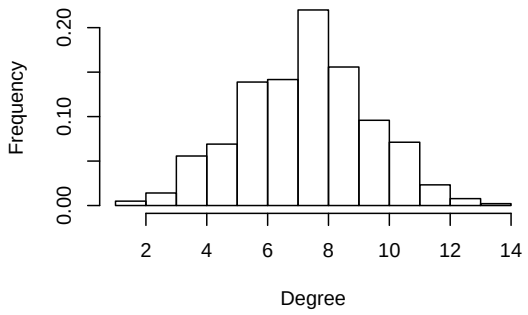

Figure 4.7: Degree distribution for the ASTRAL dataset

pathway of interest.

ASTRAL Dataset: To demonstrate the potential application of TALE for Protein Structure Matching, we use the ASTRAL [27] dataset (version 1.71). This dataset contains the 3D structures of protein domains. A domain is an independent, self-stabilizing unit of a protein, usually pertinent to the function of the protein they belong to. In biology, structure similarity is often a good indicator of function similarity. 3D structures can be translated into contact graphs, and structure matching can be achieved by approximate subgraph matching on the corresponding contact graphs. In a contact graph, nodes represent amino acids (since there are 20 different kinds of amino acids, there are 20 distinct node labels) and edges indicate that the corresponding amino acids physically interact with each other. This physical interaction is usually decided by a threshold of the contact distance. In our experiment, we used the widely used $7\mathring{A}$ threshold [25] to convert each domain 3D structure into a contact graph.

4.5.2 Parameterizations

In this section, we demonstrate how to choose the values of the parameters used in TALE for the three experimental datasets. TALE requires the setting of the following three parameters: the neighbor array size S_{bit} in the NH-Index, the approximation ratio ρ, and the fraction of important nodes P_{imp} in a query graph.

The size of the neighbor bit array is related to the number of node labels (or the number of group labels when allowing node mismatches, cf. Section 4.3.5.1) in an application. For protein structure matching, there are only 20 amino acids. Two nodes of the contact graphs can be matched only if they represent the same amino acid. Therefore, we set S_{bit} to be 32 (to make it fit in an 32-bit integer), and we use the exact value of the amino acids to set the bit array (instead of using a hash function).

For pathway analysis and protein interaction networks comparison, we need to match nodes based on function or sequence similarity. For the KEGG dataset, we utilize the KEGG Orthologous group (which classifies proteins based on function similarity). Two nodes can be matched only if they belong to the same KEGG Orthologous group. For the BIND dataset, we used CD-HIT [33] to cluster the proteins based on their sequence similarity[1]. And two nodes can be matched only if they belong to the same cluster. There are totally 8814 KEGG Orthologous groups and 22311 CD-HIT clusters. We set S_{bit} to be 64 and 96 for KEGG and BIND datasets, respectively. In fact, we have experimented with other S_{bit} values, and there is no significant difference in performance for different S_{bit} values. Therefore,

[1]Proteins in each cluster share at least 40% sequence identity.

in the interest of space, we do not show the effect of different neighbor array sizes.

The approximation ratio ρ indicates the percentage of neighbors of a query node that can have no corresponding matchings in the neighborhood of a database node. It is related to the similarity requirement of a specific application. For simplicity, we set this parameter to be 25% for all three applications.

The fraction of important nodes in a query graph P_{imp} is highly associated with the graph properties in an application. As we use degree centrality to measure the importance of a node, we study the degree distributions of the 3 applications. The representative degree distributions of the 3 datasets are shown in Figures 4.5 through 4.7. The degree distribution for the KEGG and BIND datasets is highly right skewed. In fact, studies have shown that both pathways and PINs show power-law degree distribution [37]. Only very small fraction of nodes have high degrees. We set $P_{imp} = 15\%$ for the KEGG and BIND datasets. For the ASTRAL dataset, the degree distribution is bell shape. Around 25% of nodes have degree more than 8, which we consider as important nodes ($P_{imp} = 25\%$).

4.5.3 Effectiveness Evaluation

In this section, we present results examining the effectiveness of TALE. We also compare TALE to C-Tree [22], SAGA (Chapter III), and Graemlin [17].

4.5.3.1 Protein Interaction Networks Comparison

Graph matching techniques are used on PINs to find conserved components shared between the query network and each network in the database. The PIN for a well

	# nodes	# edges
human	8470	11260
mouse	2991	3347
rat	830	942

Table 4.1: PINs of human, mouse and rat

studied species is usually a large graph with hundreds to thousands of nodes and edges. C-Tree [22] is not applicable for comparing PINs as the implementation does not allow node mismatches (nodes with different labels to be matched), which is a requirement for this application. On the other hand, TALE handles node mismatches by utilizing the group labels produced by existing protein clustering tools (see Section 4.5.2). SAGA can be used for querying PINs, but querying such large graphs using SAGA is prohibitively expensive.

For comparing PINs, the tools most closely related to TALE are NetworkBlast [40], MaWISh [35] and Graemlin [17]. Since these tools largely deal with pairwise comparison, we only focus on pairwise PIN comparison in this experiment. In [17], the authors showed that Graemlin is better at identifying conserved functional modules than the other methods. Therefore, we only compare TALE with Graemlin.

We choose the PINs of three well studied mammals: human, mouse and rat for this experiment. The statistics for these three networks are described in Table 4.1.

We use both TALE and Graemlin (using code download from `http://graemlin.stanford.edu/`) to query the rat and the mouse PINs against the human PIN. We compare the two methods using the effectiveness measures: the number of KEGGs hit and the average KEGG coverage as proposed in [17]. The number of KEGGs hit is the number of pathways in the KEGG database [34] aligned between 2 species. A

	# KEGGs hit	KEGG coverage	time (sec)
rat vs. human			
Graemlin	0	NA	910.0
TALE	6	3.2%	0.3
mouse vs. human			
Graemlin	18	5.0%	16305.5
TALE	42	13.6%	0.8

Table 4.2: Effectiveness for comparing PINs

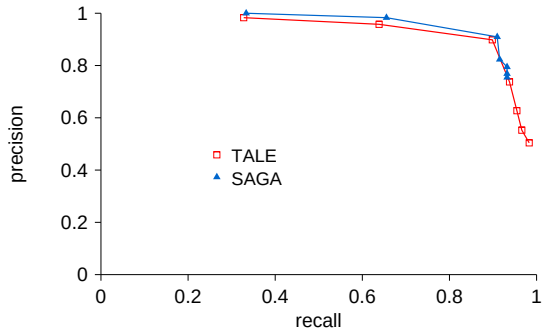

Figure 4.8: ROC curves for human pathways

KEGG pathway is considered as a hit if at least 3 proteins in the pathway are aligned to their counterparts in the pathway of the other species. KEGG coverage is the fraction of proteins aligned within a pathway.

As shown in Table 4.2, TALE achieves significant larger number of KEGGs hit and better average KEGG coverage than Graemlin. Most noticeable is the big difference in running time. TALE only takes about 1 second for the two queries while Graemlin takes 4.8 hours. In addition, TALE only takes about 1 second to build the index on the human PIN.

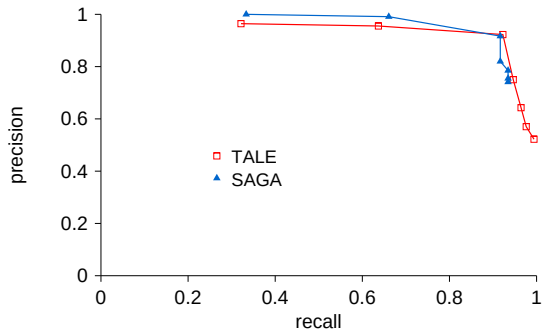

Figure 4.9: ROC curves for mouse pathways

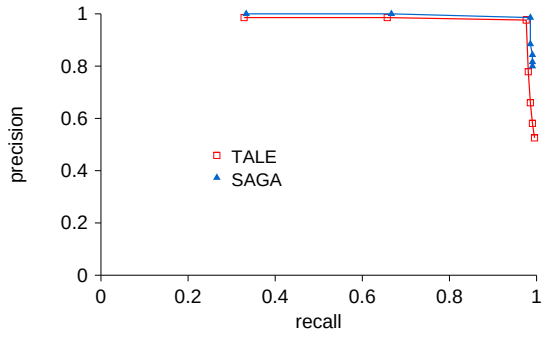

Figure 4.10: ROC curves for rat pathways

4.5.3.2 Biological Pathways Analysis

This experiment uses the KEGG pathway dataset. Again since we allow node mismatches in pathway analysis, C-Tree is not applicable. Therefore for this experiment, we only compare TALE with SAGA. SAGA was configured using the parameter settings in Section 3.3.1 of Chapter III.

Close related species in evolution (e.g. mouse and rat) share significant similarities in their corresponding pathways. For example, the WNT pathways of mouse and rat are very similar to each other, but are more different from the WNT pathway of fly.

This provides us a way to evaluate the effectiveness of TALE for the application of biological pathways analysis.

For this experiment, we choose the pathways of the 7 well-studied model species in KEGG: human, mouse, rat, fly, worm, yeast and ecoli. The statistics of this dataset is summarized in Table 4.3. Human, mouse and rat are more closely related to each other in evolutionary than the other 4 species. Therefore, we expect the results produced by TALE can reflect the fact that the pathways for these 3 species will be more similar to each other.

We used every pathway for human, mouse and rat to query the database. Note that for some large queries (e.g. human fatty acid biosynthesis pathway with 163 nodes and 151 edges), SAGA could not finish in a reasonable amount of time (taking over 1 hour), while TALE can finish every query within 1.8 seconds. For the queries SAGA can finish within 1 hour, the average running time for SAGA is about 12 seconds, while the average time for TALE is 0.14 seconds.

To evaluate the effectiveness of the results, we employ the measures: recall and precision. Recall is defined as the fraction of the retrieved relevant results out of all

	#pathways	avg #nodes	avg #edges
human	173	83.3	38.5
mouse	169	83.8	38.3
rat	161	83.6	30.3
fly	103	97.3	12.7
worm	97	100.4	13.4
yeast	87	106.1	20.8
ecoli	95	102.1	25.1
total	885	91.2	28.3

Table 4.3: The statistics of KEGG pathways for the 7 well-studied model species

the relevant results. Precision is the fraction of the retrieved relevant results out of all the retrieved results. A matching result is considered relevant if it is the same pathway from a species close in evolution. For example, if the query is human WNT pathway, then a relevant result can be human, mouse or rat WNT pathway.

To keep this experiment manageable, we kill any SAGA query if it runs over 1 hour. For fair comparison, we only compare TALE with SAGA for the query results that SAGA can finish within the time limit. We employ the SAGA distance model (using the default parameters in Section 3.3.1 of Chapter III) to rank the results returned by both TALE and SAGA. We compute the average precision and recall values for human, mouse and rat pathway queries. The ROC curves are shown in Figure 4.8 to Figure 4.10. SAGA and TALE show very comparable effectiveness for pathway analysis, with SAGA having a slight advantage.

4.5.3.3 Protein Structure Matching

In this experiment, we evaluate the effectiveness of TALE for protein structure matching using the ASTRAL dataset.

This application generally does not require node mismatches, therefore we can compare TALE with C-Tree. However, the C-Tree implementation that we got from the authors is memory-based. In other words, the whole index needs to reside in memory for query processing. Naturally, as the database size increases, the index will soon grow out of memory. For example, C-Tree cannot build an index on the entire ASTRAL dataset (which has 75626 domains). In contrast, NH-Index is a disk-based index technique and is not limited by the memory size. As we will show

in Section 4.5.4.3, TALE can easily handle the entire ASTRAL dataset, and our disk-based index structure scales nicely with increasing database sizes. For a fair comparison, we employ the similarity model used by C-Tree [22] to rank the matching results.

ASTRAL contains 75626 domains, which are classified into 7275 families. Domains in each family present significant structural similarity. This provides us with a way of evaluating the effectiveness of TALE: large fraction of the top matching results are expected to belong to the same family of the query domain.

We test TALE and C-Tree on a subset of ASTRAL, so that C-Tree can hold the index in memory. The dataset is created as follows: We randomly choose 1300 families (with more than 10 domains in each family), and then randomly choose 10 domains from each family. The average number of nodes and edges for each graph are 186.6 and 734.2, respectively.

We randomly choose 20 queries (with 346.4 nodes and 971.6 edges per graph on average) from the 13000 domains. We gradually increase the number of results returned by TALE and C-Tree, and measure the mean recall and mean precision for both methods. The recall and precision ROC curves are shown in Figure 4.11. The precision for both methods stays very high until the recall reaches round 0.6. This is because both methods return relevant results as their top results. However, as the recall further increases, the precision drops more steeply. After the recall reaches around 0.8, returning more results will not improve the recall any more. This is because the classification system in ASTRAL is not purely based on structure similarity, but also on extensive domain knowledge. No method based on pure structural sim-

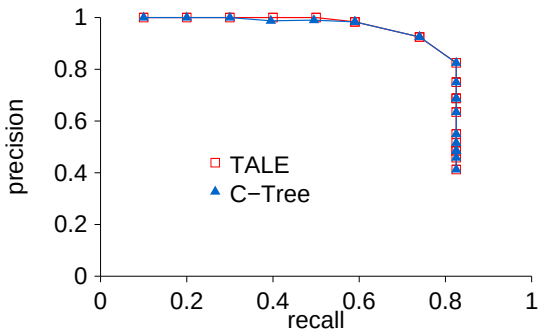

Figure 4.11: ROC curves using the ASTRAL dataset

ilarity is likely to perfectly match this classification system. However, TALE could potentially be used for classifying novel family members in combination with the domain knowledge provided by experts.

Although TALE and C-Tree are very comparable in their effectiveness for this dataset, TALE is faster than C-Tree. The average running time for the 20 queries is 34.8 seconds using TALE, but 61.9 seconds using C-Tree. TALE is almost 2 times faster than C-Tree (even though it is a disk-based implementation and is going through PostgreSQL).

	#graphs	avg #nodes	avg #edges	index size	index time
D1	10	939.1	1093.2	1.4MB	13.2s
D2	20	938.5	1691.9	2.9MB	31.1s
D3	30	939.5	1920.7	4.5MB	50.4s
D4	40	940.1	1743.6	5.7MB	62.7s

Table 4.4: Four BIND sub-datasets for the scalability experiment

4.5.4 Efficiency and Scalability Evaluation

In this experiment, we test the efficiency and scalability of TALE for the three applications.

4.5.4.1 Experiment on BIND Dataset

In this experiment, we evaluate the efficiency and scalability of TALE on the BIND dataset. BIND has PINs for 757 species, but most PINs are incomplete. We choose the largest 40 PINs from BIND. The largest graph contains 8470 nodes and 11260 edges. The smallest of these 40 PINs contains 45 nodes and 105 edges. On average, each graph has 940.1 nodes and 1743.6 edges. The characteristic of this data is that it contains large-sized graphs. To measure the scalability of TALE, we formed 4 datasets D1 to D4 with increasing sizes [2]. The statistics of the four datasets are summarized in Table 4.4. The index sizes and the index construction times are also shown in this table. As the database size increases, the index size grows at a near-linear rate and the index construction time increases steadily.

We choose the 10 graphs in dataset D1 as the queries. For this experiment, we do not restrict the number of results returned by each query. The execution time for the 10 queries on the 4 datasets is shown in Figure 4.12. Even for the largest query with 3059 nodes and 4850 edges on the largest D4 dataset, the query executes in about 0.7 seconds. The execution time grows as the size of the database increases. For most

[2]The 4 datasets are formed as follows. We first divide the 40 PINs into 4 balanced groups each with 10 PINs and roughly same total number of nodes. We randomly select one group as D1, randomly add another group to D1 to form D2, then randomly add one of the remaining groups to D2 to form D3, finally D4 contains all the 4 groups.

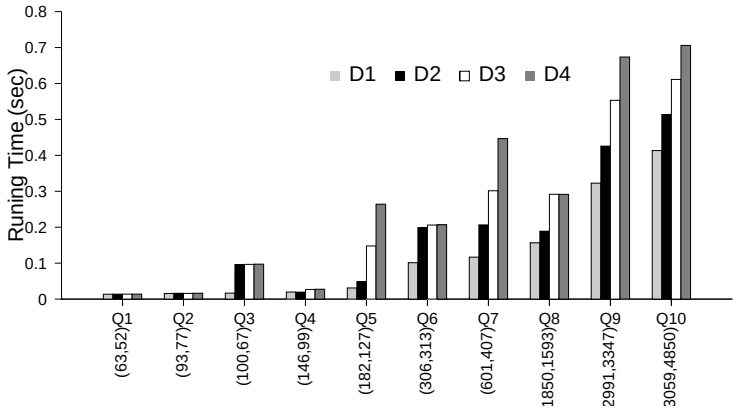

Figure 4.12: Scalability Experiment using the BIND dataset

queries, the growth ratio shows near-linear trend. Note that query execution time is not just influenced by the query and database sizes, but also by the result cardinality. In Figure 4.12, Q2, Q3 and Q4 increase in the query size, but the execution time increases from Q2 to Q3 while decreases from Q3 to Q4 for D2, D3 and D4 datasets. The reason is that Q3 has more database matches than Q2 and Q4. (Recall that in this experiment, we do not restrict the number of results returned by each query.) For Q3, there is a jump from D1 to D2, because more matching graphs are found in D2. But the number of matches remain roughly the same from D2 to D4 (and so does the execution time). Similar explanations apply to other queries in this figure.

4.5.4.2 Experiment on KEGG Dataset

In this section, we test the efficiency and scalability of TALE on increasing sizes of KEGG pathway databases. The smallest dataset contains all the human and mouse pathways. We increase the database size by including pathways of more species until

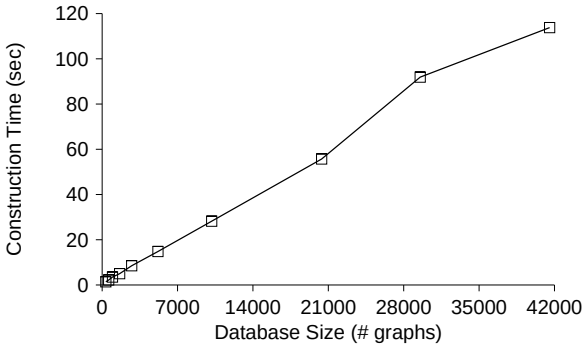

Figure 4.13: Index Construction Time with Increasing KEGG Database Size

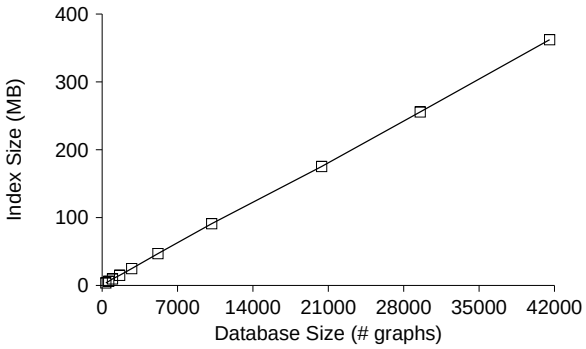

Figure 4.14: Index Size with Increasing KEGG Database Size

it contains all the 41550 KEGG pathways of 538 species. The index construction time and index sizes for these increasing databases are shown in Figure 4.13 and Figure 4.14, respectively. Our indexing technique indexes the neighborhood of each database node. This novel technique gives us the near linear increase in the index construction time and index size as shown in Figure 4.13 and Figure 4.14.

To test query execution time, we randomly selected 20 human pathways (76.7 nodes and 29.9 edges per graph on average). For each query, we ran TALE to get the top 20 results. The average running time for the 20 queries on increasing database

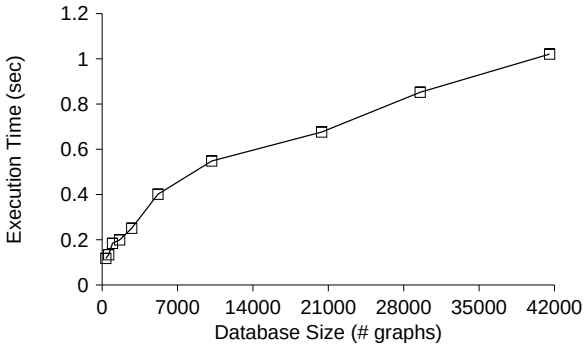

Figure 4.15: Query Execution Time with Increasing KEGG Database Size

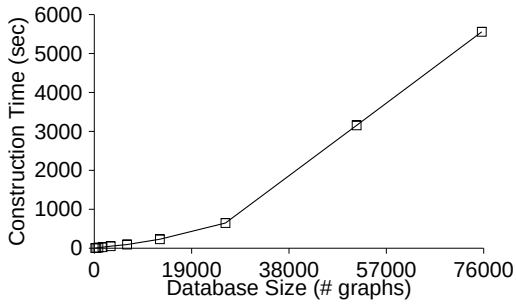

Figure 4.16: Index construction time for the ASTRAL dataset

sizes is reported in Figure 4.15. As shown in this figure, the query execution time increases steadily with the database size.

4.5.4.3 Experiment on ASTRAL Dataset

In this experiment, we evaluate the efficiency and scalability of TALE on the ASTRAL datasets with increasing sizes. The smallest dataset contains 200 graphs, while the largest one contains all the 75626 graphs in ASTRAL. As shown in Figure 4.16 and Figure 4.17, the index construction time and index size show steady growth with increasing database size.

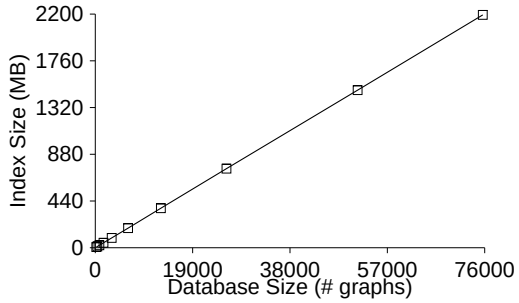

Figure 4.17: Index size for the ASTRAL dataset

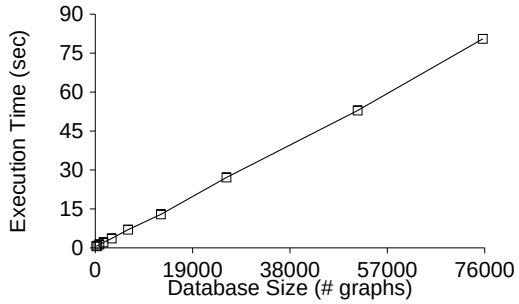

Figure 4.18: Query execution time for the ASTRAL dataset

We randomly selected 20 queries (153.1 nodes and 592.0 edges per graph on average) from the smallest dataset, and ran it on the increasing sized databases. For each query, we only retain the top 20 results. The average execution time for the 20 queries is shown in Figure 4.18. The running time scales nicely with the database size.

4.5.5 Discussion and Summary

We note that TALE is a heuristic algorithm. It does not guarantee that it will find the best or all matches. However, given that finding the best/all matches is

NP-hard [9] and infeasible in practice, heuristics are inevitable. For most real graphs, our heuristics achieve high accuracy compared with existing tools, as shown in our experiments.

In this work, we have used degree centrality to measure the importance of nodes. To show the effectiveness of this measure, we compare TALE to a variant called TALE-Random, where the "important" nodes are simply a randomly selected subset of the nodes. We ran the BIND mouse vs human test (Table 4.2, Row 3) using TALE-Random. We compare the number of matching nodes, the number of matching edges, the number of KEGGs hit and the average KEGG coverage for the two methods. The results are 106, 61, 42, 13.6% for TALE and 85, 24, 8, 5.8% for TALE-Random. This test shows the effectiveness of this node importance measure for this application.

To summarize the experimental section, our extensive empirical evaluation demonstrates the effectiveness, efficiency and scalability of TALE. We have compared TALE to three existing tools, SAGA, C-Tree and Graemlin. TALE is a flexible tool and the only tool that can easily be applied across the three applications considered in our evaluation. Furthermore, TALE produces useful and meaningful results for all the three applications, and is also significantly faster than these existing tools. Our results also show that TALE is scalable for large queries and large databases.

4.6 Conclusions

In this chapter we have presented TALE – an approximate subgraph matching tool for matching graph queries with a large number of nodes and edges. TALE

employs a novel indexing technique, which achieves a high pruning power and scales linearly with the database size. This index structure can be easily implemented in existing relational systems. The innovative matching algorithm used by TALE distinguishes nodes by their importance to the graph structure. This algorithm first matches the important nodes in the query, and then extends them to produce larger graph matches. TALE is a general tool for approximate subgraph matching queries, and can be easily customized to meet the requirement of different applications. Our empirical evaluations demonstrate the improved effectiveness and efficiency of TALE over existing methods.

CHAPTER V

Aggregation for Graph Summarization

5.1 Introduction

Besides graph matching methods, graph summarization techniques are very useful for understanding underlying characteristics of graphs. In many applications, graphs are very large, with thousands or even millions of nodes and edges. As a result, it is almost impossible to understand the information encoded in large graphs by mere visual inspection. Therefore, effective graph summarization methods are required to help users extract and understand the underlying information.

Most existing graph summarization methods use simple statistics to describe graph characteristics [11, 12, 37]; for example, researchers plot degree distributions to investigate the scale-free property of graphs, employ hop-plots to study the small world effect, and utilize clustering coefficients to measure the "clumpiness" of large graphs. While these methods are useful, the summaries contain limited information and can be difficult to interpret and manipulate. Methods that mine graphs for frequent pat-

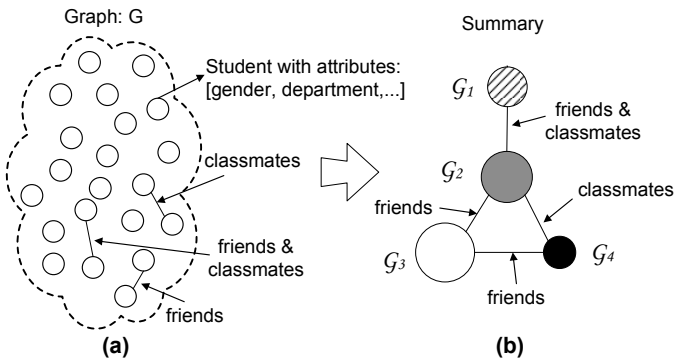

Figure 5.1: Graph summarization by aggregation

terns [26, 52, 53, 57] are also employed to understand the characteristics of large graphs. However, these algorithms often produce a large number of results that can easily overwhelm the user. Graph partitioning algorithms [38, 48, 56] have been used to detect community structures (dense subgraphs) in large networks. However, the community detection is based purely on nodes connectivities, and the attributes of nodes are largely ignored. Graph drawing techniques [3, 23] can help one better visualize graphs, but visualizing large graphs quickly becomes overwhelming.

What users need is a more controlled and intuitive method for summarizing graphs. The summarization method should allow users to freely choose the attributes and relationships that are of interest, and then make use of these features to produce small and informative summaries. Furthermore, users should be able to control the resolution of the resulting summaries and "drill-down" or "roll-up" the information, just like the OLAP-style aggregation methods in a traditional database systems.

In this chapter, we propose two operations for graph summarization that fulfills these requirements. The first operation, called *SNAP* (**S**ummarization by Grouping **N**odes on **A**ttributes and **P**airwise Relationships), produces a summary graph

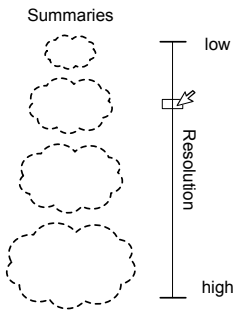

Figure 5.2: Illustration of multi-resolution summaries

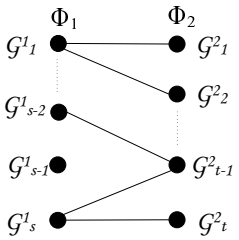

Figure 5.3: Construction of Φ_3 in the proof of Theorem 5.2.4

of the input graph by grouping nodes based on user-selected node attributes and relationships. Figure 5.1 illustrates the *SNAP* operation. Figure 5.1(a) is a graph about students (with attributes: gender, department and so on) and the relationships (classmates and friends) between them. Note that only few of the edges are shown in Figure 5.1(a). Based on user-selected gender and department attributes, and classmates and friends relationships, the *SNAP* operation produces a summary graph shown in Figure 5.1(b). This summary contains four groups of students and the relationships between these groups. Students in each group have the same gender and are in the same department, and they relate to students belonging to the same set of groups with friends and classmates relationships. For example, in Figure 5.1(b), each student in group $\mathcal{G}_1$ has at least a friend and a classmate in group $\mathcal{G}_2$. This

compact summary reveals the underlying characteristics about the nodes and their relationships in the original graph.

The second operation, called *k-SNAP*, further allows users to control the resolutions of summaries. This operation is pictorially depicted in Figure 5.2. Here using the slider, a user can "drill-down" to a larger summary with more details or "roll-up" to a smaller summary with less details.

Our summarization methods have been applied to analyze real social networking applications. In one example, by summarizing the coauthorship graphs in database and AI communities, different coauthorship patterns across the two areas are displayed. In another application, interesting linking behaviors among liberal and conservative blogs are discovered by summarizing a large political blogs network.

The main contributions of this chapter are:

(1) We introduce two database-style graph aggregation operations *SNAP* and *k-SNAP* for summarizing large graphs. We formally define the two operations, and prove that the *k-SNAP* computation is NP-complete.

(2) We propose an efficient algorithm to evaluate the *SNAP* operation, and also propose two heuristic methods (the top-down approach and the bottom-up approach) to approximately evaluate the *k-SNAP* operation.

(3) We apply our graph summarization methods to a variety of real and synthetic datasets. Through extensive experimental evaluation, we demonstrate that our methods produce meaningful summaries. We also show that the top-down approach is the ideal choice for *k-SNAP* evaluation in practice. In addition, the evaluation algorithms are very efficient even for very large graph datasets.

The remainder of this chapter is organized as follows: Section 5.2 defines the *SNAP* and the *k-SNAP* operations. Section 5.3 introduces the evaluation algorithms for these operations. Experimental results are presented in Section 5.4. Section 5.5 contains our concluding remarks.

5.2 Graph Aggregation Operations

In a graph, objects are represented by nodes, and relationships between objects are modeled as edges. In this chapter, we support a general graph model, where objects (nodes) have associated attributes and different types of relationships (edges). Formally, we denote a graph G as (V, Υ) where V is the set of nodes, and $\Upsilon = \{E_1, E_2, ..., E_r\}$ is the set of edge types, with each $E_i \subseteq V \times V$ representing the set of edges of a particular type.

Nodes in a graph have a set of associated attributes, which is denoted as $\Lambda = \{a_1, a_2, ..., a_t\}$. Each node has a value for each attribute. These attributes are used to describe the features of the objects that the nodes represent. For example, in Figure 5.1(a), a node representing a student may have attributes that represent the student's gender and department. Different types of edges in a graph correspond to different types of relationships between nodes, such as friends and classmates relationships shown in Figure 5.1(a). Note that two nodes can be connected by different types of edges. For example, in Figure 5.1(a), two students can be classmates and friends at the same time.

For ease of presentation, we denote the set of nodes of graph G as $V(G)$, the set

of attributes as $\Lambda(G)$, the actual value of attribute a_i for node v as $a_i(v)$, the set of edge types as $\Upsilon(G)$, and the set of edges of type E_i as $E_i(G)$. In addition, we denote the cardinality of a set S as $|S|$.

Our methods are applicable for both directed and undirected graphs. For ease of presentation, we only consider undirected graphs in this chapter. Adaptations of our method for directed graphs are fairly straightforward, and omitted in the interest of space.

5.2.1 SNAP Operation

The *SNAP* operation produces a summary graph through a homogeneous grouping of the input graph's nodes, based on user-selected node attributes and relationships. We now formally define this operation.

To begin the formal definition of the *SNAP* operation, we first define the concept of node-grouping.

Definition 5.2.1 ***(Node-Grouping of a Graph)*** *For a graph G, $\Phi = \{\mathcal{G}_1, \mathcal{G}_2, ..., \mathcal{G}_k\}$ is called a node-grouping of G, if and only if:*

(1) $\forall \mathcal{G}_i \in \Phi, \mathcal{G}_i \subseteq V(G)$ *and* $\mathcal{G}_i \neq \emptyset$,

(2) $\bigcup_{\mathcal{G}_i \in \Phi} \mathcal{G}_i = V(G)$,

(3) for $\forall \mathcal{G}_i, \mathcal{G}_j \in \Phi$ *and* $(i \neq j)$, $\mathcal{G}_i \cap \mathcal{G}_j = \emptyset$.

Intuitively, a node-grouping partitions the nodes in a graph into non-overlapping subsets. Each subset $\mathcal{G}_i$ is called a group. When there is no ambiguity, we simply call a node-grouping a grouping. For a given grouping Φ of G, the group that node v

belongs to is denoted as $\Phi(v)$. We further define the size of a grouping as the number of groups it contains.

Now, we define a partial order relation $\preccurlyeq$ on the set of all groupings of a graph.

Definition 5.2.2 ***(Dominance Relation)*** *For a graph G, the grouping Φ dominates the grouping Φ', denoted as $\Phi' \preccurlyeq \Phi$, if and only if $\forall \mathcal{G}'_i \in \Phi'$, $\exists \mathcal{G}_j \in \Phi$ s.t. $\mathcal{G}'_i \subseteq \mathcal{G}_j$.*

It is easy to see that the dominance relation $\preccurlyeq$ is reflexive, anti-symmetric and transitive, hence it is a partial order relation. Next we define a special kind of grouping based on a set of user-selected attributes.

Definition 5.2.3 ***(Attributes Compatible Grouping)***

For a set of attributes $A \subseteq \Lambda(G)$, a grouping Φ is compatible with attributes A or simply A-compatible, if it satisfies the following: $\forall u, v \in V$, if $\Phi(u) = \Phi(v)$, then $\forall a_i \in A, a_i(u) = a_i(v)$.

If a grouping Φ is compatible with A, we simply denote it as Φ_A. In each group of a A-compatible grouping, every node has exactly the same values for the set of attributes A. Note that there could be more than one grouping compatible with A. In fact a trivial grouping in which each node is a group is always compatible with any set of attributes.

Next, we prove that amongst all the A-compatible groupings of a graph, there is a global maximum grouping with respect to the dominance relation $\preccurlyeq$.

Theorem 5.2.4 *In the set of all the A-compatible groupings of a graph G, denoted as S_A, $\exists \Phi_A \in S_A$, s.t. $\forall \Phi'_A \in S_A$, $\Phi'_A \preccurlyeq \Phi_A$.*

Proof. We prove by contradiction. Assume that there is no global maximum A-compatible grouping, but more than one maximal grouping. Then, for every two of such maximal groupings Φ_1 and Φ_2, we will construct a new A-compatible grouping Φ_3 such that $\Phi_1 \preccurlyeq \Phi_3$ and $\Phi_2 \preccurlyeq \Phi_3$, which contradicts the assumption that Φ_1 and Φ_2 are maximal A-compatible groupings.

Assume that $\Phi_1 = \{\mathcal{G}_1^1, \mathcal{G}_2^1, ..., \mathcal{G}_s^1\}$ and $\Phi_2 = \{\mathcal{G}_1^2, \mathcal{G}_2^2, ..., \mathcal{G}_t^2\}$. We construct a bipartite graph on $\Phi_1 \cup \Phi_2$ as shown in Figure 5.3. The nodes in the bipartite graph are the groups from Φ_1 and Φ_2. And there is an edge between $\mathcal{G}_i^1 \in \Phi_1$ and $\mathcal{G}_j^2 \in \Phi_2$ if and only if $\mathcal{G}_i^1 \cap \mathcal{G}_j^2 \neq \emptyset$. After constructing the bipartite graph, we decompose this graph into connected components C_1, C_2, ..., C_m. For each connected component C_k, we union the groups inside this component and get a group $\cup(C_k)$. Now, we can construct a new grouping $\Phi_3 = \{\cup(C_1), \cup(C_2), ..., \cup(C_m)\}$. It is easy to see that $\Phi_1 \preccurlyeq \Phi_3$ and $\Phi_2 \preccurlyeq \Phi_3$. Now we prove that Φ_3 is compatible with A. From the definition of A-compatible groupings, if $\mathcal{G}_i^1 \cap \mathcal{G}_j^2 \neq \emptyset$, nodes in $\mathcal{G}_i^1 \cup \mathcal{G}_j^2$ all have the same attributes values. Therefore, every node in $\cup(C_k)$ has the same attributes values. Now, we have constructed a new A-compatible grouping Φ_3 such that $\Phi_1 \preccurlyeq \Phi_3$ and $\Phi_2 \preccurlyeq \Phi_3$. This contradicts our assumption that Φ_1 and Φ_2 are two different maximal A-compatible groupings. Therefore, there is a global maximum A-compatible grouping.

We denote this global maximum A-compatible grouping as Φ_A^{max}. Φ_A^{max} is also the A-compatible grouping with the minimum cardinality. In fact, if we consider each node in a graph as a data record, then Φ_A^{max} is very much like the result of a group-by operation for these data records on the attributes A in the relational database systems.

The A-compatible groupings only account for the node attributes. However, nodes do not just have attributes, but also participate in pairwise relationships represented by the edges. Next, we consider relationships when grouping nodes.

For a grouping Φ, we denote the neighbor-groups of node v in E_i as $NeighborGroups_{\Phi,E_i}(v) = \{\Phi(u)|(u,v) \in E_i\}$.

Now we define groupings compatible with both node attributes and relationships.

Definition 5.2.5 ***(Attributes and Relationships Compatible Grouping)*** *For a set of attributes $A \subseteq \Lambda(G)$ and a set of relationship types $R \subseteq \Upsilon(G)$, a grouping Φ is compatible with attributes A and relationship types R or simply (A,R)-compatible, if it satisfies the following:*

(1) Φ is A-compatible,

(2) $\forall u, v \in V(G)$, if $\Phi(u) = \Phi(v)$, then $\forall E_i \in R$,

$NeighborGroups_{\Phi,E_i}(u) = NeighborGroups_{\Phi,E_i}(v)$.

If a grouping Φ is compatible with A and R, we also denote it as $\Phi_{(A,R)}$. In each group of an (A,R)-compatible grouping, all the nodes are homogeneous in terms of both attributes A and relationships in R. In other words, every node inside a group has exactly the same values for attributes A, and is adjacent to nodes in the same set of groups for all the relationships in R.

As an example, assume that the summary in Figure 5.1(b) is a grouping compatible with gender and department attributes, and classmates and friends relationships. Then, for example, every student (node) in group $\mathcal{G}_2$, has the same gender and department attributes values, and is a friend of some student(s) in $\mathcal{G}_3$, a classmate of

some student(s) in $\mathcal{G}_4$, and a friend to some student(s) as well as a classmate to some student(s) in $\mathcal{G}_1$.

Given a grouping $\Phi_{(A,R)}$, we can infer relationships between groups from the relationships between nodes in R. For each edge type $E_i \in R$, we define the corresponding group relationships as $E_i(G, \Phi_{(A,R)}) = \{(\mathcal{G}_i, \mathcal{G}_j) |\ \mathcal{G}_i, \mathcal{G}_i \in \Phi_{(A,R)}$ and $\exists u \in \mathcal{G}_i, v \in \mathcal{G}_j$ s.t. $(u, v) \in E_i\}$. In fact, by the definition of (A, R)-compatible groupings, if there is one node in a group adjacent to some node(s) in the other group, then every node in the first group is adjacent to some node(s) in the second.

Similarly to attributes compatible groupings, there could be more than one grouping compatible with the given attributes and relationships. The grouping in which each node forms a group is always compatible with any given attributes and relationships.

Next we prove that among all the (A, R)-compatible groupings there is a global maximum grouping with respect to the dominance relation $\preccurlyeq$.

Theorem 5.2.6 *In the set of all the (A, R)-compatible groupings of a graph G, denoted as $S_{(A,R)}$, $\exists \Phi_{(A,R)} \in S_{(A,R)}$, s.t. $\forall \Phi'_{(A,R)} \in S_{(A,R)}$, $\Phi'_{(A,R)} \preccurlyeq \Phi_{(A,R)}$.*

Proof. Again we prove by contradiction. Assume that there is no global maximum (A, R)-compatible grouping, but more than one maximal grouping. Then, for every two of such maximal groupings Φ_1 and Φ_2, we use the same construction method to construct Φ_3 as in the proof of Theorem 5.2.4. We already know that Φ_3 is A-compatible, $\Phi_1 \preccurlyeq \Phi_3$ and $\Phi_2 \preccurlyeq \Phi_3$. Using similar arguments as in Theorem 5.2.4, we can also prove that Φ_3 is compatible with R. This contradicts our assumption that

Φ_1 and Φ_2 are two different maximal (A, R)-compatible groupings.

From the construction of Φ_3, we know that if $\mathcal{G}_i^1 \cap \mathcal{G}_j^2 \neq \emptyset$, then the nodes in $\mathcal{G}_i^1 \cup \mathcal{G}_j^2$ belong to the same group in Φ_3. Next, we prove that every node in $\mathcal{G}_i^1 \cup \mathcal{G}_j^2$ is also adjacent to nodes in the same set of groups in Φ_3.

Again we prove by contradiction. Assume that there are two nodes $u, v \in \mathcal{G}_i^1 \cup \mathcal{G}_j^2$, u is adjacent to $\cup(C_k)$ in Φ_3 but v is not. First, if both $u, v \in \mathcal{G}_i^1$ or both $u, v \in \mathcal{G}_j^2$, then as both Φ_1 and Φ_2 are (A, R)-compatible groupings, and the construction of Φ_3 does not decompose any groups in Φ_1 or Φ_2, u, v should always be adjacent to the same set of groups in Φ_3. This contradicts our assumption. Second, the two nodes can come from different groupings. For simplicity, assume $u \in \mathcal{G}_i^1$ and $v \in \mathcal{G}_j^2$. As $\mathcal{G}_i^1 \cap \mathcal{G}_j^2 \neq \emptyset$, a node $w \in \mathcal{G}_i^1 \cap \mathcal{G}_j^2$ is adjacent to the same set of groups as u in Φ_1 and adjacent to the same set of groups as v in Φ_2. As a result, every group that u is adjacent to in Φ_1 should intersect with some group that v is adjacent to in Φ_2. Since u is adjacent to $\cup(C_k)$, then u must be adjacent to at least one group in Φ_1 that is later merged to $\cup(C_k)$. This group should also intersect with a group $\mathcal{G}_l^2$ in Φ_2 that v is adjacent to. Then, by the construction algorithm of Φ_3, $\mathcal{G}_l^2$ should belong to the connected component C_k, thus should be later merged in $\cup(C_k)$. As a result, v is also adjacent to $\cup(C_k)$ in Φ_3, which contradicts our assumption.

Now we know if $\mathcal{G}_i \cap \mathcal{G}_j \neq \emptyset$, nodes in $\mathcal{G}_i \cup \mathcal{G}_j$ are all adjacent to the same set of groups in Φ_3. In each C_k, $\forall \mathcal{G}_i \in C_k$, $\exists \mathcal{G}_j \in C_k$ such that $\mathcal{G}_i \cap \mathcal{G}_j \neq \emptyset$. As a result, every node in $\cup(C_k)$ is adjacent to the same set of groups in Φ_3.

We have constructed a new (A, R)-compatible grouping Φ_3 such that $\Phi_1 \preccurlyeq \Phi_3$ and $\Phi_2 \preccurlyeq \Phi_3$. This contradicts the fact that Φ_1 and Φ_2 are two different max-

imal (A, R)-compatible groupings. Therefore, there is a global maximum (A, R)-compatible grouping.

We denote the global maximum (A, R)-compatible grouping as $\Phi^{max}_{(A,R)}$. $\Phi^{max}_{(A,R)}$ is also the (A, R)-compatible grouping with the minimum cardinality. Due to its compactness, this maximum grouping is more useful than other (A, R)-compatible groupings.

Now, we define our first operation for graph summarization, namely *SNAP*.

Definition 5.2.7 *(**SNAP Operation**) The SNAP operation takes as input a graph G, a set of attributes $A \subseteq \Lambda(G)$, and a set of edge types $R \subseteq \Upsilon(G)$, and produces a summary graph G_{snap}, where $V(G_{snap}) = \Phi^{max}_{(A,R)}$, and $\Upsilon(G_{snap}) = \{E_i(G, \Phi^{max}_{(A,R)}) | E_i \in R\}$.*

Intuitively, the *SNAP* operation produces a summary graph of the input graph based on user-selected attributes and relationships. The nodes of this summary graph correspond to the groups in the maximum (A, R)-compatible grouping. And the edges of this summary graph are the group relationships inferred from the node relationships in R.

5.2.2 k-SNAP Operation

The *SNAP* operation produces a grouping in which nodes of each group are homogeneous with respect to user-selected attributes and relationships. Unfortunately, homogeneity is often too restrictive in practice, as most real life graph data is subject to noise and uncertainty; for example, some edges may be missing because of the

failure in the detection process, and some edges may be spurious because of errors. Applying the *SNAP* operation on noisy data can result in a large number of small groups, and, in the worst case, each node may end up an individual group. Such a large summary is not very useful in practice. A better alternative is to let users control the sizes of the results to get summaries with the resolutions that they can manage (as shown in Figure 5.2). Therefore, we introduce a second operation, called *k-SNAP*, which relaxes the homogeneity requirement for the relationships and allows users to control the sizes of the summaries.

The relaxation of the homogeneity requirement for the relationships is based on the following observation. For each pair of groups in the result of the *SNAP* operation, if there is a group relationship between the two, then every node in both groups participates in this group relationship. In other words, every node in one group relates to some node(s) in the other group. On the other hand, if there is no group relationship between two groups, then absolutely no relationship connects any nodes across the two groups. However, in reality, if most (not all) nodes in the two groups participate in the group relationship, it is often a good indication of a strong relationship between the two groups. Likewise, it is intuitive to mark two groups as being weakly related if only a tiny fraction of nodes are connected between these groups.

Based on these observations, we relax the homogeneity requirement for the relationships by not requiring that every node participates in a group relationship. But we still maintain the homogeneity requirement for the attributes, i.e. all the groupings should be compatible with the given attributes. Users control how many groups are

present in the summary by specifying the required number of groups, denoted as k. There are many different groupings of size k compatible with the attributes, thus we need to measure the qualities of the different groupings. We propose the Δ-measure to assess the quality of an A-compatible grouping by examining how different it is to a hypothetical (A, R)-compatible grouping.

We first define the set of nodes in group $\mathcal{G}_i$ that participate in a group relationship $(\mathcal{G}_i, \mathcal{G}_j)$ of type E_t as $P_{E_t,\mathcal{G}_j}(\mathcal{G}_i) = \{u | u \in \mathcal{G}_i \text{ and } \exists v \in \mathcal{G}_j \text{ s.t. } (u, v) \in E_t\}$. Then we define the participation ratio of the group relationship $(\mathcal{G}_i, \mathcal{G}_j)$ of type E_t as $p^t_{i,j} = \frac{|P_{E_t,\mathcal{G}_j}(\mathcal{G}_i)|+|P_{E_t,\mathcal{G}_i}(\mathcal{G}_j)|}{|\mathcal{G}_i|+|\mathcal{G}_j|}$. For a group relationship, if its participation ratio is greater than 50%, we call it a strong group relationship, otherwise, we call it a weak group relationship. Note that in an (A, R)-compatible grouping, the participation ratios are either 0% or 100%.

Given a graph G, a set of attributes A and a set of relationship types R, the Δ-measure of $\Phi_A = \{\mathcal{G}_1, \mathcal{G}_2, ..., \mathcal{G}_k\}$ is defined as follows:

$$\Delta(\Phi_A) = \sum_{\mathcal{G}_i, \mathcal{G}_j \in \Phi_A} \sum_{E_t \in R} (\delta_{E_t,\mathcal{G}_j}(\mathcal{G}_i) + \delta_{E_t,\mathcal{G}_i}(\mathcal{G}_j)) \tag{V.1}$$

$$\delta_{E_t,\mathcal{G}_j}(\mathcal{G}_i) = \begin{cases} |P_{E_t,\mathcal{G}_j}(\mathcal{G}_i)| & \text{if } p^t_{i,j} \leq 0.5 \\ |\mathcal{G}_i| - |P_{E_t,\mathcal{G}_j}(\mathcal{G}_i)| & \text{otherwise} \end{cases} \tag{V.2}$$

Intuitively, the Δ-measure counts the minimum number of differences in participations of group relationships between the given A-compatible grouping and a

hypothetical (A, R)-compatible grouping of the same size. The measure looks at each pairwise group relationship: If this group relationship is weak ($p^t_{i,k} \leq 0.5$), then it counts the participation differences between this weak relationship and a non-relationship ($p^t_{i,k} = 0$); on the other hand, if the group relationship is strong, it counts the differences between this strong relationship and a 100% participation-ratio group relationship. The δ function, defined in Equation V.2, evaluates the part of the Δ value contributed by a group $\mathcal{G}_i$ with one of its neighbors $\mathcal{G}_j$ in a group relationship of type E_t.

It is easy to prove that $\Delta(\Phi_A) \geq 0$. The smaller $\Delta(\Phi_A)$ value is, the more closer Φ_A is to a hypothetical (A, R)-compatible grouping. $\Delta(\Phi_A) = 0$ if and only if Φ_A is (A, R)-compatible. We can also prove that $\Delta(\Phi_A)$ is bounded by $2|\Phi_A||V||R|$, as each $\delta_{E_t,\mathcal{G}_j}(\mathcal{G}_i) \leq |\mathcal{G}_i|$.

Now we will formally define the *k-SNAP* operation.

Definition 5.2.8 *(**k-SNAP Operation**) The k-SNAP operation takes as input a graph G, a set of attributes $A \subseteq \Lambda(G)$, a set of edge types $R \subseteq \Upsilon(G)$ and the desired number of groups k, and produces a summary graph $G_{k\text{-}snap}$, where $V(G_{k\text{-}snap}) = \Phi_A$, s.t. $|\Phi_A| = k$ and $\Phi_A = arg\min_{\Phi'_A}\{\Delta(\Phi'_A)\}$, and $\Upsilon(G_{k\text{-}snap}) = \{E_i(G, \Phi_A) \mid E_i \in R\}$.*

Given the desired number of groups k, the *k-SNAP* operation produces an A-compatible grouping with the minimum Δ value. Unfortunately, as we prove below, this optimization problem is NP-complete. To prove this, we first formally define the decision problem associated with this optimization problem and then prove it to be NP-complete.

Theorem 5.2.9 *Given a graph G, a set of attributes A, a set of relationship types R, a user-specified number of groups k ($\lceil|\Phi_A^{max}|\rceil \leq k \leq |V(G)|$), and a real number D ($0 \leq D < 2k|V||R|$), the problem of finding an A-compatible grouping Φ_A of size k with $\Delta(\Phi_A) \leq D$ is NP-complete.*

Proof. We use proof by restriction to prove the NP-completeness of this problem.

(1) This problem is in NP, because a nondeterministic algorithm only needs to guess an A-compatible grouping Φ_A of size k and check in polynomial time that $\Delta(\Phi_A) \leq D$. And an A-compatible grouping Φ_A of size k can be generated by a polynomial time algorithm.

(2) This problem contains a known NP-complete problem 2-Role Assignability (2RA) [42] as a special case. By restricting $A = \emptyset$, $|R| = 1$, $k = 2$ and $D = 0$, this problem becomes 2RA (which decides whether the nodes in a graph can be assigned with 2 roles, each node with one of the roles, such that if two nodes are assigned with the same role, then the sets of roles assigned to their neighbors are the same.) As proved in [42], 2RA is NP-complete.

Given the NP-completeness, it is infeasible to find the exact optimal answers for the *k-SNAP* operation. Therefore, we propose two heuristic algorithms to evaluate the *k-SNAP* operation approximately.

5.3 Evaluation Algorithms

In this section, we introduce the evaluation algorithms for *SNAP* and *k-SNAP*. It is computationally feasible to exactly evaluate the *SNAP* operation, hence the proposed

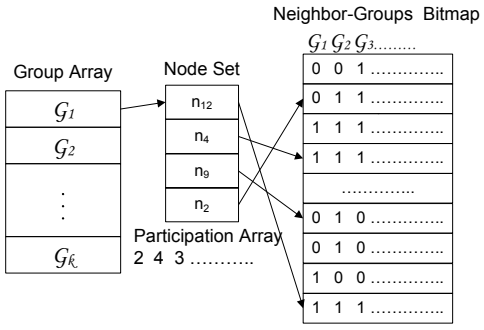

Figure 5.4: Data Structures Used in the Evaluation Algorithms

evaluation algorithm produces exact summaries. In contrast, *k-SNAP* computation was proved to be NP-complete and, therefore, we propose two *heuristic* algorithms for efficiently approximating the solution. Before discussing the details of the algorithms, we first introduce the evaluation architecture and the data structures used for the algorithms. Note that, for ease of presentation, all algorithms discussed in this section are assumed to work on one type of relationship; extending these algorithms for multiple relationship types is straightforward, hence is omitted in the interest of space.

5.3.1 Architecture and Data Structures

All the evaluation algorithms employ an architecture as follows. The input graphs reside in the database on disk. A chunk of memory is allocated as a buffer pool for the database. It is used to buffer actively used content from disk to speed up the accesses. Every access of the evaluation algorithms to the nodes and edges of graphs goes through the buffer pool. If the content is buffered, then the evaluation algorithms simply retrieve the content from the buffer pool; otherwise, the content is

read from disk into the buffer pool first. Another chunk of memory is allocated for the evaluation algorithms as the working memory, similar to the working memory space used by traditional database algorithms such as hash join. This working memory is used to hold the data structures used in the evaluation algorithms.

The evaluation algorithms share some common data structures as shown in Figure 5.4. The group-array data structure keeps track of the groups in the current grouping. Each entry in groups-array stores the id of a group and also points to a node-set, which contains the nodes in the corresponding group. Each node in the node-set points to one row of the neighbor-groups bitmap. This bitmap is the most memory consuming data structure in the evaluation algorithms. Each row of the bitmap corresponds to a node, and the bits in the row store the node's neighbor-groups. If bit position i is 1, then we know that this node has at least one neighbor belonging to group $\mathcal{G}_i$ with id i; otherwise, this node has no neighbor in group $\mathcal{G}_i$. For each group $\mathcal{G}_i$ in the current grouping, we also keep a participation-array which stores the participation counts $|P_{E,\mathcal{G}_j}(\mathcal{G}_i)|$ for each neighbor group. Note that the participation-array of a group can be inferred from the nodes' corresponding rows in the neighbor-groups bitmap. For example, in Figure 5.4, the participation-array of group $\mathcal{G}_1$ can be computed by counting the number of 1s in each column of the bitmap rows corresponding to n_{12}, n_4, n_9 and n_2. All the data structures shown in Figure 5.4 change dynamically during the evaluation algorithms. An increase in the number of groups leads to the growth of the group-array size, which also results in an increase of the width of the bitmap, as well as the sizes of the participation-arrays. The set of nodes for a group also change dynamically.

Algorithm 5 ***SNAP***$(G,\ A,\ R)$

Input: G: a graph; $A \subseteq \Lambda(G)$: a set of attributes; $R = \{E\} \subseteq \Upsilon(G)$: a set containing one relationship type E
Output: A summary graph.
1: Compute the maximum A-compatible grouping by sorting nodes in G based on values of attributes A
2: Initialize the data structures
3: **while** there is a group $\mathcal{G}_i$ with participation array containing values other than 0 or $|\mathcal{G}_i|$ **do**
4: Divide $\mathcal{G}_i$ into subgroups by sorting nodes in $\mathcal{G}_i$ based on their corresponding rows in the bitmap
5: Update the data structures
6: **end while**
7: Form the summary graph G_{snap}
8: **return** G_{snap}

For most of this chapter, we will assume that all the data structures needed by the evaluation algorithms can fit in the working memory. This is often a reasonable assumption in practice for a majority of graph datasets. However, we also consider the case when this memory assumption does not hold (see Section 5.4.4.3).

5.3.2 Evaluating SNAP Operation

In this section, we introduce the evaluation algorithm for the *SNAP* operation. This algorithm also serves as a foundation for the two *k-SNAP* evaluation algorithms.

The *SNAP* operation tries to find the maximum (A, R)-compatible grouping for a graph, a set of nodes attributes, and the specified relationship type. The evaluation algorithm starts from the maximum A-compatible grouping, and iteratively splits groups in the current grouping, until the grouping is also compatible with the relationships.

The algorithm for evaluating the $SNAP$ operation is shown in Algorithm 5. In the first step, the algorithm groups the nodes based only on the attributes by a sorting on the attributes values. Then the data structures are initialized by this maximum A-compatible grouping. Note that if a grouping is compatible with the relationships, then all nodes inside a group should have the same set of neighbor-groups, which means that they have the same values in their rows of the bitmap. In addition, the participation array of each group should then only contain the values 0 or the size of the group. This criterion has been used as the terminating condition to check whether the current grouping is compatible with the relationships in line 3 of Algorithm 5. If there exists a group whose participation array contains values other than 0 or the size of this group, the nodes in this group are not homogeneous in terms of the relationships. We can split this group into subgroups, each of which contains nodes with the same set of neighbor-groups. This can be achieved by sorting the nodes based on their corresponding entries in the bitmap. (The radix sort is a perfect candidate for this task.) After this division, new groups are introduced. One of them continues to use the same group id of the split group, and the remaining groups are added to the end of the group-array. Accordingly, each row of the bitmap has to be widened. The nodes of this split group are distributed among the new groups. As the group memberships of these nodes are changed, the bitmap entries for them and their neighbor nodes have to be updated. Then the algorithm goes to the next iteration. This process continues until the condition in line 3 does not hold anymore.

It can be easily verified that the grouping produced by Algorithm 5 is the maximum (A, R)-compatible grouping. The algorithm starts from the maximum A-

Algorithm 6 ***k-SNAP-Top-Down***$(G,\ A,\ R,\ k)$

Input: G: a graph; $A \subseteq \Lambda(G)$: a set of attributes; $R = \{E\} \subseteq \Upsilon(G)$: a set containing one relationship type E; k: the required number of groups in the summary
Output: A summary graph.
1: Compute the maximum A-compatible grouping by sorting nodes in G based on values of attributes A
2: Initialize the data structures and let Φ_c denote the current grouping
3: ***SplitGroups***$(G,\ A,\ R,\ k,\ \Phi_c)$
4: Form the summary graph $G_{k\text{-}snap}$
5: **return** $G_{k\text{-}snap}$

compatible grouping, and it only splits existing groups, so the grouping after each iteration is guaranteed to be A-compatible. In addition, each time we split a group, we always keep nodes with same neighbor-groups together. Therefore, when the algorithm stops, the grouping should be the maximum (A, R)-compatible grouping.

After we get the maximum (A, R)-compatible grouping, we can construct the summary graph. The nodes in the summary graph corresponds to the groups in the result grouping. The edges in the summary graph are the group relationships inferred from the node relationships in the original graph.

Now we will analyze the complexity of this evaluation algorithm. Sorting by the attributes values takes $O(|V| \log |V|)$ time, assuming the number of attributes is a small constant. The initialization of the data structures takes $O(|E|)$ time, where E is the only edge type in R (for simplicity, we only consider one edge type in our algorithms). At each iteration of the while loop, the radix sort takes $O(k_i|\mathcal{G}_i|)$ time, where k_i is the number of groups in the current grouping and $\mathcal{G}_i$ is the group to be split. Updating the data structures takes $|Edges(\mathcal{G}_i)|$, where $Edges(\mathcal{G}_i)$ is the set of edges adjacent to nodes in $\mathcal{G}_i$. Note that k_i is monotonically increasing, and that the

number of iterations is less than the size of the final grouping, denoted as k. Therefore, the complexity for all the iterations is bounded by $O(k^2|V|+k|E|)$. Constructing the summary takes $O(k^2)$ time. To sum up, the upper-bound complexity of the *SNAP* algorithm is $O(|V|\log|V|+k^2|V|+k|E|)$.

As the evaluation algorithm takes inputs from the graph database on disk, we also need to analyze the number of disk accesses to the database. We assume all the accesses to the database are in the units of pages. For simplicity, we do not distinguish whether an access is a disk page access or a buffer pool page access. We assume that all the nodes information of the input graph takes $\|V\|$ pages in the database, and all the edges information takes $\|E\|$ pages. Then the *SNAP* operation incurs $\|V\|$ page accesses to read all the nodes with their attributes, $\|E\|$ page accesses to initialize the data structures, and at most $\|E\|$ page accesses each time it updates the data structures. So, the total number of page accesses is bounded by $\|V\|+(k+1)\|E\|$. Note that in practice, not every page access results in an actual disk IO. Especially for the updates of the data structures discussed in Section 5.3.1, most of the edges information will be cached in the buffer pool.

5.3.3 Evaluating k-SNAP Operation

The *k-SNAP* operation allows a user to choose k, the number of groups that are shown in the summary. For a given graph, a set of nodes attributes A and the set of relationship types R, a meaningful k value should fall in the range between $|\Phi_A^{max}|$ and $|\Phi_{(A,R)}^{max}|$. However, if the user input is beyond the meaningful range, i.e.

Algorithm 7 ***SplitGroups***(G, A, R, k, Φ_c)

Input: G: a graph; $A \subseteq \Lambda(G)$: a set of attributes; $R = \{E\} \subseteq \Upsilon(G)$: a set containing one relationship type E; k: the required number of groups in the summary; Φ_c: the current grouping.
Output: Splitting groups in Φ_c until $|\Phi_c| = k$.
1: Build a heap on the CT value of each group in Φ_c
2: **while** $|\Phi_c| < k$ **do**
3: Pop the group $\mathcal{G}_i$ with the maximum CT value from the heap
4: Split $\mathcal{G}_i$ into two based on the neighbor group $\mathcal{G}_t = arg\max_{\mathcal{G}_j}\{\delta_{E,\mathcal{G}_j}(\mathcal{G}_i)\}$
5: Update data structures (Φ_c is updated)
6: Update the heap
7: **end while**

$k < |\Phi_A^{max}|$ or $k > |\Phi_{(A,R)}^{max}|$, then the evaluation algorithms will return the summary corresponding to Φ_A^{max} or $\Phi_{(A,R)}^{max}$, respectively. For simplicity, we will assume that the k values input to the algorithms are always meaningful. By varying the k values, users can produce multi-resolution summaries. A larger k value corresponds to a higher resolution summary. The finest summary corresponds to the grouping $\Phi_{(A,R)}^{max}$; and the coarsest summary corresponds to the grouping Φ_A^{max}.

As proved in Section 5.2.2, computing the exact answers for the *k-SNAP* operation is NP-complete. In this chapter, we propose two heuristic algorithms to approximate the answers. The top-down approach starts from the maximum grouping only based on attributes, and iteratively splits groups until the number of groups reaches k. The other approach employs a bottom-up scheme. This method first computes the maximum grouping compatible with both attributes and relationships, and then iteratively merges groups until the result satisfies the user defined k value. In both approaches, we apply the same principle: nodes of a same group in the maximum (A, R)-compatible grouping should always remain in a same group even in coarser

Algorithm 8 ***k-SNAP-Bottom-Up***$(G,\ A,\ R,\ k)$

Input: G: a graph; $A \subseteq \Lambda(G)$: a set of attributes; $R = \{E\} \subseteq \Upsilon(G)$: a set containing one relationship type E; k: the required number of groups in the summary.
Output: A summary graph.
1: G_{snap}=***SNAP***$(G,\ A,\ R)$
2: Initialize the data structures using the grouping in G_{snap} and let Φ_c denote the current grouping
3: ***MergeGroups***$(G,\ A,\ R,\ k,\ \Phi_c)$
4: Form the summary graph $G_{k\text{-}snap}$
5: **return** $G_{k\text{-}snap}$

summaries. We call this principle KEAT (**K**eep the **E**quivalent **A**lways **T**ogether) principle. This principle guarantees that when $k = |\Phi^{max}_{(A,R)}|$, the result produced by the *k-SNAP* evaluation algorithms is the same as the result of the *SNAP* operation with the same inputs.

5.3.3.1 Top-Down Approach

Similar to the *SNAP* evaluation algorithm, the top-down approach (see Algorithm 6) also starts from the maximum grouping based only on attributes, and then iteratively splits existing groups until the number of groups reaches k. However, in contrast to the *SNAP* evaluation algorithm, which randomly chooses a splittable group and splits it into subgroups based on its bitmap entries, the top-down approach has to make the following decisions at each iterative step: (1) which group to split and (2) how to split it. Such decisions are critical as once a group is split, the next step will operate on the new grouping. At each step, we can only make the decision based on the current grouping. We want each step to make the smallest move possible, to avoid going too far away from the right direction. Therefore, we split one group

into only two subgroups at each iterative step. There are different ways to split one group into two. One natural way is to divide the group based on whether nodes have relationships with nodes in a neighbor group. After the split, nodes in the two new groups either all or never participate in the group relationships with this neighbor group. This way of splitting groups also ensures that the resulting groups follow the KEAT principle.

Now, we introduce the heuristic for deciding which group to split and how to split at each iterative step. As defined in Section 5.2.2, the *k-SNAP* operation tries to find the grouping with a minimum Δ measure (see Equation V.1) for a given k. The computation of the Δ measure can be broken down into each group with each of its neighbors (see the δ function in Equation V.2). Therefore, our heuristic chooses the group that makes the most contribution to the Δ value with one of its neighbor groups. More formally, for each group $\mathcal{G}_i$, we define $CT(\mathcal{G}_i)$ as follows:

$$CT(\mathcal{G}_i) = \max_{\mathcal{G}_j}\{\delta_{E,\mathcal{G}_j}(\mathcal{G}_i)\} \tag{V.3}$$

Then, at each iterative step, we always choose the group with the maximum CT value to split and then split it based on whether nodes in this group $\mathcal{G}_i$ have relationships with nodes in its neighbor group $\mathcal{G}_t$, where

$$\mathcal{G}_t = arg\max_{\mathcal{G}_j}\{\delta_{E,\mathcal{G}_j}(\mathcal{G}_i)\}$$

As shown in Algorithm 7, to speed up the decision process, we build a heap on the

Algorithm 9 ***MergeGroups***$(G,\ A,\ R,\ k,\ \Phi_c)$

Input: G: a graph; $A \subseteq \Lambda(G)$: a set of attributes; $R = \{E\} \subseteq \Upsilon(G)$: a set containing one relationship type E; k: the required number of groups in the summary; Φ_c: the current grouping.
Output: Merging groups in Φ_c until $|\Phi_c| = k$.
1: Build a heap on ($MergeDist$, $Agree$, $MinSize$) for pairs of groups
2: **while** $|\Phi_c| > k$ **do**
3: Pop the pair of groups with the best ($MergeDist$, $Agree$, $MinSize$) value from the heap
4: Merge the two groups into one
5: Update data structures (Φ_c is updated)
6: Update the heap
7: **end while**

CT values of groups. At each iteration, we pop the group with the maximum CT value to split. At the end of each iteration, we update the heap elements corresponding to the neighbors of the split group, and insert elements corresponding to the two new groups.

The time complexity of the top-down approach is similar to the *SNAP* algorithm, except that it takes $O(k_0^2 + k_0)$ time to compute the CT values and build the heap, and at most $O(k_i^2 + k_i \log k_i)$ time to update the heap at each iteration, where k_0 is the number of groups in the maximum A-compatible grouping, and k_i is the number of groups at each iteration. As $k < |V|$, the upper-bound complexity of the top-down approach is still $O(|V| \log |V| + k^2|V| + k|E|)$.

Following the same method of analyzing the page accesses for the *SNAP* algorithm, the number of page accesses incurred by the top-down approach is bounded by $\|V\| + (k+1)\|E\|$.

5.3.3.2 Bottom-Up Approach

The bottom-up approach first computes the maximum (A, R)-compatible grouping using Algorithm 5, and then iteratively merges two groups until grouping size is k (see Algorithm 8). Choosing which two groups to merge in each iterative step is crucial for the bottom-up approach. First, the two groups are required to have the same attributes values. Second, the two groups must have similar group relationships with other groups. Now, we formally define this similarity between two groups.

The two groups to be merged should have similar neighbor groups with similar participation ratios. We define a measure called $MergeDist$ to assess the similarity between two groups in the merging process.

$$MergeDist(\mathcal{G}_i, \mathcal{G}_j) = \sum_{k \neq i,j} |p_{i,k} - p_{j,k}| \tag{V.4}$$

$MergeDist$ accumulates the differences in participation ratios between $\mathcal{G}_i$ and $\mathcal{G}_j$ with other groups. The smaller this value is, the more similar the two groups are.

If two pairs of groups have the same $MergeDist$, we need to further distinguish which pair is "more similar". We look at each common neighbor $\mathcal{G}_k$ of $\mathcal{G}_i$ and $\mathcal{G}_j$, and consider the group relationships $(\mathcal{G}_i, \mathcal{G}_k)$ and $(\mathcal{G}_j, \mathcal{G}_k)$. If both group relationships are strong ($p_{i,k} > 0.5$ and $p_{j,k} > 0.5$) or weak ($p_{i,k} \leq 0.5$ and $p_{j,k} \leq 0.5$), then we call it an *agreement* between $\mathcal{G}_i$ and $\mathcal{G}_j$. The total number of agreements between $\mathcal{G}_i$ and $\mathcal{G}_j$ is denoted as $Agree(\mathcal{G}_i, \mathcal{G}_j)$. Having the same $MergeDist$, the pair of groups with more agreements is a better candidate to merge.

If both of the above criteria are the same for two pairs of groups, we always prefer

merging groups with smaller sizes (in the number of nodes). More formally, we choose the pair with smaller $MinSize(\mathcal{G}_i, \mathcal{G}_j) = min\{|\mathcal{G}_i|, |\mathcal{G}_j|\}$, where $\mathcal{G}_i$ and $\mathcal{G}_j$ are in this pair.

In Algorithm 9, we utilize a heap to store pairs of groups based on the values of the triple ($MergeDist$, $Agree$, $MinSize$). At each iteration, we pop the group pair with the best ($MergeDist$, $Agree$, $MinSize$) value from the heap, and then merge the pair into one group. At the end of each iteration, we remove the heap elements (pairs of groups) involving either of the two merged groups, update elements involving neighbors of the merged groups, and insert elements involving this new group.

The time cost of the bottom-up approach is the cost of the $SNAP$ algorithm plus the merging cost. The algorithm takes $O(k_{snap}^3 + k_{snap}^2)$ to initialize the heap, then at each iteration at most $O(k_i^3 + k_i^2 \log k_i)$ time to update the heap, where k_{snap} is the size of the grouping resulting from the $SNAP$ operation, and k_i is the size of the grouping at each iteration. Therefore, the time complexity of the bottom-up approach is bounded by $O(|V| \log |V| + k_{snap}^2|V| + k_{snap}|E| + k_{snap}^4)$. Note that updating the in memory data structures in the bottom-up approach does not need to access the database (i.e. no IOs). All the necessary information for the updates can be found in the current data structures. Therefore, the upper bound of the number of page accesses for the bottom-up approach is $\|V\| + (k_{snap} + 1)\|E\|$.

5.3.3.3 Drill-Down and Roll-Up Abilities

The top-down and the bottom-up approaches introduced above both start from scratch to produce the summaries. However, it is easy to build an interactive querying

scheme, where the users can drill-down and roll-up based on the current summaries. The users can first generate an initial summary using either the top-down approach or the bottom-up approach. However, as we will show in Section 5.4.3, the top-down approach has significant advantage in both efficiency and summary quality in most practical cases. We suggest using the top-down approach to generate the initial summary. The drill-down operation can be simply achieved by calling the *SplitGroups* function (Algorithm 7). To roll up to a coarser summary, the *MergeGroups* function (Algorithm 9) can be called. However, when the number of groups in the current summary is large, the *MergeGroups* function becomes expensive, as it needs to compare every pair of groups to calculate the $MergeDist$ (see Section 5.3.3.2). Therefore, using the top-down approach to generate a new summary with the decreased resolution is a better choice to roll-up when the current summary is large.

5.4 Experimental Evaluation

In this section, we present experimental results evaluating the effectiveness and efficiency of the *SNAP* and the *k-SNAP* operations on a variety of real and synthetic datasets. All algorithms are implemented in C++ on top of PostgreSQL (`http://www.postgresql.org`) version 8.1.3. Graphs are stored in a node table and an edge table in the database, using the following schema: NodeTable(graphID, nodeID, attributeName, attributeType, attributeValue) and EdgeTable(graphID, node1ID, node2ID, edgeType). Nodes with multiple attributes have multiple entries in the node table, and edges with multiple types have multiple entries in the edge table. Accesses to

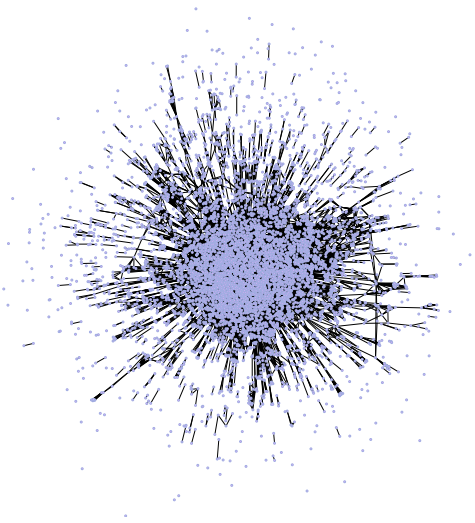

Figure 5.5: DBLP DB coauthorship graph

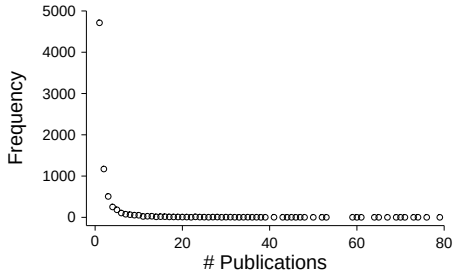

Figure 5.6: Distribution of the number of DB publications (avg: 2.6, stdev: 5.1)

nodes and edges of graphs are implemented by issuing SQL queries to the PostgreSQL database. All experiments were run on a 2.8GHz Pentium 4 machine running Fedora 2, and equipped with a 250GB SATA disk. For all experiments (except the one in Section 5.4.4.3), we set the buffer pool size to 512MB and working memory size to 256MB.

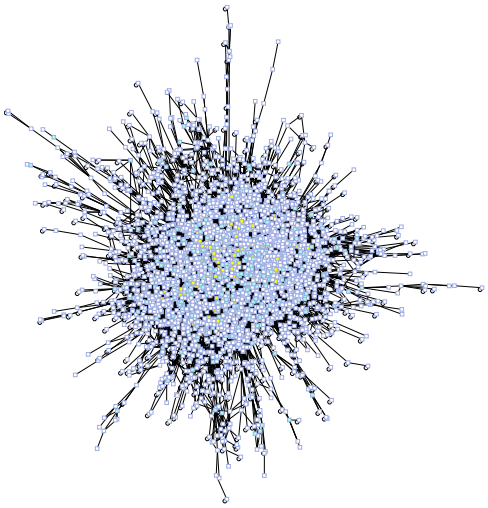

Figure 5.7: The *SNAP* result for DBLP DB dataset

	Description	#Nodes	#Edges	Avg. Degree
D1	*DB*	7,445	19,971	5.4
D2	D1+*AL*	14,533	37,386	5.1
D3	D2+*OS*+*CC*	22,435	55,007	4.9
D4	D3+*AI*	30,664	70,669	4.6

Table 5.1: The DBLP Datasets for the Efficiency Experiments

5.4.1 Experimental Datasets

In this section, we describe the datasets used in our empirical evaluation. We use two real datasets and one synthetic dataset to explore the effect of various graph characteristics.

DBLP Dataset This dataset contains the DBLP Bibliography data [32] downloaded on July 30, 2007. We use this data for both effectiveness and efficiency experiments. In order to compare the coauthorship behaviors across different research areas and construct datasets for the efficiency experiments, we partition the DBLP data into different research areas. We choose the following five areas: Database (DB), Al-

gorithms (AL), Operating Systems (OS), Compiler Construction (CC) and Artificial Intelligence (AI). For each of the five areas, we collect the publications of a number of selected journals and conferences in this area[1]. These journals and conferences are selected to construct the four datasets with increasing sizes for the efficiency experiments (see Table 5.1). These four datasets are constructed as follows: D1 contains the selected DB publications. We add into D1 the selected AL publications to form D2. D3 is D2 plus the selected OS and CC publications. And finally, D4 contains the publications of all the five areas we are interested in. We construct a coauthorship graph for each dataset. The nodes in this graph correspond to authors and edges indicate coauthorships between authors. The statistics for these four datasets are shown in Table 5.1.

Political Blogs Dataset This dataset is a network of 1490 webblogs on US politics and 19090 hyperlinks between these webblogs [1] (downloaded from `http://www-personal.umich.edu/~mejn/netdata/`). Each blog in this dataset has an attribute describing its political leaning as either liberal or conservative.

Synthetic Dataset Most real world graphs show power-law degree distributions and small-world characteristics [37]. Therefore, we use the R-MAT model [13] in the GTgraph suites [2] to generate graphs with power-law degree distributions and small-world characteristics. Based on the statistics in Table 5.1, we set the average node degree in each synthetic graph to 5. We used the default values for the other

[1]**DB**: VLDB J., TODS, KDD, PODS, VLDB, SIGMOD; **AL**: STOC, SODA, FOCS, Algorithmica, J. Algorithms, SIAM J. Comput., ISSAC, ESA, SWAT, WADS; **OS**: USENIX, OSDI, SOSP, ACM Trans. Comput. Syst., HotOS, OSR, ACM SIGOPS European Workshop; **CC**: PLDI, POPL, OOPSLA, ACM Trans. Program. Lang. Syst., CC, CGO, SIGPLAN Notices, ECOOP; **AI**: IJCAI, AAAI, AAAI/IAAI, Artif. Intell.

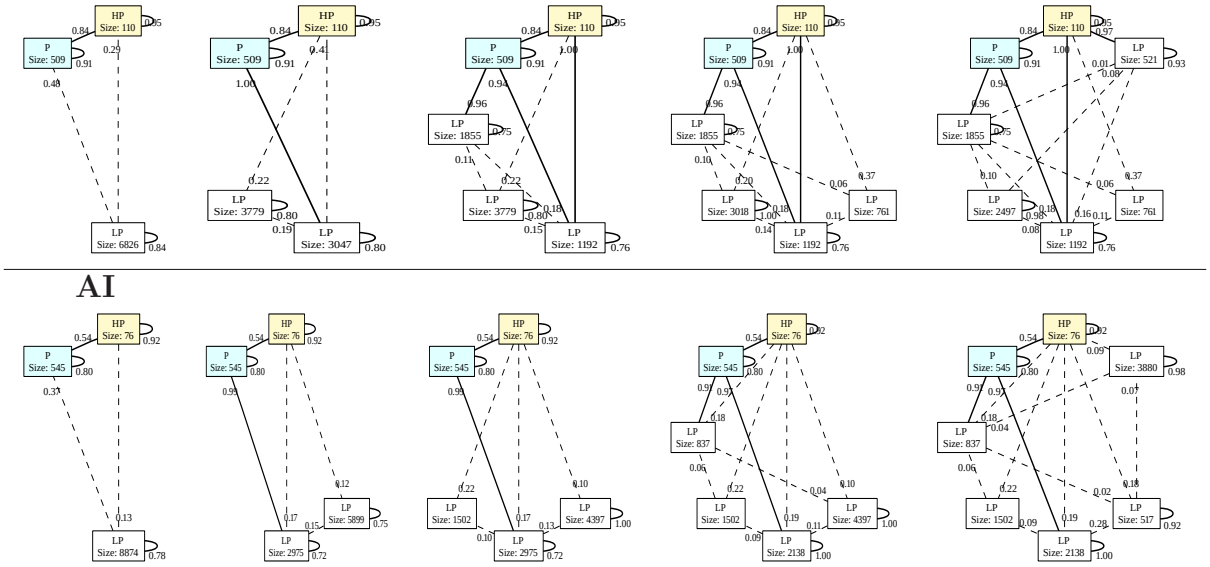

Table 5.2: The Aggregation Results for the DBLP DB and AI Subsets

parameters in the R-MAT based generator. We also assign an attribute to each node in a generated graph. The domain of this attribute has 5 values. For each node we randomly assign one of the five values.

5.4.2 Effectiveness Evaluation

We first evaluate the effectiveness of our graph summarization methods. In this section, we use only the top-down approach to evaluate the *k-SNAP* operation, as we compare the top-down and the bottom-up approaches in Section 5.4.3.

5.4.2.1 DBLP Coauthorship Networks

In this experiment, we are interested in analyzing how researchers in the database area coauthor with each other. As input, we use the DBLP DB subset (see D1 of

Table 5.1 and Figure 5.5). Each node in this graph has one attribute called *Pub-Num*, which is the number of publications belonging to the corresponding author. By plotting the distribution of the number of publications of this dataset in Figure 5.6, we assigned another attribute called *Prolific* to each author in the graph indicating whether that author is prolific: authors with ≤ 5 papers are tagged as low prolific (LP), authors with > 5 but ≤ 20 papers are prolific (P), and the authors with > 20 papers are tagged as highly prolific (HP).

We first issue a *SNAP* operation on the *Prolific* attribute and the coauthorships. The result is visualized in Figure 5.7. Groups with the HP attribute value are colored in yellow, groups with the P value are colored in light blue, and the remaining groups with the attribute value LP are in white. The *SNAP* operation results in a summary with 3569 groups and 11293 group relationships. This summary is too big to analyze. On the other hand, if we apply the *SNAP* operation on only the *Prolific* attribute (i.e. not considering any relationships in the *SNAP* operation), we will get a summary with only 3 groups as visualized in the top left figure in Table 5.2. The bold edges between two groups indicate strong group relationships (with more than 50% participation ratio), while dashed edges are weak group relationships. This summary shows that the HP researchers as a whole have very strong coauthorship with the P group of researchers. Researchers within both groups also tend to coauthor with people within their own groups. However, this summary does not provide a lot of information for the LP researchers: they tend to coauthor strongly within their group and they have some connection with the HP and P groups.

Now we make use of the *k-SNAP* operation to produce summaries with multiple

resolutions. The first row of figures in Table 5.2 shows the *k-SNAP* results for k =4, 5, 6 and 7. As k increases, more details are shown in the summaries.

When $k = 7$, the summary shows that there are 5 subgroups of LP researchers. One group of 1192 LP researchers strongly collaborates with both HP and P researchers. One group of 521 only strongly collaborates with HP researchers. One group of 1855 only strongly collaborates with P researchers. These three groups also strongly collaborate within their groups. There is another group of 2497 LP researchers that has very weak connections to other groups but strongly cooperates among themselves. The last group has 761 LP researchers, who neither coauthor with others within their own group nor collaborate strongly with researchers in other groups. They often write single author papers.

Now, in the *k-SNAP* result for $k = 7$, we are curious if the average number of publications for each subgroup of the LP researchers is affected by the coauthorships with other groups. The above question can be easily answered by applying the *avg* operation on the *PubNum* attribute for each group in the result of the *k-SNAP* operation.

With this analysis, we find that the group of LP researchers who collaborate with both P and HP researchers has a high average number of publications: 2.24. The group only collaborating with HP researchers has 1.66 publications on average. The group collaborating with only the P researchers has on average 1.55 publications. The group that tends to only cooperate among themselves has a low average number of publications: 1.26. Finally, the group of mostly single authors has on average only 1.23 publications. Not surprisingly, these results suggest that collaborating with HP

and P researchers is very helpful for the low prolific (often beginning) researchers.

Next, we want to compare the database community with the AI community to see whether the coauthorship relationships are different across these two communities. We constructed the AI coauthorship graph with 9495 authors and 16070 coauthorships from the DBLP AI subset. The distribution of the number of publications of AI authors is similar to the DB authors, thus we use the same method to assign the *Prolific* attribute to these authors. The *SNAP* operation on the *Prolific* attribute and coauthorships results in a summary with 3359 groups and 7091 group relationships. The second row of figures in Table 5.2 shows the *SNAP* result based only on the *Prolific* attribute and the *k-SNAP* results for k =4, 5, 6 and 7. Comparing the summaries for the two communities for $k = 7$, we can see the differences across the two communities: The HP and P groups in the AI community have a weaker cooperation than the DB community; and there isn't a large group of LP researchers who strongly coauthor with both HP and P researchers in the AI area.

As this example shows, by changing the resolutions of summaries, users can better understand the characteristics of the original graph data and also explore the differences and similarities across different datasets.

5.4.2.2 Political Blogs Network

In this experiment, we evaluate the effectiveness of our graph summarization methods on the political blogs network (1490 nodes and 19090 edges). The *SNAP* operation based on the political leaning attribute and the links between blogs results in a summary with 1173 groups and 16657 group relationships. The *SNAP* result based

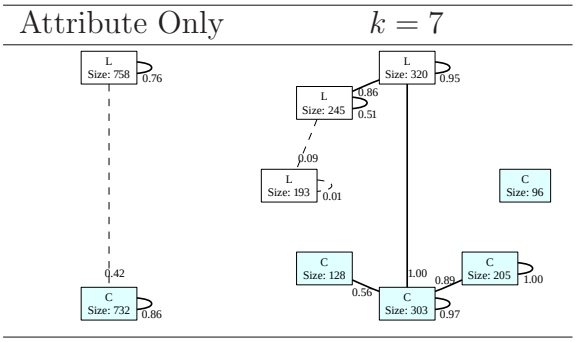

Table 5.3: Aggregation results for Political Blogs Dataset

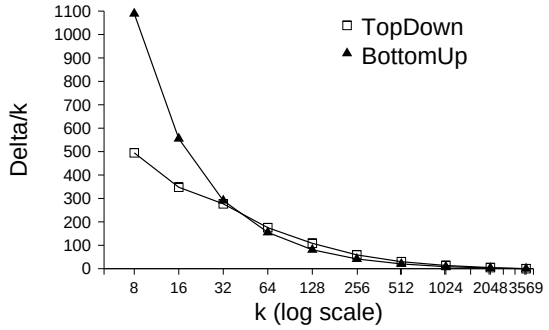

Figure 5.8: Quality of summaries: top-down vs. bottom-up

only on the attribute and the *k-SNAP* result based on both the attribute and the links for $k = 7$ are shown in Table 5.3.

From the results, we see that there are a group of liberal blogs and a group of conservative blogs that interact strongly with each other (perhaps to refute each other). Other groups of blogs only connect to blogs in their own communities (liberal or conservative), if they do connect to other blogs. There is a relatively large group of 193 liberal blogs with almost no connections to any other blogs, while such isolated blogs compose a much smaller portion (96 blogs) of all the conservative blogs. Overall, conservative blogs show a slightly higher tendency to link with each other than liberal

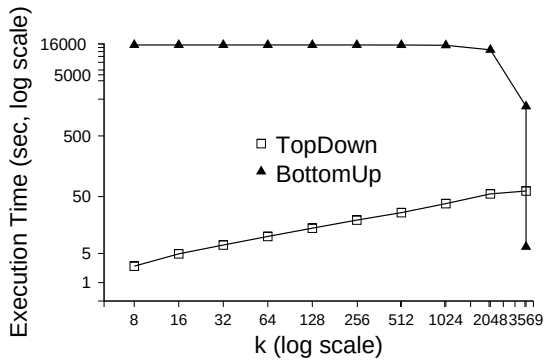

Figure 5.9: Efficiency: top-down vs. bottom-up

blogs, which is consistent with the conclusion from the analysis in [1]. Given that the blogs data was collected right after the US 2004 election, the authors in [1] speculated that the different linking behaviors in the two communities may be correlated with eventual outcome of the 2004 election.

5.4.3 k-SNAP: Top-Down vs. Bottom-Up

In this section, we compare the top-down and the bottom-up *k-SNAP* algorithms, both in terms of effectiveness and efficiency. We use the DBLP DB subset (D1 in Table 5.1) and apply both approaches for different k values.

For the effectiveness experiment, we use the Δ measure introduced in Section 5.2.2 to assess the qualities of summaries. Note that for a given k value, smaller Δ value means better quality summary, but for different k values, comparing Δ does not make sense, as a higher k value tends to result in a higher Δ value according to Equation V.1. However, if we normalize Δ by k, we get the average contribution of each group to the Δ value, then we can compare $\frac{\Delta}{k}$ for different k values.

We acknowledge that $\frac{\Delta}{k}$ is not a perfect measure for "quantitatively" evaluating the quality of summaries. However, quality assessment is a tricky issue in general, and $\frac{\Delta}{k}$, though crude, is an intuitive measure for this study.

Figure 5.8 shows the comparison of the summary qualities between the top-down and the bottom-up approaches. Note that the x-axis is in log scale and the y-axis is $\frac{\Delta}{k}$. First, as k increases, both methods produce higher quality summaries. For small k values, top-down approach produces significantly higher quality summaries than the bottom-up approach. This is because, the bottom-up approach starts from the grouping produced by the *SNAP* operation. This initial grouping is usually very large, in this case, it contains 3569 groups. The bottom-up approach has to continuously merge groups until the number of groups decreases to a small k value. Each merge decision is only made based on the current grouping, and errors can easily accumulate. In contrast, the top-down approach starts from the maximum A-compatible grouping, and only needs a small number of splits to reach the result. Therefore, small amount of errors is accumulated. As k becomes larger, the bottom-up approach shows slight advantage over the top-down approach.

The execution times for the two approaches are shown in Figure 5.9. Note that both axes are in log scale. The top-down approach significantly outperform the bottom-up approach, except when k is equal to the size of the grouping resulting from the *SNAP* operation. Initializing the heap takes a lot of time for the bottom-up approach, as it has to compare every pair of groups. This situation becomes worse, if the size of the initial grouping is very large.

In practice, users are more likely to choose small k values to generate summaries.

	# Groups	# Group Relationships	Time(sec)
D1	3569	11293	6.4
D2	7892	26031	16.1
D3	11379	35682	27.9
D4	15052	44318	44.0

Table 5.4: The *SNAP* Results for the DBLP Datasets

The top-down approach significantly outperforms the bottom-up approach in both effectiveness and efficiency for small k values. Therefore, the top-down approach is preferred for most practical uses. For all the remaining experiments, we only consider the top-down approach.

5.4.4 Efficiency Experiment

This section evaluates the efficiency the *SNAP* and the *k-SNAP* operations.

5.4.4.1 SNAP Efficiency

In this section, we apply the *SNAP* operation on the four DBLP datasets with increasing sizes (see Table 5.1). Table 5.4 shows the number of groups and group relationships in the summaries produced by the *SNAP* operation on the attribute *Prolific* (defined in the same way as in Section 5.4.2.1) and coauthorships, as well as the execution times. Even for the largest dataset with 30664 nodes and 70669 edges, the execution is completed in 44 seconds. However, all of the *SNAP* results are very large. The summary sizes are comparable to the input graphs. Such large summaries are often not very useful for analyzing the input graphs. This is an anecdotal evidence of why the *k-SNAP* operation is often more desired than the *SNAP* operation in

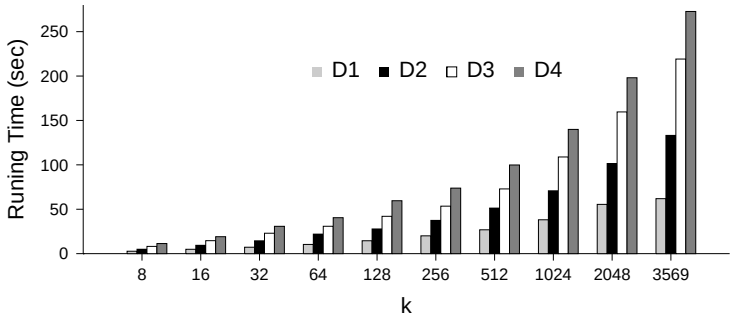

Figure 5.10: Efficiency experiment for DBLP datasets

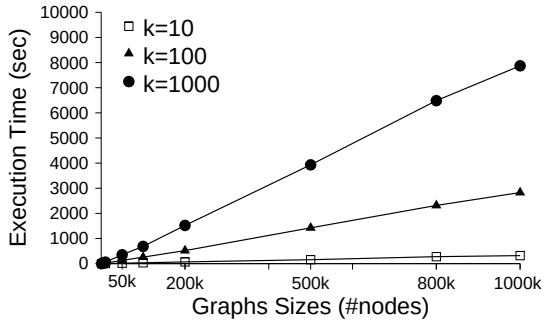

Figure 5.11: Efficiency experiment for synthetic datasets

practice.

5.4.4.2 k-SNAP Efficiency

This section evaluates the efficiency of the top-down *k-SNAP* algorithm on both the DBLP and the synthetic datasets.

DBLP Data In this experiment, we apply the top-down *k-SNAP* evaluation algorithm on the four DBLP datasets shown in Table 5.1 (the *k-SNAP* operation is based on *Prolific* attribute and coauthorships). The execution times with increasing graph sizes and increasing k values are shown in Figure 5.10. For these datasets, the

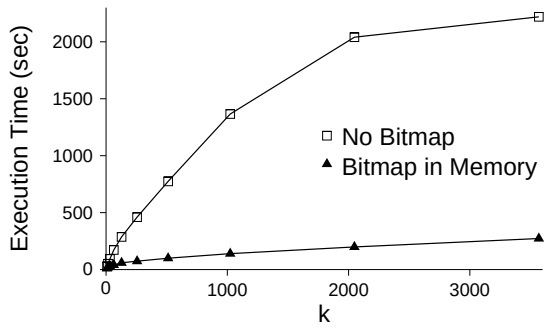

Figure 5.12: Bitmap in memory vs. no bitmap

performance behavior is close to linear, since the execution times are dominated by the database page accesses (as discussed in Section 5.3.3.1).

Synthetic Data We apply the *k-SNAP* operation on different sized synthetic graphs with three k values: 10, 100 and 1000. The execution times with increasing graph sizes are shown in Figure 5.11. When $k = 10$, even on the largest graph with 1 million nodes and 2.5 million edges, the evaluation algorithm finishes in about 5 minutes. For a given k value, the algorithm scales nicely with increasing graph sizes.

5.4.4.3 Evaluation with Very Large Graphs

So far, we have assumed that the amount of working memory is big enough to hold all the data structures (shown in Figure 5.4) used in the evaluation algorithms. This is often the case in practice, as large multi GB memory configurations are common and many graph datasets can fit in this space (especially if a subset of large graph is selected for analysis). However, our methods also work, when the graph datasets are extremely large and this in-memory assumption in not valid. In this section, we discuss the behaviors of our methods for this case. We only consider the most

practically useful top-down *k-SNAP* algorithm for this experiment.

In the case when the most memory consuming data structure, namely the neighbor-groups bitmap (see Figure 5.4), cannot fit in memory, the top-down approach drops the bitmap data structure. Without the bitmap, each time the algorithm splits a group, it has to query the edges information in the database to infer the neighbor-groups. We have implemented a version of the top-down *k-SNAP* algorithm without the bitmap data structure, and compared it with the normal top-down algorithm.

To keep this experiment manageable, we scaled down the experiment settings. We used the DBLP D4 dataset in Table 5.1, and set the buffer pool size and working memory size to 16MB and 8MB, respectively. This "scaled-down" experiment exposes the behaviors of the two versions of the top-down algorithm, while keeping the running times reasonable. As shown in Figure 5.12, the version of the top-down approach without bitmap is much slower than the normal version. This is not surprising as the former incurs more disk IOs.

Given the graph size and the k value, our current implementation can decide in advance whether the bitmap can fit in the working memory, by estimating the upper bound of the bitmap size. It can then choose the appropriate version of the algorithm to use. In the future, we plan on designing a more sophisticated version of the top-down algorithm in which part of the bitmap can be kept in memory when the available memory is small.

5.5 Conclusions

This chapter has introduced two aggregation operations *SNAP* and *k-SNAP* for controlled and intuitive database-style graph summarization. Our methods allow users to freely choose node attributes and relationships that are of interest, and produce summaries based on the selected features. Furthermore, the *k-SNAP* aggregation allows users to control the resolutions of summaries and provides "drill-down" and "roll-up" abilities to navigate through the summaries. We have formally defined the two operations and proved that evaluating the *k-SNAP* operation is NP-complete. We have also proposed an efficient algorithm to evaluate the *SNAP* operation and two heuristic algorithms to approximately evaluate the *k-SNAP* operation. Through extensive experiments on a variety of real and synthetic datasets, we show that of the two *k-SNAP* algorithms, the top-down approach is a better choice in practice. Our experiments also demonstrate the effectiveness and efficiency of our methods. As part of future work, we plan on designing a formal graph data model and query language that allows incorporation of *k-SNAP*, along with a number of other additional common and useful graph matching methods.

CHAPTER VI

Related Work

6.1 Graph Matching Methods

There is a long history of database research on methods for querying graphs. However, most previous works have focused on exact graph or subgraph matching, i.e. graph or subgraph isomorphism. Subgraph isomorphism was proved to be NP-complete in [19]. Ullmann [51] proposed a subgraph matching algorithm based on a state space search method with backtracking. However, this algorithm is prohibitively expensive for querying against database with a large number of graphs. To reduce the search space, GraphGrep [45], GIndex [58] and TreePi [61] index substructures of the database (paths, frequent subgraphs and trees respectively) to filter out graphs that do not match the query.

Several index-based methods for approximate subgraph matching have also been proposed. However, most of these techniques only apply to small graphs and allow limited approximation. Grafil [59] and PIS [60] are both built on top of the exact

subgraph matching method GIndex. However, neither method allows node insertion or deletion in their match models. CDIndex [55] only applies to graphs with limited sizes, as it exhaustedly enumerates and indexes all the subgraphs in the database. GString [28] utilizes sequence matching to answer graph queries, but it only applies to applications in which the graphs contain a small number of basic substructures. C-Tree [22], which employs an R-tree like index structure, is a more general tool than the above methods. However, C-Tree, as well as Grafil, PIS, CDIndex and GString, utilize memory-based indexing techniques, which require the indexes to be memory resident during query processing. As the database size increases, these indexes quickly grow out of memory. On the contrary, the indexing approaches in SAGA and TALE are disk-based.

The life science community has produced vast amount of protein interaction networks. Several tools for comparing protein interaction networks have been proposed. These include PathBlast [30], its successor NetworkBlast [40], MaWIsh [35], and Graemlin [17]. Of these, Graemlin is the latest method and in many ways superior to the other methods for comparing protein interaction networks.

6.2 Graph Summarization Methods

Graph summarization has attracted a lot of interest from both the sociology and the database research communities. Most existing works on graph summarization use statistical methods to study graph characteristics, such as degree distributions, hop-plots and clustering coefficients. Comprehensive surveys on these methods are

provided in [11] and [37]. A-plots [12] is a novel statistical method to summarize the adjacency matrix of graphs for outlier detection. Statistical summaries are useful but hard to control and navigate. Methods for mining frequent graph patterns [26, 52, 57] are also used to understand the characteristics of large graphs. Washio and Motoda [53] provide an elegant review on this topic. However, these mining algorithms often produces an overwhelmingly large number of frequent patterns. Various graph partitioning algorithms [38, 48, 56] are used to detect community structures (dense subgraphs) in large graphs. SuperGraph [43] employs hierarchical graph partitioning to visualize large graphs. In [7], Frey and Dueck proposed a novel clustering method by passing messages between nodes in the graph. However, graph partitioning or clustering techniques largely ignore the node attributes in the summarization. Studies on graph visualization are surveyed in [3, 23]. For very large graphs, these visualization methods are still not enough. Unlike these existing methods, we introduce two database-style operations to summarize large graphs. Our method allows users to easily control and navigate through summaries.

Previous research [4, 6, 41] have also studied the problem of compressing large graphs, especially Web graphs. However, these graph compression methods mainly focus on compact graph representation for easy storage and manipulation, whereas graph summarization methods aim at producing small and understandable summaries.

Regular equivalence is introduced in [54] to study social roles of nodes based on graphs structures in social networks. It shares resemblance with the *SNAP* operation. However, regular equivalence is defined only based on the relationships between nodes.

Node attributes are largely ignored. In addition, the *k-SNAP* operation relaxes the stringent equivalence requirement of relationships between node groups, and produces user controllable multi-resolution summaries.

The *SNAP* algorithm shares similarity with the automorphism partitioning algorithm in [16]. However, the automorphism partitioning algorithm only partitions nodes based on node degrees and relationships, whereas *SNAP* can be evaluated based on arbitrary node attributes and relationships that a user selects.

CHAPTER VII

Conclusions

The rapidly growing graph datasets have made graph querying systems critical for many modern applications. To allow users to perform complex analysis on graph data, this thesis develops a graph querying toolkit, called Periscope/GQ. This toolkit provides a uniform schema for storing graphs in the database and supports various graph query operations, especially sophisticated query operations.

Approximate graph matching query is one of the sophisticated query operations that Periscope/GQ supports, due to its usefulness and advantage over the exact graph matching query. Chapter III introduces an efficient approximate graph matching method, called SAGA. SAGA employs a flexible graph similarity model and an index-based matching algorithm to efficiently evaluate approximate graph matching queries.

To further handle the case of large query graphs, Chapter IV proposes another approximate graph matching technique TALE. TALE utilizes a novel indexing technique, which achieves high pruning power and linear index size with the database size. The TALE matching algorithm first uses the index to match the important nodes in

the query, and then progressively extends these matches.

Chapter V proposes two aggregation operations for efficient graph summarization. The *SNAP* operation, produces a summary graph by grouping nodes based on user-selected node attributes and relationships. The *k-SNAP* operation allows users to further control the resolutions of summaries and provides the "drill-down" and "roll-up" abilities to navigate through summaries with different resolutions.

Extensive experiments on a variety of real applications have demonstrated the effectiveness and efficiency of the Periscope/GQ toolkit.

APPENDICES

APPENDIX A

Statistical Evaluation for Approximate Graph Matching Results

In many applications, especially life sciences applications, producing the approximate graph matching results is not enough. It is also very important to evaluate the statistical significance of the matching results, i.e. assess whether an approximate graph match constitutes a meaningful result or a random accident. Periscope/GQ employs the Monte Carlo simulation approach to evaluate the statistical significance of the matches for general applications. However, this approach brings significant computation overhead. Making use of domain knowledge, experts can develop more light-weight statistical evaluation methods. As an anecdotal example, we introduce a specific statistical scoring model designed for the application of matching parsed literature graphs (see Section 3.3.4 of Chapter III).

1.1 Monte Carlo Simulation

The Monte Carlo simulation approach relies on matching a query against random graphs to estimate the p-value of a query result (the probability of obtaining a result at least as good as the one that was actually observed by change). Periscope/GQ generates random graphs by randomly shuffling edges of the graphs in the database preserving the node degrees, and randomizing the orthologous groups of each node preserving the number of orthologous groups that each node belongs to. For a given query, in addition to querying the real database, the query is also run against a large number of random graphs. A p-value of a match from the real database is estimated as the fraction of matches from the random graphs with the same or a larger size (in number of nodes) and the same or a better similarity value (e.g. SAGA graph distance value).

1.2 An Application Specific Statistical Scoring Model

For the application of matching parsed literature graphs (see Section 3.3.4 of Chapter III), we developed a light-weight method to assess the statistical significance based on domain knowledge. This method does not need any simulation on random graphs. Instead, it can be simply calculated based on the query graph, the target graph and the match, thus significantly speeds up the statistical evaluation.

The statistical score for a query to a target graph in the database is defined as follows:

$$S = -log(P(N_{me}|N_{mn}) * P(N_{mn}))$$

Notation	Description
N_g	The total number of genes in a species
N_n	The number of nodes in the target graph
N_{mn}	The number of matching nodes in the target graph
N_e	The number of edges connecting the matching nodes in the target graph
N_{me}	The number of matching edges in the target graph

Table 1.1: The Notation Table

In the above equation, N_{mn} is the number of matching nodes, N_{me} is the number of matching edges, $P(N_{mn})$ is the probability of selecting N_{mn} matching nodes at random from the target graph, and $P(N_{me}|N_{mn})$ is the probability of selecting N_{me} matching edges given N_{mn} matching nodes. Table 1.1 lists the descriptions of the notations used in the computation of this statistical scoring method.

1.2.1 Calculating $P(N_{mn})$

$P(N_{mn})$ is the probability of selecting N_{mn} matching nodes at random from the target graph. Assume that all nodes, which represent genes in this application, are equally likely to be matched, and there are N_g genes in a typical genome (N_g = 20,000 in general). For a target graph, containing N_n nodes and matching N_{mn} of these with the query, the number of ways to choose the N_{mn} matching nodes in the target is $N_{cn} = N_n!/(N_{mn}! * (N_n - N_{mn})!)$. And the number of ways to choose N_{mn} nodes (or genes) from the genome is $N_{cg} = N_g!/(N_{mn}! * (N_g - N_{mn})!)$. Therefore, the probability that the matching nodes are selected at random is $P(N_{mn}) = N_{cn}/N_{cg}$.

1.2.2 Calculating $P(N_{me}|N_{mn})$

$P(N_{me}|N_{mn})$ is the probability of selecting N_{me} matching edges given N_{mn} matching nodes. Let N_e be the number of edges connecting the N_{mn} matching nodes in the target graph. Assume that each edge from the N_e edges are equally likely to be selected as a matching edge ($P_e = 1/N_e$). The number of ways to choose N_{me} matching edges is $N_{ce} = (N_e!/(N_{me}! * (N_e - N_{me})!)$. The probability that there are N_{me} matching edges given N_{mn} matching nodes is $P(N_{me}|N_{mn}) = N_{ce} * P_e^{N_{me}}$ (The chance that an edge is in the matching set raised to the power of number of edges multiplied by the number of ways to choose the matching set from the set of possible edges).

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] L. A. Adamic and N. Glance. The political blogosphere and the 2004 us election. In *WWW Workshop on the Weblogging Ecosystem*, 2005.

[2] D. A. Bader and K. Madduri. GTgraph: A suite of synthetic graph generators. `http://www.cc.gatech.edu/~kamesh/GTgraph`.

[3] G. Battista, P. Eades, R. Tamassia, and I. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[4] D. K. Blandford, G. E. Blelloch, and I. A. Kash. Compact representations of separable graphs. In *Proceedings of SODA'03*, pages 679–688, 2003.

[5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[6] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proceedings of WWW'04*, pages 595–602, 2004.

[7] F. Brendan J and D. Dueck. Clustering by passing messages between data points. *Science*, 315:972–976, 2007.

[8] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *CACM*, 16(9):575–577, 1973.

[9] H. Bunke. On a relation between graph edit distance and maximum common subgraph. *Pattern Recogn. Lett.*, 18(8):689–694, 1997.

[10] C. Alfarano et al. The biomolecular interaction network database and related tools 2005 update. *Nucleic Acids Res.*, 33:D418–D424, 2005.

[11] D. Chakrabarti and C. Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Comput. Surv.*, 38(1), 2006.

[12] D. Chakrabarti, C. Faloutsos, and Y. Zhan. Visualization of large networks with min-cut plots, A-plots and R-MAT. *Int. J. Hum.-Comput. Stud.*, 65(5), 2007.

[13] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *SDM*, 2004.

[14] M. Chen and R. Hofestaedt. *Applied Bioinformatics*, 3(4):241–252, 2004.

[15] A. Coppen. The biochemistry of affective disorders. *Br J Psychiatry*, 113(504):1237–64, 1967.

[16] D. G. Corneil and C. C. Gotlieb. An efficient algorithm for graph isomorphism. *J. ACM*, 17(1):51–64, 1970.

[17] J. Flannick, A. Novak, B. S. Srinivasan, H. H. McAdams, and S. Batzoglou. Græmlin: General and robust alignment of multiple large interaction networks. *Genome Res.*, 16:1169–1181, 2006.

[18] G. Joshi-Tope et al. Reactome: a knowledgebase of biological pathways. *Nucleic Acids Res.*, 33:D428–32, 2005.

[19] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[20] R. Giugno and D. Shasha. Graphgrep: A fast and universal method for querying graphs. In *IEEE International Conference in Pattern recognition*, 2002.

[21] R. H. Guting. Graphdb: Modeling and querying graphs in databases. In *VLDB*, pages 297–308, 1994.

[22] H. He and A. K. Singh. Closure-tree: an index structure for graph queries. In *ICDE*, 2006.

[23] I. Herman, G. Melançon, and M. S. Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Trans. Vis. Comput. Graph.*, 6(1), 2000.

[24] D. S. Hochbaum. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Co., Boston, 1997.

[25] J. Hu, X. Shen, Y. Shao, C. Bystroff, and M. J. Zaki. Mining protein contact maps. In *BIOKDD*, 2002.

[26] J. Huan, W. Wang, J. Prins, and J. Yang. Spin: mining maximal frequent subgraphs from graph databases. In *KDD*, 2004.

[27] J. Chandonia et al. The astral compendium in 2004. *Nucleic Acids Res.*, 32:D189–D192, 2004.

[28] H. Jiang, H. Wang, P. S. Yu, and S. Zhou. Gstring: A novel approach for efficient search in graph databases. In *ICDE*, 2007.

[29] D. Kalderon. Similarities between the hedgehog and wnt signaling pathways. *Trends in Cell Biology*, 12(11):523–531, 2002.

[30] B. P. Kelley, B. Yuan, F. Lewitter, R. Sharan, B. R. Stockwel, and T. Ideker. Pathblast: a tool for alignment of protein interaction networks. *Nucleic Acids Res.*, pages W83–W88, 2004.

[31] M. Koyuturk, A. Grama, and W. Szpankowski. Pairwise local alignment of protein interaction networks guided by models of evolution. In *International Conference on Research in Computational Molecular Biology*, pages 48–65, 2005.

[32] M. Ley. DBLP Bibliography. `http://www.informatik.uni-trier.de/~ley/db/`.

[33] W. Li, L. Jaroszewski, and A. Godzik. Clustering of highly homologous sequences to reduce the size of large protein database. *Bioinformatics*, 17:282–283, 2001.

[34] M. Kanehisa et al. The kegg resources for deciphering the genome. *Nucleic Acids Res.*, 32:D277–D280, 2004.

[35] M. Koyuturk et al. Pairwise alignment of protein interaction networks. *Journal of Computational Biology*, 13(2):182–199, 2006.

[36] M. Tourigny et al. Cdk inhibitor p18ink4c is required for the generation of functional plasma cells. *Immunity*, 17(2):179–189, 2002.

[37] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45, 2003.

[38] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69, 2004.

[39] R. Nusse. Wnts and hedgehogs: lipid-modified proteins and similarities in signaling mechanisms at the cell surface. *Development*, 130:5297–5305, 2003.

[40] R. Sharan et al. Conserved patterns of protein interaction in multiple species. *PNAS*, 102:1974–1979, 2005.

[41] S. Raghavan and H. Garcia-Molina. Representing Web graphs. In *Proceedings of ICDE'03*, pages 405–416, 2003.

[42] F. S. Roberts and L. Sheng. How hard is it to determine if a graph has a 2-role assignment? *Networks*, 37(2), 2001.

[43] J. F. Rodrigues, A. J. M. Traina, C. Faloutsos, and C. T. Jr. SuperGraph visualization. In *ISM*, 2006.

[44] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1983.

[45] D. Shasha, J. T.-L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, 2002.

[46] L. Sheng, Z. M. Ozsoyoglu, and G. Ozsoyoglu. A graph query language and its query processing. In *ICDE*, 1999.

[47] E. Sprinzak, S. Sattath, and H. Margalit. How reliable are experimental protein-protein interaction data? *Journal of Molecular Biology*, 327(5):919–923, 2003.

[48] J. Sun, Y. Xie, H. Zhang, and C. Faloutsos. Less is more: Compact matrix decomposition for large sparse graphs. In *SDM*, 2007.

[49] T. Luedde et al. p18(ink4c) collaborates with other cdk-inhibitory proteins in the regenerating liver. *Hepatology*, 37(4):833–841, 2003.

[50] R. L. Tatusov, E. V. Koonin, and D. J. Lipman. A genomic perspective on protein families. *Science*, 278(5388):631–7, 1997.

[51] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[52] W. Wang, C. Wang, Y. Zhu, B. Shi, J. Pei, X. Yan, and J. Han. Graphminer: a structural pattern-mining system for large disk-based graph databases and its applications. In *SIGMOD*, 2005.

[53] T. Washio and H. Motoda. State of the art of graph-based data mining. *SIGKDD Explor. Newsl.*, 5(1), 2003.

[54] D. R. White and K. P. Reitz. Graph and semigroup homomorphisms on semigroups of relations. Social Networks, 1983.

[55] D. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *ICDE*, 2007.

[56] X. Xu, N. Yuruk, Z. Feng, and T. A. J. Schweiger. SCAN: a structural clustering algorithm for networks. In *KDD*, 2007.

[57] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, 2002.

[58] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *SIGMOD*, 2004.

[59] X. Yan, P. S. Yu, and J. Han. Substructure similarity search in graph databases. In *SIGMOD*, 2005.

[60] X. Yan, F. Zhu, J. Han, and P. S. Yu. Searching substructures wuth superimposed distance. In *ICDE to appear*, 2006.

[61] S. Zhang, M. Hu, and J. Yang. Treepi: A new graph indexing method. In *ICDE*, 2007.