MORGAN & CLAYPOOL PUBLISHERS

Human Interaction with Graphs

A Visual Querying Perspective

Sourav S. Bhowmick
Byron Choi
Chengkai Li

SYNTHESIS LECTURES ON DATA MANAGEMENT

H.V. Jagadish, Series Editor

Human Interaction with Graphs

A Visual Querying Perspective

Synthesis Lectures on Data Management

Editor
H.V. Jagadish, ***University of Michigan***

Founding Editor
M. Tamer Özsu, ***University of Waterloo***

Synthesis Lectures on Data Management is edited by H.V. Jagadish of the University of Michigan. The series publishes 80–150 page publications on topics pertaining to data management. Topics include query languages, database system architectures, transaction management, data warehousing, XML and databases, data stream systems, wide scale data distribution, multimedia data management, data mining, and related subjects.

Human Interaction with Graphs: A Visual Querying Perspective
Sourav S. Bhowmick, Byron Choi, and Chengkai Li
2018

Answering Queries Using Views
Foto Afrati and Rada Chirkova
2017

Databases on Modern Hardware: How to Stop Underutilization and Love Multicores
Anatasia Ailamaki, Erieta Liarou, Pınar Tözün, Danica Porobic, and Iraklis Psaroudakis
2017

Instant Recovery with Write-Ahead Logging: Page Repair, System Restart, Media Restore, and System Failover, Second Edition
Goetz Graefe, Wey Guy, and Caetano Sauer
2016

Generating Plans from Proofs: The Interpolation-based Approach to Query Reformulation
Michael Benedikt, Julien Leblay, Balder ten Cate, and Efthymia Tsamoura
2016

Veracity of Data: From Truth Discovery Computation Algorithms to Models of Misinformation Dynamics
Laure Berti-Équille and Javier Borge-Holthoefer
2015

Datalog and Logic Databases
Sergio Greco and Cristina Molinaro
2015

Big Data Integration
Xin Luna Dong and Divesh Srivastava
2015

Instant Recovery with Write-Ahead Logging: Page Repair, System Restart, and Media Restore
Goetz Graefe, Wey Guy, and Caetano Sauer
2014

Similarity Joins in Relational Database Systems
Nikolaus Augsten and Michael H. Böhlen
2013

Information and Influence Propagation in Social Networks
Wei Chen, Laks V.S. Lakshmanan, and Carlos Castillo
2013

Data Cleaning: A Practical Perspective
Venkatesh Ganti and Anish Das Sarma
2013

Data Processing on FPGAs
Jens Teubner and Louis Woods
2013

Perspectives on Business Intelligence
Raymond T. Ng, Patricia C. Arocena, Denilson Barbosa, Giuseppe Carenini, Luiz Gomes, Jr., Stephan Jou, Rock Anthony Leung, Evangelos Milios, Renée J. Miller, John Mylopoulos, Rachel A. Pottinger, Frank Tompa, and Eric Yu
2013

Semantics Empowered Web 3.0: Managing Enterprise, Social, Sensor, and Cloud-based Data and Services for Advanced Applications
Amit Sheth and Krishnaprasad Thirunarayan
2012

Data Management in the Cloud: Challenges and Opportunities
Divyakant Agrawal, Sudipto Das, and Amr El Abbadi
2012

Query Processing over Uncertain Databases
Lei Chen and Xiang Lian
2012

Foundations of Data Quality Management
Wenfei Fan and Floris Geerts
2012

Incomplete Data and Data Dependencies in Relational Databases
Sergio Greco, Cristian Molinaro, and Francesca Spezzano
2012

Business Processes: A Database Perspective
Daniel Deutch and Tova Milo
2012

Data Protection from Insider Threats
Elisa Bertino
2012

Deep Web Query Interface Understanding and Integration
Eduard C. Dragut, Weiyi Meng, and Clement T. Yu
2012

P2P Techniques for Decentralized Applications
Esther Pacitti, Reza Akbarinia, and Manal El-Dick
2012

Query Answer Authentication
HweeHwa Pang and Kian-Lee Tan
2012

Declarative Networking
Boon Thau Loo and Wenchao Zhou
2012

Full-Text (Substring) Indexes in External Memory
Marina Barsky, Ulrike Stege, and Alex Thomo
2011

Spatial Data Management
Nikos Mamoulis
2011

Database Repairing and Consistent Query Answering
Leopoldo Bertossi
2011

Managing Event Information: Modeling, Retrieval, and Applications
Amarnath Gupta and Ramesh Jain
2011

Fundamentals of Physical Design and Query Compilation
David Toman and Grant Weddell
2011

Methods for Mining and Summarizing Text Conversations
Giuseppe Carenini, Gabriel Murray, and Raymond Ng
2011

Probabilistic Databases
Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch
2011

Peer-to-Peer Data Management
Karl Aberer
2011

Probabilistic Ranking Techniques in Relational Databases
Ihab F. Ilyas and Mohamed A. Soliman
2011

Uncertain Schema Matching
Avigdor Gal
2011

Fundamentals of Object Databases: Object-Oriented and Object-Relational Design
Suzanne W. Dietrich and Susan D. Urban
2010

Advanced Metasearch Engine Technology
Weiyi Meng and Clement T. Yu
2010

Web Page Recommendation Models: Theory and Algorithms
Sule Gündüz-Ögüdücü
2010

Multidimensional Databases and Data Warehousing
Christian S. Jensen, Torben Bach Pedersen, and Christian Thomsen
2010

Database Replication
Bettina Kemme, Ricardo Jimenez-Peris, and Marta Patino-Martinez
2010

Relational and XML Data Exchange
Marcelo Arenas, Pablo Barcelo, Leonid Libkin, and Filip Murlak
2010

User-Centered Data Management
Tiziana Catarci, Alan Dix, Stephen Kimani, and Giuseppe Santucci
2010

Data Stream Management
Lukasz Golab and M. Tamer Özsu
2010

Access Control in Data Management Systems
Elena Ferrari
2010

An Introduction to Duplicate Detection
Felix Naumann and Melanie Herschel
2010

Privacy-Preserving Data Publishing: An Overview
Raymond Chi-Wing Wong and Ada Wai-Chee Fu
2010

Keyword Search in Databases
Jeffrey Xu Yu, Lu Qin, and Lijun Chang
2009

Copyright © 2018 by Morgan & Claypool

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopy, recording, or any other except for brief quotations in printed reviews, without the prior permission of the publisher.

Human Interaction with Graphs: A Visual Querying Perspective

Sourav S. Bhowmick, Byron Choi, and Chengkai Li

www.morganclaypool.com

ISBN: 9781681733760 paperback
ISBN: 9781681733777 ebook
ISBN: 9781681733784 hardcover

DOI 10.2200/S00855ED1V01Y201805DTM047

A Publication in the Morgan & Claypool Publishers series
SYNTHESIS LECTURES ON DATA MANAGEMENT

Lecture #47
Series Editor: H.V. Jagadish, *University of Michigan*
Founding Editor: M. Tamer Özsu, *University of Waterloo*
Series ISSN
Print 2153-5418 Electronic 2153-5426

Human Interaction with Graphs

A Visual Querying Perspective

Sourav S. Bhowmick
Nanyang Technological University

Byron Choi
Hong Kong Baptist University

Chengkai Li
The University of Texas, Arlington

SYNTHESIS LECTURES ON DATA MANAGEMENT #47

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

Interacting with graphs using queries has emerged as an important research problem for real-world applications that center on large graph data. Given the syntactic complexity of graph query languages (e.g., SPARQL, *Cypher*), visual graph query interfaces make it easy for non-programmers to query such graph data repositories. In this book, we present recent developments in the emerging area of visual graph querying paradigm that bridges traditional graph querying with human computer interaction (HCI). Specifically, we focus on techniques that emphasize *deep integration* between the visual graph query interface and the underlying graph query engine. We discuss various strategies and guidance for constructing graph queries visually, interleaving processing of graph queries and visual actions, visual exploration of graph query results, and automated performance study of visual graph querying frameworks. In addition, this book highlights open problems and new research directions. In summary, in this book, we review and summarize the research thus far into the integration of HCI and graph querying to facilitate user-friendly interaction with graph-structured data, giving researchers a snapshot of the current state of the art in this topic, and future research directions.

KEYWORDS

graph querying, human-data interaction, deep integration, visual query formulation, visual query interface, query processing, results exploration, performance benchmarking

Contents

Foreword xv

Preface xvii

Acknowledgments xxi

1 Introduction 1

1.1 Interaction with Graphs Using Queries 1
1.2 Graph Query Construction Using Visual Interfaces 2
1.3 Integration of Visual Query Interface and Query Engine 4
1.4 Overview of this Book 6
1.5 Scope 7

2 Background 9

2.1 Graph Terminology 9
2.1.1 Subgraph Isomorphism-Related Terminology 10
2.1.2 Types of Graph Collection 10
2.1.3 Frequent and Infrequent Features 11
2.2 Visual Graph Query Interface 13
2.2.1 Structure of Visual Graph Query Interfaces 13
2.2.2 Visual Graph Query Formulation 14
2.2.3 Query Formulation-Related Terminology 15
2.3 Summary 17

3 Guidance for Visual Query Formulation 19

3.1 Overview of AutoG 21
3.2 Query Composition 23
3.2.1 Definition 23
3.2.2 Query Autocompletion Modes 24
3.2.3 C-prime Features 25
3.3 Autocompletion Framework in AutoG 28
3.3.1 Query Decomposition 28

3.3.2 Generation of Candidate Suggestions 30
3.3.3 Ranking Candidate Suggestions 31
3.4 Indexed Autocompletion–AutoGI 40
3.4.1 Feature DAG (FDAG) Index 40
3.4.2 Autocompletion by Using FDAG 42
3.5 Performance Study 42
3.5.1 Suggestion Quality 45
3.5.2 Efficiency 47
3.6 Guidance for Queries Over Large Networks 47
3.7 Bibliographic Notes 48
3.8 Conclusion 50

4 Blending Human Interactions and Graph Query Processing 51

4.1 Visual Substructure Search Problem 53
4.2 A Unified Framework 55
4.2.1 The Framework 55
4.2.2 Generality of the Framework 58
4.2.3 An Instantiation 58
4.3 Action-Aware Indexing 60
4.3.1 Key Features of Action-Aware Index 60
4.3.2 Action-Aware Frequent (A^2F) Index 61
4.3.3 Action-Aware Infrequent (A^2I) Index 65
4.4 Spindle-Shaped Graph (SPIG) 67
4.4.1 Algorithm for SPIG Construction 69
4.4.2 Analysis of SPIG Construction 71
4.5 Substructure Similarity Search 72
4.5.1 Exact Substructure Candidates Set Generation 72
4.5.2 Similar Substructure Candidates Set Generation 72
4.5.3 Generation of Approximate Query Results 75
4.6 Supporting Query Modification 76
4.7 Performance Study 79
4.7.1 Experimental Setup 79
4.7.2 Performance on Real Graph Dataset 80
4.7.3 Performance on Synthetic Graph Dataset 83
4.8 Bibliographic Notes 86
4.9 Conclusions 87

5 Blending Interactions and Query Processing on Large Networks 89

5.1 Overview and Contributions 91
5.1.1 Visual Substructure Search Problem Revisited 91
5.1.2 Overview 91
5.2 Decomposition of a Large Network 96
5.2.1 Graphlets and Adjacent Graphlets 96
5.2.2 Supergraphlets 98
5.3 Indexing Frequent and Infrequent Fragments 100
5.3.1 Frequent and Infrequent Fragments 101
5.3.2 Fragment Join 102
5.3.3 Generation of Frequent Fragments and SIFs 105
5.3.4 Index Construction 110
5.4 Graphlet-Based SPIG 110
5.4.1 Structure of G-SPIG 110
5.4.2 Algorithm 113
5.5 Blending Visual Subgraph Query 116
5.5.1 Candidate Data Graphs Generation 116
5.5.2 Generation of Query Results 118
5.6 Performance Study 119
5.6.1 Experimental Setup 120
5.6.2 System Response Time (SRT) 122
5.6.3 Index Size 124
5.6.4 Prefetching Time 128
5.6.5 Performance on a Million-Nodes Network 129
5.7 Bibliographic Notes 131
5.8 Conclusions 132

6 Human Interaction with Query Results 133

6.1 Results Exploration for Small- or Medium-Sized Data Graphs 133
6.1.1 PICASSO 134
6.2 Results Exploration on Large Networks 138
6.2.1 Region-Based Exploration 138
6.2.2 Exemplar-Based Exploration 140
6.2.3 Feature-Based Exploration 141
6.3 Bibliographic Notes 143
6.4 Conclusions 143

7 Simulation of Visual Subgraph Query Formulation 145

7.1 Overview of ViSual 147
7.2 Index-Based Generation of Subgraph Queries 148
7.3 Quantitative Modeling of Visual Query Formulation 151
7.3.1 Modeling Query Formulation Time 153
7.3.2 Model Extensibility 158
7.4 Simulation of Visual Subgraph Query Construction 160
7.4.1 Graph Representation of Query Formulation 160
7.4.2 The ViSual Algorithm 160
7.4.3 Finding Minimal and Maximal QFS 164
7.5 Performance Study 164
7.5.1 Performance of Test Subgraph Query Generation 164
7.5.2 Performance of the Query Formulation Model and ViSual 166
7.5.3 Application of ViSual 170
7.6 Bibliographic Notes 170
7.7 Conclusions 171

8 The Road Ahead 173

8.1 Summary 173
8.2 Future Research 174

Bibliography 177

Authors' Biographies 185

Foreword

Databases do an excellent job of organizing data and efficiently answering complex queries, yet they remain notoriously hard to use. Usability of database systems has grown into an important sub-field of database research over the past decade and more. A common misconception about usability is that it can be achieved by slapping a user interface on top of an existing system. For example, many relational databases come with a visual query builder interface. While such interfaces can be helpful, they do not address the more fundamental problem of bridging the gap between the user's desire and the system's representation.

What we need is a more fundamental alignment between query specification and query evaluation. Imagine, for example, getting a hint of the possible query results, at interactive speeds, even as you are specifying the query. This book is a significant step toward such a future.

Graph data is easier to visualize than most other types of data. At least for small graphs, humans can quickly learn a great deal just by "eyeballing" the graph. However, this benefit quickly disappears as the graphs grow bigger. A visual display of the graph begins to look like a "hairball" to the human eye. In short, graphs are particularly well suited for human communication, but come with substantial challenges.

This book studies human interaction with graph data and develops the intellectual framework to have joint query formulation and query evaluation on large and complex graphs. Whereas this book is focused only on graph data, I believe it has usability lessons that are more broadly applicable. Written by three scholars who have each made substantial contributions to this topic, this book provides the framework with which to understand these lessons and apply them elsewhere.

H. V. Jagadish
July 2018

Preface

"Sometimes you stir, sometimes you shake, and sometimes you blend. And sometimes you just serve it up neat, as it is."

Helen Rothberg, The Perfect Mix

Since the inception of database technology several decades ago, the classical paradigm of database querying has always been served "neat". A query is formulated by an end user or application and the underlying query processor is responsible for evaluating it efficiently once it is completely formulated. That is, *query formulation* and *query processing* are treated as two independent activities and they were seldom "stirred", "shaken", or "blended".

A popular approach to make formulation of database query accessible to a wider audience is to provide a visual query interface for interactive construction of queries. However, the two key enablers of visual query formulation and query processing, namely human-computer interaction (HCI) and data management, have evolved into two disparate and vibrant scientific fields, rarely making any systematic effort to leverage techniques and principles from each other towards superior realization of these efforts. Specifically, the data management community has traditionally focused on "under-the-hood" techniques such as indexing, query processing, and transactions. However, the HCI community has focused on "outside-the-hood" issues such as user task modeling, menu design models, human factors, etc. We believe that this chasm between these two vibrant fields sometimes create obstacles in providing superlative visual querying and data management services to end users. In this book, we report recent work that bridge this chasm by "blending" and "stirring" visual query formulation and processing in the context of graphs.

Graphs are used widely to model data in a wide variety of domains (e.g., biology, chemistry, ecology, social science, journalism). Consequently, it is increasingly important to devise frameworks for efficient formulation and processing of graph queries. This book presents frameworks, as they stand today, that enable visual interactions with such graph data through querying. In particular, a broad goal of this book is to emphasize the deep integration (i.e., "blending") of HCI and database query in the context of graphs. Drawing on well-founded principles in HCI and data management, we present human-data interaction paradigms in the context of graph data to address this broad goal. Note that it is reasonable to expect this picture to evolve with time.

As representative examples of graph-structured data, we utilize a large set of small or medium-sized data graphs such as chemical compounds and a single large network such as DBLP in majority part of this book. Our discussion is divided into five parts. First, we review, as accurately as possible, a spectrum of approaches to guide visual subgraph query formulation by providing high quality suggestions. The results of such suggestions facilitate end users to formulate graph queries efficiently.

Second, we review frameworks that blend visual subgraph query formulation and processing. Specifically, we introduce recent visual querying frameworks that depart from the traditional "neat" query processing paradigm by exploring a paradigm that blends query construction and query processing to prune false results and prefetch partial query results by exploiting the availability of GUI latency during visual query formulation. In particular, a pervasive desire of this book is to emphasize the deep integration of HCI and graph query processing and highlight why a panoply of traditional graph querying algorithms proposed by the database community cannot be leveraged to address this paradigm effectively.

Third, we discuss recent techniques for visual exploration of graph query results. In particular, we highlight the challenges associated with visualizing and exploring query results especially in the context of large graphs and report early efforts to mitigate these challenges.

Although comprehensive user studies are paramount for exhaustive and systematic evaluation of visual graph querying frameworks, they are expensive and time consuming to conduct. We discuss a recent technique that addresses this challenge by automatically generating many test subgraph queries having different user-specified characteristics and simulating their formulation based on different query formulation sequences without requiring human query writers. Specifically, it employs an HCI-inspired quantitative model to estimate subgraph query formulation time and an algorithm to simulate visual query formulation of the generated test queries by leveraging the quantitative model. We show an application of this technique by realizing it on top of the aforementioned visual query processing framework. In other words, this book once again emphasizes on the deep integration of HCI and database query performance evaluation.

The last topic consists of several open problems of this young field. The list presented should by no means be considered exhaustive, and is centered around challenges and issues currently in vogue. Nevertheless, readers can benefit by exploring the research directions given in this part.

This book is suitable for use in advanced undergraduate- and graduate-level courses on graph data management. It has sufficient material that can be covered as part of a semester-long course, thereby leaving plenty of room for an instructor to choose topics. An undergraduate course in algorithms, graph theory, database technology, and basic HCI should suffice as a prerequisite for most of the chapters. A good knowledge of C++/Java programming language is sufficient to code the algorithms described herein. For completeness, we have provided background information on several topics in Chapter 2: fundamental graph terminology and visual

graph query interface. The knowledgeable reader may omit this chapter, and perhaps refer back to it while reading later chapters of the book.

We hope that this book will serve as a catalyst in helping this burgeoning area at the intersection of data management and HCI (i.e., human-data interaction) grow and have practical impact.

Sourav S. Bhowmick, Byron Choi, and Chengkai Li
June 2018

Acknowledgments

It is a great pleasure for us to acknowledge the assistance and contributions of a large number of individuals. First, we would like to thank our publisher Morgan & Claypool for their support. In particular, we would like to acknowledge the efforts, help, and patience of Diane Cerra, our primary contact for this edition.

The majority of the work reported in this book grew out of the **H**uman **I**nteraction with **N**etwork Da**T**a (HINT) project at the Nanyang Technological University (NTU), Singapore and Hong Kong Baptist University (HKBU), Hong Kong. In this project, we explored issues on building frameworks that allow bridging the long-standing chasm between graph data management and HCI. Specifically, some of the chapters are published in reputable journals and conferences in the area of data management.

Prof. Shuigeng Zhou of Fudan University, who was a key collaborator for this project, deserves the first thank you. Not only did he continuously provided high quality students to crystalize several ideas in this project, he was also always willing to discuss ideas with us.

In addition, we would also like to express our gratitude to all the group members and collaborators, past and present, in our data management research group at NTU (DANTE) and the database research group at HKBU. In particular, Dr. Huey Eng Chua (NTU), Changjiu Jin (NTU), Ba Quan Truong (NTU), Ho Hoang Hung (NTU), Kai Huang (NTU), Benji Thian (NTU), Yinglong Song (NTU), Miao Xie (NTU), Peipei Yi (HKBU), Nandish Jayaram (UTA), Prof. Curtis Dyreson (Utah State University), and Prof. Jianliang Xu (HKBU) made substantial contributions to the broader aspect of our research in human interactions with graphs.

Quite a few people helped us with the initial vetting of the text for this book. It is our pleasure to acknowledge them all here. We would like to thank Sara Kreisman of Rambling Rose Press, Inc. for carefully proofreading the complete book in a short span of time and suggesting the changes which have been incorporated.

We would also like to acknowledge our parents and family members who gave their incredible support throughout the years. They were the major force behind our continuous strive to break out of the comfort zone of computer science to explore problems that are at the intersection of two or more disparate areas. It has been a great learning experience for us. A special thanks goes to Prof H. V. Jagadish (UMich, USA) for giving us the opportunity to author this book.

Finally, we would like to thank the MOE Singapore AcFR Tier 1 and Tier 2 for the generous financial support provided for the HINT project. We would also like to thank the School of Computer Science and Engineering at the Nanyang Technological University for allowing the use of their resources to help complete the book. The work done at the Department of Com-

puter Science at HKBU is partially supported by HK-RGC General Research Funds (GRFs) 12201315 and 12232716. We would also thank the Faculty of Science at HKBU for the Faculty Research Grant II FRG2/16-17/093.

Sourav S. Bhowmick, Byron Choi, and Chengkai Li
June 2018

CHAPTER 1

Introduction

Graph is an extensively studied subject in mathematics and many areas of computer science as it provides a natural way of modeling data in a wide variety of domains. For example, in chem-informatics, graphs are used to represent atoms and bonds in chemical compounds. In bioinformatics, protein interaction networks are graphs where nodes represent molecules and edges represent interactions between them. The Semantic Web represents information on the Web as a graph with a set of entities and explicit relationships. Even the relational data can be modeled as a graph, where each tuple is a node and there is an edge between two nodes if their corresponding tuples have foreign key relationships. Therefore, it is evident that graph data stores are growing rapidly in size in a wide spectrum of applications. As a result, there is a critical need for user-friendly, efficient, and scalable frameworks that enable end users to interact with these growing graph data stores effectively.

1.1 INTERACTION WITH GRAPHS USING QUERIES

Querying graphs has emerged as an important research problem for real-world applications that are centered on large graph data as it provides an effective means to interact with such data by searching for relevant information [Yan et al., 2004]. Querying graph data involves two key steps: *query formulation* and *efficient processing* of the formulated query. A number of declarative query languages have been proposed for graphs which can be used to formulate a query in textual form. For instance, PQL [Leser, 2005] is a pathway query language designed for biological networks. SPARQL[1] is a graph query language designed for querying RDF data. *Cypher* is another graph query language proposed by Neo4j.[2]

All these languages assume that a user has programming and debugging expertise to formulate queries correctly. Unfortunately, this assumption makes it harder for non-programmers to take advantage of a graph querying framework as it requires significant time and effort to learn these languages. Consequently, in many real life domains (e.g., life sciences, drug discovery) it is unrealistic to assume that users are proficient in expressing such queries textually.

Example 1.1 Consider a graph data store containing chemical compounds such as *Pubchem*.[3] Hazel, a chemist, wishes to search for compounds that contain the structure shown in Figure 1.1a. In order to realize her search intention, she needs to formulate a query using a graph

[1] https://www.w3.org/TR/rdf-sparql-query/
[2] https://neo4j.com/developer/cypher-query-language/
[3] https://pubchem.ncbi.nlm.nih.gov/

query language such as *Cypher*. Figure 1.1b shows the corresponding query in *Cypher*. Unfortunately, although Hazel is an expert chemist, she is a non-programmer and cannot be expected to learn the complex syntax of *Cypher* in order to formulate meaningful queries.

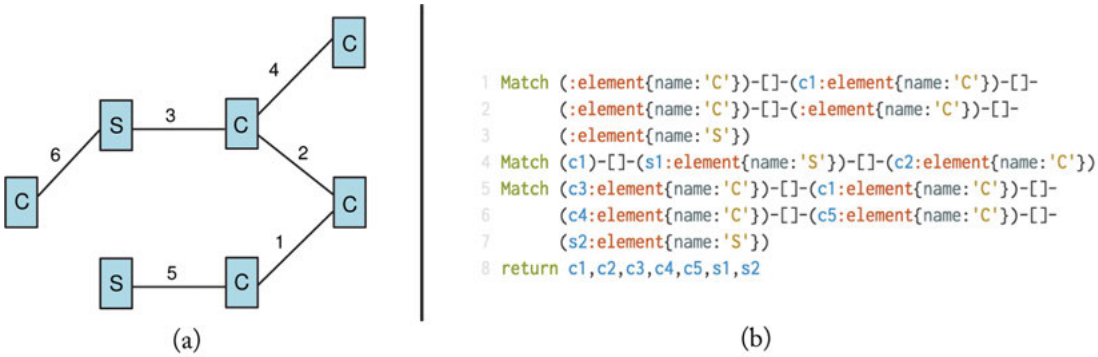

Figure 1.1: Query formulation using *Cypher*.

> *With the democratization of data management, most end users will not learn query languages; rather such language is notation for professional programmers.*

It is paramount for designers of graph querying frameworks to understand the aforementioned resistance to query languages as a signal to consider alternative query paradigms, without which we will see the erosion of users' ability to influence the design and usage of such interaction frameworks that surround them.

1.2 GRAPH QUERY CONSTRUCTION USING VISUAL INTERFACES

Fortunately, unlike SQL, graph queries are more intuitive to draw than to compose them in textual format. Hence, a popular approach to make the formulation of graph queries accessible to non-programmers is to provide a user-friendly visual query interface (GUI) for interactive construction of queries. For instance, *Pubchem* provides a visual query interface for structure-based chemical compound search.[4] Similarly, *Drugbank* allows end users to draw a subgraph search query using a visual interface.[5] Interestingly, both these data sources do not expose interfaces to formulate textual queries using a graph query language, highlighting the reluctance of end users to use such programming languages. In fact, this idea that visual representations—rather

[4] https://pubchem.ncbi.nlm.nih.gov/search/search.cgi
[5] https://www.drugbank.ca/structures/search/small_molecule_drugs/structure

than text—can facilitate the understanding and creation of queries, or make them palatable to non-programmers is several decades old. In the context of graph queries, a GUI for graph query formulation is usually composed of several panels, such as a panel to display the set of labels or attributes of nodes or edges of the underlying data graphs, a panel to construct a query graphically, and a panel to view query results. A user can simply drag-and-drop various components of the GUI to formulate a query.

Example 1.2 Reconsider Hazel's problem in Example 1.1. Suppose she is provided with a visual query interface as depicted in Figure 1.2. Then she can draw her query in *Panel 3* by simply dragging the relevant attributes in *Panel 2* to create nodes and clicking on a pair of nodes to create an edge. That is, Hazel can now formulate her query without requiring the knowledge of a graph query language.

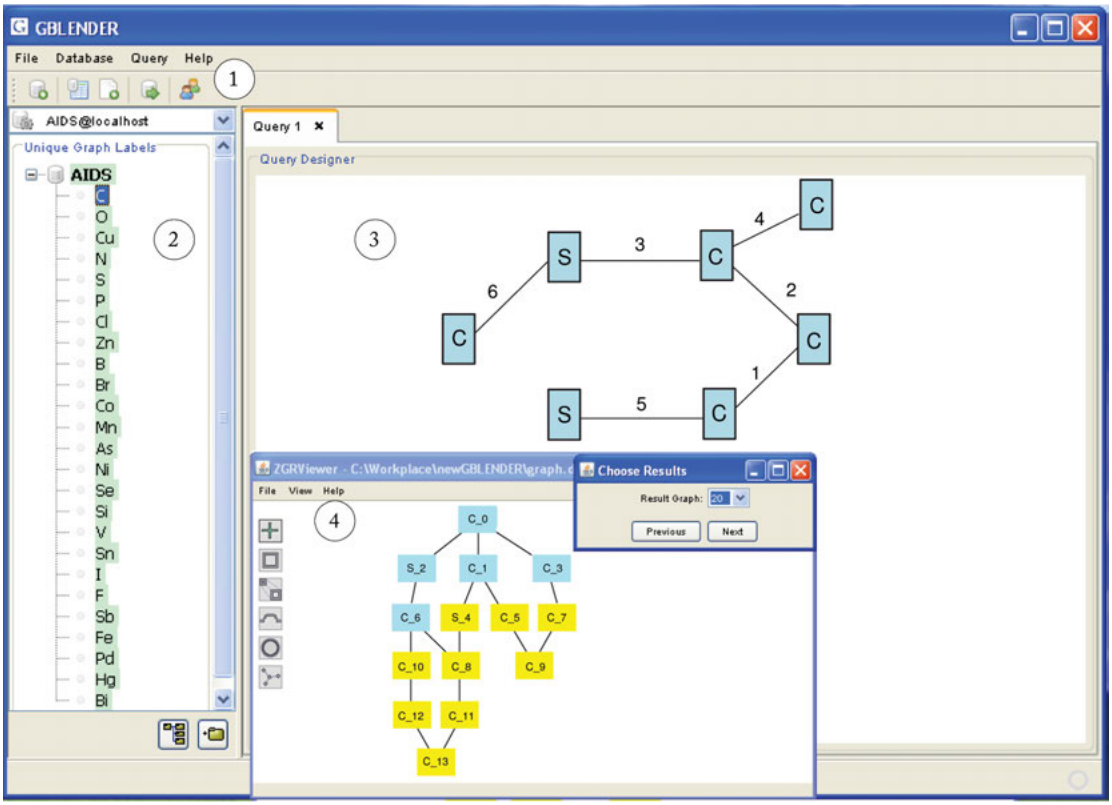

Figure 1.2: Visual interface for formulating graph queries in Jin et al. [2012].

In traditional visual querying framework, once visual construction of a query is completed (by clicking on the Run icon), it may be internally translated into corresponding textual query

using a graph query language (e.g., *Cypher*) and then executing by the underlying query processor. The result matches of the query is then returned to the user and depicted in the visual query interface (e.g., *Panel 4* in Figure 1.2). In fact, this visual querying paradigm is not only limited to graphs but also adopted for other types of data such as XML [Braga, Campi, and Ceri, 2005, Comai, Damiani, and Fraternali, 2001] to enhance usability of databases.

1.3 INTEGRATION OF VISUAL QUERY INTERFACE AND QUERY ENGINE

In a classical visual graph querying framework, the visual query interface (GUI) is "loosely coupled" with the underlying query engine. Typically, a visual query interface is designed and implemented by leveraging principles from the human-computer interaction (HCI) area to enhance its usability. On the other hand, the query engine is realized using data management principles to ensure efficient and scalable execution of graph queries. Seldom is there any meaningful interaction between these two components *concurrently*. Consequently, when an end user is visually formulating a graph query, the underlying query engine remains idle as the human interactions at the GUI level are rarely communicated to the query engine. The query engine is only invoked when the complete query has been formulated and the `Run` icon is clicked to execute it. For instance, reconsider Example 1.2. Hazel's interactions with the GUI in Figure 1.2 during visual query construction are transparent to the underlying graph query engine. Only when Hazel clicks the `Run` icon is the constructed query shipped to the query engine for efficient processing. In other words, visual query formulation and query processing occur in a *sequential* fashion, as shown in Figure 1.3a. Note that this traditional query paradigm (visual or textual) is not limited to graphs but also prevalent in other types of querying systems (e.g., relational, XML). We refer to this loose coupling of these two key components of a visual graph querying framework as *shallow integration*. The advantage of such shallow integration is that any state of the art graph querying engine can be combined with any superior visual graph query interface with minimum effort.

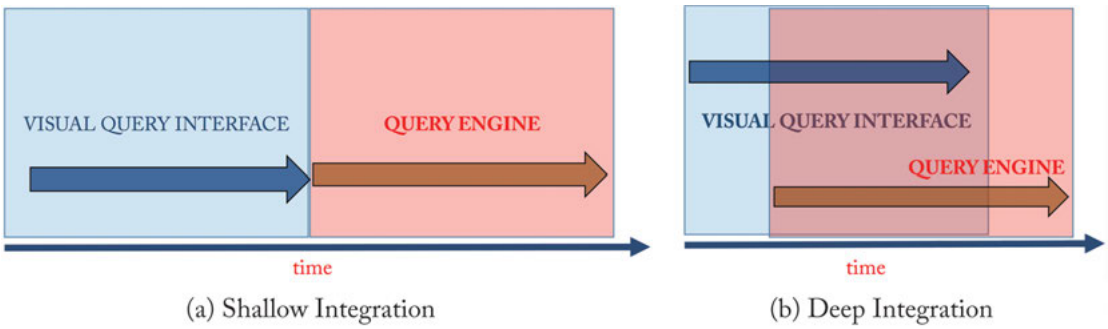

Figure 1.3: Shallow vs. deep integration.

We advocate that the shallow integration of visual query interface and query engine is primarily due to the fact that the two key enablers of these efforts, namely HCI and database management, have evolved into two disparate and vibrant scientific fields, rarely making any systematic effort to leverage techniques and principles from each other toward superior realization of these efforts. Specifically, data management researchers and practitioners have traditionally focused on "under-the-hood" techniques such as indexing, query processing, and transactions. On the hand, the HCI community has focused on "outside-the-hood" issues such as user task modeling, menu design models, human factors, etc. Data management community often have a tendency to avoid outside-the-hood challenges with HCI flavors whereas the HCI community are traditionally reluctant to look at under-the-hood challenges that may influence the way they build visual interfaces among others.

The visual graph query formulation process demonstrates two key characteristics. First, a query is gradually (i.e., incrementally) exposed to the underlying query engine during its construction. Second, it gives rise to *GUI latency* (i.e., the time spent by a human to complete certain query formulation task such as construction of an edge). These features provide opportunities to the underlying query engine during visual query formulation to provide superlative visual querying and data management services to end users. For instance, a graph query engine can provide valuable suggestions, feedbacks, and guidance to an end user during query formulation by utilizing the partial query drawn by her [Bhowmick et al., 2015, Yi et al., 2017a]. This can potentially enhance the usability of a visual graph querying framework. Similarly, it can utilize the partial query information along with the GUI latency to retrieve candidate matches containing the query fragment and refine them iteratively as the user adds more nodes and edges to her query [Bhowmick, Choi, and Zhou, 2013]. This cannot only improve the query response time of the framework [Hung et al., 2014, Jin et al., 2012] but also pave the way for *exploratory search* [Huang et al., 2017]. However, due to the aforementioned chasm between these two vibrant fields, the shallow integration paradigm of classical visual graph querying framework fails to exploit such opportunities efficiently.

We advocate that "tight coupling" between visual graph query interface and query engine components is paramount to exploit the above opportunities. Instead of the query engine being oblivious to the human interactions in the GUI during visual query formulation, it should "track" these interactions by communicating with the visual query interface and process them judiciously during query formulation by exploiting the GUI latency. In other words, activities of the query engine and visual query interface components should be *blended* (i.e., interleaved), as shown schematically in Figure 1.3b. We refer to this tight coupling of the visual query interface and query engine as *deep integration*.

Deep integration of visual graph query interface and query engine paves the way toward superior query response time and usability of graph query systems.

In this book, we report recent efforts that question the traditional reluctance of tight coupling between visual graph query interfaces and underlying graph query engines. Specifically, we describe work on deep integration between these two components to realize the vision of bridging the long-standing chasm between traditional data management and HCI in the context of graph querying. As we shall see later, realization of deep integration entails significant rethinking of several long-standing strategies for database query formulation and processing.

1.4 OVERVIEW OF THIS BOOK

Although research on visual querying schemes dates back to the relational database era [Catarchi et al., 1997], deep integration of visual query interface and query processing engine was first explored by Bhowmick and Prakash [2006], Truong and Bhowmick [2013] and Zhou et al. [2009] in the context of XML. In this work, visual formulation of XML queries are interleaved (i.e., blended) with their processing, resulting in superior *system response time* compared to shallow integration-based classical XML querying frameworks. This book focuses on work that realize deep integration in the context of graphs. That is, it reports a variety of visual graph querying techniques that facilitate human interaction with graph through deep integration between the visual graph query interface and the underlying query processing engine. To this end, we organize the discussions in this book as follows.

- In Chapter 2 we present the elements that serve as background for the remaining chapters of the book. In particular, we focus on fundamental ingredients of a visual graph querying framework. We discuss relevant terminologies related to graphs that are necessary to understand graph querying. We also discuss the components and associated terminologies of a typical visual graph query interface.

- In Chapter 3 we discuss techniques that guide users in visual query formulation. Specifically, we report recent work that leverage on partially constructed query information during query formulation to present suggestions to end users toward completion of the query. These efforts realize deep integration between the visual query interface and underlying query engine by generating data-driven suggestions while a graph query is being visually formulated.

- In Chapters 4 and 5 we describe a unified framework that utilizes partial query information and GUI latency to realize deep integration by blending visual graph query formulation with its processing. Specifically, we review work that instantiate this framework to two categories of data graphs: a large collection of small- or medium-sized data graphs (e.g., chemical compounds) and a large network.

- A visual graph querying framework should support rich exploration and visualization of result matches of graph queries in order to facilitate superior human interactions with underlying data graphs. In fact, query results exploration is only the beginning of the

sensemaking process for an end user. In Chapter 6 we review seminal collection of work for facilitating exploration and visualization of graph query results.

- In a shallow integration-based visual querying framework, performance study of a large number of graph queries can be easily undertaken by automatically extracting a random collection of subgraphs from the underlying data and executing them. However, a similar study for various deep integration-based graph querying techniques demands exhaustive user study due to the blending of visual query formulation and query processing activities. Such study is expensive and time-consuming. In Chapter 7, we present a framework that draws upon the literature in HCI and graph data management to simulate visual subgraph query construction process, which enables us to automate performance study in deep integration environment without requiring human users.

- Finally, in Chapter 8 we summarize the contributions of this book and list down interesting open research problems in this burgeoning area of human interaction with graphs.

1.5 SCOPE

In summary, the scope of this book is as follows.

- Human interaction with graphs can be broadly classified into *by-query*, *by-example*, and *by-analytics*. In this book, we focus on visual query-based interaction. We assume that humans play the central role in interacting with data graphs through visual subgraph search. Hence, human interaction with graph data through visual graph analytics or visual examples is beyond the scope of this book.

- Since our focus is on *deep integration* of visual graph query interface and the query engine, we do not discuss traditional graph query evaluation techniques [Cheng et al., 2007, Han et al., 2013, Yan et al., 2005, Yang et al., 2014, Zhu et al., 2012] that embody shallow integration, i.e., classical graph query processing techniques that can be easily incorporated in a visual graph querying framework by simply building a visual query interface on top of it (Figure 1.3a).

- In subsequent chapters, we assume that the data graphs are stored in a single machine. That is, we do not discuss visual graph querying in distributed graph environment. This is primarily because, to the best of our knowledge, there is no existing work that realizes deep integration between visual graph query interface and distributed graph query engine.

CHAPTER 2

Background

This chapter provides an overview of key topics that serve as background for the rest of the book. First, we discuss relevant terminology related to graphs. This is followed by a brief discussion on visual graph query interfaces and associated terminology. Table 2.1 shows the key symbols related to graphs that we will be using throughout this book.

Table 2.1: Key symbols related to graphs used in this book

Symbol	Definition
$\mathcal{D}$	A graph database
g, G	A (sub)graph
q, Q	A query graph (fragment) or a subgraph query
$\mathcal{D}_g$	A set of FSGs of g
$fsgIds(g)$	Set of identifiers of the data graphs in $\mathcal{D}_g$
$\mathcal{I}$	Set of infrequent fragments in $\mathcal{D}$
dif_i	A discriminative infrequent fragment (DIF)
inf_i	A non-discriminative infrequent fragment (NIF)
$\mathcal{I}_d$	A set of DIFs in $\mathcal{D}$
α	Minimum support threshold

2.1 GRAPH TERMINOLOGY

We denote a graph as $G = (V, E)$, where V is a set of nodes and $E \subseteq V \times V$ is a set of (directed or undirected) edges. We refer to the nodes and edges of G as $G.V$ and $G.E$ (simply V and E when the context is clear), respectively. A node in $G = (V, E)$ has an identifier j and is referred to as $v_j \in V$. Nodes and edges can have labels as attributes specified by mappings $\phi : V \rightarrow \sum_{V_\ell}$ and $\psi : E \rightarrow \sum_{E_\ell}$, respectively, where $\sum_{V_\ell}$ is the set of node labels and $\sum_{E_\ell}$ is the set of edge labels. In this book, we assume that G (data or query graph) is a connected graph with at least one edge. The *size* of G is defined as $|G| = |E|$. For ease of presentation, we present data graphs and visual subgraph queries using undirected graphs with labeled nodes. However, the techniques

discussed in this book can be easily extended to directed graphs. We also focus on *simple* graphs, i.e., undirected graphs with neither self-loops nor multiple edges.

Given a graph $G = (V, E)$, let M be the adjacency matrix of edges of G. Then, the CAM code of G [Huan et al., 2003], denoted as $cam(G)$ is the sequence formed by concatenating lower triangular entries of M (including entries on the diagonal) in the order from top to bottom and from the leftmost entry to the rightmost entry.

2.1.1 SUBGRAPH ISOMORPHISM-RELATED TERMINOLOGY

Given two vertex-labeled graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the problem of subgraph isomorphism is to find a 1-1 mapping from V_1 to V_2 such that each vertex in V_1 is mapped to a distinct vertex in V_2 with the same label, and each edge in E_1 is mapped to an edge in E_2.

Definition 2.1 (**Subgraph Isomorphism**) *Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, there exists a **subgraph isomorphism** from G_1 to G_2, denoted by $G_1 \subseteq G_2$, if there exists an injective function $f : V_1 \rightarrow V_2$, such that (1) $\forall u \in V_1$, $\phi_1(u) = \phi_2(f(u))$, and (2) $\forall (u, v) \in E_1$, $(f(u), f(v)) \in E_2$ and $\psi_1(u, v) = \psi_2(f(u), f(v))$.*

A graph $G_1 = (V_1, E_1)$ is a *subgraph* of another graph $G_2 = (V_2, E_2)$ (or G_2 is a *supergraph* of G_1) if there exists a subgraph isomorphism from G_1 to G_2, i.e., $G_1 \subseteq G_2$ (or $G_2 \supseteq G_1$). We may also simply say that G_2 *contains* G_1. The graph G_1 is called a *proper subgraph* of G_2, denoted as $G_1 \subset G_2$, if $G_1 \subseteq G_2$ and $G_1 \not\supseteq G_2$. Lastly, let $G_1 \subseteq G_2$ and $|G_1| = |G_2| - 1$. Then, we refer to G_1 and G_2 as *parent* and *child* graphs, respectively.

Multiple subgraph isomorphic *embeddings* of G_1 in G_2 may exist. Therefore, subgraph isomorphism of G_1 and G_2 can also be viewed as a relation between G_1 and G_2, where each record is an embedding of G_1 in G_2. Furthermore, we may use *graph isomorphism* of G_1 and G_2 if $G_1 \subseteq G_2$ and $|V_1| = |V_2|$, and *graph automorphism* of G_1, which is a graph isomorphism to itself.

2.1.2 TYPES OF GRAPH COLLECTION

There are two different streams of research for processing subgraph queries in the literature that are based on the type of data graph collection. One stream handles a large number of small- or medium-sized data graphs such as chemical compounds. Several existing techniques belong to this stream [Cheng et al., 2007, Lin et al., 2012, Mongiovi et al., 2010, Shang et al., 2010, Yan et al., 2004, 2005]. The other stream handles a single large data graph such as citation network and protein-protein interaction network [Fan et al., 2010, Han et al., 2013, Khan and Elnikety, 2014, Wu et al., 2013, Xie and Yu, 2011, Zhang et al., 2010, Zhu et al., 2012]. In this book, we discuss human-graph interaction techniques that are devised for these two streams of data graphs.

2.1.3 FREQUENT AND INFREQUENT FEATURES

Next, we introduce some features associated with a set of data graphs that we shall be utilizing in the sequel. Given a graph database $\mathcal{D}$ containing a set of small- or medium-sized data graphs, we assign a unique identifier (i.e., id) to each data graph in $\mathcal{D}$. A data graph G with id i is denoted as G_i. Let g be a subgraph of $G_i \in \mathcal{D}$ $(0 < i \leq |\mathcal{D}|)$ and has at least one edge. Then, g is a *fragment* in $\mathcal{D}$. Informally, we use the term *fragment* to refer to a small subgraph in a data graph or a query graph. Given a fragment $g \subseteq G$ and $G \in \mathcal{D}$, G is referred to as the *fragment support graph* (FSG) of g [Jin et al., 2010, 2012]. We denote the set of FSGs of g as $\mathcal{D}_g$. We refer to $|\mathcal{D}_g|$ as (*absolute*) *support*, denoted by $sup(g)$. We denote the set of identifiers of the data graphs in $\mathcal{D}_g$ as $fsgIds(g)$. Note that in this book we shall refer to a fragment in a query graph as *query fragment* in order to distinguish it from a fragment in a data graph.

A fragment g is *frequent* if its support is no less than $\alpha|\mathcal{D}|$ where α is the minimum support threshold. That is, if $g \in \mathcal{D}$ and $sup(g) \geq \alpha|\mathcal{D}|$ where $0 < \alpha < 1$ then g is a *frequent* fragment in $\mathcal{D}$. We denote the set of frequent fragments in $\mathcal{D}$ as $\mathcal{F}$. For example, let $|\mathcal{D}| = 10000$ and $\alpha = 0.1$. Then, all the fragments with support larger than or equal to 1000 are frequent fragments. The fragments $f_0 - f_6$ in Figure 2.1 are frequent fragments (support values are shown in parenthesis). We refer to a frequent fragment g as *frequent edge* if $|g| = 1$. For example, f_0 and f_1 are frequent edges in Figure 2.1. On the other hand, if $sup(g) < \alpha|\mathcal{D}|$ then g is an *infrequent* fragment. We denote the set of infrequent fragments in $\mathcal{D}$ as $\mathcal{I}$. Specifically, we classify infrequent fragments into two types, *discriminative* and *non-discriminative* [Jin et al., 2010, 2012]. Given $g \in \mathcal{I}$, let $\mathbb{S}(g)$ be the set of all subgraphs of g. If $\mathbb{S}(g) \subset \mathcal{F}$ or $|g| = 1$, then g is a *discriminative infrequent fragment* (DIF) in $\mathcal{D}$. We denote a set of DIFs in $\mathcal{D}$ as $\mathcal{I}_d$. Likewise, we refer to an infrequent fragment that is not a DIF as *non-discriminative infrequent fragment* (NIF). For example, consider Figure 2.1 that depicts a list of frequent and infrequent fragments in a chemical compound database. Observe that as all the subgraphs of dif_0 are frequent fragments, dif_0 is a DIF. Due to the same reasons, dif_1 and dif_2 are DIFs. Observe that dif_0 (as an infrequent fragment) is a subgraph of inf_0 and inf_2. Hence, inf_0 and inf_2 are NIFs but not DIFs. It is easy to observe that an infrequent fragment may contain more than one DIFs.

Lemma 2.2 *Let $g' \in \mathcal{I}_d$ and $g \in \mathcal{D}$. If $g' \subset g$ then $g \in \mathcal{I}$.*

Proof. Since $g' \in \mathcal{I}_d$, $|\mathcal{D}_{g'}| < \alpha|\mathcal{D}|$. Also as $g' \subset g$, $|\mathcal{D}_g| \leq |\mathcal{D}_{g'}|$. Therefore, $|\mathcal{D}_g| \leq |\mathcal{D}_{g'}| < \alpha|\mathcal{D}|$. Hence, g is an infrequent fragment. □

On the other hand, if a graph is an infrequent fragment, then it must contain at least one DIF.

Lemma 2.3 *Given $g \in \mathcal{I}$, $\exists g' \in \mathcal{I}_d$ such that $g' \subseteq g$.*

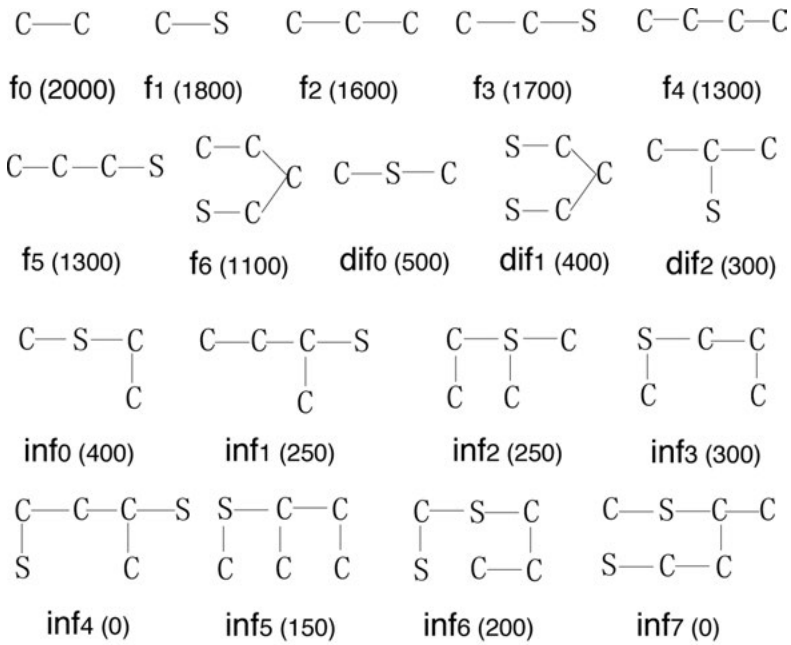

Figure 2.1: Examples of frequent and infrequent fragments.

Proof. If $\nexists\ (g' \subset g$ and $g' \in \mathcal{I}_d)$, then $\mathbb{S}(g) \subset \mathcal{F}$. Consequently, g itself is a DIF. Hence, the above lemma holds. □

Next, we define the notion of *largest subgraph*. Let $g' \subset g$ and $|g'| = |g| - 1$. Then, g' is the largest subgraph of g. Note that g can have more than one largest subgraphs. We denote a set of largest subgraphs of g as $Lsub(g)$. If all the largest subgraphs of an infrequent fragment are frequent fragments, then the infrequent fragment must be a DIF.

Theorem 2.4 *Given $g \in \mathcal{I}$, if $Lsub(g) \subset \mathcal{F}$, then $g \in \mathcal{I}_d$.*

Proof. Assume that $g \in \mathcal{I}$ but $g \notin \mathcal{I}_d$. Then, $\exists g' \subset g$, $g' \in \mathcal{I}_d$ (based on Lemma 2.3). Hence, $\exists g'' \in Lsub(g)$, such that $g' \subseteq g''$. Therefore, $g'' \in \mathcal{I}$ (based on Lemma 2.2), which contradicts $Lsub(g) \subset \mathcal{F}$. Therefore, g is a DIF. □

Theorem 2.5 *Given a graph database $\mathcal{D}$ and user-defined support threshold α, $|\mathcal{I}_d| < |\mathcal{I}|$.*

Proof. Let k-edge fragment be a graph fragment with k edges. Let $\mathcal{I}_{d_1}$ be the set of 1-edge DIFs. Then the 2-edge DIFs are infrequent fragments with two edges (denoted as $\mathcal{I}_2$) that do not contain any subgraph in $\mathcal{I}_{d_1}$. Therefore, the number of 2-edge DIFs is: $|\mathcal{I}_{d_2}| = |\overline{\mathcal{I}_{d_1} \cap \mathcal{I}_2}|$. Then, the number of k-edge DIFs can be expressed as follows:

$$|\mathcal{I}_{d_k}| = |\overline{(\mathcal{I}_{d_1} + \mathcal{I}_{d_2} \ldots + \mathcal{I}_{d_{k-1}}) \cap \mathcal{I}_k}| < |\mathcal{I}_k|. \tag{2.1}$$

The number of DIFs is given as follows:

$$|\mathcal{I}_d| = \sum_{n=1}^{M} |\mathcal{I}_{d_i}| < \sum_{n=1}^{N} |\mathcal{I}_i| = |\mathcal{I}|, \tag{2.2}$$

where M and N ($M \leq N$) are the maximum sizes of frequent fragment and DIFs, respectively. □

Based on the above discussion, it follows that if one of the subgraphs of g is a DIF, g is an infrequent fragment. Therefore, a DIF plays a central role in the formation of infrequent fragment and can be used in turn to identify an infrequent fragment. In practice, the number of DIFs is significantly smaller than the total number of infrequent fragments.

2.2 VISUAL GRAPH QUERY INTERFACE

Recall from the preceding chapter, a popular approach to make formulation of graph query accessible to non-programmers is to provide a visual query interface (a.k.a GUI) for interactive construction of queries. In this book, we assume that a user interact with data graphs through such a visual query interface. Here, we first briefly describe the structure of a visual graph query interface that we consider in the subsequent chapters. Next, we briefly describe related terminology and concepts that we shall utilize in the sequel.

2.2.1 STRUCTURE OF VISUAL GRAPH QUERY INTERFACES

In recent times, there has been increasing efforts to create user-friendly visual graph query interfaces from academia [Chau et al., 2008, Haag et al., 2014, Jin et al., 2012, Pienta et al., 2016, Yang et al., 2014] and industry (e.g., *PubChem*,[1] *eMolecule*,[2] *DrugBank*[3]) to ease the burden of query formulation. In the context of subgraph queries, a GUI for graph query formulation is usually composed of several panels, such as a panel to display the set of labels or attributes of nodes or edges of the underlying data graph(s), a panel to construct a subgraph query graphically, and a panel to view query results. Figures 1.2 and 2.2 depict two examples of screenshots of GUIs for subgraph query formulation in Jin et al. [2012] and *DrugBank*, respectively. Several of these interfaces are comprised of at least the following three key panels.

[1] https://pubchem.ncbi.nlm.nih.gov/edit2/index.html?cnt=0
[2] https://www.emolecules.com/
[3] https://www.drugbank.ca/structures/search/small_molecule_drugs/structure

Figure 2.2: Visual interface for formulating queries in *Drugbank*.

1. An *Attribute Panel* to display a set of labels or attributes of nodes or edges of the underlying data. In this book, for simplicity we assume that this panel consists of a set of node or edge labels[4] (e.g., Panel 2 in Figure 1.2).
2. A *Query Panel* for constructing a graph query graphically by adding a node or edge iteratively (e.g., Panel 3 in Figure 1.2).
3. A *Results Panel* that displays the query results (e.g., Panel 4).

2.2.2 VISUAL GRAPH QUERY FORMULATION

Given the aforementioned structure of a visual graph query interface, a query can be constructed by performing the following sequence of steps.

1. Move the mouse cursor to the *Attribute Panel*.
2. Scan and select a label (e.g., label `C`).

[4]Attributes associated with nodes or edges of multi-attribute graphs can be easily added in the panel.

3. Drag the selected item to the *Query Panel* and drop it. Each such action represents formulation of a single node in the query graph.
4. Repeat, if necessary, Steps 1–3 for constructing another node.
5. Construct edges between relevant nodes in the constructed subgraphs by clicking on them.
6. Repeat Steps 4 and 5 until the complete query graph is formulated.

Note that some GUIs expose a set of *canned patterns* (e.g., Benzene ring for querying chemical compounds) to aide query formulation. An example is shown in Figure 2.2. However, in this book we do not assume that such canned patterns are utilized during visual query formulation. That is, we assume that queries are formulated using an *edge-at-a-time* approach as described above, unless specified otherwise.

Recently, some visual graph query interfaces support *Graph Query By Example* (GQBE) such as Jayaram et al. [2013]. GQBE allows users to provide some *entity tuples* as input and discovers the query graph of the input tuples. Here the query graph is a carefully defined structured called *maximum query graph* (MQG) where the edges are associated with weights to capture the number of edges incident to the query entity tuples and the size of neighborhood (related entities). The answer graphs of the MQG can be extended to answer tuples, which consist of answer entities corresponding to the input entities. Figure 2.3 depicts an example of GQBE in Jayaram et al. [2013]. The input entity is "*Donald Kruth*." Part of the MQG is shown in the right-hand side of the figure. On the left-hand side of the user interface, GQBE shows other matching answer entities. Users may explore the knowledge graph by selecting some entities and retrieve more related entities. For example, the users may select the first two answers and can obtain the two matching subgraphs from the knowledge graph, shown in Figure 2.4. As mentioned in the preceding chapter, in this book we do not consider example-based interaction with graphs.

2.2.3 QUERY FORMULATION-RELATED TERMINOLOGY

GUI Actions. We first introduce a set of GUI *actions* (*actions* for brevity) that a user takes to formulate a query graph (a.k.a subgraph query) using the aforementioned type of visual graph query interface. The set of fundamental GUI actions considered in this book are $Action = \{\mathsf{add}, \mathsf{modify}, \mathsf{run}\}$, where each action corresponds to a common user's action. For better readability, we illustrate these few essential actions and provide their narrative definitions.

- $\mathsf{add}(q,s)$: The add action denotes a user manually adding an edge or a node s to an existing query graph q, and returns the augmented query. It is typical that a user adds an edge to q in edge-at-a-time query formulation mode.[5]
- $\mathsf{modify}(q,s)$: This action denotes that a user revokes (deletes) an edge or a node s, and returns the modified query graph.

[5]If a GUI exposes a set of canned patterns for query formulation, then the add action can be easily extended to support it. Specifically, a canned pattern can be represented as a set of edges and hence $\mathsf{add}(q,S)$ simply adds a set of edges S to q.

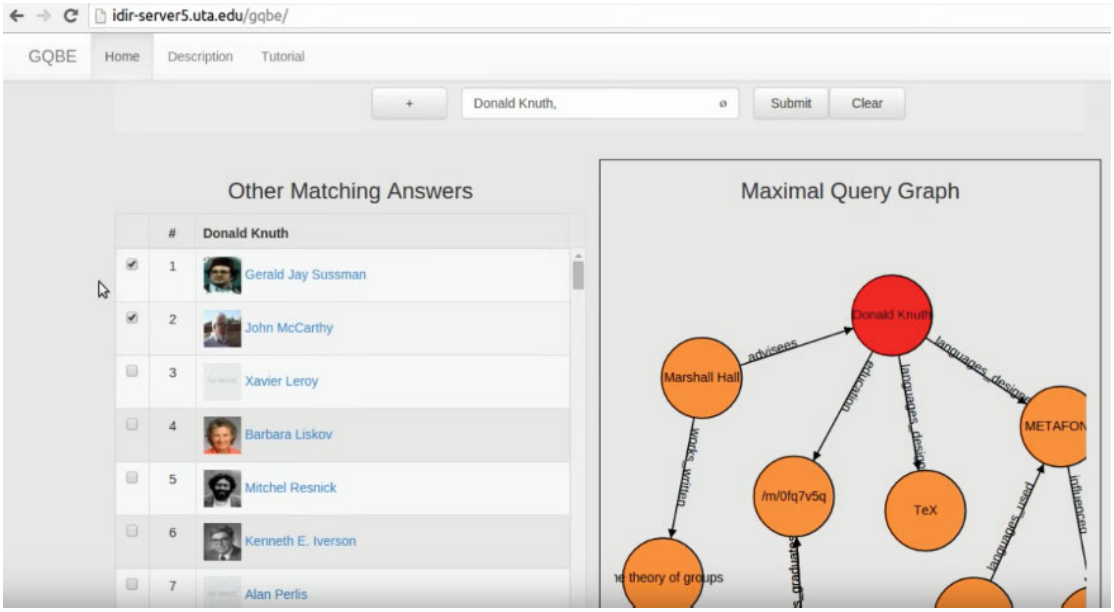

Figure 2.3: The GQBE interface [Jayaram et al., 2013].

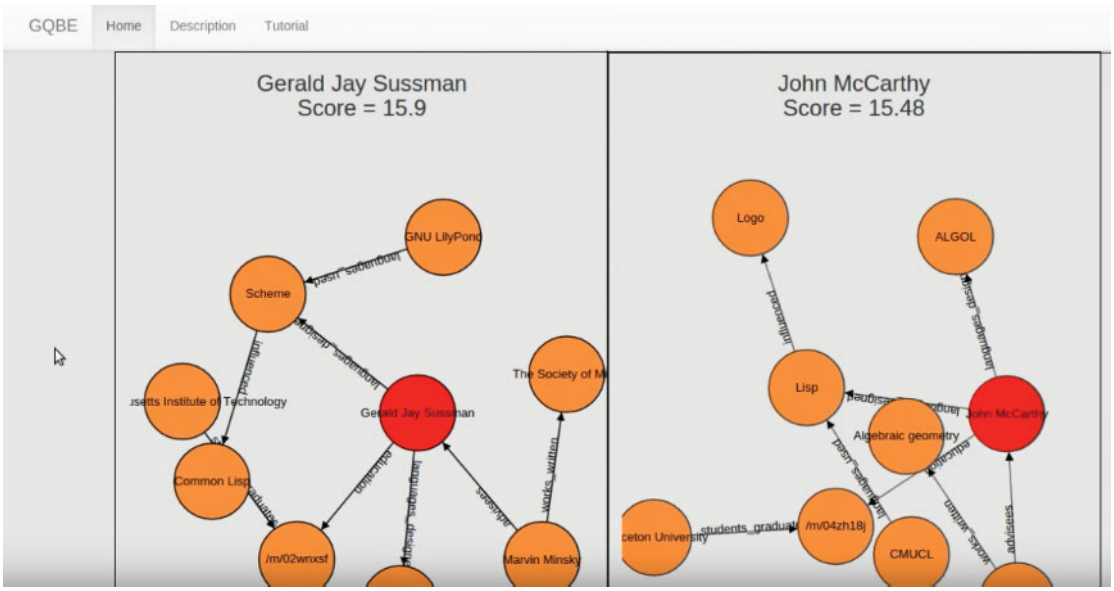

Figure 2.4: The GQBE interface (continued).

- run(q): The run action models the execution of the visually formulated query by clicking on the Run icon (or equivalent of Run) in the GUI. Note that the run action does not change the structure of the query, unlike the previous actions.

We remark that we do not model low-level actions (e.g., mouse click, mouse hover, drag-and-drop) as different GUIs may follow different sequence of low level actions to realize the aforementioned three GUI actions. For example, in one GUI addition of a node may be simply realized by right clicking on an empty space of the *Query Panel* followed by addition of the label whereas in another interface this action may require selection of a node label from the *Attribute Panel* and dragging-and-dropping it on the *Query Panel*. Hence, we model GUI actions at a higher level of abstraction.

Query Formulation Sequence (QFS). A *query formulation sequence* (QFS) is a sequence of GUI actions, which specifies how a user formulates a query graph. For example, Figure 2.5 shows two QFSs for formulating the query graph depicted in Figure 1.2. Specifically, both the sequences are sequence of six add actions (in different order) followed by the run action to execute the query. Note that typically the run action is the last action for query formulation. However, as we shall see in Chapter 6, it may be interleaved with other actions during *exploratory search*.

GUI Latency. We use the term *GUI latency* to refer to the time taken to construct an edge of a query visually. That is, it is the time taken to perform the add action. For example, the GUI latency available at Step 2 in *Sequence 1* (Figure 2.5) is the time taken to construct the edge C–C. As we shall see later, this latency will be exploited to process partially formulated subgraph queries.

2.3 SUMMARY

This chapter can be summarized as follows.

- A fragment in a data graph is frequent if its support is no less than $\alpha|\mathcal{D}|$ where α is the minimum support threshold. Otherwise, it is an infrequent fragment. We consider two types of infrequent fragments, namely, discriminative and non-discriminative infrequent fragments.

- Visual graph query interfaces (GUI) enable end users to formulate a query graph simply by drawing it without the need to know any graph query languages. Due to its ease of use, there have been increasing number of such visual query interfaces introduced by academia and industry for querying graph data.

- Interactions of users with a GUI for query formulation can be modeled as a set of GUI actions. A query graph can be visually formulated by following different query formulation sequences (a set of GUI actions). A GUI action in a formulation sequence gives rise to GUI latency that can be exploited judiciously to process partial query fragments.

Steps	Sequence 1			Sequence 2		
	Action	Current Graph	Status	Action	Current Graph	Status
Step 1	C—C	C-1-C	Frequent	C—S	C-1-S	Frequent
Step 2	C—C	C-1-C-2-C	Frequent	S—C	C-1-S-2-C	Infreq
Step 3	C—S	C-1-C-2-C-3-S	Frequent	C—C	C-1-S-2-C-3-C	Infreq
Step 4	C—C	C-1-C-2-C(-3-S)(-4-C)	Infreq	C—C	C-1-S-2-C(-3-C)(-4-C)	Infreq
Step 5	C—S	S-5-C-1-C-2-C(-3-S)(-4-C)	Similar	C—C	C-1-S-2-C(-3-C)-4-C-5-C	Infreq
Step 6	C—S	C-6-S-3-C(-4-C)-2-C-1-C-5-S	Similar	C—S	C-1-S-2-C(-3-C)-4-C-5-C-6-S	Similar
Step 7	Click RUN	C-6-S-3-C(-4-C)-2-C-1-C-5-S	Verify	Click RUN	C-1-S-2-C(-3-C)-4-C-5-C-6-S	Verify

Figure 2.5: Query formulation steps.

CHAPTER 3

Guidance for Visual Query Formulation

The first step to facilitate human interaction with a graph repository in a visual environment is to enable visual formulation of search queries. In the preceding chapter, we introduced the *edge-at-a-time* approach to formulate visual subgraph queries. Recall that in this approach a query is incrementally constructed by adding one edge at a time. Consequently, it may be time consuming to formulate a query with a large number of edges using this approach. For instance, if a query contains a benzene ring, then a user needs to draw six edges iteratively to formulate it. One way to address this challenge is to provide a list of patterns (i.e., subgraphs) that can be utilized to formulate a query. These patterns can be generated in a *static* or *dynamic* mode. In the former case, the set of patterns is provided *a priori*, typically fixed, and are displayed in the GUI prior to query formulation (e.g., Figure 2.2). In the latter case, the patterns are *dynamically* suggested during visual query formulation by utilizing the knowledge of partially formulated query fragments. In this chapter, we focus on the latter case. Observe that such suggestions make visual query formulation more efficient than the edge-at-a-time approach as it typically takes lesser time to construct a query. For instance, if a benzene ring is suggested as a pattern then instead of drawing six edges, we can construct it with a single click-and-drag action from the list of suggested patterns.

As a concrete example, consider the visual subgraph query interface in Figure 3.1 for querying a publicly available chemical database *Pubchem*. Suppose Mike wishes to search for compounds containing the chlorobenzene substructure. The partial subgraph query constructed by him is depicted in the *Visual Graph Editor* panel. It will be indeed helpful to Mike if the query system can suggest top-k possible query fragments (subgraphs) that he may add to his query in the next step.[1] An example of such top-4 suggestions is shown at the bottom panel. Observe that each suggestion is composed by adding small *increments* (i.e., subgraphs) to the query graph in the middle panel (indicated in grey). Mike may select the fourth suggestion by clicking on it, thus saving his effort to manually formulate the new nodes and edges. He may then continue formulating the final query graph in subsequent steps by leveraging the query suggestion capability iteratively.

[1]Ideally, the user interface may automatically show useful suggestions to users. The current GUI (Figure 3.1) provides an "Autocomplete" button for users to fetch the top-k suggestions to allow an explicit comparison of the experiences with and without suggestions for user studies.

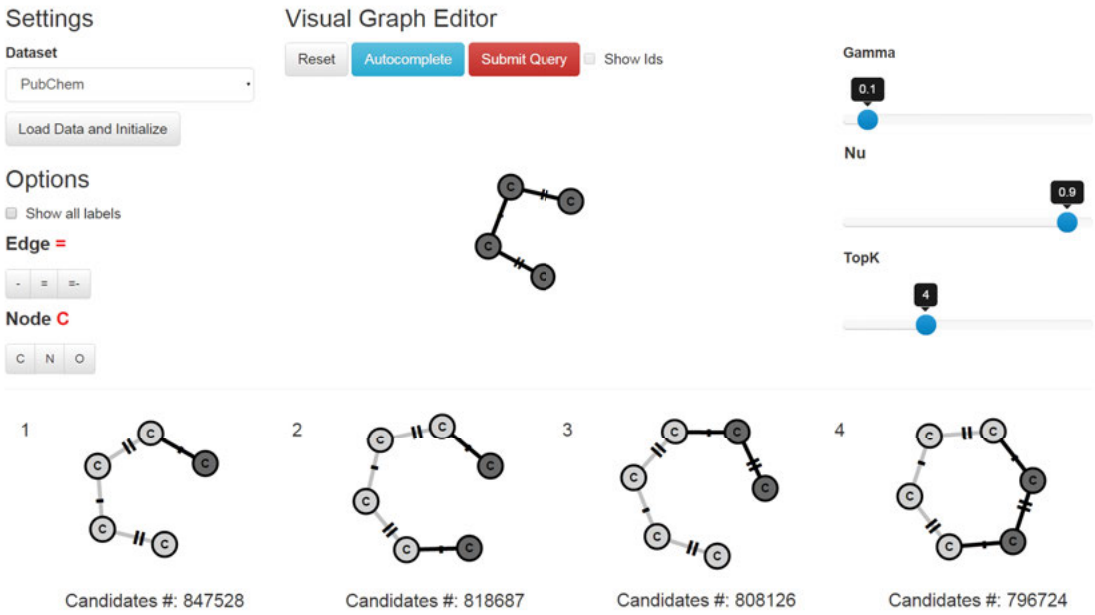

Figure 3.1: Suggestions during visual subgraph query formulation.

> *Due to the complex topology of graph-structured data, subgraph queries can be challenging to formulate as users precisely need to know the topological structure of what they are seeking. There are many instances in which the user has a clear goal in mind but only a vague idea of how the query should be specified. Consequently, suggestions and guidance to formulate visual subgraph queries can be immensely helpful to users.*

In the literature, such suggestions that assist query formulation are often referred to as *query autocompletion*. Techniques for query autocompletion have been proposed for web search and XML search (e.g., Feng and Li [2012]). For instance, search engine companies use their proprietary algorithms for providing keyword suggestions during query formulation. However, a corresponding capability for graph query engine is in its infancy.

There are at least two key challenges for providing guidance for visual query formulation, via autocompleting subgraph queries. First, in web search, the natural logical increments (i.e., tokens) of queries are keywords. However, the notion of "increments" of subgraph queries can be subgraphs, which are more flexible than keywords. Furthermore, subgraph queries are structures,

not a sequence of tokens. In particular, there are many ways to compose the queries. Second, there can be potentially many candidate query suggestions. Consequently, it is paramount to return a ranked list of query suggestions at interactive time.

In this chapter, we report recent autocompletion framework for visual subgraph queries. We classify the techniques into two categories, namely autocompletion frameworks designed for subgraph queries on a large collection of small- or medium-sized graphs and those designed for large networks. In Sections 3.1–3.5, we report AutoG [Yi et al., 2017a], a novel framework to address the former whereas in Section 3.6 we report recent efforts on the latter. We review related research in Section 3.7. The last section concludes the chapter. The key notations specific to this chapter are given in Table 3.1.

Table 3.1: Key symbols for this chapter

Symbol	Definition
$\mathsf{cs}(g_1, g_2)$	Common subgraph of g_1 and g_2
$\lambda^i_{g,g'}$	An embedding of g on g'
$\mathcal{M}_q$	A set of embeddings in a query q
Q'	A set of candidate query suggestions
util	Ranking function of query suggestions
$\vec{u}$	User preference
$\mathcal{F}_g$	A set of features of graph g
γ	A user-specified parameter to specify the degree of overlapping features
ν	A user preference component

3.1 OVERVIEW OF AUTOG

Consider a graph database $\mathcal{D}$ containing a set of small- or medium-sized data graphs $\{g_1, g_2, \ldots, g_n\}$. Each graph is a 3-ary tuple $g = (V, E, \phi, \varphi)$, where V and E are the vertex and edge sets of g, respectively, and ϕ and φ are the node and edge label functions, respectively. Given a graph database $\mathcal{D}$ and a query graph q, the answer set of q is $\mathcal{D}_q = \{g | q \subseteq g, g \in \mathcal{D}\}$. The goal of the AutoG framework [Yi et al., 2016, 2017a] is to provide query formulation guidance to an end user during visual construction of a subgraph query q on a large collection of small- or medium-sized data graphs. Figure 3.2 depicts an overview of the framework and consists of the following key steps: (i) decomposition of a query fragment, (ii) determining the candidate query suggestions, and (iii) ranking the suggestions, with respect to users' preference. Specifically, a visual query fragment q of a user is decomposed into a set of embeddings $\mathcal{M}_q$ of *c-prime features*

in q, which are logical increments of subgraph queries. In this chapter, we illustrate c-prime features with *frequent subgraphs, but they can be any structural features that capture the structural characteristics of data graphs* (e.g., Cheng et al. [2007], Fan et al. [2014], Yan et al. [2004]) that users may be interested in. Particularly, c-prime features are the first structural features defined with *feature composability*—the number of ways that a feature can be composed from other small features. In short, a c-prime feature is a feature whose composability is no more than c.

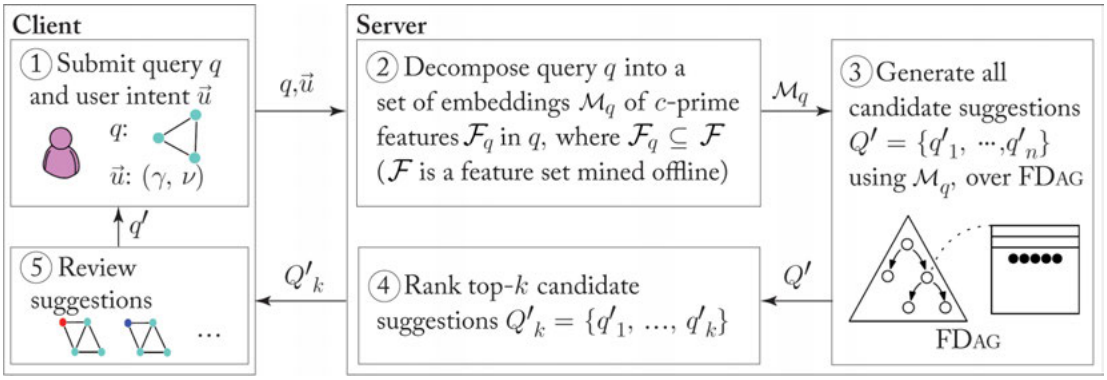

Figure 3.2: Autocompletion framework for subgraph queries in AutoG [Yi et al., 2017a].

When possible increments (i.e., subgraphs) are many, query autocompletion can be inefficient. To optimize query autocompletion time, AutoG omits non-c-prime features because they may be formed from c-prime features/queries anyway. Hence, c-prime features enables the framework to limit the number of possible increments.

Given $\mathcal{M}_q$, in the next step AutoG generates and ranks candidate suggestions by utilizing it. It formalizes the query autocompletion problem as a novel *ranked subgraph query suggestion problem* (RSQ). The goal of RSQ problem is to efficiently determine a candidate query suggestion set Q', where each suggestion specifies how a c-prime feature is added to the current query. It should be noted that *queries, as opposed to data graphs, are being ranked*. It is also not surprising that RSQ is an NP-hard problem. To optimize RSQ, AutoG leverages an index for c-prime features, called *feature* DAG (FDAG). The key features of FDAG are as follows: (i) it provides efficient support for determining the subgraphs or supergraphs of indexed features and their embeddings, (ii) it prunes redundant suggestions via graph automorphism, (iii) it enumerates possible query compositions of c-prime feature pairs offline, and (iv) it indexes some auxiliary structures for computing structural dissimilarity of suggestions online. A ranking function that is in favor of query suggestions of high selectivities and structural diversity is described in this chapter. It should be noted that the AutoG framework is not tightly coupled with this ranking strategy. Interested users may plug into AutoG their ranking functions that fit their potential applications as well. To rank query suggestions efficiently, AutoG uses a two-level greedy algorithm. In addition, it exploits a necessary condition for non-empty suggestions to prune useless

suggestions. A sampling approach is adopted to efficiently estimate the selectivity of query suggestions. An algorithm is developed to efficiently compute structural differences of suggestions.

In summary, the key contributions of the AutoG framework are as follows.

- AutoG proposes c-prime features as logical units for autocompletion for subgraph queries.
- AutoG proposes a query autocompletion framework that takes a user's current query and preference as input and produces ranked query suggestions as output.
- AutoG proposes FDAG to efficiently optimize the generation and ranking of query suggestions.
- AutoG studies the query suggestion quality with simulations and real users and conducts an extensive experimental study on performance with both real and synthetic datasets. The result verifies the usefulness of the suggestions and the effectiveness and efficiency of our query autocompletion framework.

3.2 QUERY COMPOSITION

In this section, we formalize the notion of *query composition* used in our subsequent discussions.[2] The guidance for query formulation offered by AutoG is then the query suggestions that are automatically composed from the existing user query and subgraph increments.

3.2.1 DEFINITION

We recall that queries are complex structures; and larger queries may be constructed from smaller queries in many ways. To facilitate the discussions, we define how a large query is constructed from two smaller queries by specifying how a *common subgraph* connects them.

Definition 3.1 [Common subgraphs (CS)] *Given two graphs g_1 and g_2, a* common subgraph of g_1 and g_2 *is a connected subgraph containing at least one edge and it is a subgraph of g_1 and g_2 (denoted as $\mathsf{cs}(g_1, g_2)$, or simply cs when g_1 and g_2 are clear from the context), i.e., $\mathsf{cs} \subseteq g_1$ and $\mathsf{cs} \subseteq g_2$. We define $\mathsf{CS}(g_1, g_2)$ to be the set of common subgraphs of g_1 and g_2.*

A subtlety is that in the literature *maximal* common subgraphs are extensively studied. However, we present *common subgraphs* because in query composition, large query graphs may not be necessarily formed via the maximal common subgraphs of small graphs.

Definition 3.2 [Query composition] $\mathsf{compose}$ *is a function that takes two graphs, g_1 and g_2, and the corresponding embeddings, λ_1 and λ_2, of a common subgraph* cs *as input, returns the graph g*

[2] Query composition refers to an intuitive step when users are composing their queries, which is obviously different from the compositions used in the functional programming literature.

that is composed by g_1 *and* g_2 *via* λ_1 *and* λ_2 *of* cs, *respectively, denoted as* g = compose(g_1, g_2, cs, λ_1, λ_2).

Example 3.3 Figure 3.3 shows an example of query composition by using *Pubchem*. f_{10} can be considered as a user's query and f_{13} is an increment to f_{10} that may be helpful to guide a user to complete his/her query. The chemical elements are vertices and the numbers near them are their IDs. f_4 is a common subgraph of f_{10} and f_{13}. We have $f_4 \subseteq f_{10}$ where $\lambda^0_{4,10}$={0↦0, 1↦2} or $\lambda^1_{4,10}$={0↦2, 1↦0}, $f_4 \subseteq f_{13}$ where $\lambda^0_{4,13}$={0↦1, 1↦2} or $\lambda^1_{4,13}$={0↦2, 1↦1}. There are four ways to compose f_{10} and f_{13} via f_4:

1. compose(f_{10}, f_{13}, f_4, $\lambda^0_{4,10}$, $\lambda^0_{4,13}$) results in f_{22};
2. compose(f_{10}, f_{13}, f_4, $\lambda^0_{4,10}$, $\lambda^1_{4,13}$) results in f_{18};
3. compose(f_{10}, f_{13}, f_4, $\lambda^1_{4,10}$, $\lambda^0_{4,13}$) results in f_{18}; and
4. compose(f_{10}, f_{13}, f_4, $\lambda^1_{4,10}$, $\lambda^1_{4,13}$) results in f_{22}.

□

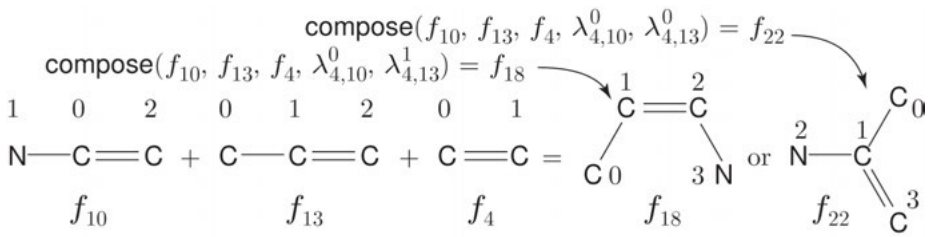

Figure 3.3: Query graph composition.

3.2.2 QUERY AUTOCOMPLETION MODES

Next, we present the possible query composition modes for automatically generating suggestions and the one that is assumed by this chapter.

Connected vs. disconnected subgraphs. In this chapter, we assume that the query suggestions (formed by a composition of the query graphs and increments) are connected graphs. Hence, the common subgraphs are connected graphs, too. A minor relaxation on the requirement of connected existing queries and increments will easily support disconnected query suggestions.

Edge increments vs. subgraph increments. A simple query autocompletion mode is to generate edge increments to the current query (e.g., , Jayaram, Goyal, and Li [2015]). However, edge

suggestions have at least two drawbacks: (1) the query formulation process may take many steps; and (2) users can express limited structural information regarding their desired queries in each step. On the other hand, AutoG guides the query formulation by incrementing the query with subgraphs. In the previous experimental investigation on the query suggestions, the average number of edges of query increments is always more than one. For simplicity of presentation, *we assume connected subgraphs suggestions* unless otherwise specified. Putting these together, the problem being addressed in this chapter can be described as follows.

Problem statement. *Given an existing query q, a ranking function* util, *a user preference $\vec{u}$ and a parameter k, compute a query suggestion set Q'_k: $\{q'_1, q'_2, \ldots, q'_k\}$ such that for $i \in [1, k]$, q'_i is composed by adding an increment to q and Q'_k is the top-k suggestions w.r.t.* util *and $\vec{u}$.*

3.2.3 C-PRIME FEATURES

Subgraph increments to an existing query are obviously structures. Without prior information of the target query that a user trying to formulate, it is desirable to introduce structures that contain important characteristics of the graph data to the existing query.

Structural features of graphs have been extensively studied since the last decade, for the purpose of optimizing the performances of structural queries, among other things. Their intuition is to determine a set of subgraphs that carry various structural characteristics of data graphs $\mathcal{D}$ (e.g., discriminative frequent subgraph [Yan et al., 2004]). Data graphs are then indexed by the features. Given a query q, it is decomposed into a set of features F_q. A data graph that does not contain F_q cannot be an answer of q and, hence, can be pruned. Previous work shows that this approach can effectively prune non-answers. However, this approach does not consider query autocompletion.

In this section, we recall c-prime features. They are defined by how many ways they can be composed from smaller features. Intuitively, c-prime features are features that can be formed by smaller features in only a few ways. c-prime features are specially designed for suggesting *query increments not filtering data graphs*. We remark c-prime features are orthogonal to existing features, i.e., users may integrate their existing features with c-prime features for their specific applications.

The design rationales of c-prime features are that (i) some features are important to query autocompletion because their absence leads to fewer possible suggestions; and (ii) some other features are less important because they can be constructed incrementally from small ones in numerous ways and can be suggested by query autocompletion anyway.

To discuss c-prime features, we start with *frequent features* [Yan and Han, 2002] (refer to Section 2.1.3 for further discussions). Frequent features are adopted because, without prior knowledge, we may assume each data graph in the database $\mathcal{D}$ has the same chance of being retrieved by users' queries. Hence, frequent subgraphs of $\mathcal{D}$ have higher chances to appear in users' queries.

Given a graph g and a frequent feature set $\mathcal{F}$ of a database $\mathcal{D}$, we may decompose g into a set of features $\mathcal{F}_g$: $\{f_1, \ldots, f_n\}$, where $\forall f_i \in \mathcal{F}_g$ implies $f_i \in \mathcal{F}$ and $f_i \subseteq g$. Similarly, we may decompose a query into a set of features and their embeddings in g.

Definition 3.4 *A query q of* AutoG *is represented as a binary tuple $(\mathcal{F}_q, \lambda)$, where $\mathcal{F}_q$ is a set of features of q, and λ takes a feature $f_q \in \mathcal{F}_q$ as input and returns the embedding of f_q in q.*

Example 3.5 Consider Figure 3.4. It depicts a set of frequent features extracted from *Pubchem* by gSpan [Yan and Han, 2002]. In the figure, the vertices `C`, `N`, and `O` represent the chemical elements carbon, nitrogen and oxygen, respectively. The edges between the chemicals (i.e., `C-C` and `C=C`) signify the single and double bonds between two elements. Suppose f_{18} is a query. A possible $\mathcal{F}_q$ is $\{f_4, f_6, f_7, f_{10}, f_{13}, f_{18}\}$. One may easily derive embeddings of the double-bond `C=C` (f_4), the single-bond `C-C` (f_7), and other features in f_{18}. □

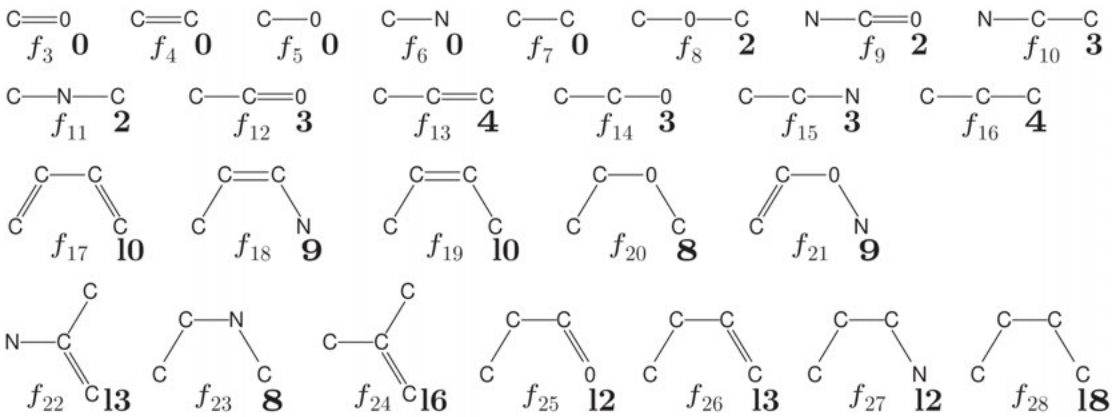

Figure 3.4: Frequent features (partial) with their composabilities.

From the above example, we can see that queries can be considered as compositions of features.[3] However, how features are composed together to form queries requires some elaboration. Graph query composition here is structural.[4] Feature embeddings are required to specify how large structures are formed (see Definition 3.2). In Definition 3.6, we define *feature composability* as a measurement of the number of embeddings of feature pair compositions that form the feature f.

Definition 3.6 **[Feature composability]** *The composability of a frequent feature f with respect to the feature set $\mathcal{F}$, denoted as* $\mathtt{c}(f, \mathcal{F})$*, is*

[3] AutoG assumes the non feature parts of the queries are inputted by users. That is, they are not composed from AutoG.
[4] In contrast, for keyword search (of strings), key phrases are simply composed by a *union/concatenation* of keywords.

$$\sum \frac{|\{(f_i, f_j, \mathsf{cs}, \lambda_i, \lambda_j) \mid f = \mathsf{compose}(f_i, f_j, \mathsf{cs}, \lambda_i, \lambda_j)\}|}{|A(\mathsf{cs})|},$$

where $\mathsf{cs} \in \mathsf{CS}(f_i, f_j)$ *and* $f_i, f_j \in \mathcal{F}$*, the equality "=" denotes graph isomorphism, and* $\mathcal{A}(\mathsf{cs})$ *denotes the automorphism relation of* cs.

In Definition 3.6, the numerator of the composablity is the number of distinct feature embeddings that form the feature. The denominator $|\mathcal{A}(\mathsf{cs})|$ is needed in Definition 3.6 because the queries that are constructed from features via automorphic common subgraphs are structurally equivalent.

Given this background, we are ready to present c-prime features, which are features that have a composability smaller than or equal to c.

Definition 3.7 **[c-prime feature]** *A feature f is a c-prime feature if and only if* $\mathsf{c}(f, \mathcal{F}) \leq c$*. The feature is non-$c$-prime feature if and only if it is not a c-prime feature.*

Assuming that each composition is equally likely, non-c-prime features have a high chance of being formed from c-prime features. Therefore, non-c-prime features have a higher chance of being recovered from query autocompletion. On the other hand, c-prime features may not be suggested as they may not be composed from other features. Ignoring c-prime features leads to less comprehensive query suggestions. In addition, c-prime features have the properties of anti-monotonicity and downward-closure properties, which are similar to those of frequent features.

Example 3.8 Reconsider the features shown in Figure 3.4. We annotate each feature with its composability at its lower right-hand side. When the value of c was 4, the c-prime features (4-prime features) are $\{f_3, f_4, \ldots, f_{16}\}$. While larger features may still be constructed from 4-prime features, they may require multiple construction steps. When c is set to 16, only f_{28} is not c-prime feature. It is because f_{28} can be constructed from f_7 and f_{16} in many ways. In the case where the dataset has many features and their possible compositions are voluminous, query autocompletion may become too costly. Hence, the non-16-prime features may be omitted from query autocompletion as these features are more probably suggested later. In other words, users may not lose query suggestion candidates even omitting non-16-prime features. □

Example 3.9 Consider Figure 3.4 again. The feature f_{13} can be constructed from f_4 and f_{13} via f_4. The cs is f_4. The cs has two embeddings in f_{13} and f_4, respectively. $\mathcal{A}(\mathsf{cs})$ is 2. The same counts are obtained from composing f_7 and f_{13} via f_7. The composability of f_{13} is 2×2 / 2 + 2×2 /2 equals 4. Thus, f_{13} is a 4-prime feature. f_{13} is a subgraph of f_{18}. After some counting, we note that f_{18} is a 9-prime feature. By the property of antimonotonicity, f_{13} is certainly a 9-prime feature. Since f_{18} is not a 4-prime feature, by the downward-closure property, any supergraphs of f_{18} are not 4-prime features. □

Integration of c-prime features with other features. We remark that this section illustrates c-prime features with frequent features (as their underlying features). However, depending on users' applications, AutoG may integrate other features into c-prime features. For instance, the structure search *PubChem*[5] provides *some templates of query structures*[6] for users to compose their queries, based on domain knowledge of chemical applications. In other applications such as web searches [Pandey and Punera, 2012], the query templates can be automatically derived from query logs by using machine learning or data mining methods. *When* AutoG *adopts such approaches, it adds the templates into the feature set of* $\mathcal{D}$. AutoG simply determines whether they are c-prime as before.

3.3 AUTOCOMPLETION FRAMEWORK IN AUTOG

In this section, we discuss in detail the three steps in AutoG for providing query formulation guidance, namely, query decomposition, determining candidate suggestions, and ranking them w.r.t a ranking function and a user's preference.

3.3.1 QUERY DECOMPOSITION

Following the subgraph query processing method in the literature (e.g., Cheng et al. [2007], Yan et al. [2004]), AutoG assumes the (c-prime) features of data graphs are mined offline. At runtime, it decomposes a query into a feature set. However, AutoG requires the embeddings (intuitively, the locations) of the features in the query which show how they are connected.

It is evident that the decomposition of a query is not unique. Further, the design rationales of query decomposition in AutoG are competing ones: firstly, the larger the features, the more structural semantics they preserve; secondly, the larger the features, the higher the chance the features overlap. The overlapping features contain redundant information, and, hence, should be avoided. Therefore, a user-specified parameter γ is introduced to specify the desirable degree of overlapping features. The larger the value of γ, the more likely the overlapping of decomposed features.

Example 3.10 Consider the graphs shown in Figure 3.3. Suppose that f_{18} is a user query. A possible decomposition is $\{f_{10}\ f_{13}\}$, of which the embeddings are overlapping. □

Algorithm 3.1 is a greedy algorithm for decomposing a query q with respect to the user-specified parameter γ. Initially, all the edges of q are not covered. Algorithm 3.1 iteratively determines an embedding of a c-prime feature to cover q until no more uncovered edges can be found. The output is a set of embeddings of the c-prime features $\mathcal{M}_q$ in q.

[5] `https://pubchem.ncbi.nlm.nih.gov/edit2/index.html?cnt=0`

[6] Almost all query templates of *Pubchem* on their user interface are also frequent subgraphs, returned by *gSpan* using its default parameters. This shows that the domain experts indeed set frequent subgraphs as query templates. Yet, we omit the advanced templates of *PubChem* after user's click on the UI.

Algorithm 3.1 ***Query Decomposition***

Input : a query q, feature set $\mathcal{F}$ (determined by gSpan [Yan and Han, 2002] offline) and user preference component γ
Output : a set of embeddings of the c-prime features $\mathcal{M}_q$ in q

1: Let $\mathcal{F}_q$ be the c-prime features of q
2: Let $\mathcal{M}_{\mathcal{F}_q}$ be the embeddings of each $f \in \mathcal{F}_q$ in q // e.g., using VF2
3: Let E be the edge set of q, and $\mathcal{M}_q$ be an empty set
4: Initialize $e.w = 1$ for $\forall e \in E$, and $\lambda_f.\texttt{unused} = \texttt{true}$ for $\forall \lambda_f \in \mathcal{M}_{\mathcal{F}_q}$
5: **while** λ_f = FIND$(E, \mathcal{M}_{\mathcal{F}_q})$ **do**
6: $\quad \mathcal{M}_q \leftarrow \mathcal{M}_q \cup \lambda_f$
7: **end while**
8: **return** $\mathcal{M}_q$
9: **Function** FIND$(E, \mathcal{M}_{\mathcal{F}_q})$
10: determine $\lambda_f \in \mathcal{M}_{\mathcal{F}_q}$, where $\lambda_f.\texttt{unused} = \texttt{true}$,
λ_f covers at least one uncovered edge
$\nexists \lambda'_f \in \mathcal{M}_{\mathcal{F}_q}$ and $\mathsf{util}(\lambda'_f) > \mathsf{util}(\lambda_f)$
11: $e.w \leftarrow e.w * \gamma$ for $\forall e \in \lambda_f.E$
12: $\lambda_f.\texttt{unused} \leftarrow \texttt{false}$
13: **return** λ_f
14: **Function** $\mathsf{util}(\lambda_f)$
15: **return** $\sum_{e \in \lambda_f.E} e.w$

More specifically, first, we determine the c-prime features $\mathcal{F}_q$ of q by invoking a feature extraction program [Yan and Han, 2002]. We determine the embeddings $\mathcal{M}_{\mathcal{F}_q}$ (called feature-query embeddings) of the extracted features by invoking VF2 (Line 2). We initialize the result feature-query embeddings $\mathcal{M}_q$ to be an empty set, and E to be the edge set of q to be covered (Line 3). We assume each edge e of q has a weight w, which denotes if e has been covered by some features in $\mathcal{M}_q$. Each feature-query embedding λ_f has a flag (called `unused`) to indicate whether the feature-query embeddings is in $\mathcal{M}_q$. Second, in Lines 5–6, we iteratively add the next λ_f with the largest utility (defined by util) to $\mathcal{M}_q$. The weights of the edges of the added λ_f are degraded by γ (Line 12).

We make two remarks on FIND (Lines 8–14). In general, there are multiple decompositions of a query. FIND enumerates larger features, where the feature size is determined by the sum of edge weights (implemented by util in Lines 15–16). Larger features are used because they require more human effort to compose, i.e., they may preserve more of the user's intention. Hence, when a user has drawn exactly a large feature in his/her initial query, FIND leads

AutoG to consider it as a whole, as opposed to small feature(s). However, large features may overlap. Thus, if an edge is covered by an λ_f and added to $\mathcal{M}_q$, the weight of the covered edge is reduced by a factor of γ (Line 12).

It should be remarked that queries may contain infrequent edges, which are not in $\mathcal{F}$, and will not be handled by AutoG. The analogy is that in web searches, infrequent keywords are not suggested; similarly, in AutoG, infrequent logical units are not suggested. By definition, infrequent edges lead to small answer sets. In this case, users may need less guidance from AutoG.

Complexity analysis. The time complexity of Algorithm 3.1 is $O(|\mathcal{F}_q| \times T_{subiso} + |E|^2 \times |\mathcal{M}_{\mathcal{F}_q}|)$, where (a) the first term is the time for determining the embeddings of $\mathcal{F}_q$ in q and T_{subiso} is the complexity for a subgraph isomorphism call, and (b) the second term is for scanning the $|\mathcal{M}_{\mathcal{F}_q}|$ embeddings to cover $O(|E|)$ edges in the FIND function, which is invoked $O(|E|)$ times.

We provide several observations on the two terms in the above-mentioned complexity. A worst-case exponential-time subgraph isomorphism algorithm is needed to determine the embeddings of $\mathcal{F}_q$ in q (Lines 1–2). AutoG uses a practical subgraph isomorphism algorithm called VF2. Moreover, users typically draw small queries via a visual graph editor, e.g., a graph containing fewer than 24 edges. Hence, the size and the number of features of the query ($\mathcal{F}_q$) are small. As a result, T_{subiso} is small in practice, and VF2 is invoked only few times. Regarding the complexity for scanning the embeddings, the terms $|E|$ and $|\mathcal{M}_{\mathcal{F}_q}|$ are small, again, due to small query sizes. Finally, the calculations of Lines 5–16 do not incur large constants in the asymptotic complexity. Hence, Algorithm 3.1 decomposes queries efficiently.

3.3.2 GENERATION OF CANDIDATE SUGGESTIONS

After the query decomposition step, the query q is represented by a set of c-prime features and their embeddings in q. The next step is to generate candidate query suggestions. In this subsection, we present connected feature increments, which are the most technically intriguing query composition mode, as discussed in Section 3.2.2.

Query increments can be added to the current query in multiple ways. Specifically, given a set of c-prime features, the number of compositions is, in the worst case, exponential to the query and feature sizes. However, in practice, many possible composed queries may not make sense that do not retrieve any data graphs. Such queries are also known as *empty queries*. Further, it is known that deciding the emptiness of a subgraph query is NP-hard.

This subsection formalizes a *necessary* condition for non-empty query compositions. We illustrate how to efficiently prune empty queries using the necessary condition and *the unpruned queries are considered candidate suggestions*.

Baseline. We present the condition with *node labels* for presentation simplicity, which can be readily extended to support edge labels. Consider a graph $g = (V, E, \phi, \varphi)$. Denote Σ to be the

node label set, where $\Sigma = \{\phi(v) \mid v \in V\}$. For each node $v \in V$, we determine a vector of the counts of its neighboring node's label $\vec{\phi_v}$, where

$$\vec{\phi_v}[l'] = |\{v' | (v, v') \in E, \phi(v') = l'\}|.$$

The nodes of the graphs can be represented by such vectors and, hence, as data points in a Σ-dimensional space. Denote S to a skyline in the Σ-dimensional space of the data point representations of the nodes of the graphs. Given a query q, *if it is non-empty, q does not contain a node whose vector dominates the nodes in S in some dimensions.* This condition is formalized in Proposition 3.11. Its proof can be established by a proof by contradiction.

Proposition 3.11 *A query q is a* non-empty query *(also refers to candidate suggestion) only if $\nexists v_q \in q.V$ such that $\forall g \in \mathcal{D}$, $\forall v \in g.V$, $\exists l \in \Sigma$, $l = \phi(v) = \phi(v_q)$, and $\vec{\phi_v}[l] < \vec{\phi_{v_q}}[l]$.* □

Suppose that there are $|S|$ data points on the skyline. The check of Proposition 3.11 requires $O(|q.V| \times |\Sigma| \times |S|)$ comparisons. As discussed, numerous possible queries may be generated and *they are checked with S at runtime*. Thus, we relax the check for efficiency.

Relaxed necessary condition. For each label l_1 in Σ, we determine the maximum number of each neighboring label l_2 of $\mathcal{D}$,

$$d_{l_1,l_2} = \max(|\{v_2 | \phi(v_1) = l_1, \phi(v_2) = l_2, g \in \mathcal{D}, (v_1, v_2) \in g.E\}|).$$

The necessary condition for non-empty queries is then expressed in terms of d_{l_1,l_2}, as shown in Proposition 3.12. The number comparisons are reduced to $O(|q.V| \times |\Sigma|)$. It has been validated by experiments that this simplification is both efficient and effective [Yi et al., 2017a].

Proposition 3.12 *A query q is non-empty* only if $\nexists v_q \in q.V$, *such that* $\forall\ d_{l_q,l_2}$, $d_{l_q,l_2} < |\{v_2 \mid \phi(v_q)=l_q, \phi(v_2)=l_2, (v_q,v_2) \in q.E\}|$, *where* $l_q \in \Sigma$. □

3.3.3 RANKING CANDIDATE SUGGESTIONS

Candidate suggestions can be numerous. Since users may only be able to interpret a small subset of them, AutoG returns top-k suggestions w.r.t. a *ranking function* and a *user preference* component. When users formulate their queries, they may *rank* candidate suggestions differently, because of different query formulation scenarios: for example, expert users may use AutoG to speed up their manual query formulation, whereas novice users may prefer diversified suggestions for exploring a database. In this section, we model the preferences between different criteria with a ranking function and a user preference component.

Ranking function and user preference component. We present a ranking function for possibly novice users who prefer query suggestions that *(i) return more answer graphs and (ii) are*

structurally diversified. The first preference simply reflects users' intent to retrieve more answers whereas the second one recognizes the importance of avoiding similar suggestions (e.g., Gollapudi and Sharma [2009], Qin, Yu, and Chang [2012], Vierira et al. [2011]). These two preferences can be quantified as the following objective functions.

1. $\mathsf{sel}(q)$: the selectivity of q on $\mathcal{D}$, is $|\mathcal{D}_q|/|\mathcal{D}|$.

2. $\mathsf{dist}(q_i, q_j)$: the "intra-dis-similarity" between a pair of suggestions, q_i and q_j. The total pairwise distance of suggestions reflects how diversified a set of suggestions are. For illustration purposes, we adopt the *maximum common edge subgraph* (mces) for dist (see Definition 3.13 [Bunke and Kim, 1998, Kriege, Mutzel, and Schäfer, 2014]). mces is adopted because adding edges (as opposed to nodes) to an existing query appears a natural logical step of composing queries. The distance definition is denoted as dist.

Definition 3.13 [Distance] *Given two graphs g_1 and g_2, the graph distance based on the maximum common edge subgraph (*mces*) is defined as follows:*

$$\mathsf{dist}(g_1, g_2) = 1 - \frac{|\mathsf{mces}(g_1, g_2)|}{\mathsf{max}\{|g_1|, |g_2|\}},$$

where mces*(g_1, g_2) is a subgraph of g_1 with as many edges as possible that is isomorphic to a subgraph of g_2.*

Example 3.14 Take f_{18} and f_{22} in Figure 3.4 as an example; $\mathsf{dist}(f_{18}, f_{22}) = 1 - \frac{2}{3} = \frac{1}{3}$, according to Definition 3.13.

The dist function has a few nice properties [Bunke and Kim, 1998, Kriege, Mutzel, and Schäfer, 2014]. It is a metric. It is a reflexive and symmetric function, which can be observed from its definition. Other graph distance functions (such as Kriege, Mutzel, and Schäfer [2014], Venero and Valiente [2001], Wallis et al. [2001]) can be adopted to implement dist.

Definition 3.15 [User intent value of query suggestions] *Given a set of query suggestions $Q' : \{q'_1, q'_2, \ldots, q'_k\}$ and a user preference component ν, the* user intent value *of Q' (*util*) is defined as follows:*

$$\mathsf{util}(Q') = \frac{\nu}{k} \sum_{q' \in Q'} \mathsf{sel}(q') + \frac{1-\nu}{k(k-1)} \sum_{q'_i, q'_j \in Q', i \neq j} \mathsf{dist}(q'_i, q'_j),$$

where $\nu \in [0, 1]$.

The overall ranking function (util) is presented in Definition 3.15, which is the normalized weighted sum of the two objective functions mentioned above. The two objective functions can

be competing: for example, it can be observed that in practice, the sel of smaller queries are often larger as more data graphs contain smaller queries; in contrast, smaller queries may have smaller structural differences between them and, consequently, dist returns smaller values and their diversities are relatively low. With the util function, we are ready to formulate the ranking problem of query suggestions and report its hardness.

Definition 3.16 [Ranked Subgraph Query Suggestion Problem] *Given a query q, a set of query suggestions Q', the ranking function* util*, a user preference component ν, and a user specified constraint k, the* ranked subgraph query suggestion *problem* [RSQ] *is to determine a subset Q'',* util*(Q'') is maximized.*

Theorem 3.17 *The* RSQ *problem is* NP*-hard.*

The RSQ problem is an NP-hard problem, which can be established by a reduction from the maximum independent set (MIS) problem.

Proof sketch. [MIS] Given a graph G=(V, E), where V and E are the sets of vertices and edges, respectively, an independent set (IS) is a set of vertices V', such that $V' \subseteq V$, there does not exist $v_i, v_j \in V'$ and $(v_i, v_j) \in E$. The MIS problem is to determine an IS V' such that there does not exist an IS whose size is larger than V'.

Reduction. We start with an instance of MIS problem G=(V, E). We construct a query suggestion set Q_V such that for each vertex v in V, we have a query suggestion q_v corresponding to each v in V. We construct Q_V such that each query q_v in Q_V has exactly $|Q_V|$ edges. The structure of each query is a star, with a common first edge (v_a, v_b) and the other edges encode the following (also illustrated with Figure 3.5).

1. If $(v_i, v_j) \in E$, then, (v_b, vi, j) is an edge of q_{v_i} and q_{v_j}. That is, mces between q_{v_i} and q_{v_j} is $\{(v_a, v_b), (v_b, vi, j)\}$.

2. Otherwise, an edge $(v_b, v^i i, j)$ is introduced to q_{v_i} and $(v_b, v^j i, j)$ is introduced to q_{v_j}. Then, the mces between q_{v_i} and q_{v_j} is $\{(v_a, v_b)\}$ only.

The maximum independent set is at most of the size $|V|/2$. Therefore, we invoke RSQ on Q_V, where k is ranged from 1 to $|V|/2$ and α is set to 0. That is, only the diversity component of the ranking function is considered.

Case (1): Suppose Q'_v is a solution of RSQ. If for some $q_i, q_j \in Q'_v$ and mces(q_i, q_j) is not just (v_a, v_b); then it guarantees there does not exists Q''_v such that $|Q'_v| = |Q''_v|$ and *for all* $q'_i, q'_j \in Q''_v$, such that mces(q'_i, q'_j) is just (v_a, v_b). This is because Q''_v would have been ranked higher, according to util (i.e., Q''_v is more diversified than Q'_v). And by the reduction above, there is an edge (v_i, v_j) in E. The corresponding V' is not an IS.

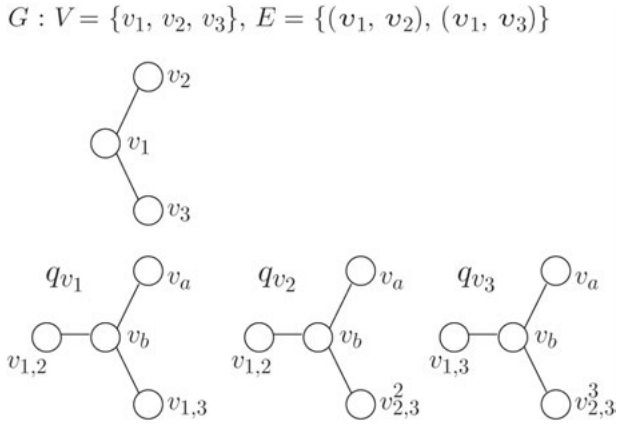

Figure 3.5: An illustration of the query suggestions generated from an MIS instance.

Case (2): Suppose Q'_v is a solution of RSQ and for all $q_i, q_j \in Q'_v$, $\mathsf{mces}(q_i, q_j)$ is (v_a, v_b). By the reduction above, there is no edge between v_i and v_j, for all v_i, v_j. Thus, the corresponding V' is an IS.

Putting these together, let Q'_v is the largest set returned by invoking RSQ's for k ranging from 1 to $|V|/2$, whose corresponding V' is an IS. Suppose Q''_V is returned by RSQ and larger than Q'_v. Then, Q''_V belongs to Case (1). According to Case (1), there is no IS of the size $|Q''_V|$ can be obtained. Therefore, V' is the maximum independent set. □

Template values of user preference ν. The user preference ν in Definition 3.15 expresses the relative importance of selectivities and suggestion diversities. To help users to set ν, AutoG derives a set of templates of predefined preference components with intuitive semantics (such as *selectivity-oriented* and *diversity-oriented* suggestions) from the underlying dataset. Users may start with a predefined template and subsequently refine ν after reviewing some query suggestions returned.

Alternatively, AutoG starts with a template of user preference. Based on the suggestions adopted by a user, AutoG may learn whether he/she prefers selectivities or diversities. The details of learning parameters from users' feedbacks, however, are beyond the scope of this chapter.

Integration of other ranking functions. It should be remarked that the ranking function util presented in this section is for illustration purposes. That is, other objective functions can be readily plugged into the AutoG framework. Take the structure search of *PubChem* as a concrete example. We may include application-specific semantics in an additional objective function. Suppose $\mathcal{F}_T$ is the set of query templates provided by *PubChem* and the templates in $\mathcal{F}_T$ are c-prime features, as discussed at the end of Section 3.2.3. Suppose users are favorable to sug-

gestions that contain the query templates $\mathcal{F}_T$. This can be achieved by introducing a function $\mathsf{app}(q')$ that returns the number of features in $\mathcal{F}_T$ contained in q'. AutoG adds app to util and sets its preference as other objective functions.

Efficient Selectivity and Diversity Computation

Next, we present efficient algorithms for determining sel and dist, which enable efficient ranking candidate suggestions.

1. Candidate answer selectivity estimation. We first recall some standard notations. We denote $\mathcal{D}_q = \mathsf{eval}(q, \mathcal{D})$ as the query evaluation of q on $\mathcal{D}$ and $\mathcal{D}_q$ is the query result. The selectivity of q (denoted as $\mathsf{sel}(q, \mathcal{D})$) is $|\mathsf{eval}(q, \mathcal{D})|/|\mathcal{D}|$.

Recall that $\mathsf{eval}(q, \mathcal{D})$ is NP-hard due to subgraph isomorphism tests. Hence, AutoG leverages feature-based query processing to efficiently estimate eval. The benefits of this approach are twofold. (1) c-prime features can be seamlessly integrated into existing feature-based approaches. This estimation does not incur much overhead. (2) It has been known that the feature-based approach (e.g., Yan et al. [2004]) can efficiently determine candidate answer sets of subgraph queries, which are close to the actual answer sets.

In a nutshell, each feature is associated with a set of IDs of the graphs that contain the feature. Given the features of a query $\mathcal{F}_q$, the candidate set is obtained by the *intersection* of the sets of IDs associated with each feature in $\mathcal{F}_q$. The numerous intersections of large ID sets may be costly, especially when ranking suggestions online. Hence, we estimate the selectivity by adopting a systematic sampling after a uniformly random permutation of the *graph IDs* [Broder, 1997].

W.l.o.g, we assume two sets A and B, and $|A| < |B|$. Then, $|A|$ is the population size. The real selectivity, $|A \cap B|$, is the number of success states in the population. $S = |A|/m$ is the number of draws, where m is *the user-specified sampling interval*. The number of observed successes is denoted as k. The probability that the observed successes exactly equals to k is given by

$$P(X = k) = \frac{\binom{|A \cap B|}{k}\binom{|A| - |A \cap B|}{S - k}}{\binom{|A|}{S}}.$$

The error ϵ of our estimation method can be analyzed based on hypergeometric distribution, that describes the probability of k successes in n draws, *without replacement*:

$$|A \cap B|\frac{S}{|A|} - \epsilon \leq k \leq |A \cap B|\frac{S}{|A|} + \epsilon.$$

2. Diversity approximation. The second component of util is the structural diversity of query suggestions in Q'. dist makes the overall ranking function util submodular, so that greedy algorithms are its natural heuristic.

To efficiently implement the dist function of two suggestions, our main idea is to trim the common parts from them before calling the exponential-time algorithm for mces. This can be efficiently computed because (i) the query suggestions are composed by adding different increments on the *same* existing query graph, and (ii) some auxiliary structures between possible composable features can be computed *offline*. For brevity, we omit the tedious pseudo-code but illustrate the major steps with the following example.

Example 3.18 Given a current query q which is simply a feature f_{57}, consider two possible compositions (shown in Figure 3.6) that construct query suggestions from f_{57} by adding either f_{19} (denoted as q'_1) or f_{28} (denoted as q'_2). Note that the existing query q and increments f_{19} and f_{28}, in this example, are features and their compositions can be enumerated offline. Suppose we compute mces of q'_1 and q'_2. Some parts of q'_1 and q'_2 are trivially common and are not necessary to perform the costly mces computation. Thus, we reduce q'_1 and q'_2 to the trimmed subgraphs $\bar{q_1}$ and $\bar{q_2}$ for computing the non-trivial mces of q'_1 and q'_2. Further, some intermediate results are indexed offline. Specifically, the major offline steps are presented as follows.

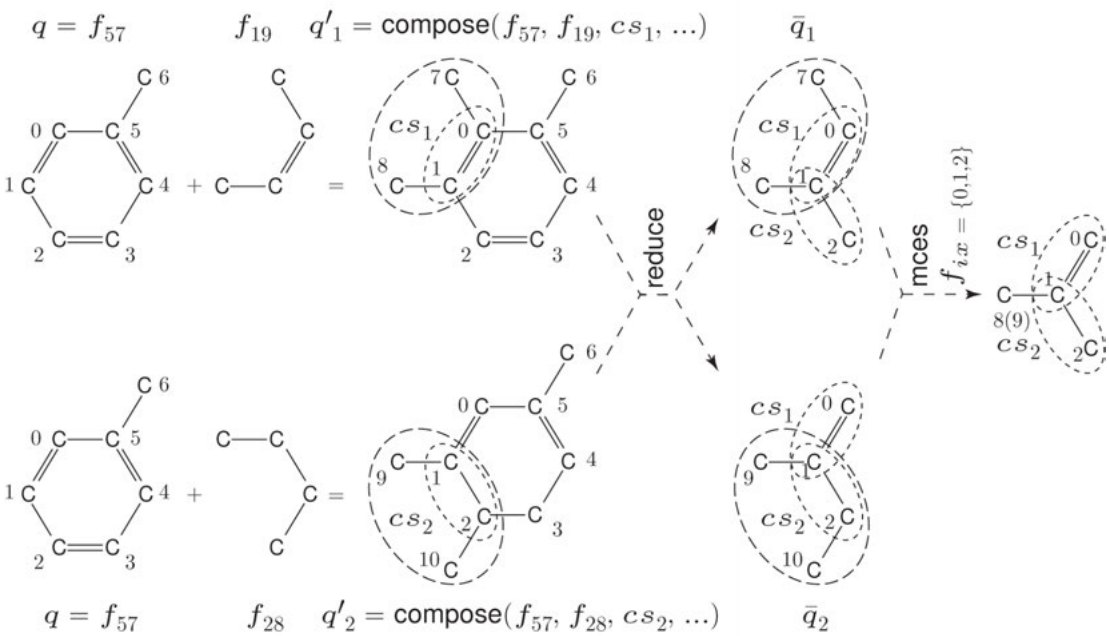

Figure 3.6: Trimming compositions for mces computation.

1. Denote cs_1 (resp. cs_2) to be the common subgraph between f_{57} and f_{19} (resp. f_{57} and f_{28}).

2. The subgraph s computed by f_{57} - cs_1 - cs_2 is trivially a part of the mces of q'_1 and q'_2.

3. $\bar{q_1}$ is obtained by q'_1 - s. Similarly, $\bar{q_2}$ is q'_2 - s.

4. The embedding of cs_1 (resp. cs_2) in f_{57} is computed offline. It also specifies its location in $\bar{q_1}$ (resp. $\bar{q_2}$), which minimizes the search of mces. In particular, the nodes 0, 1, and 2 of $\bar{q_1}$ must map to the nodes 0, 1, and 2 of $\bar{q_2}$.

5. An mces algorithm determines the mces of $\bar{q_1}$ and $\bar{q_2}$ offline, and it returns $s_{1,2}$.

6. In $s_{1,2}$, the nodes {0,1,2} are from cs_1 and cs_2 (i.e., the existing query q). The non-trivial mces is C-C (1, 8 (or 9)), which contains one edge.

When candidate suggestions are ranked *online*, the size of mces between q'_1 and q'_2 is obtained by simply adding the sizes of the existing query q (i.e., f_{57}) and the non-trivial mces: $|f_{57}| + 1 = 8$. □

We provide two remarks on the above mces computations. (i) While the query is provided by users online, features and their compositions can be enumerated offline. Therefore, sizes of the non-trivial mces between compositions can be indexed offline. (ii) The above optimization significantly speeds up the online mces computation because query suggestions contain the same existing query graph.

Greedy Ranking Algorithm

We now present a ranking algorithm for a set of candidate suggestions with respect to the ranking function util and the user preference ν. Note that the util function has two monotone submodular components. It can be easily established that the util function is also submodular. A decent property of a submodular function is that greedy algorithms work and guarantees a 1/2OPT approximation ratio [Borodin et al., 2012].

We present a two-level greedy algorithm (Algorithm 3.2) to rank the candidate suggestions. A two-level algorithm has been adopted because the candidate suggestions can be many and computing diversity between every possible pairs of them involves dist, which is time consuming. In a nutshell, at the first level, for each feature f embedded in a specific location λ of the query, the algorithm greedily determines its top-k suggestions (denoted as $Q_k^{f,\lambda}$) that increment f in location λ. At the second level, it greedily determines the overall top-k suggestions from $Q_k^{f,\lambda}$s, for all (f, λ)s. Hence, it avoids computing dist of all possible pairs of candidate suggestions. The pseudo-code of the greedy algorithm is presented in Algorithm 3.2. We elaborate its details below.

Greedy_local: First, Greedy_local (Lines 14–15) determines *the possible suggestions composed*

Algorithm 3.2 ***Ranking Candidate Suggestions***

Input : a query q represented by $\mathcal{M}_q$, user preference component ν, number of suggestions requested k and max. increment size δ
Output : the top-k query suggestions Q'_k

1: **Function** GREEDY_GLOBAL(q, ν, k)
2: **for all** $(f, \lambda) \in \mathcal{M}_q$ **do**
3: $\quad Q_k^{f,\lambda}$ = GREDDY_LOCAL(f, λ, q) //local ranking w.r.t (f, λ)
4: $\quad Q' = Q' \cup Q_k^{f,\lambda}$
5: **end for**
6: compute dist between each pair of suggestions in Q'
7: $Q'_k = \{\}$
8: **for all** $i = 1 \ldots k$ **do**
9: $\quad q_{max} = \text{argmax}(q', \text{util}(q', Q'_k))$, where $q' \in Q'$ //local ranking w.r.t (f, λ)
10: $\quad Q'_k = Q'_k \cup q_{\max}$
11: $\quad Q' = Q' \setminus q_{\max}$
12: **end for**
13: **return** Q'_k
14: **Function** GREDDY_LOCAL(q, ν, k) // local ranking
15: $Q_k^{f,\lambda} = \{\}$
16: Q_C is the possible suggestions composed by adding a feature to the feature-query embedding (f, λ) of q, where
17: $\quad q_c \in Q_C$ implies $|q_c|$-$|q| \leq \delta$
18: compute dist between each pair of suggestions in Q_C
19: **for all** $i = 1 \ldots k$ **do**
20: $\quad Q_k^{f,\lambda} = Q_k^{f,\lambda} \cup q_{\max}$
21: $\quad Q_C = Q_C \setminus q_{\max}$
22: **end for**
23: **return** $Q_k^{f,\lambda}$

by adding a feature to the feature-query embedding (f, λ) of q, denoted as Q_C. For efficiency purposes, we further restrict that q' is increased by at most δ edges (Line 15). Line 16 computes the pairwise dist between suggestions in Q_C. In each iteration step (Lines 17–20), it adds the composed suggestion q_c that makes the util function of $Q_k^{f,\lambda}$ the largest. This step is repeated until it obtains k suggestions in $Q_k^{f,\lambda}$.

Greedy_global: In GREEDY_GLOBAL, GREEDY_LOCAL is invoked $O(|\mathcal{M}_q|)$ times (Line 2). In

each GREEDY_LOCAL call (Line 3), we obtain candidate suggestions (Line 4) and take the union with the top-k suggestions from each possible (f, λ) obtained so far (denoted as Q'). Line 5 computes the pairwise dist between suggestions in Q' *online*. In each iteration step (Lines 6–10), it adds the composed suggestion q' that makes the util function of Q'_k the largest. This step is repeated until it obtains k suggestions in Q'_k.

Remarks. The greedy algorithm involves a tradeoff between the efficiency and suggestion quality. GREEDY_LOCAL obtains 1/2OPT suggestions w.r.t a specific (f, λ), whose computation can be further optimized offline. Since $\mathcal{M}_q$ is only available online and the time for computing the user intent value of a query set, T_{util}, can be potentially long, GREEDY_GLOBAL is run only on Q' instead of Q_Cs of all possible (f, λ)s. Alternatively, when the query suggestion time is already acceptable, one may tune AUTOG to produce more suggestions for ranking as follows. (i) One may include more features, e.g., by lowering the minimum support and/or increasing the composability of c-prime features. (ii) AUTOG can be tuned to allow more overlapping features in query decomposition by using the parameter γ. (iii) One may set the maximum query increment size δ to a large value.

Complexity analysis. We analyze the asymptotic complexity of the greedy ranking algorithm (Algorithm 3.2). We remark that when determining the user intent value of query suggestions for ranking, the sampling method of selectivity estimation (presented in Section 3.3.3) is significantly more efficient when compared to the worst-case exponential-time computation of the query suggestions' diversity, which involves computing structural difference between graphs (dist) multiple times. For a succinct analysis, we assume that there is an oracle that efficiently provides selectivity estimation of a query and omit it in the analysis.

Greedy_local: The time complexity of the local greedy algorithm is simply $O(|Q_C| \times T_{compose} + k \times |Q_C| \times T_{util} + k \times |Q_k^{f,\lambda}|)$.

1. The first term is due to the addition of a feature to (f, λ), where λ is the embedding of f in the current query. $|Q_C|$ is the number of possible suggestions at (f, λ), which is of a modest value. $T_{compose}$ is the time for adding a feature in $\mathcal{F}$ to (f, λ) of q. Such an addition requires to check whether the feature contains a common subgraph with f. This requires subgraph isomorphism calls.

2. The second term is due to the time for computing the composed suggestion q_c that makes the util of $Q_k^{f,\lambda}$ the largest. Denote T_{util} to be the time for computing the user intent value of a set of suggestions. Recall that in each iteration of the greedy algorithm we augment $Q_k^{f,\lambda}$ with the suggestion from Q_C that gives the largest user intent value, which takes $O(|Q_C| \times T_{util})$ to find the largest one. The iteration is repeated k times. We remark that to compute the user intent value of $Q_k^{f,\lambda}$, it requires to compute the structural diversity of the $Q_k^{f,\lambda}$ and it involves the dist function. We adopt the technique presented at the end of Section 3.3.3 to optimize this step.

3. The last term is the time for outputting the top-k suggestions $Q_k^{f,\lambda}$ of the feature embedding (f, λ).

Greedy_global: The time complexity of the overall greedy algorithm is $O(|\mathcal{M}_q| \times T_{local} + k \times |Q'| \times T_{util} + k \times |Q'_k|)$.

1. The first term is due to the calls of local greedy algorithm of feature embeddings. Greedy_global calls Greedy_local once for *each* embedding. There are $|\mathcal{M}_q|$ embeddings in total. Recall humans often draw small queries. Hence, $|\mathcal{M}_q|$ is small in practice.

2. The second term is due to the ranking of Q' to obtain the overall top-k suggestions Q'_k. We remark that each Greedy_local call returns k suggestions. Hence, the number of query suggestions ranked by Greedy_global $|Q'|$ is $k \times |\mathcal{M}_q|$, which is of a modest number. In each iteration, the suggestion from $|Q'|$ that makes the value of util of Q'_k the largest (computed in T_{util} time) is added to Q'_k. The iteration is repeated k times.

3. Finally, the last term is the time for outputting the top-k suggestions $|Q'_k|$.

Based on the above analysis, we observe that the algorithm calls two worst-case exponential-time subroutines with graphs of small or modest sizes. In particular, they are (i) the subgraph isomorphism calls when determining the possible common subgraphs of feature pairs, and (ii) the structural difference between query suggestions for computing their diversities. In Section 3.3.3, we presented an optimization for computing suggestion diversities. Moreover, in the next section, we present an index to further optimize them.

3.4 INDEXED AUTOCOMPLETION–AUTOGI

To optimize the query autocompletion framework presented in Section 3.3, AutoG presents an index, called *Feature* DAG *Index* (FDAG), *and* its associated algorithms. It is the first structure that indexes features and records their structural information for query autocompletion, including *subgraph isomorphisms between features*, *features' automorphisms* and *auxiliary structural differences between query compositions*. We report the definition of FDAG and its operations in this subsection.

3.4.1 FEATURE DAG (FDAG) INDEX

Prior to the definition of FDAG, we present the design rationales of FDAG.

1. Greedy_local (Algorithm 3.2, Lines 14–15) involves adding a feature $f_j \in \mathcal{F}$ to a query q, via a feature f_i embedded in q, where f_j and f_i have a common subgraph cs, i.e., cs $\subseteq_{\lambda_i} f_i$ and cs $\subseteq_{\lambda_j} f_j$. Numerous suggestions may be potentially generated (i.e., $|Q_C|$ can be large). Determining all possible common subgraphs of two features is costly. Hence, FDAG indexes all subgraph isomorphic embeddings between features.

2. Suggestions that are formed via common subgraphs which are automorphic to each other are structurally equivalent. FDAG indexes automorphic embeddings of each feature so that automorphic suggestions are generated only once.

3. All possible pairwise feature compositions are enumerated and indexed so that the time for adding an increment to a feature ($T_{compose}$) is done in an FDAG lookup.

4. As motivated in Section 3.3.3, it has been known that determining structural differences between graphs is potentially costly (i.e., T_{util} can be large). Thus, FDAG indexes the auxiliary structure for determining structural differences between compositions (illustrated with Example 3.18) so that T_{util} is significant reduced.

5. The ranking function presented earlier involves selectivity estimations. FDAG indexes the graph IDs of a feature with a predefined sampling interval m.

The feature DAG (FDAG) index is then formally presented in Definition 3.19.

Definition 3.19 [FDAG] FDAG *is a* DAG (V, E, $\mathcal{M}$, anc, des, $\mathcal{A}$, ζ, η, $\mathcal{D}$), *where*

1. *V is a set of index nodes. Each node v represents a feature denoted as f_v. For presentation simplicity, we may often use f_v to refer to the index node;*

2. *$E : V \times V$ is a set of edges, $(v_i, v_j) \in E$ iff $f_{v_i} \subseteq f_{v_j}$; further, $\mathcal{M}$ is a function that takes an edge (v_i, v_j) as input, i.e., $f_{v_i} \subseteq f_{v_j}$, and returns the subgraph isomorphism embeddings of f_{v_i} in f_{v_j}, often denoted as $\mathcal{M}_{f_i,f_j}$; anc and des are functions that take an index node v as input and return its ancestor and descendant nodes, respectively;*

3. *$\mathcal{A}$ takes a feature f_v as input and returns the automorphism embeddings of f_v, often denoted as $\mathcal{A}_{f_v}$, where $\mathcal{A}$ is used to prune structurally identical composed graphs;*

4. *ζ is a function that takes a c-prime feature f_v as input and returns a set of composition records C as output, where each record in C is a 6-ary tuple (f_v, f_{v_j}, cs, λ_v, λ_{v_j}, F_l), where cs is the common subgraph of f_v and f_{v_j}, λ_v (resp. λ_{v_j}) specifies the embedding of cs in f_v (resp. f_{v_j}) and F_l is a set of features embedded in the composed graph;*

5. *η is a function that takes a pair of compositions as input and returns the auxiliary structural difference between the pair; and*

6. *$\mathcal{D}$ takes an index node f_v as input and outputs a sample of the IDs of the graphs that contain f_v. We often denote the graph IDs of f_v as D_{f_v}.*

In the worst case, FDAGs can be large. However, in practice, their sizes are far from the worst-case ones. We report some experimental results of AutoG on *PubChem* [Yi et al., 2017a] here. (i) The number of edges of the FDAG $|E|$ is $O(|\mathcal{F}|^2)$, where $\mathcal{F}$ is the feature set. The

experiments showed that the $|E|$ values were only $18\times|\mathcal{F}|$ on average. (ii) A feature pair may have exponentially many subgraph isomorphism embeddings between them. In practice, the average number of such embeddings per feature pair is around 2. (iii) Regarding automorphisms, the experiments show that about 12% of the features have multiple automorphisms. (iv) Each feature pair has around 120 possible compositions on average (indexed in ζ).

Example 3.20 Figure 3.7 illustrated a partial FDAG for *Pubchem* with a sketch. Suppose f_{10} is the current query. f_{13} is a possible increment to f_{10} that forms f_{22}, via f_4. Query suggestion f_{22} can be efficiently determined by using FDAG as follows.

f_4 is located from $\mathsf{anc}(f_{10})$. f_{13} is in $\mathsf{des}(f_4)$. f_{22} is retrieved from the compositions of f_{10} (i.e., $\zeta_{f_{10}}$) via an FDAG lookup. A trivial query composition is needed when the composed suggestion is not a feature. The composed suggestion (i.e., f_{22}) is ranked against other possible candidate suggestions (formed by other compositions of f_{10}). This is efficient because the intermediate results of the structural differences between compositions of f_{10} are recorded in η. □

3.4.2 AUTOCOMPLETION BY USING FDAG

We end this section by highlighting how FDAG optimizes the online ranking (Algorithm 3.2). Determining the set of possible candidate suggestions Q_C (Line 14) that have large util values (Line 18) are computationally costly. In Line 14 of Greedy_local, the query increments that can be composed with the feature f (at the location λ of q) is fixed. Therefore, the FDAG indexes all possible compositions C of f in ζ offline. Then, they can be efficiently retrieved online. Given a composition $c = (f, f', \mathsf{cs}, \lambda_1, \lambda_2)$ and a suggestion q', the constraint $|q'| - |q| \leq \delta$ can be then easily checked (Line 15). Then, the util function has the sel and dist components (Lines 16–18). sel can be efficiently estimated from D of FDAG. Regarding dists between q' and other suggestions constructed from (f, λ), some intermediate results had been indexed in η of FDAG. Further, since q' differs from q by at most δ edges, dist can be efficiently derived from η. Hence, T_{util} is reduced.

3.5 PERFORMANCE STUDY

This section reports some highlights of an experimental evaluation of the AutoG framework. It studies the suggestion quality of user studies and simulations on popular real and synthetic datasets.

Software. A prototype of AutoG has been implemented mainly in C++. The prototype uses VF2 for subgraph isomorphism test and the McGregor's algorithm [McGregor, 1982] (with minor adaptation) for determining mces. The interface of the prototype is shown in Figure 3.1. The gSpan implementation from Yan and Han [2002] is used for frequent subgraph mining.

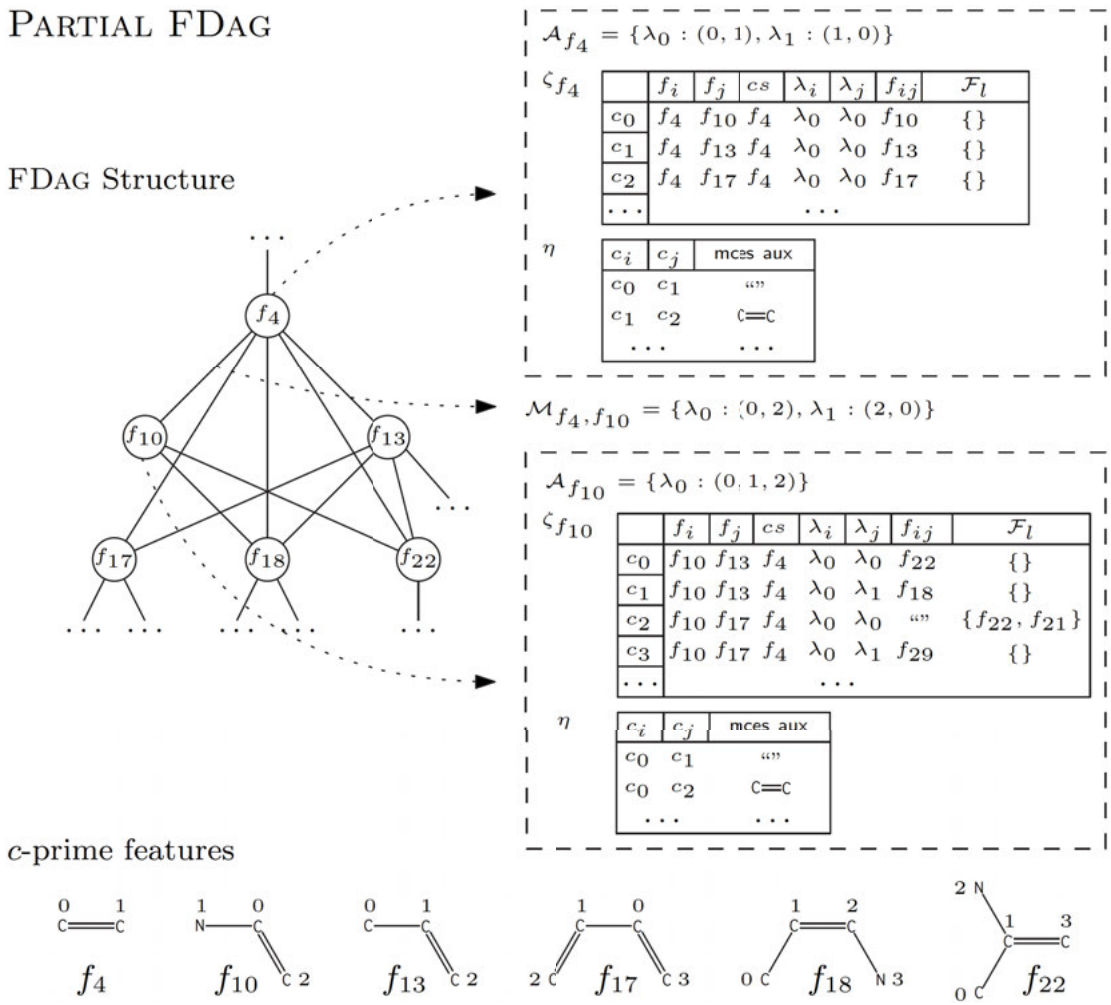

Figure 3.7: Major structures of the (partial) FDAG of *Pubchem*.

Hardware. All the experiments were run on a machine with a 2.67 GHz processor and 64 GB memory running the Linux OS. All the indexes were built offline once and loaded from a hard disk and were then fully memory-resident for online query suggestions.

Datasets. The datasets and query sets provided by iGraph [Han et al., 2010]. Their default settings were used. The two popular benchmarked real datasets are: (i) PubChem, a real chemical compound dataset consisting of 1 million graphs; and (ii) Aids(the AIDS Antiviral dataset), which consists of 10,000 graphs. The synthetic datasets are synthetic.10K.E30.D5.L20 and synthetic.10K.E30.D5.L80 (hereafter referred to as Syn-1 and Syn-2), both of which consist of

10,000 graphs. Table 3.2 shows some characteristics of the datasets: the number of graphs ($|\mathcal{D}|$), the average number of vertices and edges ($avg(|V|)$ and $avg(|E|)$), and the number of vertex and edge labels ($|\phi(V)|$ and $|\varphi(E)|$).

Table 3.2: Some characteristics of the datasets

Dataset	$\|\mathcal{D}\|$	avg($\|V\|$)	avg($\|E\|$)	$\|\phi(V)\|$	$\|\varphi(E)\|$
Aids	10 K	25.42	27.40	51	4
Syn-1	10 K	11.02	30.53	20	20
Syn-2	10 K	11.02	30.53	57	80
PubChem	1 M	23.98	25.76	81	3

Query sets. The query sets were taken from Han et al. [2010], with the query size ranging from 4–24. Each query set of a particular size contained 100 queries.

For time measurements, we reported the elapsed wall-clock times. It is known that there are large variations in subgraph query times. To avoid the reported times being governed by few long (or short) queries, we discarded the runtimes that were beyond two standard deviations from the mean. The reported runtime was the average of the remaining runtimes.

Mining of c-prime features. We ran gSpan to obtain sufficient features for AutoG to compose suggestions. In particular, the default minimum support value (*minSup*) for the real datasets is 10%. *minSup* is set to 5% for synthetic datasets simply because their frequent subgraphs are relatively scarce. The maximum feature size *maxL* was set to 10 for all datasets. Some statistics of the features are summarized in Table 3.3.

Table 3.3: Statistics of the feature sets

Dataset	minSup	$\|\mathcal{F}\|$	avg($\|V\|$)	avg($\|E\|$)
Aids	0.10	460	6.15	5.20
Syn-1	0.05	1,860	3.81	3.15
Syn-2	0.05	1,453	4.40	4.04
PubChem	0.10	1,206	7.44	6.47

Default AutoG settings. The default maximum increment size (i.e., δ) was set to 5, which is large enough to provide a large number of candidate suggestions. We set the default composability c to infinity unless we specifically studied its effects. There are five parameters (i.e., m, γ, ν, k, and $|q|$) of online AutoG processing. In the default setting, we set m to 4, γ and ν to 0.5, k to 10 and $|q|$ to 8, unless otherwise specified.

3.5.1 SUGGESTION QUALITY

Simulations. We first report the qualities of the suggestions from simulations under a large variety of parameter settings. Each simulation started with a random 2-edge subgraph as a target query. In each step, AutoG is called. Then, a useful suggestion (if present) with the largest size is chosen. If no suggestion was useful, the query was augmented with a random edge toward the target query. Each target query set contained 100 queries, which are publicly available [Yi et al., 2017]. To show suggestion qualities, we present several popular metrics, listed in Table 3.4.

We report some representative results from PubChem in Tables 3.5, 3.6, and 3.7. Table 3.5 shows the quality metrics of Q8 with various δs. We remark that same trends were observed on Q12 and Q16. Table 3.6 shows the quality metrics of various target query sizes. 1 Hit shows that AutoG suggestions were almost always used; and #AutoG shows that the suggestions were used in multiple iterations of query formulation; the numbers of edges added by AutoG were around 30%; and TPM shows that AutoG saved roughly 42% mouse clicks in query formulation.

Table 3.7 shows the suggestion quality when we varied k. Table 3.7 showed the qualities increased with k. It is not surprising because as more suggestions are returned, the higher chance some of them are adopted.

Table 3.8 shows the quality metrics of Q8 with various ν. The results showed that the suggestion qualities increased as the value of ν increased and were stable when $\nu \geq 0.1$. That is, when the factor of the selectivities of suggestions (e.g., $\nu \geq 0.1$) was adequately significant in the ranking, AutoG produced high quality suggestions. To obtain helpful suggestions, ν may be set to some values greater than 0.1 so that both selectivities and diversities involve in suggestion ranking.

User study. Next, we reported a user study with 10 volunteers. Each user was given 2 queries with high, medium and low TPM values, respectively, from the simulation. We randomly shuffled these 6 queries.[7] The users were asked to formulate the target queries via the visual aid shown in Figure 3.1. They expressed their level of agreement to the statement "AutoG *is useful when I draw the query.*" via a symmetric 5-level agree-disagree Likert scale, where 1 means "*strongly disagree*" and 5 means "*strongly agree*").[8]

The result showed that the correlation coefficient between TPMs and users' points is 0.96 and the p-value is 0.002. Therefore, TPM is a good quality indication of AutoG. The average ratings of the queries with high, medium and low TPM values are 4.55 (between "*strongly agree*" and "*agree*"), 2.95 ("*neither agree nor disagree*") and 1.65 (between "*disagree*" and "*strongly disagree*"), respectively.

[7] Readers may find the full list of queries for investigating the suggestion quality from the project site `https://goo.gl/Xr9MRY`. Further, a short video shows how users may interact with the AutoG prototype.
[8] The questionnaire used in the tests can be found at `http://goo.gl/dFRdwj`.

Table 3.4: Quality metrics and their meanings

Metric	Meaning
1 Hit	The % of queries that AutoG returned at least one useful suggestion
#AutoG	The average number of suggestions accepted in the simulation
AutoG |E|	The average number of edges obtained from suggestions
TPM	The *total profit metric* (TPM) adopted from Nandi et al. [2007], which quantifies the % of mouse clicks saved by AutoG in visual graph query formulation

Table 3.5: Quality metrics vs. δ (PubChem)

δ	1 Hit (%)	#AutoG	AutoG $\|E\|$	TPM (%)
1	99%	4.7	4.7	59%
2	99%	2.7	3.8	52%
3	98%	2.2	3.3	45%
4	99%	2.2	3.0	42%
5	98%	2.3	3.1	43%

Table 3.6: Quality metrics vs. $|q|$ (PubChem when $\delta = 3$)

$\|q\|$	1 Hit (%)	#AutoG	AutoG $\|E\|$	TPM (%)
8	98%	2.2	3.3	45%
12	99%	3.3	5.1	44%
16	100%	4.0	6.7	42%

Table 3.7: Quality metrics vs. k (PubChem when $\delta = 3$)

k	1 Hit (%)	#AutoG	AutoG $\|E\|$	TPM (%)
4	86%	1.5	2.0	26%
6	92%	1.8	2.5	34%
8	94%	2.0	2.9	40%
10	98%	2.2	3.3	45%

Table 3.8: Quality metrics vs. ν (PubChem when $\delta = 3$)

ν	1 Hit (%)	#AutoG	AutoG $\lvert E \rvert$	TPM (%)
0.00	63%	0.8	1.2	17%
0.02	86%	1.3	2.2	31%
0.04	83%	1.2	2.0	29%
0.06	86%	1.3	2.4	34%
0.08	90%	1.6	2.6	36%
0.10	93%	1.6	2.7	39%
0.20	98%	1.8	2.9	41%
0.40	97%	2.1	3.1	44%
0.60	98%	2.2	3.2	44%
0.80	98%	2.2	3.2	45%
1.00	99%	2.1	3.1	44%

3.5.2 EFFICIENCY

We report the average response times of query suggestions of all datasets (see Figures 3.8–3.11). For Aids and PubChem, we observed that the runtimes of global ranking (detailed in Greedy_global) increased much faster than those of the local one (detailed in Greedy_local), due to the costly online mces computations. For Syn-1 and Syn-2, we observed the (average) response times of the two phases exhibited linear trends. In all, the average response times are small when the query size is smaller than 16.

3.6 GUIDANCE FOR QUERIES OVER LARGE NETWORKS

In the preceding sections, we have described a guidance scheme for visual subgraph queries on a large collection of small- or medium-sized data graphs. In this section, we briefly describe efforts toward building such query formulation guidance strategies on large networks. Surprisingly, scant attention has been paid to this problem in the literature. *Orion* is an interactive visual query builder with suggestions [Jayaram, Goyal, and Li, 2015]. It iteratively suggests edges based on their relevance to the user's query intent, according to the partial query graph so far. Orion proposes query-specific random *decision paths* for suggestion edge ranking. The edge suggestion ranking uses statistics of data graphs, query logs, among others.

Figure 3.12 shows the user interface of the prototype of Orion. The red graph is the current query provided by a user. By using the Orion's ranking algorithm, several edge suggestions are produced and visualized. In addition, some tips for possible (user) actions are dynamically shown at the left hand side of the interface. If an edge suggestion is accepted by an user, it is called a

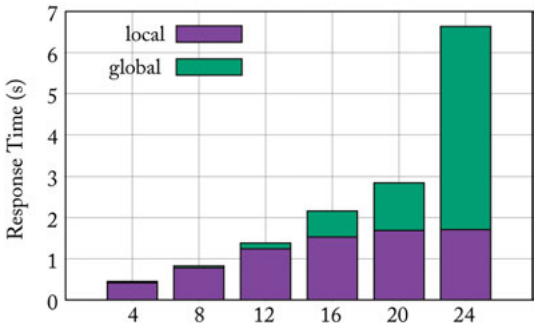

Figure 3.8: Performance breakdown (AIDS)—vary $|q|$.

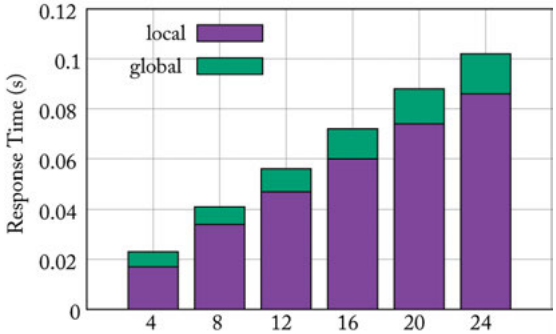

Figure 3.9: Performance breakdown (SYN-1)—vary $|q|$.

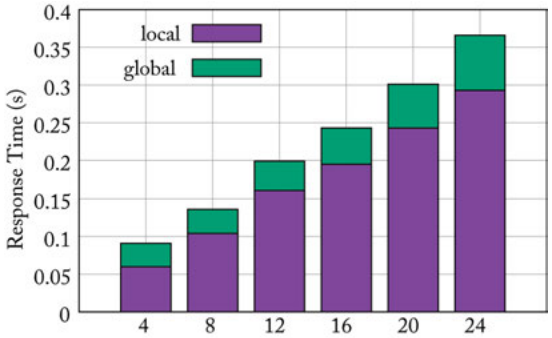

Figure 3.10: Performance breakdown (SYN-2)—vary $|q|$.

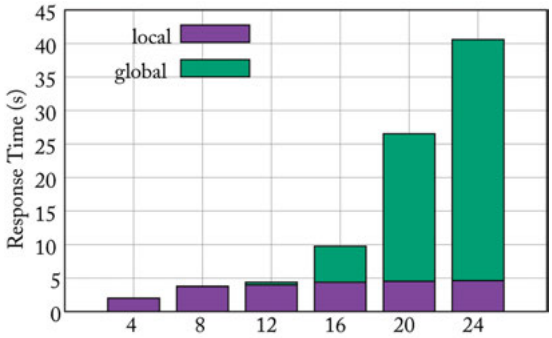

Figure 3.11: Performance breakdown (PUBCHEM)—vary $|q|$.

positive edge (the blue one). Otherwise, the edge suggestion is not adopted and it is considered a *negative edge*. These are considered user feedback and the user intent is derived from them.

3.7 BIBLIOGRAPHIC NOTES

The work reported in this chapter appeared in Jayaram, Goyal, and Li [2015], Yi et al. [2017a] and Yi et al. [2016]. Specifically, Yi et al. [2016, 2017a] are designed for a collection of small- or medium-sized data graphs whereas [Jayaram, Goyal, and Li, 2015] is an early effort for large networks. However, the general idea of query autocompletion has been studied in the context of XML since the last decade. Feng and Li [2012] proposed to extend keyword search autocompletion to XML queries. In Li, Yu, and Jagadish [2008], structures are associated with the query keywords. However, keyword searches are inherently difficult (if possible at all) to express struc-

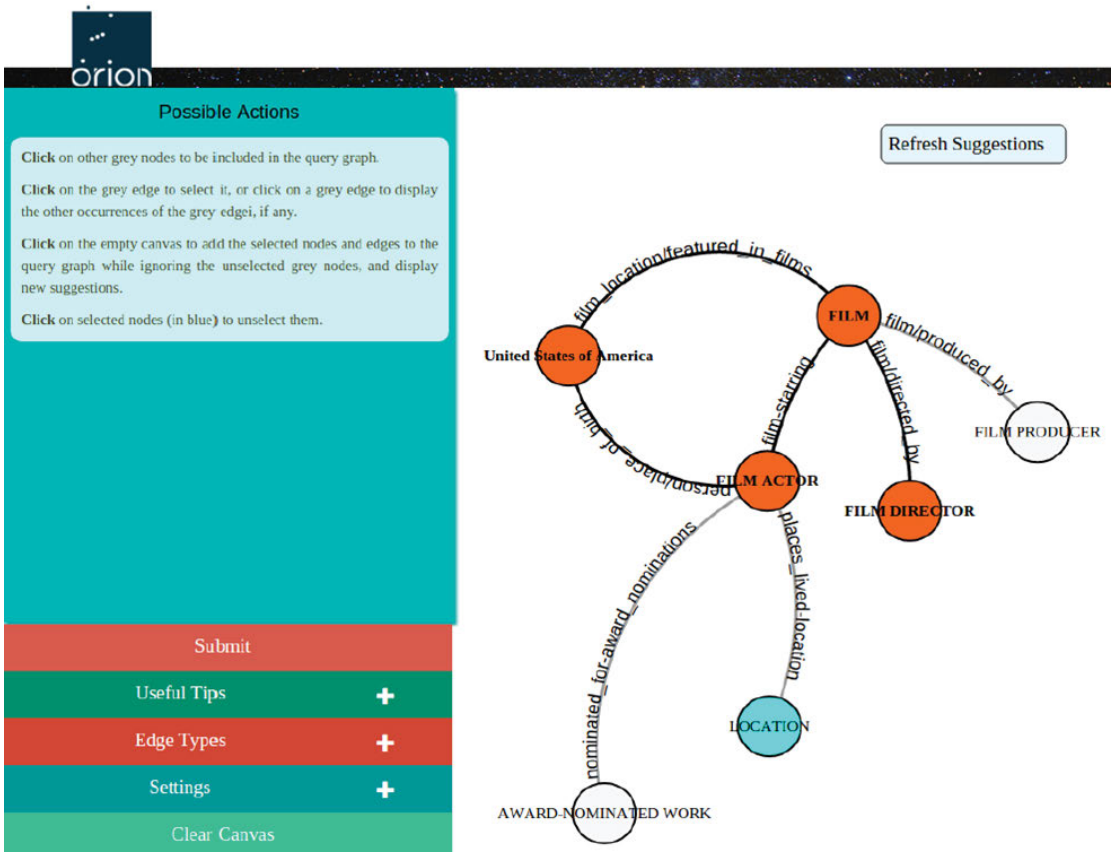

Figure 3.12: Orion interface [Jayaram, Goyal, and Li, 2015].

tural queries. LotusX provides position-aware autocompletion capability for XML [Lin et al., 2012]. Autocompletion learning editor for XML provides intelligence autocompletion [Abiteboul et al., 2012].

To alleviate the burdens of structural query composition, visual aids (or GUIs) have also been studied in the context of XML queries [Braga, Campi, and Ceri, 2005, Comai, Damiani, and Fraternali, 2001, Papakonstantinou, Petropoulos, and Vassalos, 2002]. For example, graphical constructs of XML queries (XML-GL) have been proposed [Comai, Damiani, and Fraternali, 2001]. QURSED provides a query editor for building reports [Papakonstantinou, Petropoulos, and Vassalos, 2002]. The XML Query By Example (XQBE) provides tools to express graphical constructs of complicated XML queries [Braga, Campi, and Ceri, 2005]. One possible reason why GUIs have received significant research attention is that XML data are structures. Their queries are tedious to compose, and they are naturally visualized as pictures.

The same arguments can be applied to graph databases, but their data and query languages are even more complex.

3.8 CONCLUSION

Subgraph queries on graph-structured data can be topologically complex. Hence, it is paramount to guide users in a visual environment to formulate such queries efficiently and easily. In this chapter, we present subgraph query autocompletion frameworks to alleviate this challenge. Specifically, we describe the framework of AutoG, which provides query suggestions to guide users to formulate their queries on a collection of small- or medium-sized data graphs. On the other hand, the query autocompletion problem on large networks is still in its infancy with Orion framework being the only notable solution.

CHAPTER 4

Blending Human Interactions and Graph Query Processing

Consider a large collection of small- or medium-sized data graphs. A popular query primitive on such collection is *subgraph search*, which retrieves one or more data graphs containing *exact* or *approximate* match of a user-specified query graph (i.e., subgraph query). In particular, given a *subgraph containment query* q on a set of data graphs $\mathcal{D}$, the aim is to find all data graphs in $\mathcal{D}$ in which q is a subgraph [Yan et al., 2004] (i.e., exact match). Note that q is a subgraph of a data graph $g \in \mathcal{D}$ if there exists a subgraph isomorphism from q to g. A common problem for this type of query is that in many occasions there may not exists any data graph that contains a subgraph which is isomorphic to the query. For example, consider the subgraph containment query in Figure 4.1a and the data graphs in Figures 4.1b and c. Observe that the query is not a subgraph of any of these data graphs. In this case, it is often useful to find out data graphs that *approximately* contain q. This problem is referred to as the *substructure similarity search* problem [Yan et al., 2005] and q is called a *subgraph similarity query*. For example, if we are allowed to miss at most two edges from the query in Figure 4.1a, then both the data graphs match it as they contain subgraphs that nearly (or approximately) contain the query graph (shown by dotted box). In this chapter, we refer to these exact and approximate subgraph search problems collectively as *substructure search* problem.

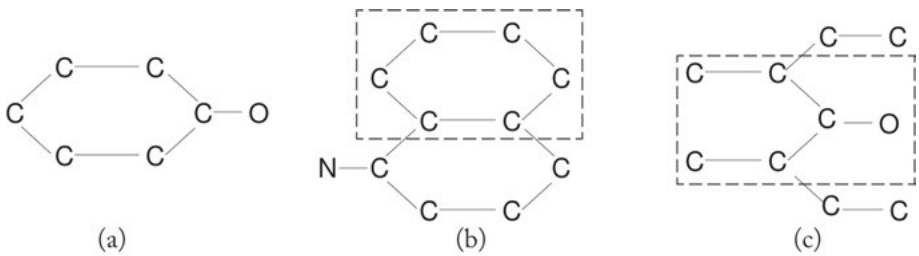

Figure 4.1: A query graph (a) and data graphs ((b) and (c)).

Due to the complexity of graph query languages, we can visually formulate a subgraph query by utilizing a graphical query interface (e.g., Figure 1.2) as introduced in preceding chapters. In traditional visual query processing paradigm, query evaluation in such a graphical framework can be performed in two key steps. First, the visual query is transformed into its textual

or algebraic form. Second, the transformed query is evaluated using an existing state-of-the-art graph query processing method. Observe that although the final query that a user intends to pose is revealed gradually in a step-by-step manner during query construction, it is not exploited by the query processor prior to clicking of the `Run` icon to execute the query. That is, query processing is initiated only *after* the user has finished drawing the query. This often results in slower *system response time* (SRT), which is the duration between the time a user presses the `Run` icon to the time when the user gets the query results [Jin et al., 2010]. As remarked earlier, this traditional view of visual graph query processing is primarily due to the fact that until recently the data management community has traditionally considered visual interface-related issues more relevant to the HCI community and orthogonal to data processing.

In this chapter, we take a non-traditional step toward exploring a novel graph query processing paradigm by blending these two orthogonal areas. Specifically, we *interleave* (i.e., *blend*) query construction and query processing to prune false results and prefetch partial query results in a single-user environment by exploiting the latency offered by the GUI-based query formulation [Jin et al., 2010, 2012]. The key benefits of this paradigm are at least two-fold. First, it significantly improves the SRT[1] as the query processor does not remain idle during visual query formulation by processing the potential subgraph query early based on "hints" received from the user. In traditional subgraph query processing paradigm, SRT is identical to the time taken to evaluate the entire query. In contrast, in this new paradigm SRT is the time taken to process a part of the query that is yet to be evaluated (if any). Second, as a visual query is iteratively processed during query formulation, it paves way for realizing efficient techniques that can enhance usability of graph databases such as query suggestion, empty result feedback, and exploratory search.

Query processing on graphs in this new paradigm is challenging for a number of reasons. First, the naïve strategy of matching every edge a user draws on the query canvas to the underlying database can be prohibitively expensive due to multiple subgraph isomorphism tests. How can we blend query evaluation and query construction so that it can minimize subgraph isomorphism tests? Further, the number of candidate graphs for subgraph isomorphism should be manageable during the entire period. Second, what type of indexing schemes should we have to support such query processing paradigm? Indexing mechanism in this new paradigm should be effective even when the entire query is not known and must be able to exploit typical users' interaction behaviors with the visual interface for efficient pruning and retrieval. As we shall see later, traditional state-of-the-art graph indexing schemes are not suitable for this purpose as they are oblivious to visual *actions* (or *steps*) taken by users during query construction. Further, they are primarily designed based on the assumption that the entire graph query is available. Third, the *prefetching-based* graph query processing (i.e., we prefetch partial results) must be completely transparent from the user. A user's interaction behavior with the visual interface

[1] From an end user's perspective, the SRT is crucial as it is the time a user has to wait before she can view the results.

should not be affected by the query processing strategy. In this chapter, we address all these issues. In summary, this chapter makes the following contributions.

- We introduce an innovative substructure search paradigm that blends visual graph query construction and query processing to iteratively prefetch candidate matches during query formulation.
- We present a unified framework for blending visual substructure search query formulation and processing that can be instantiating into different approaches.
- We describe novel offline and online indexes that exploit typical visual interaction characteristics of users to facilitate efficient pruning and retrieval of partial results matching query fragments.
- We present algorithms that implement this querying paradigm for substructure search queries by exploiting the indexing schemes. We also present an algorithm that supports efficient modification to candidate matches when a user modifies her query during formulation.
- Using real-world and synthetic datasets, we show the effectiveness and superiority of our querying paradigm in supporting substructure search over state-of-the-art methods based on the traditional paradigm.

The rest of the chapter is organized as follows. We begin by formally defining the *visual substructure search problem*. Next, we introduce our unified framework to address this problem in Section 4.2. We present the offline and online indexes in Sections 4.3 and 4.4, respectively. The blending of visual subgraph query formulation and processing is presented in Section 4.5. We discuss support for modification to visual queries in Section 4.6. Experimental results are presented in Section 4.7. We review related research in Section 4.8. The last section concludes the chapter. The key notations specific to this chapter are given in Table 4.1. In the following, for clarity, we distinguish between a node in a query/data graph and a node in indexes by using the terms "node" and "vertex", respectively.

4.1 VISUAL SUBSTRUCTURE SEARCH PROBLEM

Existing subgraph similarity search techniques measure similarity between two graphs using distance measures that are either based on *graph edit distance* [Gao et al., 2010] or *maximum common subgraph* [Jin et al., 2012, Shang et al., 2010, Yan et al., 2005]. The former measure defines similarity of two graphs by the least edit operations (insertion, deletion, and relabeling) used to transform one graph into another. The latter measure detects the *maximum common subgraph* (MCS) or *maximum connected common subgraphs* (MCCS) between the query graph and the data graphs, and measures the similarity based on the difference of the query graph and the MCS or MCCS. Similar to Jin et al. [2012] and Shang et al. [2010], we adopt MCCS to measure similarity between two graphs. Given two graphs G and Q, the MCCS is the largest

Table 4.1: Key symbols for this chapter

Symbol	Definition
$delId(g)$	A subset of $fsgIds(g)$ used in indexes
$a2fId(\cdot)$	Identifier of each vertex in A^{2}F-index
$a2iId(\cdot)$	Identifier of a DIF in A^{2}I-index
σ	Subgraph distance threshold
β	Fragment size threshold
e_ℓ	A new edge added by user
$S_\ell = (V_\ell, E_\ell)$	A SPIG
$\mathcal{L}_E(g)$	*Edge List* associated with the vertex $v \in V_\ell$ containing a list of labels of edges in g
$\mathcal{L}_{frag}(g)$	The *Fragment List* of a vertex $v \in V_\ell$ representing g
$freqId(g)$	Frequent id attribute of $\mathcal{L}_{frag}(g)$
$difId(g)$	DIF id attribute of $\mathcal{L}_{frag}(g)$
$\Phi(g)$	Frequent subgraph id set attribute of $\mathcal{L}_{frag}(g)$
$\Upsilon(g)$	DIF subgraph id set attribute of $\mathcal{L}_{frag}(g)$
S	A set of SPIGs
R_q	Identifiers of data graphs containing q
R_{free}	Identifiers of verification-free candidate graphs
R_{ver}	Candidate graphs that need verification
$dist(G, Q)$	Subgraph similarity distance between G and Q

connected subgraph of Q that is subgraph isomorphic to G, denoted as $mccs(G, Q)$. Specifically, we define the *subgraph similarity distance* between G and Q as the maximum number of edges that are allowed to be missed or deleted in Q in order to match G. Formally, given two graphs G and Q, the *subgraph similarity distance*, denoted as $dist(G, Q)$, is defined as follows:

$$dist(G, Q) = \left\lfloor (1 - \frac{|mccs(G, Q)|}{|Q|})|Q| \right\rfloor.$$

Observe that the smaller the $dist(G, Q)$, the more similar G and Q are. If $dist(G, Q) = 0$, then $Q \subseteq G$ and $G \subseteq Q$ (isomorphic to each other). It is worth noting that we chose the MCCS-based similarity measure instead of an edit distance-based measure due to difficulty in

obtaining accurate edit costs for the latter as well as more intuitive and easier interpretation of the former in a visual graph querying environment.

Definition 4.1 [Visual Substructure Search Problem] *Given a subgraph query Q visually constructed on a visual graph query interface, a graph database $\mathcal{D} = \{g_1, g_2, \ldots, g_n\}$, and a subgraph similarity distance threshold δ, the goal of visual substructure search problem is to retrieve all the graphs $g_i \in \mathcal{D}$ with $dist(g_i, Q) \leq \delta$ by interleaving (i.e., blending) formulation and processing of Q.*

Note that when $\delta = 0$, our problem reduces to finding all data graphs in $\mathcal{D}$ that are subgraph isomorphic to Q (i.e., subgraph containment search).

4.2 A UNIFIED FRAMEWORK

In this section, we present a unified framework to address the visual substructure search problem.

4.2.1 THE FRAMEWORK

Algorithm 4.3 outlines the unified framework for implementing the visual substructure search problem. Specifically, it provides a framework for blending formulation and processing of visual substructure search query (both subgraph containment and subgraph similarity search) as well as query modification during construction. We assume that an offline index I_O has been constructed from $\mathcal{D}$ prior to formulation of any query. Let q be a visual query being formulated by a user. Let *simFlag* be a boolean variable to indicate if q is a subgraph similarity query or a containment query (*true* or *false*, respectively). The framework monitors four visual actions, namely, `NewEdgeSet` for adding a set of edges to q, `SimQuery` for invoking subgraph similarity search, `Modify` for removing an existing edge in q, and `Run` for executing the current query graph. Observe that `NewEdgeSet`, `Modify`, and `Run` represent the add, modify, and run GUI actions, respectively, as introduced in Section 2.2.3.

When a user adds a new edge set *eStr* to q, the algorithm constructs and maintains an adaptive online index I_L for the edge set to facilitate blending of visual query formulation and processing (Line 5). If the user is interested in exact subgraph matches (i.e., *simFlag* is *false*), then it retrieves the candidates of q (stored in R_q) by leverage the offline and online indexes (Line 7). If R_q is empty, then it means that there is no exact match for q after the addition of e. Consequently, the user can either modify q or retrieve similar matches to q (Lines 8–9). If the user chooses the latter, then q is regarded as a subgraph similarity query and corresponding candidate matches are retrieved (Lines 15–17) by leveraging the indexes. On the other hand, if the user chooses the former then a user-selected edge is removed and I_L is updated (Lines 18–19). If the user clicks the `Run` button, then the constructed query q is processed to retrieve result matches (Lines 21–22). If q is a subgraph containment query, the exact results *Results* will be returned after conducting candidates verification (i.e., subgraph isomorphism test), if necessary,

Algorithm 4.3 *A Unified Framework*

Input : GUI *action*, a boolean variable *simFlag*, a query graph q, a candidate set R_q, a subgraph distance threshold δ, an offline index I_O, a graph database $\mathcal{D}$
Output : Query results *Results*

```
1:  Initialize eStr ← ∅
2:  if action is NewEdgeSet eStr then
3:     for all edge e ∈ eStr do
4:        q ← q + e
5:        I_L ← ConstructOnlineIndex(I_O, I_L, q, e)
6:        if simFlag = false then
7:           R_q ← ExactCandidates(I_O, I_L, q)
8:           if R_q = ∅ then
9:              action ← Modify q or invoke similarity search
10:          end if
11:       else
12:          R_q ← SimilarCandidates(I_O, I_L, q, δ)
13:       end if
14:    end for
15: else if action is SimQuery then
16:    simFlag ← true
17:    R_q ← SimilarCandidates(I_O, I_L, q, δ)
18: else if action is Modify with edge e then
19:    Update I_L
20: else
21:    if action is Run then
22:       Results ← ProcessQuery(I_O, I_L, q, δ)
23:    end if
24: end if
```

on R_q. Otherwise, if it is already a subgraph similarity query (i.e., *simFlag* is true), then a list of data graphs that match q approximately is returned to the user.

Note that in the above framework the expensive candidate verification step is performed only after the `Run` button is clicked and not during visual construction of the query fragments. Second, it allows a user to execute a query fragment any time during query formulation and not wait until the entire query graph is visually formulated. This facilitates exploratory search

as a user may formulate a partial query graph, execute it, browse the results, and then continue expanding the query [Huang et al., 2017].

Example 4.2 Let us illustrate our visual subgraph querying paradigm based on the aforementioned framework. Consider a graph-structured chemical compounds dataset.[2] We first mine and extract the frequent and infrequent graph fragments from this dataset using an existing frequent graph mining algorithm [Yan and Han, 2002]. These fragments are then used to construct the *feature-based indexes* to support efficient matching of frequent and infrequent query fragments, respectively, while formulating a visual query.

Suppose now a user formulates a visual subgraph containment query over this dataset using the GUI in Figure 1.2. The sequence of steps taken by the user to formulate this query is shown in Figure 2.5 (*Sequence 1*). After every visual step taken by the user, the current query fragment is evaluated by exploiting the latency offered by the GUI. For instance, after Step 1 the query fragment is a frequent fragment (see the *Status* column) and is efficiently evaluated using the offline feature-based index and an adaptive on-the-fly online index, and a set of identifiers of data graphs containing this fragment (denoted by R_q) is retrieved. Next, when the user draws Step 2, R_q is refined by filtering irrelevant matches using the indexes (the query fragment is still frequent). Observe that at Step 4, the query fragment evolves from frequent to an infrequent one. Consequently, the index related to infrequent fragments is probed and R_q is refined accordingly by leveraging on the online index. This continues until the user clicks on the `Run` icon, when the final query results are computed by performing subgraph isomorphism test *if necessary*.

For more than four decades, query formulation is traditionally considered orthogonal to query processing in data management research. Our visual querying paradigm brings a shift in this traditional thinking by blending human interactions during visual query formulation with query processing in the context of graph data.

Remark 4.3 Note that all the instantiations of the unified framework discussed in this book support edge-at-a-time query formulation paradigm. Consequently, this paradigm is not palatable for formulating large visual substructure queries as it is cumbersome and time consuming for a user to draw a large query by adding edges repeatedly. Hence, our subsequent solution designs for blending visual query formulation and query processing are catered for small-sized

[2] For simplicity, in the sequel we shall ignore the type of chemical bonds (i.e., single or double) among the atoms and simply show the interaction as an unweighted edge.

subgraph queries. It is worth noting that in practice subgraph queries are often small; 98% of real-life pattern queries have radius 1 and 1.8% have radius 2 [Gallego et al., 2011]. A more recent study of real-life SPARQL queries by Bonifati et al. [2017] also reported that 90.8% queries uses at most 6 edges.

4.2.2 GENERALITY OF THE FRAMEWORK

The above framework is general and can be instantiated into different algorithms as follows. In this chapter and the next we present instantiations where the offline index I_O comprises of two sets of feature-based *action-aware indexes* to handle a large collection of small- or medium-sized graphs and large networks. In Line 2, we only allow edge-at-a-time query construction instead of a set of edges (i.e., pattern-at-a-time) in the form of canned pattern. Line 5 can adopt different online indexing schemes. Particularly, in this chapter I_L is instantiated to a *spindle-shaped graph* (SPIG) for *each* new edge. In the next chapter, we shall instantiate it with an index called gSPIG that is designed for supporting our querying paradigm on large networks. The generation of exact and similar candidates in Lines 7, 12, and 17 can also leverage different strategies. Line 19 can adopt different modification policies. In this chapter, the SPIG set is modified whereas in the next chapter the gSPIG index is updated. Line 22 also can be implemented using different subgraph verification techniques and different instantiations of I_O and I_L.

4.2.3 AN INSTANTIATION

An instantiation of the unified framework for a large collection of small- or medium-sized data graphs is outlined in Algorithm 4.4. When a user adds a new edge e_ℓ to q, the algorithm first constructs the SPIG S_ℓ (Line 4). If *simFlag* is *false*, it retrieves the FSG identifiers of q (R_q) by invoking the *ExactCandidates* procedure (Line 6). If R_q is empty, then there is no exact match for q after the addition of e_ℓ. Consequently, it gives the user options to either modify q ($action$ is `Modify`) or enable retrieval of approximate matches (*action* is `SimQuery`) by popping out an option dialogue box (Line 8). If the user chooses to modify q, then it provides suggestion on which edge she should delete in order to ensure R_q is not empty. The user may select the suggested modification or perform a different modification to q. These steps are encapsulated in the procedure *QueryModification* (Line 14). On the other hand, if the user intends to continue formulating the query without modification (*action* is `SimQuery`), then the algorithm identifies q as a subgraph similarity query. The *SimilarCandidates* procedure retrieves the candidate data graphs that match approximately with q by exploiting the SPIG set $\mathcal{S}$ (Line 17). These steps are repeated for each new edge until the user clicks the `Run` icon (Line 18). If $simFlag$ is *false*, then the exact results *Results* are returned from the candidate graphs (Line 20). If *Results* is empty after candidate verification (subgraph isomorphism test) then the substructure similarity search is invoked to retrieve approximate matches (Lines 22–23). Otherwise, if it is already a substructure similarity search (*simFlag* is *true*), then a list of data graphs that match the query approximately

Algorithm 4.4 *An instantiation of the framework for a collection of small- or medium-sized graph*

Input : GUI $action$, query q, candidate set R_q, subgraph distance threshold σ, graph database $\mathcal{D}$
Output : Query results *Results*

```
1:  Initialize SPIG set S = ∅
2:  if action is NewEdge then
3:     q ← q + e_ℓ
4:     S_ℓ ← SpigConstruct(q, Q, e_ℓ, S) /*Algorithm 4.7*/
5:     if simFlag = false then
6:        R_q ← ExactCandidates(S_ℓ.v_target) /*Algorithm 4.8*/
7:        if R_q = ∅ then
8:           action ← OptionDialogueDisplay()
9:        end if
10:    else
11:       (R_free, R_ver) ← SimilarCandidates(q, σ, S) /*Algorithm 4.9*/
12:    end if
13: else if action is Modify then
14:    q ←QueryModification(q, R_q, S, σ) /*Algorithm 4.11*/
15: else if action is SimQuery then
16:    Set simFlag = true
17:    (R_free, R_ver) ← SimilarCandidates(q, σ, S)
18: else if action is Run then
19:    if simFlag = false then
20:       Results ← ExactVerification(R_q)
21:       if Results = ∅ then
22:          (R_free, R_ver) ← SimilarCandidates(q, σ, S)
23:          Results ← SimilarResultsGen(q, R_free, R_ver, σ) /*Algorithm 4.10*/
24:       end if
25:    end if
26: else
27:    Results ←SimilarResultsGen(q, R_free, R_ver, σ)
28: end if
```

is returned to the user. This step is encapsulated in the procedure *SimilarResultsGen* (Line 27). In subsequent sections, we shall elaborate on these procedures.

4.3 ACTION-AWARE INDEXING

In this section, we present two indexing schemes, namely *action-aware frequent index* (A^2F) and *action-aware infrequent index* (A^2I), to support efficient matching of frequent and infrequent query fragments, respectively, while formulating a visual query graph. Our indexing schemes are *user action-aware*. That is, the structure of the index is designed to take advantage of typical actions a user undertakes in order to formulate a visual graph query. We begin by identifying the key features of such an *action-aware* index. In the sequel, we assume that frequent fragments are mined from the database using an existing technique e.g., *gSpan* [Yan and Han, 2002].

4.3.1 KEY FEATURES OF ACTION-AWARE INDEX

A visual graph query can be formulated in different ways by following different sequences of GUI actions. Figure 2.5 shows two different sequences of visual actions (also referred to as *steps*), denoted by *Sequence 1* and *Sequence 2*, a user may undertake to formulate the query in Figure 1.2. We can make the following observations related to the query formulation process.

- Visual query formulation in our framework follows a "edge-at-a-time" approach where a user incrementally adds new nodes or edges in the *Query Panel* (Panel 3 in Figure 1.2). Consequently, after every step the size of the query fragment grows by one. In the case, a query is modified then the size may decrease by one due to deletion of an edge. Consequently, the query size typically increases or decreases incrementally. Besides, the structure of the query fragment can evolve from a path to a tree or graph.
- At any step, the partial query graph formulated thus far, is either a frequent or infrequent fragment. Typically, as more edges are added, the chance of a query to remain frequent diminishes. Once it becomes infrequent, it remains as infrequent for rest of the formulation steps (unless one or more edges are deleted). For instance, in Figure 2.5 the partial query evolved from a frequent fragment to an infrequent one after Step 4 in *Sequence 1* whereas it becomes infrequent after the second step in *Sequence 2*.

In our proposed paradigm of blending visual query formulation and query processing, it is important to filter negative results after every visual action taken by a user. Consequently, we need an efficient indexing scheme which can exploit the above visual interaction characteristics effectively to prune false results. We envisage that such an *action-aware* indexing scheme should support the following key features.

- It should be able to prune a part of irrelevant results even if only *partial* query graph is known during query formulation.

- Since the size of a partial query graph g' grows by one, given a list of graphs that satisfy the fragment g' in *Step* i, it is important to support efficient strategy for identifying the graphs that match the fragment g'' (generated at *Step* $i+1$) where $g' \subset g''$ and $|g''| = |g'| + 1$.

- A partial query graph may evolve from being a frequent fragment to an infrequent one in the database. Furthermore, it may also evolve from a simple path to a complex graph structure. Hence, the proposed strategy should be able to support pruning based on both graph-structured frequent and infrequent fragments.

- Since smaller fragments always appear more often in different visual queries compared to larger-sized fragments, smaller-sized graph fragments should be efficiently indexed to support fast retrieval.

- Last, since subgraph isomorphism testing is known to be NP-complete, the indexing scheme should minimize expensive candidate verification while retrieving partial results.

4.3.2 ACTION-AWARE FREQUENT (A^2F) INDEX

A challenge in creating an index for frequent fragments is that the frequent fragment set can be large for a small α and hence the index built on the frequent fragments can be too large to fit in the main memory. Then, the performance of repeated evaluation of partial query fragments may degrade as the processing needs frequent disk access. To address this issue, similar to FG-Index [Cheng et al., 2007], we create a *memory-resident* and a *disk-resident* components of A^2F index. We refer to them as *memory-based frequent index* (MF-index) and *disk-based frequent index* (DF-index), respectively.

How do we determine which frequent fragment should reside where? To answer this question, we take a different strategy compared to FG-Index by exploiting a key feature of visual query formulation. Recall that the construction of visual queries always grows incrementally from small to larger-sized query fragments. Consequently, smaller frequent fragments are processed more frequently in various visual queries compared to their larger counterparts. We exploit this feature to determine where a frequent fragment should reside. Specifically, small-sized frequent fragments (frequently utilized) are stored in MF-index whereas larger frequent fragments (less frequently utilized) reside in DF-index. Formally, let $\beta \geq 1$ be the *fragment size threshold*. If $g \in \mathcal{F}$ and $|g| \leq \beta$, then index g into the MF-index. Otherwise, index g into the DF-index. Note that the sizes of MF-index and DF-index can be tuned by adjusting β based on the average size of typical queries and availability of memory. For instance, when β is the maximal size of frequent fragments, all the frequent fragments are indexed in MF-index. Even though it is faster to match frequent fragments in MF-index, it occupies larger memory space. In contrast, if β is too small, most of the frequent fragments are indexed in the DF-index and query processing needs to frequently access the disk. We now elaborate on the structure of these two types of frequent index.

Disk-based A^2F index (DF-Index). Informally, DF-index is an array of *fragment clusters*. A *fragment cluster* is a directed graph $\mathcal{C} = (V_C, E_C)$ where each vertex $v \in V_C$ is a frequent fragment g where $|g| > \beta$. There is an edge $(v', v) \in E_C$ *iff* $g' \subset g$ and $|g| = |g'| + 1$. We denote the root vertex (vertex with no incoming edge) of $\mathcal{C}$ as $root(\mathcal{C})$. Each fragment g of v is represented by its CAM code [Huan et al., 2003], denoted as $cam(g)$. We choose the maximal code among all possible codes of a graph by lexicographic order as this graph's canonical code.

Each vertex with fragment g in $\mathcal{C}$ points to a set of FSG identifiers of g (denoted as $delId(g)$ where $delId(g) \subseteq fsgId(g)$). Note that it is not space efficient to attach the complete list of FSG identifiers of g on each frequent fragment in the index as the size can be large when α is close to 1. Fortunately, the following property holds: given $g, g' \in \mathcal{F}$, if $g' \subset g$ then $fsgId(g) \cap fsgId(g') = fsgId(g)$ [Cheng et al., 2007]. That is, vertex v' (representing g') and its child vertex v (representing g) shares a large number of FSGs. We exploit this property to make the index more space-efficient. We elaborate on this with a simple example. Figure 4.2 depicts an example of DF-index ($\beta = 3$) based on the frequent fragments in Figure 2.1. In the fragment cluster $\mathcal{C}_2$, we assign those FSG ids to f_9 that are not in f_{14}. Since $fsgId(f_{14}) \subset fsgId(f_9)$, $|delId(f_9)| = |fsgId(f_9)| - |fsgId(f_{14})| = 200$. For the leaf vertex (i.e., vertex with no outgoing edges) f_{14}, $delId(f_{14}) = fsgId(f_{14})$. Also observe that we can retrieve the identifers of all FSGs of g by traversing all its children and adding them together. For instance, in the case of fragment cluster $\mathcal{C}_1$, $fsgId(f_8) = delId(f_8) \cup delId(f_{11}) \cup delId(f_{12}) \cup delId(f_{13})$.

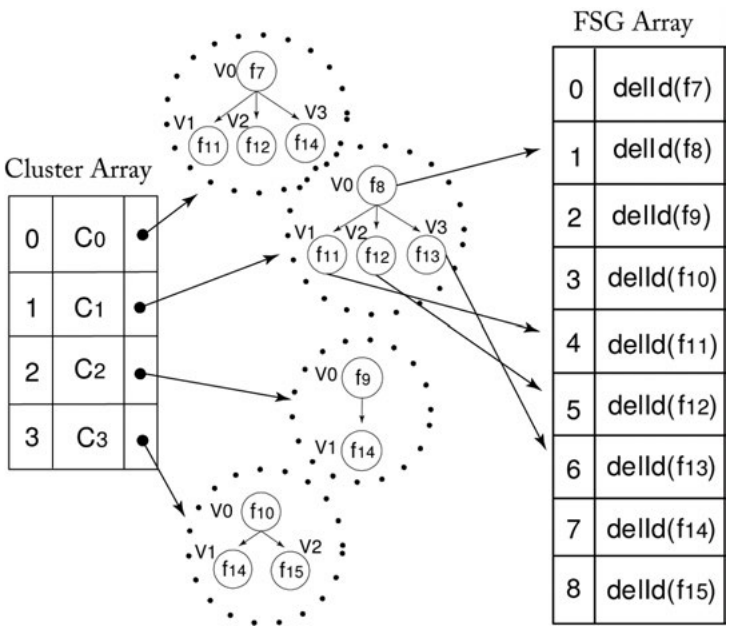

Figure 4.2: An example of DF-index.

Definition 4.4 (**DF-index**) *Given a set of frequent fragments $\mathcal{F}$ in a graph database $\mathcal{D}$ and fragment size threshold β, an* DF*-index constructed on $\mathcal{F}$ consists of the following components.*

- *An array, called Cluster Array (*CA*), stores a collection of fragment clusters. Let $CA[i]$ be the i-th entry in the* CA*. The fragment cluster stored in $CA[i]$ is assigned an identifier $\mathcal{C}_i$.*
- *A fragment cluster $\mathcal{C}_i$ is a graph $\mathcal{C}_i = (V_{\mathcal{C}_i}, E_{\mathcal{C}_i})$ where $v \in V_{\mathcal{C}_i}$ represents a frequent fragment $g \in \mathcal{F}$ such that $|g| > \beta$. Each $(v', v) \in E_{\mathcal{C}_i}$ represents the parent-child relationship between two vertexes. Fragment g is the child of g' iff $g' \subset g$ and $|g| = |g'| + 1$.*
- *An array, called* FSG *Array (*FA*), stores delId list of distinct frequent fragments in* CA*.*
- *$\forall\, v \in V_{\mathcal{C}_i}$ and $\forall i$, $v = (cam(g), j))$ where $cam(g)$ is the* CAM *code of g and $FA[j]$ contains $delId(g)$.*

Main memory-based A^2F index (MF-Index). The MF-index indexes all frequent fragments having size less than or equal to β. Similar to a fragment cluster, it is a directed graph $G_M = (V_M, E_M)$ where the vertices and edges have same semantics as $\mathcal{C}$. In addition, vertices representing frequent fragments of size β are leaf vertices in G_M and do not have any child fragments. Each leaf vertex $v \in V_M$ representing a fragment g, is additionally associated with a *fragment cluster list* $\mathcal{L}$ where each entry $\mathcal{L}_i$ points to a fragment cluster $\mathcal{C}_j$ in the DF-index such that $g \subset root(\mathcal{C}_j)$.

Definition 4.5 (**MF-index**) *Given a set of frequent fragments $\mathcal{F}$ in a graph database $\mathcal{D}$ and fragment size threshold β, an* MF*-index constructed on $\mathcal{F}$ is a graph $G_M = (V_M, E_M)$ where $v \in V_M$ represents a frequent fragment $g \in \mathcal{F}$ and satisfies the following conditions.*

- *For each $v \in V_M$, $|g| \leq \beta$.*
- *If v is not a leaf vertex then $v = (cam(g), delId(g))$ where $cam(g)$ is the* CAM *code of g and $delId(g)$ is a list of* FSG *identifiers of g s.t. $delId(g) \subset fsgId(g)$.*
- *If v is a leaf vertex then $v = (cam(g), delId(g), \mathcal{L})$ where $\mathcal{L}$ is a list of fragment cluster identifiers of g and $delId(g) = fsgId(g)$. Let $\mathcal{L}_i$ be the i-th entry of $\mathcal{L}$. Then, $\mathcal{L}_i$ contains an index j of* CA *such that $CA[j] = \mathcal{C}_j$ and $g \subset root(\mathcal{C}_j)$.*
- *Each $(v', v) \in E_M$ represents the parent-child relationship between two vertexes. Fragment g is the child of g' iff $g' \subset g$ and $|g| = |g'| + 1$.*

Example 4.6 Figures 4.2 and 4.3 depict DF-index and MF-index, respectively, built based on the fragments listed in Figure 2.1 and $\beta = 3$. The fragments f_0 and f_1 are chosen as the root vertices in the MF-index as they have the least size ($|f_0| = |f_1| = 1$). Since f_2 and f_3 are supergraphs of f_0 and f_1 with one additional edge, they are connected to f_0, f_1 as their children,

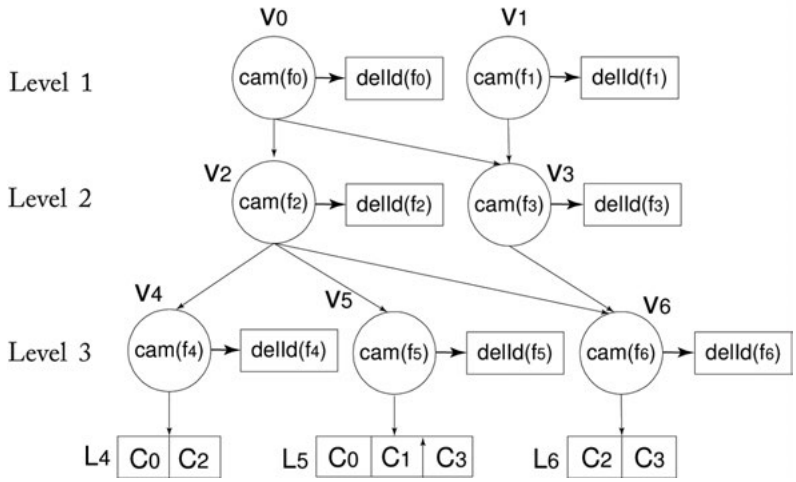

Figure 4.3: An example of MF-index.

respectively. Similarly, f_4, f_5, and f_6 are inserted into the MF-index. Since the sizes of these fragments are 3, they are leaf vertices in the MF-index (Figure 4.3).

Next, we create a set of fragment clusters for each leaf node in the MF-index and insert them into the Cluster Array of DF-index (Figure 4.2). Since f_7 is the child of f_4 with size 4, we create a fragment cluster, denoted as $\mathcal{C}_0$, containing f_7 and it's children f_{11}, f_{12}, and f_{14}. Note that $root(\mathcal{C}_0) = f_7$. $\mathcal{C}_0$ is added to f_4's cluster list $\mathcal{L}_4$. We also add $delId(f_7)$, $delId(f_{11})$, $delId(f_{12})$, and $delId(f_{14})$ in the array FA of the DF-index. Similarly, we build the fragment clusters $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$ and add them in CA.

Algorithm for building A^2F index. Algorithm 4.5 shows the top-down approach of the A^2F-index construction. Firstly, the frequent fragments are sorted in ascending order based on their size (Line 1). All frequent edges are indexed in the MF-index (Line 2). Given $g_i \in$ A^2F-index, $g_j \in \mathcal{F}$, if $g_i \subset g_j$ and g_j has one more edge than g_i, then g_j is a child of g_i (Line 4). Note that the cost of subgraph isomorphism test here is not significant as the frequent fragments are already sorted by their size. Consequently, for a given frequent fragment we just need to check only those fragments that have one additional edge. If $|g_i| = \beta$, g_i is a leaf vertex in the MF-index. Consequently, g_j should reside in the DF-index. If $g_j \notin$DF-index, the algorithm indexes g_j as the root of fragment cluster $\mathcal{C}_k$ and insert $\mathcal{C}_k$ in the cluster array of DF-index (Lines 6–9). It inserts g_j's fragment cluster id in g_i's fragment cluster list (Line 10). If $|g_i| > \beta$, then g_j is indexed in the same fragment cluster as g_i. Otherwise, it indexes g_j in the MF-index (Lines 12–16). Then it connects g_i and g_j with an edge and updates g_i's FSG id entries by deleting g_j's FSG ids (Lines 17–18). This process is repeated until all the frequent fragments are indexed.

Algorithm 4.5 ***BuildA2FIndex***

Input : A set of frequent fragments $\mathcal{F}$, fragment size threshold β
Output : MF-index and DF-index

```
1:  Sort F by size ascending order
2:  Index each |g| = 1, g ∈ F in MF-index
3:  for g_i ∈ A²F-index, g_j ∈ F do
4:     if g_i ⊂ g_j and |g_j| = |g_i| + 1 then
5:        if |g_i| = β then
6:           if g_j ∉DF-index then
7:              Index g_j in C_k, k++
8:              Insert C_k in DF-index
9:           end if
10:          Add g_j's fragment cluster id in g_i.L
11:       else
12:          if |g_i| > β then
13:             Index g_j in the same fragment cluster as g_i
14:          else
15:             Index g_j in MF-index
16:          end if
17:          Connect g_i and g_j with an edge
18:          delId(g_i) = delId(g_i) − fsgId(g_j)
19:       end if
20:    end if
21: end for
```

4.3.3 ACTION-AWARE INFREQUENT (A^2I) INDEX

The A^2I-index indexes infrequent fragments to prune the candidate space for infrequent queries. In order to ensure that the index is space-efficient, we index only the discriminative infrequent fragments (DIFs) $\mathcal{I}_d$ instead of infrequent fragments $\mathcal{I}$ as often in practice $|\mathcal{I}_d| \ll |\mathcal{I}|$. Intuitively, it consists of an array of discriminative fragments (denoted as DFA) arranged in ascending order of their sizes. Each entry in DFA stores the CAM code of $g \in \mathcal{I}_d$ and a list of FSG identifiers of g ($fsgId(g)$). Figure 4.4 depicts an example of A^2I-index constructed using the discriminative fragments in Figure 2.1. For instance, $cam(f_{16})$ is stored in $DFA[0]$. Also, $DFA[0]$ has a pointer to the list of FSG identifiers of f_{16}. Note that as the support of each DIF is less than $\alpha|\mathcal{D}|$, it is possible to store A^2I-index in the memory.

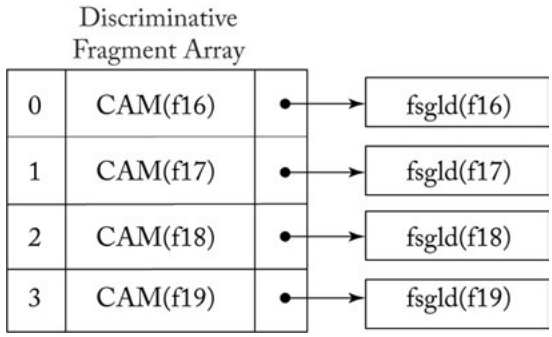

Figure 4.4: An example of A^2I index.

Definition 4.7 (**A^2I-index**) *Given a set of discriminative infrequent fragments $\mathcal{I}_d$ in a graph database $\mathcal{D}$, an* A^2I*-index constructed on $\mathcal{I}_d$ consists of an array, called Discriminative Fragment Array (*DFA*), which stores $\mathcal{I}_d$. Let DFA$[i]$ be the i-th entry in the* DFA*. Then, DFA$[i] = (cam(g_k), fsgId(g_k))$ where $g_k \in \mathcal{I}_d$. Further, if $i < j$ are indexes of* DFA *then $|g_i| \leq |g_j|$.*

Algorithm for building A^2I index. The algorithm for building A^2I-index is shown in Algorithm 4.6. We denote the sets of infrequent and frequent fragments with only one edge as $\mathcal{I}_1$ and $\mathcal{F}_1$, respectively. First, we retrieve $\mathcal{I}_1$ from $\mathcal{D}$ and index them in the A^2I-index (Line 1). Next, we add a frequent edge e_j on the frequent fragment g_i to form a new graph g_{new} (Lines 2–3). Obviously, there are different ways to construct such a new graph by adding e_j to different nodes of g_i as elaborated later. If g_{new} is not a frequent fragment and does not exist in DFA, then it checks if g_{new} is a discriminative fragment using Theorem 2.4 (Line 4). The algorithm retrieves the identifiers of FSGs of g_{new} from $fsdId(g_i)$ using subgraph isomorphism test (Line 5). If g_{new} exists, then it adds g_{new} and $fsgId(g_{new})$ in the A^2I-index (Lines 6–7). The algorithm repeats this process until no new fragment is generated.

Observe that there are two ways to add a frequent edge on a frequent fragment: (a) connect a new node on a node of the frequent fragment and (b) connect two existing nodes of a frequent fragment without introducing a new node. Let $g \in \mathcal{F}$ has n nodes and K be the number of frequent edges in $\mathcal{F}$. Let $|g_{new}|$ be the number of newly generated graphs of g. Then the largest possible value of $|g_{new}|$ is as follows: $|g_{new}| = Kn + (\max(|g|) - \min(|g|))$. The computational complexity to evaluate this equation is $O(n^2)$. We remove the frequent fragments, newly generated fragments that do not exist in the database, and existing discriminative fragments from the newly generated graphs at each step. Furthermore, as subgraph isomorphism test is used to retrieve $fsgId(g_{new})$ (Line 5), it is important to reduce the size of FSG space. We achieve this by considering only the FSGs in $\mathcal{D}_{g_i}$ instead of $\mathcal{D}$.

Algorithm 4.6 ***BuildA2IIndex***

Input : $\mathcal{F}$, $\mathcal{D}$
Output : MF-index and DF-index

```
1:  Get I_1 from D to A^2I-index
2:  for g_i ∈ F, e_j ∈ F_1 do
3:      g_new = g_i + e_j
4:      if g_new ∈ I_d and g_new ∉ DFA then
5:          fsgId(g_new) ← Retrieve FSG identifiers of g_new
6:          if |fsgId(g_new)| > 0 then
7:              Add g_new in A^2I-index with fsgId(g_new)
8:          end if
9:      end if
10: end for
```

4.4 SPINDLE-SHAPED GRAPH (SPIG)

We now present the adaptive online index called *spindle-shaped graph* (SPIG) that is created on the fly during query formulation. For each *new* edge e_ℓ created by the user, we create a SPIG. We allocate each edge a unique identifier according to their formulation sequence. That is, the ℓ*-th* edge constructed by a user is denoted as e_ℓ where ℓ is the *label* of the edge. The edge with the *largest* ℓ is referred to as *new edge* (most recently added). For example, in Figure 2.5 (Sequence 1) after Step 4, four edges have been constructed and they are uniquely identified as e_1 to e_4. The new edge is e_4 (C–C) as $\ell = 4$ is largest in this step.

A SPIG is a directed graph $S_\ell = (V_\ell, E_\ell)$ where each vertex $v \in V_\ell$ represents a subgraph g of the query fragment containing the new edge e_ℓ. In the sequel, we refer to a vertex v and its associated query fragment g interchangeably. There is a directed edge from vertex v' to vertex v if $g' \subset g$ and $|g| = |g'| + 1$. Each v is associated with the CAM code of the corresponding g, a list of labels of edges of g, and a list of identifier set called *Fragment List* to capture information related to frequent or infrequent nature of g or its subgraphs.

A *Fragment List* contains four attributes, namely *frequent id*, DIF *id*, *frequent subgraph id set*, and DIF *subgraph id set*.

- If g is in A^{2}F-index or A^{2}I-index, then the identifier of the vertex or entry representing g in the corresponding index is stored in *frequent id* or DIF *id* attribute, respectively. We denote the identifier of a vertex or an entry in A^{2}F-index or A^{2}I-index by *a2fId*(g) or *a2iId*(g), respectively.

- If g is neither in A^2F-index nor in A^2I-index, then the *frequent subgraph id set* stores the frequent ids of all *largest* proper subgraphs of g that are in A^2F-index. Note that the size of these subgraphs is $|g| - 1$. The DIF *subgraph id set* of g contains the DIF ids of all subgraphs of g that are indexed by A^2I-index.

The *source* vertex (vertex with no incoming edge) in the first level of S_ℓ, denoted by $S_\ell.v_{source}$, represents e_ℓ and the *target* vertex (vertex with no outgoing edge) in the last level, denoted by $S_\ell.v_{target}$, represents the entire query fragment at a specific step. Since there is only one vertex at the first and the last level and a set of vertices in the "middle" levels, the shape of S_ℓ is like a spindle.

Definition 4.8 (**Spindle-shaped Graph (SPIG)**) *Let e_ℓ be the new edge added to a visual graph query q during Step ℓ. Then, the SPIG of e_ℓ is a directed graph $S_\ell = (V_\ell, E_\ell)$ that satisfies the following conditions.*

- *For each $v \in V_\ell$, $\exists$ an injective function $f : v \rightarrow f(g)$ s.t. e_ℓ is contained in g and $g \subseteq q$.*
- *By abusing the notations of trees, each $(v', v) \in E_\ell$ represents the parent-child relationship between two vertices v' and v where v is the child of v' iff $g' \subset g$ and $|g| = |g'| + 1$.*
- *Each $v \in V_\ell$ is a 3-tuple $v = (cam(g), \mathcal{L}_E(g), \mathcal{L}_{frag}(g))$ where $cam(g)$ is the* CAM *code of g, $\mathcal{L}_E(g)$ is the Edge List containing a list of labels of edges in g, and $\mathcal{L}_{frag}(g) = (freqId(g), difId(g), \Phi(g), \Upsilon(g))$ is the Fragment List. $freqId(g)$, $difId(g)$, $\Phi(g)$, and $\Upsilon(g)$ refer to frequent id,* DIF *id, frequent subgraph id set, and* DIF *subgraph id set, respectively, such that:*

  1. *if $g \in A^2F$-index, then $freqId(g) = a2fId(g)$ and $difId(g) = \Phi(g) = \Upsilon(g) = \emptyset$;*
  2. *if $g \in A^2I$-index, then $difId(g) = a2iId(g)$ and $freqId(g) = \Phi(g) = \Upsilon(g) = \emptyset$; and*
  3. *if $g \notin A^2F$-index and $g \notin A^2I$-index, then $\forall g' \subset g$ where $|g'| = |g| - 1$, if $g' \in A^2F$-index, then $\Phi(g)$ contains $a2fId(g')$, and $freqId(g) = difId(g) = \emptyset$. Also, $\forall g' \subset g$ where $g' \in A^2I$-index, $\Upsilon(g)$ contains $a2iId(g')$.*

- *Each v is uniquely identified by the pair (ℓ, k) where k is the position of v based on depth-first traversal order starting from $S_\ell.v_{source}$.*

Example 4.9 Consider Step 5 (*Sequence 1*) in Figure 2.5. Figure 4.5a depicts the SPIG S_5 after the addition of the new edge labeled 5 (e_5). Each vertex represents a subgraph of the query fragment containing e_5 and is identified by a pair of identifiers containing label of e_5 and its position. For instance, $v_{5,3}$ refers to the third vertex in S_5. Information associated with each vertex in S_5 is shown in Figure 4.5b. Particularly, the entries from left to right in the Fragment List are *freqId*, *difId*, Φ, and Υ, respectively (we follow this sequence in all relevant figures). Note

that $v_{5,1}, v_{5,2}, v_{5,3}$, and $v_{5,4}$ represent the frequent fragments f_1, f_3, f_5, and f_6 (Figure 2.1), respectively. Therefore, their *freqId*s are 1, 3, 5, and 6, respectively. Since $v_{5,5}$ represents dif_1, the *difId* is 1 (Figure 4.3b). However, $v_{5,6}$ represents the NIF inf_4. Hence, it satisfies the Condition 3 in Definition 4.8 as inf_4 is neither indexed by A^{2}F-index nor by A^{2}I-index. Consequently, $freqId(v_{5,6}) = difId(v_{5,6}) = \varnothing$. Among all the largest proper subgraphs of inf_4 (size of these subgraphs is $|inf_4| - 1$), the subgraph f_6 (see Figure 2.1) is a frequent fragment and hence stored in the A^{2}F-index (vertex id 6 in Figure 4.3a). Hence, $\Phi(v_{5,6}) = \{6\}$. Also, among all the subgraphs of inf_4, the subgraphs dif_1 and dif_2 (see Figure 2.1) are DIFs and are indexed by A^{2}I-index (having entry ids 1 and 2 in Figure 4.3b). Consequently, $\Upsilon(v_{5,6}) = \{1, 2\}$.

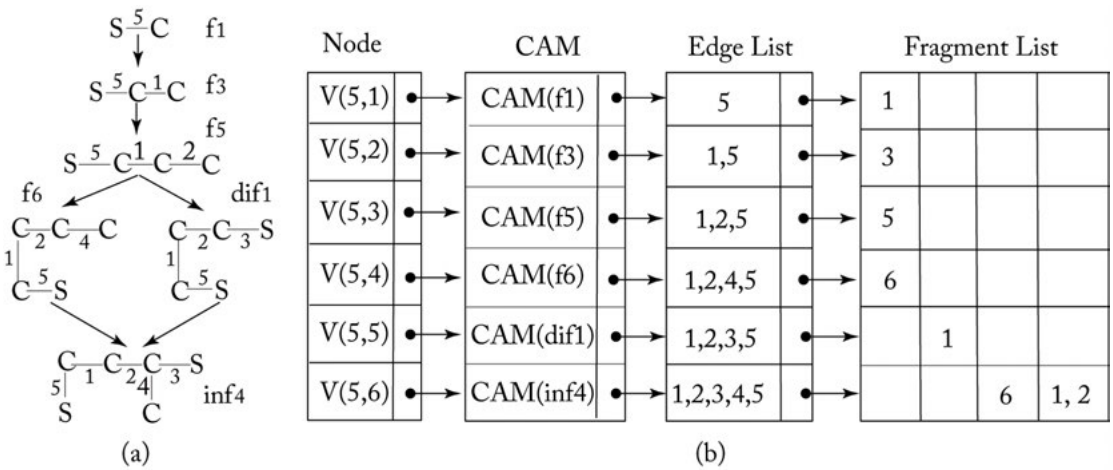

Figure 4.5: The vertices of the spindle-shaped graph in step 5.

4.4.1 ALGORITHM FOR SPIG CONSTRUCTION

The algorithm for building a SPIG is shown in Algorithm 4.7. It takes as input the new edge e_ℓ added to the query fragment q, a set of SPIGs $\mathcal{S}$ from previous step, and a queue $\mathbb{Q}$ to temporarily store the vertexes of S_ℓ. The building process starts from the new edge (Lines 1–2). It first attaches the CAM code and edge label of e_ℓ to vertex $v_{\ell,1}$. Let $v_{\ell,i}$ be the vertex dequeued from $\mathbb{Q}$ (Line 5). For each $v_{\ell,j}$ in S_ℓ, if $v_{\ell,j}$ is the parent of $v_{\ell,i}$, then $v_{\ell,i}$ inherits the frequent and DIF ids of $v_{\ell,j}$. That is, it attaches $freqId(v_{\ell,j})$ to $\Phi(v_{\ell,i})$, $difId(v_{\ell,j})$ and $\Upsilon(v_{\ell,j})$ to $\Upsilon(v_{\ell,i})$ (Lines 6–8). If g_i is not a DIF or a frequent fragment (Line 9), then the algorithm first extracts the largest subgraph of g_i without e_ℓ (denoted by g'_i). Let ℓ' be the new edge in g'_i where $\ell' < \ell$. Since $S_{\ell'}$ has already been constructed and stored in $\mathcal{S}$, the algorithm retrieves $v'_{\ell,i}$ from the $|g'_i|$-*th* level of $S_{\ell'}$ (Lines 10–11). Then it attaches the relevant ids in *FragmentList* of $v'_{\ell,i}$ to $v_{\ell,i}$ (Line 12). Note that as all the largest subgraphs of $v_{\ell,i}$ can be found in $\mathcal{S}$, the identifiers

Algorithm 4.7 ***SpigConstruct***

Input : Query q, Vertex queue $\mathbb{Q}$, new edge e_ℓ, set of SPIGs $\mathcal{S}$
Output : Spindle-shaped graph S_ℓ

```
1:  v_{ℓ,1} ← f(e_ℓ)
2:  ENQUEUE(v_{ℓ,1}, Q)
3:  INSERT(v_{ℓ,1}, S_ℓ)
4:  while Q ≠ ∅ do
5:     v_{ℓ,i} ← DEQUEUE(Q)
6:     for all v_{ℓ,j} ∈ S_ℓ is the parent of v_{ℓ,i} do
7:        Add v_{ℓ,j}'s FragmentList to v_{ℓ,i}
8:     end for
9:     if g_i ∉ A²F-index or A²I-index then
10:       g'_i ← g_i − e_ℓ
11:       v'_{ℓ,i} ← Search cam(g'_i) in the |g'_i|-th level of S_ℓ'
12:       Attach v'_{ℓ,i}'s FragmentList to v_{ℓ,i}
13:    else
14:       Attach v_{ℓ,i} with difId(g_i) or freqId(g_i)
15:    end if
16:    if |g_i|=|q| then
17:       Add S_ℓ in S
18:       return S_ℓ
19:    else
20:       for all g_i ⊂ g_j ⊂ q and |g_j|=|g_i|+1 do
21:          if v_{ℓ,j} ∉ Q then
22:             v_{ℓ,j} ← f(g_j)
23:             ENQUEUE(v_{ℓ,j}, Q)
24:          end if
25:          INSERT(v_{ℓ,j}, S_ℓ)
26:          Connect edge(v_{ℓ,i}, v_{ℓ,j})
27:       end for
28:    end if
29: end while
```

of frequent and infrequent fragments can be efficiently inherited *without* decomposing them to its subgraphs and retrieving them by probing action-aware-indexes.

If g_i is a DIF or a frequent fragment, then it attaches frequent fragment id or DIF id of g_i on $v_{\ell,i}$'s *freqId* or *difId*, respectively (Line 14). If $|g_i| = |q|$, then the SPIG construction process is terminated and S_ℓ is added to $\mathcal{S}$ (Lines 16–18). Otherwise, vertex $v_{\ell,j}$ is constructed as the child of $v_{\ell,i}$ in S_ℓ. For each $g_j \supset g_i$ in q, if $v_{\ell,j}$ does not exist in $\mathbb{Q}$ then it attaches the CAM code and edge labels of g_j to $v_{\ell,j}$ and inserts the vertex in $\mathbb{Q}$. Lastly, it adds $v_{\ell,j}$ in S_ℓ and connects $v_{\ell,i}$ and $v_{\ell,j}$ with a directed edge (Lines 20–27).

Observe that the aforementioned procedure *does not incrementally* build S_ℓ from $S_{\ell'}$ ($\ell' < \ell$) as e_ℓ is different in each formulation step. Consequently, the fragments represented by the vertices of S_ℓ are often different from those in $S_{\ell'}$. For instance, Figure 4.6 shows a set of SPIGs constructed for Steps 1–6 in *Sequence 1* in Figure 2.5. Observe that the fragments in two consecutive SPIGs (e.g., S_5 and S_6) can be quite different.

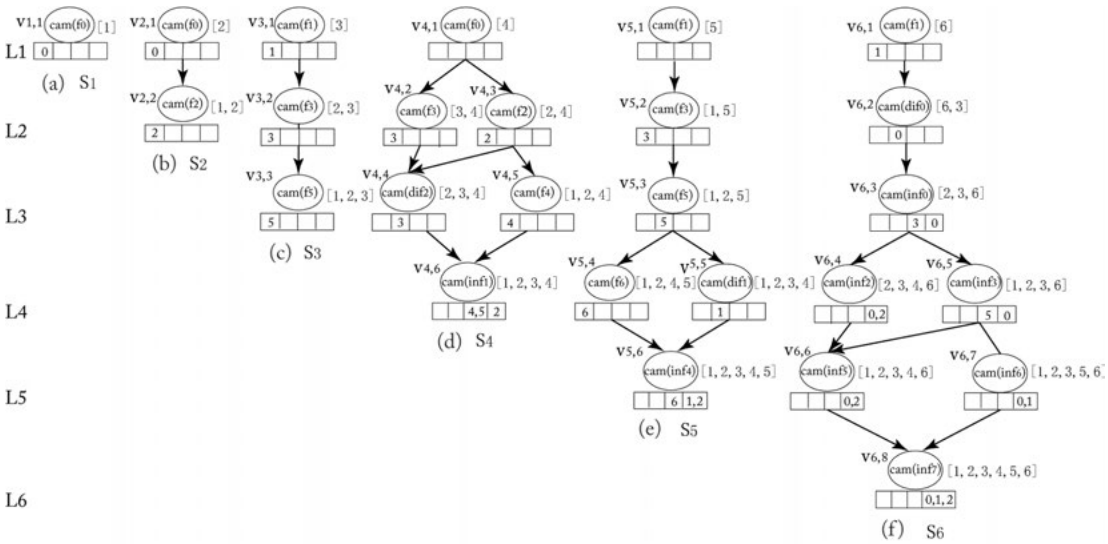

Figure 4.6: The SPIG set for Sequence 1 (Edge Lists are in square brackets and Fragment Lists are shown in rectangular boxes).

4.4.2 ANALYSIS OF SPIG CONSTRUCTION

The cost of SPIG construction depends on the number of edges in the query as it influences the number of levels and vertex set size of the SPIG. Let q be a visual query graph fragment with n *distinct* edges. That is, q has n edges with unique node label pairs (v_i, v_j). Then the maximum number of vertexes in the k-*th* level of S_ℓ is C_{n-1}^{k-1}. Consequently, the total number of vertices in S_ℓ is: $\sum_{k=1}^{n} C_{n-1}^{k-1}$. However, in practice, often some nodes in q share the same

vertex labels. For example, in the query in Figure 1.2 there are only two distinct edges (`C-S` and `C-C`). Consequently, the number of unique vertexes in the *k-th* level of S_ℓ is much less than the worst-case scenario. For instance, only two vertexes are in the fourth level of S_6 (Figure 4.6f).

Lemma 4.10 *The total number of vertexes in the k-th levels of* SPIG*s in* $\mathcal{S}$ *is:* $N(k) \leq \mathcal{C}_n^k$.

Proof. *(Sketch)* $\forall g_i \subseteq q, v_i \in \mathcal{S}$. The number of g_i with k edges $\leq \mathcal{C}_n^k, n = |q|$, so $N(k) \leq \mathcal{C}_n^k$. □

Different sequence of formulation steps for a query q (e.g., *Sequences 1* and *2* in Figure 2.5) will result in different SPIG sets. However, the total number of vertexes in the *k-th* level will remain identical in different SPIG sets. That is, given $\mathcal{S}_i$ and $\mathcal{S}_j$ generated by two distinct sequence of formulation steps for q, $N_i(k) = N_j(k)$.

4.5 SUBSTRUCTURE SIMILARITY SEARCH

We begin by describing SPIG-based candidates generation for exact substructure search (*ExactCandidates* procedure). Note that this will be exploited by substructure similarity search and our query modification strategy.

4.5.1 EXACT SUBSTRUCTURE CANDIDATES SET GENERATION

Algorithm 4.8 outlines the SPIG-based procedure for retrieving R_q at a specific step. Given the target vertex v in the SPIG S_ℓ representing the query fragment q, if v represents a frequent fragment, then it retrieves FSG identifiers of v from A^2F-index (Lines 1–3). Otherwise, if v represents a DIF, then the algorithm retrieve the FSG identifiers from A^2I-index (Lines 4–6). If v represents a NIF then for each identifier in the frequent subgraph id set ($\Phi(v)$) and DIF subgraph id set ($\Upsilon(v)$) of v, it retrieves the corresponding FSG identifiers from A^2F-index and A^2I-index, respectively, and then intersect them with R_q to generate the candidate set (Lines 8–9).

4.5.2 SIMILAR SUBSTRUCTURE CANDIDATES SET GENERATION

A key challenge in substructure similarity search is that the similar subgraph verification for a large candidate set is prohibitively expensive. Our strategy for reducing the verification cost is as follows: (a) retrieve only candidates that are "nearly" similar to the query fragment and (b) identify verification-free candidates among them.

Algorithm 4.9 describes the *SimilarCandidates* procedure. It separates the candidate set into two parts, namely R_{free} and R_{ver}. R_{free} stores the identifiers of verification-free candidate graphs whereas R_{ver} stores identifiers of candidate data graphs that need verification. Given the subgraph distance threshold σ, the algorithm exploits the SPIG set $\mathcal{S}$ to identify the relevant subgraphs of q that need to be matched for retrieving approximate candidate sets. Specifically,

Algorithm 4.8 ***ExactCandidates***

Input : Target vertex v in S_ℓ, A^2F-index, A^2I-index
Output : Set of candidate identifiers R_q

```
1:  if freqId(v) ≠ ∅ then
2:      i = freqId(v)
3:      R_q ← retrieve fsgIds(g_i) from A²F-index
4:  else if difId(v) ≠ ∅ then
5:      i = difId(v)
6:      R_q ← retrieve fsgIds(g_i) from A²I-index
7:  else
8:      for all i ∈ Φ(v), j ∈ ϒ(v) do
9:          R_q ← R_q ∩ fsgIds(g_i) ∩ fsgIds(g_j)
10:     end for
11: end if
```

these subgraphs are query fragments represented by the vertices at levels $|q|-1$ to $|q|-\sigma$ in $\mathcal{S}$ (Line 1). Let $R_{free}(i)$ and $R_{ver}(i)$ store the verification free candidates and candidates that need verification in the i-*th* ($|q|$-$\sigma \leq i < |q|$) level of $\mathcal{S}$, respectively. For each vertex v_j in the i-*th* level, if it is a frequent fragment or DIF, then the algorithm retrieves the candidates satisfying v_j using the *ExactCandidates*(v_j) procedure and combine them with $R_{free}(i)$ (Lines 3–4). Otherwise, v_j is a NIF. Consequently, $R_{ver}(i)$ is computed by combining $R_{ver}(i)$ with the candidates returned by *ExactCandidates*(v_j) (Lines 5–6). Next, it removes the candidates that exist in both $R_{free}(i)$ and $R_{ver}(i)$ from $R_{ver}(i)$ as these are already identified as verification-free candidates (Line 9). Finally, it adds $R_{ver}(i)$ and $R_{free}(i)$ in R_{ver} and R_{free}, respectively (Line 10).

Example 4.11 Reconsider the SPIG set in Figure 4.6 generated based on the query formulation sequence in *Sequence 1* (Figure 2.5). In the first step, edge e_1 is added and S_1 is constructed as shown in Figure 4.6a (Line 4 in Algorithm 4.4). As $freqId(v_{1,1}) = 0$, *ExactCandidates* procedure is invoked (Line 6 in Algorithm 4.4) to locate it in A^2F-index and retrieve $fsgIds(f_0)$ as the candidate set of current query fragment (Lines 1–3 in Algorithm 4.8). In the second step, the SPIG S_2 is constructed. Since $freqId(v_{2,2}) = 2$ ($v_{2,2}$ is the target vertex), its FSG identifiers are again retrieved by probing A^2F-index using the *ExactCandidates* procedure. After Step 3, $v_{3,3}$ is the target vertex in S_3 and $freqId(v_{3,3}) = 5$. Hence, $fsgIds(f_5)$ is retrieved by probing A^2F-index as the candidate set for exact match. Observe that so far the query is a frequent fragment. After Step 4, the target vertex $v_{4,6}$ in S_4 is a NIF and $\Psi(v_{4,6}) = \{4, 5\}$ and $\Upsilon(v_{4,6}) = 2$. Hence, the FSG identifiers of these fragments are retrieved by executing Lines 8–9 in Algorithm 4.8 ($fsgIds(v_{4,6})$=$fsgIds(dif_2) \cap fsgIds(f_4) \cap fsgIds(f_5) = fsgIds(inf_1)$). Also, $|fsgIds(v_{4,6})| = 250$.

Algorithm 4.9 ***SimilarCandidates***

Input : Query fragment q, σ, SPIG set $\mathcal{S}$
Output : R_{free}, R_{ver}

```
1:  for i=|q|-1 to |q|-σ do
2:     for all v_j in ith level of S do
3:        if freqId(v_j) ≠ ∅ or difId(v_j) ≠ ∅ then
4:           R_free(i) ← R_free(i) ∪ ExactCandidates(v_j)
5:        else
6:           R_ver(i) ← R_ver(i) ∪ ExactCandidates(v_j)
7:        end if
8:     end for
9:     R_ver(i) ← R_ver(i) − (R_free(i) ∩ R_ver(i))
10:    Add R_free(i) in R_free and R_ver(i) in R_ver
11: end for
```

After Step 5 since the target vertex is $v_{5,6}$ in S_5 and $fsgIds(inf_4) = 0$, the user is given an option to either modify the query or relax it to a subgraph similarity query and retrieve approximate matches (Line 9 in Algorithm 4.4). Suppose the user chose the latter option. Then, the *SimilarCandidates* procedure is invoked. Assume that $\sigma = 2$. That is, in Step 5 two edges are allowed to be missed in the results of substructure similarity search ($i \in \{3, 4\}$ in Algorithm 4.9). Therefore, Lines 2-10 in Algorithm 4.9 are executed twice. $R_{free}(3)$ and $R_{ver}(3)$ are generated for the vertexes in the third levels of the SPIGs in $\mathcal{S}$ ($v_{3,3}$, $v_{4,4}$, $v_{4,5}$, and $v_{5,3}$). $R_{ver}(3) = \emptyset$ and $R_{free}(3) = fsgIds(v_{3,3}) \cup fsgIds(v_{4,4}) \cup fsgIds(v_{4,5}) \cup fsgIds(v_{5,3}) = fsgIds(f_5) \cup fsgIds(dif_3) \cup fsgIds(f_4) \cup fsgIds(dif_5)$. Observe that $|R_{free}(3)| \geq 1300$. $R_{free}(4)$ and $R_{ver}(4)$ are generated for the vertexes in the fourth levels of the SPIGs in $\mathcal{S}$ ($v_{5,4}$, $v_{5,5}$, and $v_{4,6}$). Consequently, $R_{free} = R_{free}(4) = fsgIds(v_{5,4}) \cup fsgIds(v_{5,5}) = fsgIds(f_6) \cup fsgIds(dif_1)$ (Line 4) whereas $R_{ver} = R_{ver}(4) = fsgIds(v_{4,6})$ (Lines 9–10). Observe that $|R_{free}(4)| \geq 1100$ and $|R_{ver}(4)| = 250$. If the user clicks on the `Run` icon now, at most 250 candidate graphs in R_{ver} need candidate verification. However, at least 2400 candidate graphs in R_{free} are returned directly without verification.

When another edge is added in Step 6, R_{free}= $R_{free}(4) \cup R_{free}(5)$ and R_{ver}= $R_{ver}(4) \cup R_{ver}(5)$ where

$$\begin{aligned} R_{ver}(4) &= R_{ver}(4) \cup fsgIds(v_{6,4}) \cup fsgIds(v_{6,5}) \\ &= R_{ver}(4) \cup fsgIds(inf_2) \cup fsgIds(inf_3). \end{aligned}$$

Note that $|R_{ver}| \leq 800$. Similarly, $R_{free}(5) = 0$ and $R_{ver}(5) = fsgIds(v_{5,6}) \cup fsgIds(v_{6,6}) \cup fsgIds(v_{6,7})$. Since $fsgIds(v_{5,6}) = 0$, $fsgIds(v_{6,6})$= $fsgIds(dif_0) \cap fsgIds(dif_2)$ and $fsgIds(v_{6,7})$= $fsgIds(dif_0) \cap fsgIds(dif_1)$. Also, since $|fsgIds(v_{6,7})| = 200$ and $|fsgIds(v_{6,6})| = 150$, $|R_{ver}(5)| \leq 350$.

Analysis of candidate graph set. Observe that the candidate set is equal to the union of the FSG identifiers of vertexes in the levels $|q|$-σ to $|q| - 1$ of the SPIGs in $\mathcal{S}$.

Lemma 4.12 *Let R_{cand} be the candidate set for a query fragment at a specific formulation step. Then,*

$$R_{cand} = \bigcup_{k=|q|-\sigma}^{|q|-1} \bigcup_{i=0}^{N(k)} fsgIds(v_i).$$

Proof. *(Sketch)* $\forall g_i \subset q, |g_i| \geq |q| - \sigma, \sum_{i=0}^{N(k)} v_i$ is the sum of vertices in the k-th level of $\mathcal{S}$. $\bigcup_{i=0}^{N(k)} fsgIds(v_i)$ is its candidates. Therefore, the candidates satisfying σ edge missing are given as follows: $\bigcup_{k=|q|-\sigma}^{|q|-1} \bigcup_{i=0}^{N(k)} fsgIds(v_i)$. □

Notably, the query formulation sequences *do not* have any effect on the candidate graphs set for both subgraph containment and similarity queries. That is, given two SPIG sets $\mathcal{S}_i$ and $\mathcal{S}_j$ of a query q, $R_{cand}(i) = R_{cand}(j)$. *Consequently, different formulation sequences do not have significant effect on the* SRT *as it is primarily influenced by the size of candidate set.*

4.5.3 GENERATION OF APPROXIMATE QUERY RESULTS

As the data graphs in the result set of a substructure similarity search have varying degree of similarity with respect to the query graph, we order them based on the following rule. Let g_1 and g_2 be two candidate graphs that approximately match the query q. If $dist(g_1, q) < dist(g_2, q)$ then $Rank(g_1) < Rank(g_2)$. Note that a lower rank of g indicates that g is more similar to q.

Algorithm 4.10 outlines the procedure for generating ordered query results. As the subgraph distance of candidate graphs associated with the i-*th* level of SPIGs in $\mathcal{S}$ is $|q| - i$, the higher level the candidate graph is in $\mathcal{S}$, the more similar it is to the query graph. For the candidate graphs that are associated with level i, first the verification-free candidates ($R_{free}(i)$) are added in *Results* (Line 2). Next, it generates the result set from the candidates in $R_{ver}(i)$ (Lines 3–4). Here we extend VF2 [Cordella et al., 2001] to handle MCCS-based similarity verification. This procedure is encapsulated by the *SimVerify* procedure. The verified candidates are then added to *Results* (Line 4). The aforementioned procedure is repeatedly executed up to ($|q|$-1)-*th* level of the SPIGs. The results are returned ordered by increasing σ values.

Note that our focus here is not to develop an efficient similar subgraph verification technique. In fact, we can easily replace the implementation of *SimVerify* with a more efficient technique (e.g., Han et al. [2013]).

Algorithm 4.10 ***SimilarResultsGen***

Input : q, R_{free}, R_{ver} and σ
Output : Ordered result set *Results*

```
1: for i=|q|-σ to |q|-1 do
2:     Results ← Results ∪ R_free(i)
3:     R_ver(i) ← R_ver(i) ∩ Results
4:     Results ← Results∪ SimVerify(q, R_ver(i), i)
5: end for
```

4.6 SUPPORTING QUERY MODIFICATION

Any practical visual graph querying system should allow users to modify a query fragment at any time during query formulation. In our framework, a user may modify a visual query due to two key reasons: (a) if the candidate set of the formulated query fragment is empty then she may modify the query when prompted by the system; and (b) she may commit a mistake or may change her mind during query formulation and modify the query fragment accordingly. We now discuss how such query modification is efficiently supported.

In the current version of our framework, modification to a query is achieved by edge deletion.[3] The user can delete any edge as long as the modified query graph is a connected graph at all times. For clarity, we introduce our query modification algorithm based on single edge deletion at a time. It is trivial to extend it to support multiple edge deletions.

Algorithm 4.11 outlines our SPIG-based strategy for handling query modifications. Let e_ℓ be the most recently added edge in q and e_d be the edge deleted from q where $0 < d \leq \ell$. When the candidate set of subgraph containment query fragment q becomes empty and the user opts for query modification then Lines 3–9 are executed to provide modification suggestion to her. For each possible deleted edge in q, it matches the corresponding vertex v_i in the $|q'|$-*th* level of the SPIGs in $\mathcal{S}$ by performing the graph isomorphism test of q' and v_i. Note that two graphs g and g' are isomorphic to each other, if and only if $cam(g) = cam(g')$ [Huan et al., 2003]. It recommends the edge, that returns the largest candidate set $R_{q'}$, for deletion to the user (Lines 6–8). On the other hand, if e_d is selected by the user at any time during query formulation, the new query fragment q' is formed by deleting e_d from q (Line 13). The SPIG set $\mathcal{S}$ is updated by removing SPIGs and vertexes related to e_d (Lines 15–18). Finally, if modification occurs when the query fragment is already a subgraph similarity query, then the new candidate set is

[3]Node relabeling can be expressed as deletion of edge(s) following by insertion of new edge(s) and node.

Algorithm 4.11 ***QueryModification***

Input : Query q, Deleted edge e_d, $\mathcal{S}$, R_q, σ
Output : R_q

```
1:  Initialize e_d to be deleted edge
2:  if R_q=∅ and e_d = ∅ then
3:     for all e_i ⊂ q do
4:        q' ← q − e_i
5:        v_i ←Match q' in the |q'|-th level of S
6:        if |fsgIds(v_i)| > |R_q'| then
7:           e_d ← e_i
8:           R_q' ← fsgIds(v_i)
9:        end if
10:    end for
11: else
12:    e_d ← edge deleted by the user
13:    q' ← q − e_d
14: end if
15: Remove S_d from S
16: for all v_i ∈ S_j, S_j ∈ S, e_d ∈ L_E(v_i) do
17:    S_j ← Delete v_i and its edges in S_j
18: end for
19: if R_q=∅ then
20:    SimilarCandidates(q', σ, S)
21: else
22:    ExactCandidates(q')
23: end if
```

generated by *SimilarCandidates* procedure (Line 20). Otherwise, the candidate set is generated by invoking the *ExactCandidates* procedure.

Example 4.13 Consider Step 5 of *Sequence 1* in Figure 2.5. The state of the query fragment is depicted in Figure 4.7a. Assume that the user selects the query modification option when prompted by our framework (Figure 4.7b). Since $|fsgIds(v_{5,4})| = |fsgIds(f_6)| = 1300$ is larger than both $|fsgIds(v_{4,6})|$ and $|fsgIds(v_{5,5})|$ in the fourth level of the SPIGs in Figure 4.6, q' is modified to f_6 and the edge 3 is suggested for deletion (Figure 4.7c). Figure 4.7d shows the modified query fragment q' after the user accepted the suggestion. At the same time, the SPIG

set is updated by removing S_3 and updating the SPIGs S_4 and S_5 by deleting the vertexes with edge 3 in their Edge Lists. The updated $\mathcal{S}$ is shown in Figure 4.8.

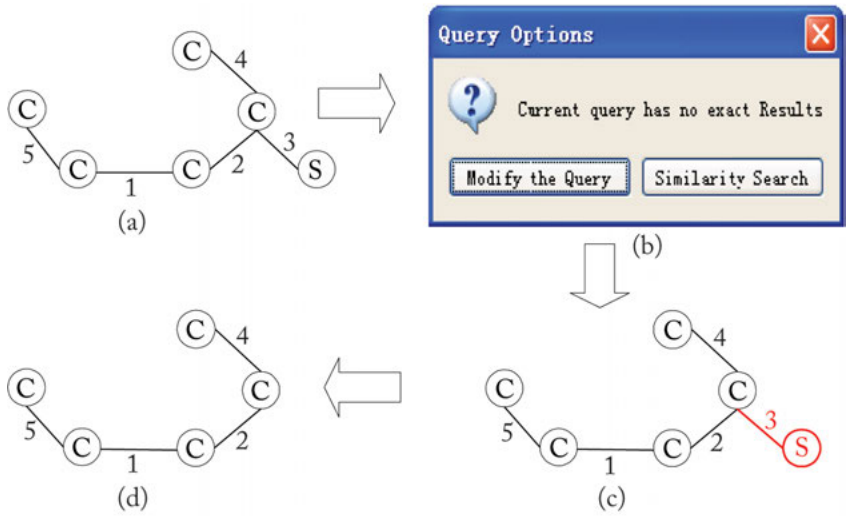

Figure 4.7: The query modification procedure in Step 5.

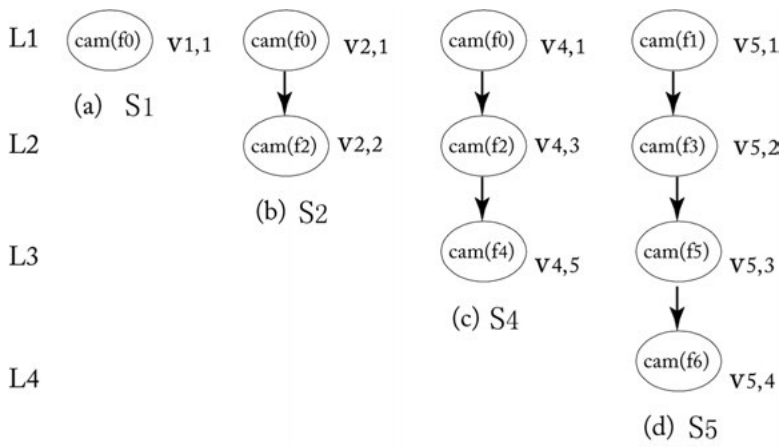

Figure 4.8: The updated SPIG set after deleting edge 3 in Step 5.

Now suppose the user chooses to invoke substructure similarity search instead at Step 5 and then deletes edge 6 after Step 6. Now q' matches $v_{5,6}$ and the target vertex of S_5. Hence, the updated $\mathcal{S}$ consists of $(\mathcal{S} - S_6)$. At last, the new candidates are calculated based on this updated $\mathcal{S}$.

4.7 PERFORMANCE STUDY

Our proposed visual subgraph querying system, referred to as PRAGUE, is implemented in Java. We run all experiments on an AMD 3.4 GHz machine with 3.25 GB RAM, running Ubuntu 9.10 system. Since there is no existing system that realizes our new paradigm in the context of substructure similarity search, we are confined to compare PRAGUE (denoted by PRG for brevity) against the following state-of-the-art MCS-based substructure similarity search methods based on traditional paradigm: *Grafil* [Yan et al., 2005] (denoted by GR), SIGMA [Mongiovi et al., 2010] (denoted by SG), and *restricted version* of *DistVP* [Shang et al., 2010] (denoted by DVP). These programs are all implemented in C++.

4.7.1 EXPERIMENTAL SETUP

Datasets. We use the popular AIDS Antiviral dataset containing 40K (40,000) graphs as real-world dataset. The average size of a graph is 25 vertices and 27 edges. The maximum size of a graph is 222 vertices and 251 edges. There are 63 different types of atoms in the dataset. We use the *Graphgen* of FG-Index [Cheng et al., 2007] to generate five synthetic datasets with sizes from 10,000–80,000 (denoted by 10K–80K). The average number of graph edges in each dataset is set to 30 and the average graph density is 0.1.

Queryset. Figure 4.9 depicts the queryset. Q_1–Q_4 are queries on the AIDS dataset whereas Q_5–Q_8 are queries on the synthetic datasets. As remarked earlier, our framework is designed for small-sized query graphs as it is not realistic to expect a user to formulate large queries visually in edge-at-a-time mode. Therefore, we chose query graphs whose sizes do not exceed 10. Additionally, since queries here are visually formulated by real users, it is not possible to generate a large number of visual queries as our preliminary study revealed that such aspiration strongly deters end users to participate in the empirical study.

The labels on the edges of a query in Figure 4.9 represent the default sequence of steps for query formulation in PRG. For example, in Q_3 the default sequence of steps for query formulation is: [(Hg, O), (O, C), (C, C), (C, N), (N, N), (N, N), (N, N), (C, N)]. Unless mentioned otherwise, we shall be using the default sequence for formulating a particular query. The specific step in a query when R_q becomes empty is shown by **bold** edge (e.g., Step 4 in Q_1).

Recall that the candidate set of PRG consists of two parts: R_{free} and R_{ver}. Obviously, the more candidates are in R_{free}, the better it is for PRG as these candidates are verification-free. Hence, we chose the query set to study *best* and *worst* case behaviors of PRG with respect to R_{free} and R_{ver}. Specifically, all candidates of Q_1 is in R_{free} ("best" case). In contrast, all candidates of Q_2, Q_3, Q_5–Q_8 are in R_{ver} ("worst" case).

Participants profile. Eight unpaid male volunteers (ages from 21–27) participated in the experiments. None of them are familiar with any graph query languages. They were first trained to use the GUI of PRG. For every query, the participants were given some time to determine the steps that are needed to formulate the query visually. This is to ensure that the effect of thinking time

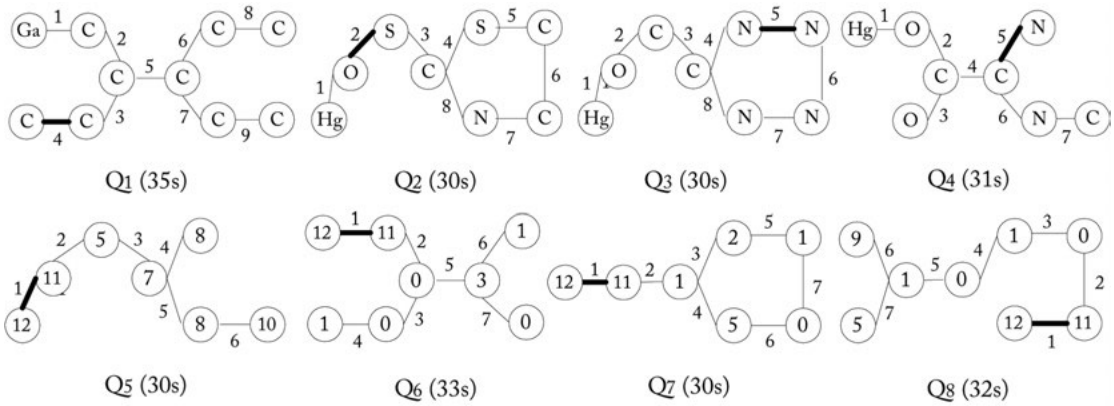

Figure 4.9: Queries on real and synthetic datasets.

is minimized during query formulation. Note that faster a user formulates a query, the lesser time PRG has for SPIG construction. Each query was formulated five times by each participant (using the default sequence unless specified otherwise) and reading of the first formulation was ignored. The average query formulation time (QFT) for a query by all participants is shown in parenthesis in Figure 4.9.

4.7.2 PERFORMANCE ON REAL GRAPH DATASET

We discuss the performance of PRG on the AIDS dataset from a variety of aspects. We set $\alpha = 0.1$, $\beta = 8$ for PRG and $\sigma = 3$ for all techniques unless specified otherwise. Note that the effect of different values of β has negligible effect on frequent subgraph containment queries (candidate pruning depends on frequent fragments) [Jin et al., 2010]. For subgraph similarity query, the candidate pruning is mainly based on DIFs. Hence, the variation of β has even lesser effect on similarity queries. For other parameters, we use the default settings of GR, SG, and DVP as suggested in Yan et al. [2005], Mongiovi et al. [2010], and Shang et al. [2010], respectively.

Candidate size and system response time (SRT). We compare the performances of the representative systems for evaluating subgraph similarity queries by varying σ from 1–4. Figures 4.10–4.13 report the candidate sizes and SRTs of Q_1–Q_4 on AIDS dataset for different values of σ. Note that in PRG, GR, and SG, candidate size refers to $|R_{free} \cup R_{ver}|$. In fact, GR and SG do not separate the candidates into these two categories. However, candidate size in DVP refers to $|R_{ver}|$ only. Observe that for most cases the candidate size of PRG is significantly less than GR, SG, and DVP. In Figures 4.11 and 4.12 ("worst" case queries), although the candidate size of PRG is larger than GR and SG when $\sigma \in \{1, 2\}$, it becomes less than these approaches when σ increases to 3 and 4. The candidate pruning of PRG depends on the DIFs and frequent fragments. Typically, DIFs have stronger pruning ability. In contrast, pruning of SG and GR

mainly depends on the frequent fragments. In the worst cases, there are less DIFs in the queries with smaller σ, which weakens pruning ability of PRG. Besides, the candidate sizes of DVP in Q_1 ("best" case) is significantly lesser than PRG for $\sigma \in \{3, 4\}$. This is primarily because DVP only reports $|R_{ver}|$. For Q_1, $|R_{ver}| = 0$ in PRG. For Q_2–Q_4, the candidate sizes of DVP are close to the entire dataset ($\sim$ 40K).

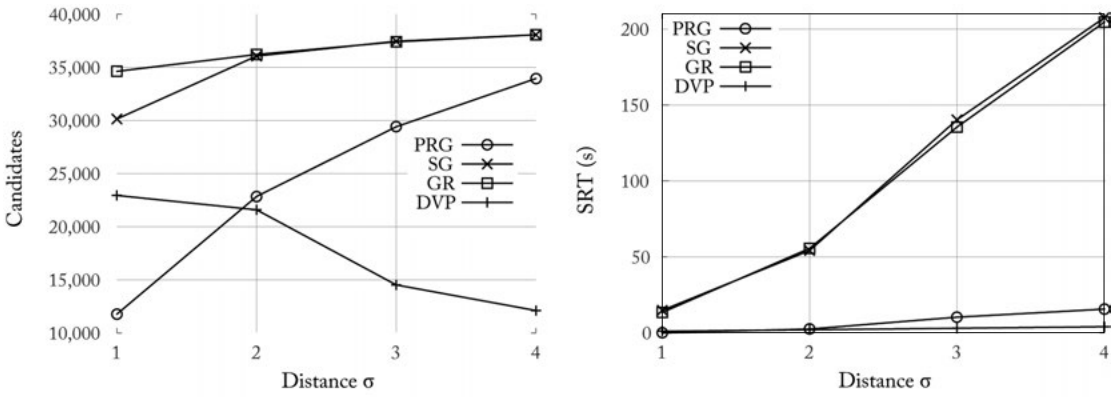

Figure 4.10: Candidate size and SRT of Q_1.

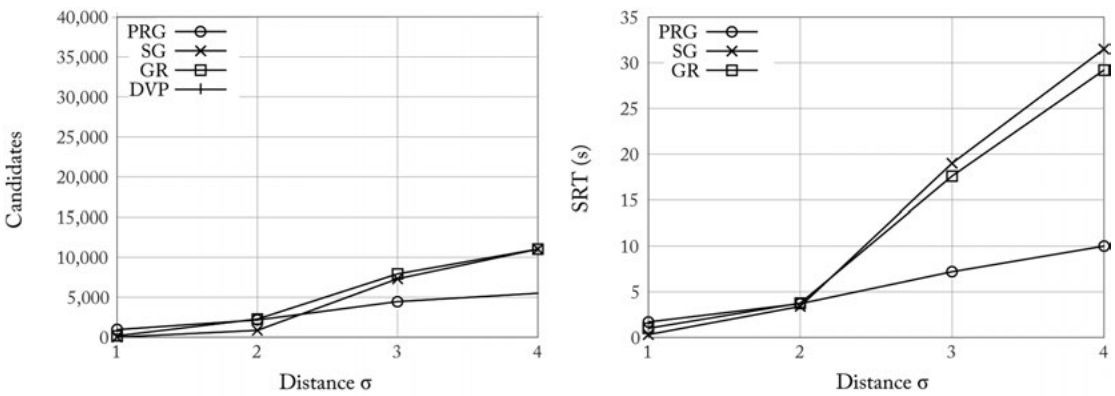

Figure 4.11: Candidate size and SRT of Q_2.

Now consider the SRTs for different values of σ. In GR, SG, and DVP, SRT refers to the execution time of a query. Each query was executed five times in each approach and the results from the first run were always discarded. Observe that we only display the SRTs of DVP for Q_1. This is because in contrast to the remaining approaches, DVP returns empty results for the remaining queries.

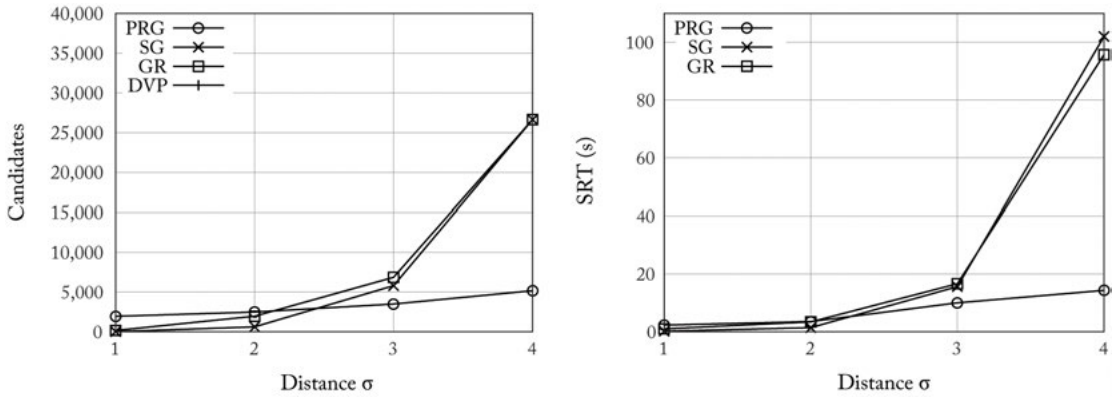

Figure 4.12: Candidate size and SRT of Q_3.

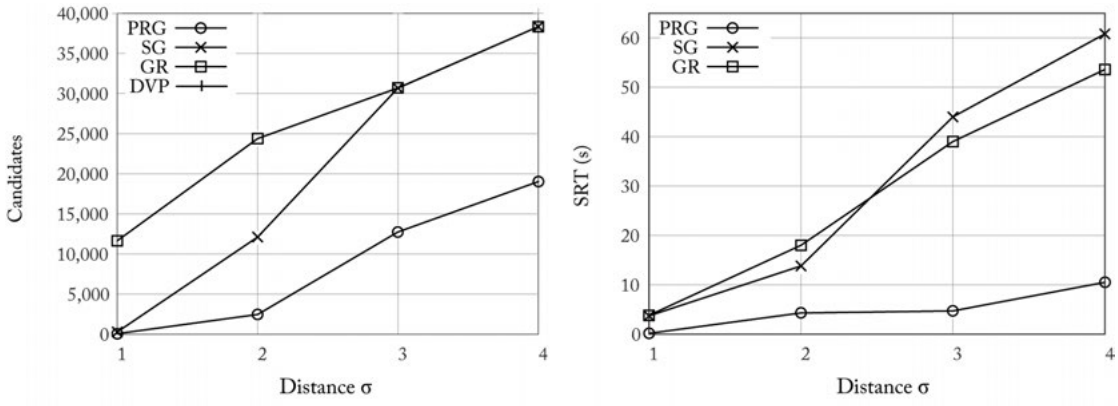

Figure 4.13: Candidate size and SRT of Q_4.

It is evident that the performance of PRG is better than the existing strategies. Although in Figures 4.11 and 4.12 (worst-case queries) the SRT of PRG is a little bit longer than GR and SG for $\sigma \in \{1, 2\}$, it is less than these approaches for larger σ. SG/GR converts the subgraph similarity verification problem to the exact subgraph isomorphism verification problem. The latter is typically faster than the former. In the worst cases, all the candidates in PRG need to be verified. Note that SG/GR loses this advantage when σ increases as they have to perform a large number of exact subgraph verification. More importantly, the SRT of PRG grows gracefully with σ. Lastly, *only* PRG orders the query results according to their subgraph distance. Inevitably, this increases the SRT of PRG.

Our experimental study makes a strong connection between usability and performance—the usability provided by visual graph query interfaces can be judiciously exploited to enhance performance of subgraph query processing.

Query modification costs. We now compare the cost of modifying a visual query using Algorithm 4.11. We vary the steps when a user performs modification, namely from addition of the 4-*th* edge (e_4) to the 9-*th* edge (e_9) if any. We always delete the first edge (e_1) from Q_1–Q_4 to simulate worst-case scenario. Table 4.2 reports the performances. Observe that the modification cost of PRG is cognitively negligible (virtually "zero"). This also implies that the cost of updating the SPIG set is negligible. Since the time taken to construct an edge in PRG typically is at least 2 sec, query modification can easily be completed by exploiting the GUI latency.

Table 4.2: Query modification costs for AIDS dataset (msec)

Query	e_4	e_5	e_6	e_7	e_8	e_9
Q_1	20	36	36	36	37	37
Q_2	0	0	0	15	15	
Q_3	0	0	0	0	0	
Q_4	16	16	16	16		

SPIG construction cost and query formulation sequence. Next, we assess the effect of different formulation sequences on the SPIG construction time and SRT. Table 4.3 lists different formulation sequences for Q_1–Q_8 and the average time (all participants) to construct the SPIGs at different steps. Observe that the SPIG construction process at each step is very efficient and takes negligible time. It is significantly lower (almost an order of magnitude) than the available GUI latency (at least 2 sec to draw an edge[4]). Also, SPIG construction is not adversely affected by addition of new edges to a query fragment. In summary, SPIGs can easily be constructed by exploiting the latency offered by the GUI. Lastly, the formulation sequences only have minor effect on the SPIG construction time and SRT highlighting the robustness of our technique.

4.7.3 PERFORMANCE ON SYNTHETIC GRAPH DATASET

We now assess the scalability of PRG using the synthetic datasets and the queries Q_5–Q_8. For synthetic datasets, we set $\beta = 4$ and $\alpha = 0.05$ for PRG and $\sigma = 3$ for PRG, SG, and GR. Note

[4]Here we ignore the "user thinking time." As the thinking time increases, the latency offered by the GUI increases as well at each step.

Table 4.3: Effect of variation in query formulation sequence on SPIG construction (in seconds)

Query	Sequence	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8	Step 9	Avg. SRT
Q_1	1, 2, 3, 4, 5, 6, 7, 8, 9	0	0	0	0	0.004	0.006	0.045	0.069	0.477	9.3
	1, 2, 3, 4, 5, 6, 8, 7, 9	0	0	0	0	0.005	0.007	0.021	0.285	0.478	9.5
Q_2	1, 2, 3, 4, 5, 6, 7, 8	0	0.15	0.3	0.134	0.343	0.21	0.312	0.24		8.2
	1, 2, 3, 8, 7, 6, 5, 4	0	0.15	0.3.	0	0.135	03	0.34	0.35		8.4
Q_3	1, 2, 3, 4, 6, 8, 7, 5	0.002	0.003	0	0	0	0.004	0.004	0.168		10
	3, 2, 1, 5, 7, 8, 6, 4	0	0.002	0.002	0	0	0.002	0.004	0.128		10.2
Q_4	1, 2, 3, 4, 5, 6, 7	0.1	0.32	0.15	0.21	0.35	0.12	0.32			4.8
	1, 2, 4, 3, 6, 7, 5	0.1	0.32	0.33	0.14	0.2	0.4	0.15			5
Q_5	1, 2, 3, 4, 5, 6	0.015	0	0	0.01	0.015	0.016				0.58
	1, 2, 3, 5, 6, 4	0.015	0	0	0.03	0.02	0				0.54
Q_6	1, 2, 3, 4, 5, 6, 7	0	0	0.01	0.03	0.01	0.015	0.024			1.1
	4, 3, 2, 1, 5, 7, 6	0.015	0	0.015	0.015	0.026	0	0.03			1
Q_7	1, 2, 3, 4, 5, 6, 7	0.01	0.014	0	0.015	0.015	0.02	0.015			0.73
	1, 2, 4, 6, 7, 5, 3	0.01	0.014	0.02	0.032	0.016	0	0.015			0.6
Q_8	1, 2, 3, 4, 5, 6, 7	0.023	0.013	0.023	0.02	0.013	0.016	0.015			0.62
	7, 6, 5, 4, 3, 2, 1	0.015	0	0.015	0.03	0.02	0	0.023			0.54

that since DVP failed to build indices for the synthetic datasets we are not able to compare its performance here.

SRT and size of candidate graphs. Figures 4.14–4.17 depict the SRTs and sizes of candidate graphs of Q_5–Q_8 for the five datasets. Clearly, SRT of PRG is lower than SG and GR and it has the least candidates across all datasets and queries, confirming the strengths of PRG. More importantly, our proposed paradigm enables PRG to scale gracefully. Note that the sharp increase in SRT for Q_5 (for 80K dataset) in PRG is primarily due to the simple verification method we have used rather than its candidates pruning ability.

Query modification cost. Table 4.4 reports the modification costs of Q_5–Q_8. For each query we modify at the last step and the first edge is always deleted. Observe that the modification is very efficient for PRG and scales gracefully across all datasets. Importantly, it can be easily completed during the latency provided by the GUI.

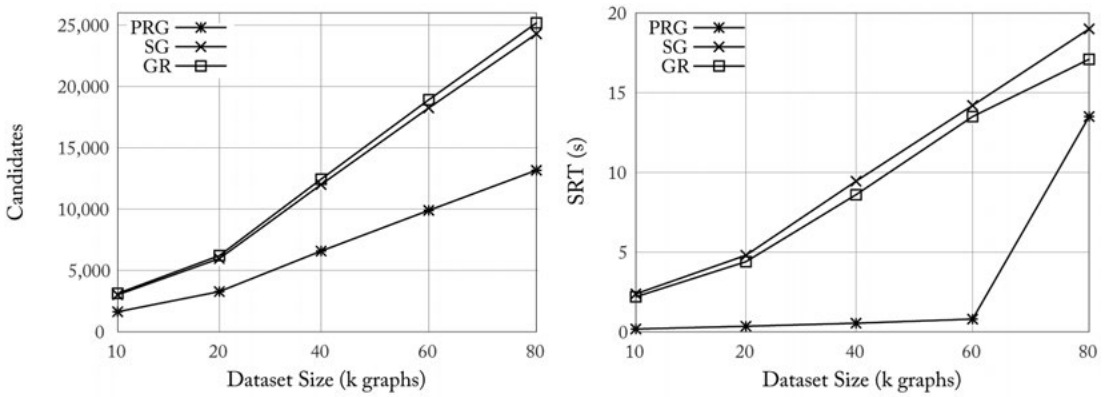

Figure 4.14: Candidate size and SRT of Q_5.

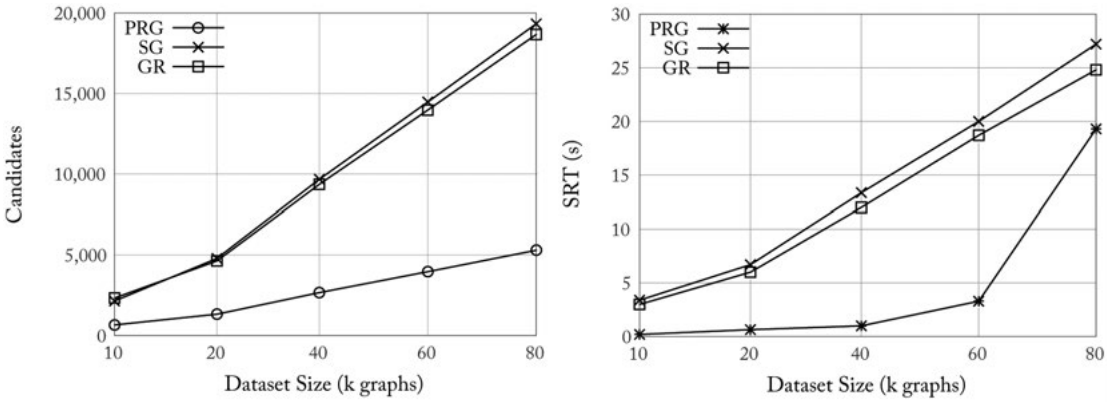

Figure 4.15: Candidate size and SRT of Q_6.

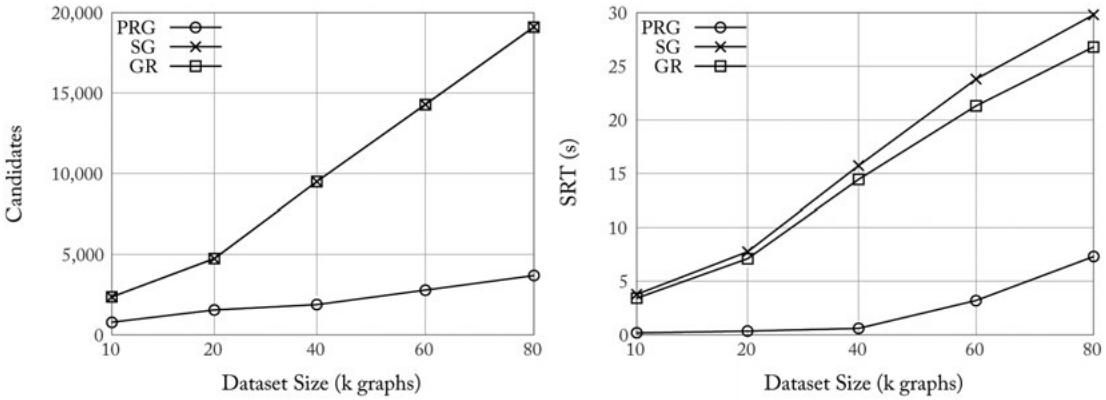

Figure 4.16: Candidate size and SRT of Q_7.

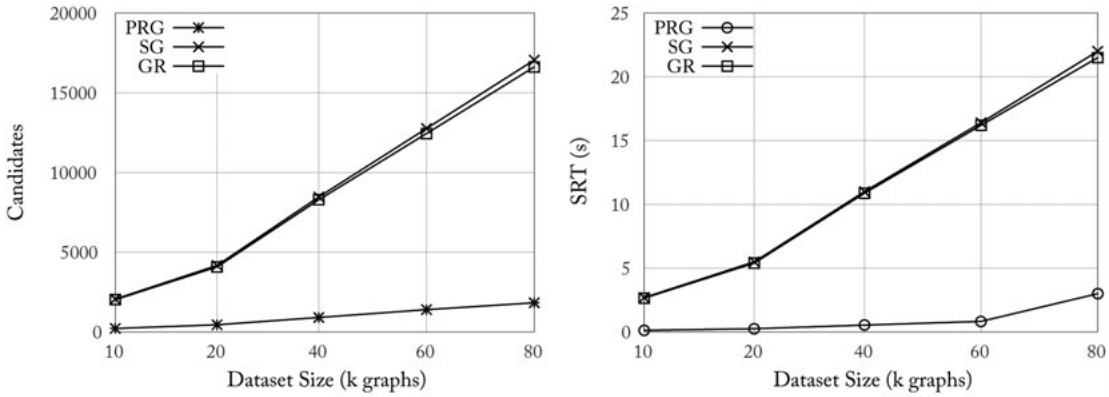

Figure 4.17: Candidate size and SRT of Q_8.

Table 4.4: Query modification cost for synthetic dataset (msec)

Query	10 K	20 K	40 K	60 K	80 K
Q_5	0	0	0	16	16
Q_6	0	0	0	0	15
Q_7	0	0	0	15	30
Q_8	0	0	15	30	40

4.8 BIBLIOGRAPHIC NOTES

Since the last decade there has been numerous efforts on designing efficient and scalable techniques to address the substructure search problem. These efforts follow the conventional query processing paradigm where the formulation of a query graph is independent of its evaluation against the database. Typically, the *complete* query is first specified before it is processed. In contrast, in this chapter we describe a novel query processing paradigm by blending two traditionally independent areas, namely HCI and database query processing. Specifically, when a substructure search query is visually formulated, its evaluation is interleaved with the formulation activities. Hence, this method is orthogonal to traditional studies related to subgraph query processing.

The work reported in this chapter appeared in Jin et al. [2010, 2012]. However, the general idea of blending visual query formulation with query processing was first proposed in 2006 in the context of XML query processing [Bhowmick and Prakash, 2006]. Similar to this work, a technique that blends visual XML query formulation and query processing by exploiting the latency offered by the GUI-based query formulation to prefetch portions of the query results

was proposed. A more refined version of this paradigm was presented in Truong and Bhowmick [2013]. The technique discussed in this chapter differs in the following ways. First, we focus on graph queries instead of tree-structured XML queries. Evaluation of graph queries is typically more challenging than tree queries due to its inherent computational complexity. Second, as Bhowmick and Prakash [2006] and Truong and Bhowmick [2013] are built on top of a relational backend, they leverage on existing well-known indexing schemes and SQL queries to efficiently prefetch partial results. In contrast, the techniques reported here design novel users' action-aware indexing scheme to support efficient computation of partial results that match different fragments of a visual subgraph query.

4.9 CONCLUSIONS

In this chapter, we presented a unified framework that supports the non-traditional paradigm of processing of visual substructure search queries by blending their evaluation with query formulation. Particularly, we show that how human interactions on a visual subgraph query interface can be exploited to improve the performance (i.e., SRT) of substructure search queries. Thus, this chapter makes a strong connection between usability of a visual graph querying system and the performance benefits it can bring to the underlying query processor when it is exploited.

CHAPTER 5

Blending Interactions and Query Processing on Large Networks

In the previous chapter, we presented a unified framework for blending visual query formulation and processing. An instantiation of this framework is designed for graph databases containing a large collection of small- or medium-sized graphs. Consequently, its indexing schemes and query processing strategy are designed to efficiently support query matching on such data graph collection. Unfortunately, these schemes cannot be easily adopted to support subgraph queries on large networks containing thousands of nodes and edges. This is primarily because the frequent and infrequent fragments-based indexing strategy discussed earlier is impractical for this case. Specifically, a graph fragment is considered frequent if the number of data graphs containing it is no less than a certain support threshold α. Now suppose we have a single large graph with 100,000 nodes and 200,000 edges. Then, the proposed technique can only identify frequent fragments if α is set to either zero or one. If α is set to one, then all subgraphs of the data graph can be considered as frequent. However, this is prohibitively expensive to index as there are more than one billion subgraphs. Even if an index could be constructed, a NIF query would require a subgraph isomorphism test against a data graph having 100,000 nodes, which is prohibitively expensive. In contrast, it is not explosive in the context of small- or medium-sized graph collection as we only need to keep track of identifiers of data graphs that contain a frequent fragment or DIF and the subgraph isomorphism test is against small-sized graphs.

At first glance, it may seem that the aforementioned bottleneck is due to the way frequent and infrequent fragments are defined in our framework. However, in general, generating frequent subgraphs for indexing is itself a bottleneck for the case of large networks [Xie and Yu, 2011]. This is because the time complexity of subgraph isomorphism, the core routine of any frequent subgraph mining algorithms, grows exponentially with the graph size. Furthermore, these subgraphs may suffer from low selectivity issue [Xie and Yu, 2011], reducing the effectiveness of the indexes. Small-sized frequent fragments typically have low selectivity as they may occur many times. As a result, they may generate a large number of candidates against small-sized query fragments. In this chapter, we present another instantiation of our unified framework (Algorithm 4.3) which addresses this limitation to realize our vision of the visual subgraph querying paradigm on large networks.

The rest of the chapter is organized as follows. In Section 5.1, we give an overview of the visual graph querying paradigm for large networks and highlight the key contributions. We elaborate on each step of our solution in Sections 5.2–5.5. Section 5.6 presents a systematic analysis of the performance of the framework. We review related literature in Section 5.7. The last section concludes the chapter. The key notations specific to this chapter are given in Table 5.1.

Table 5.1: Key symbols

Symbol	Definition
$G_p = (V_p, E_p)$	A partition graph with id p
$G_b = (V_b, E_b)$	A bridge
$G_\ell = (V_\ell, E_\ell)$	A graphlet
$gid(.)$	Identifier of a graphlet
$\mathcal{D}_\ell$	The set of all graphlets
Δ	Adjacent set
$G_\Delta = (V_\Delta, E_\Delta)$	A supergraphlet
$sgId(.)$	Identifier of a supergraphlet
$\mathcal{D}_\Delta$	A set of all graphlets and supergraphlets
$fsgIds(g)$	Set of supergraphlet identifiers of FSGs of g
$\mathbb{J} = g_1 \boxtimes g_2$	Fragment join of g_1 and g_2
σ	Subgraph distance threshold
$a2fId(.)$	Identifier of each node in A^2F-index
$a2iId(.)$	Identifier of a SIF in A^2I-index
e_m	A new edge added by user
$S_m = (V_m, E_m)$	A G-SPIG
$\mathcal{L}_{ind}(g)$	*Indexed Fragment List* of a vertex $v \in V_m$ representing g
$freqId(g)$	Frequent id attribute of $\mathcal{L}_{ind}(g)$
$sifId(g)$	SIF id attribute of $\mathcal{L}_{ind}(g)$
$\Omega(g)$	Supergraphlet id set attribute of $\mathcal{L}_{ind}(g)$
$\mathbb{S}$	A set of G-SPIGs
R_q	Identifiers of (super)graphlets containing q

5.1 OVERVIEW AND CONTRIBUTIONS

In this section, we first revisit the visual substructure search problem in the context of large networks and then present an overview of the instantiation of our unified framework to address it.

5.1.1 VISUAL SUBSTRUCTURE SEARCH PROBLEM REVISITED

Recall the definition of visual substructure search problem in Chapter 4 in the context of a large collection of small- or medium-sized data graphs. We now revisit it in the context of large networks. Similar to the preceding framework, here also we support both subgraph containment and subgraph similarity queries. Given two graphs G and Q, let $C_Q \subseteq G$ be a *connected common subgraph* (CCS) of Q and G. Then the *subgraph distance*, denoted as $dist_C(Q, G)$, is defined as follows: $dist_C(Q, G) = |Q| - |C_Q|$. Specifically, the subgraph distance measures the number of edges that are allowed to be missed in Q in order to match G. There can be many subgraphs of G that are CCS of Q and G. Hence, subgraphs with smaller *dist* are more similar to Q. Note that if $dist_C(G_1, G_2) = 0$, then G_1 and G_2 are subgraph isomorphic to each other.

Definition 5.1 [Visual Substructure Search Problem] *Given a query graph Q visually constructed on a visual graph query interface, a large network G, and a subgraph distance threshold σ, the goal of visual substructure search problem for large networks is to retrieve all connected common subgraphs C_i of Q and G by interleaving (i.e., blending) formulation and processing of Q s.t $dist_{C_i}(Q, G) \leq \sigma$.*

In contrast to the preceding chapter, observe that we use CCS instead of MCCS for similarity search. This is because we aim to find all similar matches whose size may be smaller than that of an MCCS as long as it is within σ. We believe that this feature is especially important in large networks having intricate structures or noise.

5.1.2 OVERVIEW

Can we somehow leverage the action-aware indexes and SPIGs because they have efficiently supported our visual query paradigm on a large collection of small- or medium-sized graphs? Unfortunately, this is challenging as it requires us to determine frequent fragments in a large network which is prohibitively expensive operation and a long-standing problem [Xie and Yu, 2011]. Hence, techniques described in the preceding chapter cannot be directly adopted to this new scenario. Furthermore, it is also highly space consuming to index location of all possible occurrences of a feature in a large intricate network as it may appear numerous times. We address these challenges here by taking the following steps. First, we decompose a large network into pieces of small data graphs while ensuring that no structural information is lost during this process. Consequently, the decomposed graph set can be viewed as a collection of small- or medium-sized data graphs. Second, we discover *approximate* sets of frequent and infrequent

fragments from this collection and *identify* their occurrences in the data graphs. This associates each fragment with a list of data graph identifiers instead of a full location list in the original network, which is very storage efficient. Third, we *redefine* and *build* action-aware indexes and SPIGs over these decomposed graphs to support substructure search. We now briefly describe these steps.

> *For the first time in the literature of querying large networks, our approach demonstrated that it is indeed possible to decompose a large network into smaller data graphs so that at the logical level we can transform the problem of querying a large network to querying a large collection of small decomposed data graphs.*

Action-aware index construction. Algorithm 5.12 outlines the procedure for generating action-aware indexes. We decompose a large network to pieces of small data graphs called *graphlets* by exploiting METIS [Karypis and Kumar, 1999], a fast and widely used minimum cut-based graph partitioning algorithm (Lines 1–2). A *graphlet* is either a *partition graph* or a *bridge*. Informally, *partition graphs* are partitions generated by the graph partitioning algorithm on the original network. On the other hand, *bridges* are graphs that are constructed from cut edges that link certain pairs of partition graphs. For instance, consider the network in Figure 5.1. The subgraphs with ids G_1 to G_4 (subgraphs encompassed by thick lines) are partition graphs generated by the graph partitioning technique. The bridges link certain pairs of these partition graphs (shown by patterned nodes encompassed by dotted lines) and are denoted by G_5 to G_7

Algorithm 5.12 ***IndexGeneration***

Input : A large network G, support threshold α, partition threshold p
Output : Action-aware indexes

1: $\mathcal{G} \leftarrow$ METIS(G, p)
2: $\mathcal{D}_\ell \leftarrow$ Construct graphlets from $\mathcal{G}$
3: $\mathcal{F} \leftarrow$ GSPAN($\mathcal{D}_\ell$, α)
4: Group frequent fragments $\mathcal{F}$ by size
5: $(\mathcal{F}, \mathcal{I}) \leftarrow$ TWOEDGEFRAGMENT($\mathcal{F}$, $\mathcal{I}$, $\mathcal{D}_\ell$) /* Algorithm 5.16 */
6: $\mathcal{F} \leftarrow$ COMPLETEFREQFRAG($\mathcal{F}$, $\mathcal{D}_\Delta$) /* Algorithm 5.17 */
7: $\mathcal{I} \leftarrow$ COMPLETESIF($\mathcal{I}$, $\mathcal{D}_\ell$) /* Algorithm 5.18 */
8: Build action-aware indexes using $(\mathcal{F}, \mathcal{I})$

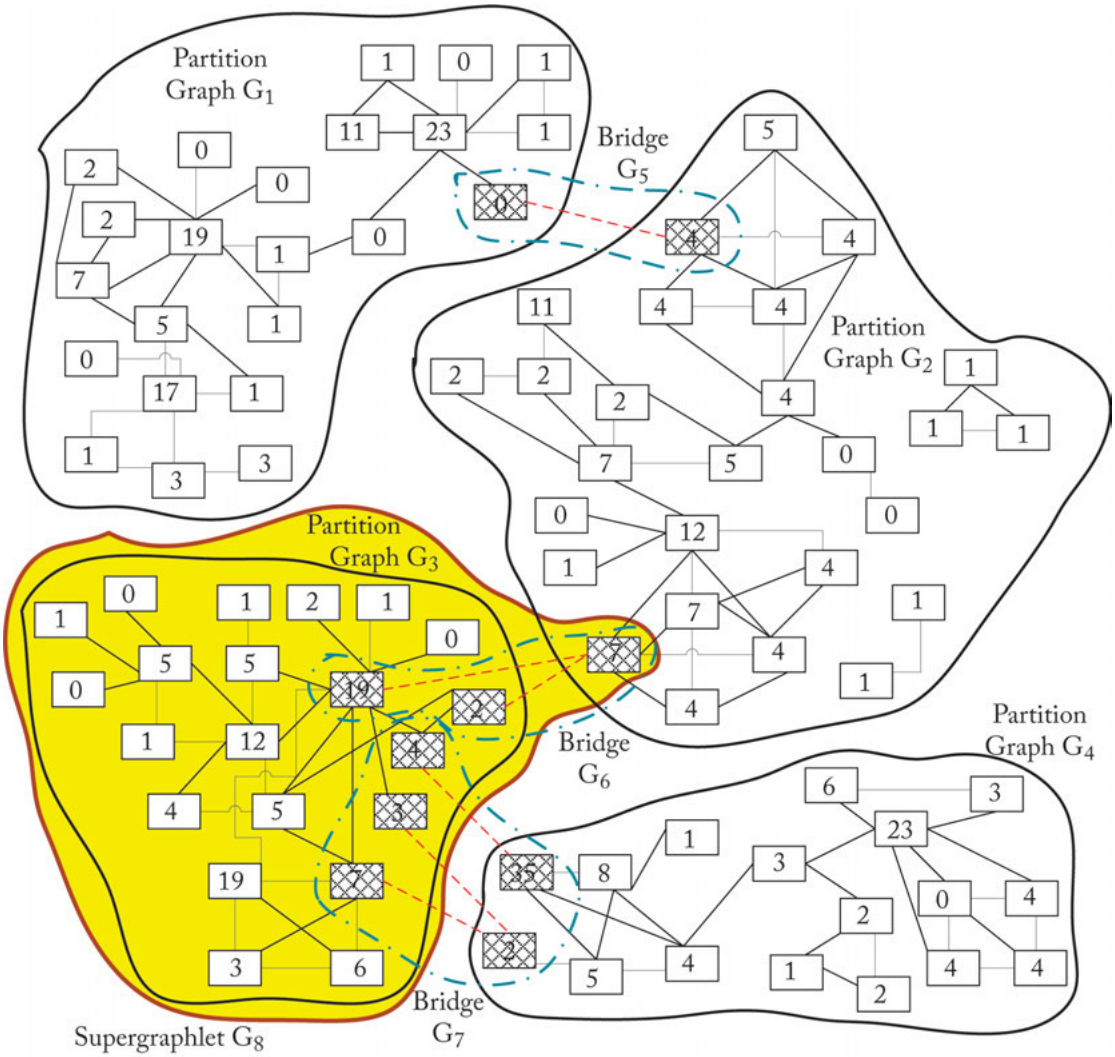

Figure 5.1: A network. The numbers within the nodes are labels and not node ids.

(e.g., G_5 linking G_1 and G_2). These seven data graphs are collectively referred to as graphlets. We elaborate on the construction of graphlets in Section 5.2.

Next, we mine these graphlets to extract frequent and *small* infrequent fragments (SIF) and their occurrences, and use them to create *graphlet-based indices* and *graphlet-based* SPIG*s* (G-SPIG), which are variants of the action-aware indices and SPIGs, respectively. At first glance, it may seem that we can use an existing frequent subgraph mining algorithm (e.g., *gSpan* [Huan

et al., 2003]) to identify all frequent fragments from the graphlet set (Line 3). Unfortunately, such an approach can only identify all *size-one* frequent fragments as an edge can only belong to exactly one graphlet. However, due to cut-based partitioning of the network, it fails to find *all* frequent fragments (having size two or more) as well as their occurrences as such a fragment may be contained in a subgraph involving multiple *adjacent* graphlets instead of a single graphlet. For instance, consider the graphlets in Figure 5.1 and the fragment g_1 in Figure 5.2a. Observe that although there are three occurrences of g_1 in the original network, only one of them occurs in the graphlet G_2. The remaining two are subgraphs of adjacent graphlets (G_3, G_6) and (G_4, G_7). Hence, if the *support threshold* is set to 2, then g_1 will be identified as an infrequent fragment instead of a frequent one. Furthermore, only the occurrence of g_1 in G_2 will be identified by the aforementioned approach. Consequently, we need to devise strategies to address this challenge.

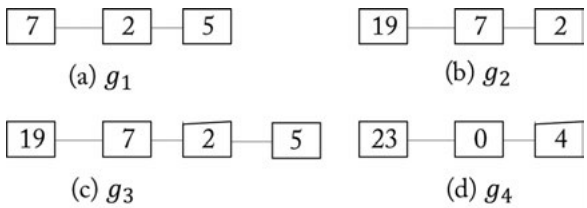

Figure 5.2: Query fragments.

The aforementioned challenge raises two important issues. First, do we need to identify *all* frequent fragments or a *partial* set is sufficient? How do we identify them? Second, irrespective of whether a fragment is frequent or infrequent, we need to devise a technique to obtain *complete* sets of occurrences of these fragments to facilitate subgraph search. How can we identify and index them efficiently?

Fortunately, as we shall see in Section 5.6, it is not necessary to identify *all* frequent fragments to support efficient visual subgraph query processing in our paradigm. A *partial* set of frequent fragments is sufficient for our goal. Consequently, frequent fragments that are not identified as frequent are categorized as infrequent (e.g., g_1 is classified as infrequent in the above example). Importantly, as we shall see later, such "miscategorization" does not adversely impact the accuracy of our framework. The second issue, however, needs to be addressed carefully to support efficient subgraph query processing. Regardless of whether a fragment is frequent or SIF, all occurrences of the fragment must be identified and indexed (e.g., all three occurrences of g_1 must be identified). We exploit the notion of *supergraphlets* to identify an approximate set of frequent fragments as well as finding all occurrences of frequent fragments and SIFs.

Notice that some nodes in Figure 5.1 belong to multiple graphlets. Graphlets that share some nodes but not edges are referred to as *adjacent*. Hence, we can combine adjacent graphlets together to create a new graph called *supergraphlet*. For example, in Figure 5.1, G_3 and G_6 are ad-

jacent graphlets which are combined together to form the supergraphlet G_8 (subgraph shaded in yellow). Observe that $g_1 \subseteq G_8$. Obviously, constructing all possible supergraphlets is prohibitively expensive. Hence, they are *selectively* constructed to identify all occurrences of frequent fragments and SIFs. Specifically, the procedures in Line 4–7 are invoked to achieve this (detailed in Section 5.3).

> *Since the last decades, traditional subgraph query processing research on large networks considered query formulation to be orthogonal to query processing. Our visual querying paradigm shifts this traditional thinking by blending human interactions during visual query formulation with query processing.*

Blending of visual query. When a user constructs a visual query graph step-by-step, these graphlet-based indices are leveraged to generate candidate graphlets and supergraphlets. After every visual *action* taken by a user, the current query fragment is evaluated by exploiting the latency offered by the GUI. Algorithm 5.13 outlines the instantiation of Algorithm 4.3 for large networks. It is invoked whenever a user adds an edge e_m during visual query formulation. Let q be the visual query being formulated by a user. The algorithm first constructs the *graphlet-based spindle-shaped graph* (G-SPIG) S_m for e_m (Line 3). It then computes the identifiers of candidate graphlets and supergraphlets that contain q using S_m and the action-aware indexes by invoking the *ExactCandidates* procedure (Line 4). Next, for a given σ, identifiers of candidates that match approximately with q is retrieved by exploiting the G-SPIG set $\mathbb{S}$. This is encapsulated in the procedure *SimilarCandidates* (Line 5). The above steps are repeated for each new edge to incrementally update candidate identifiers until the `Run` icon is clicked (Line 8). Subgraphs that exactly match the query are verified (if necessary) from the candidate graphs and stored in *Results* (Line 9). Next, candidates that match the query approximately are added to *Results* (Line 10). Note that here we retrieve both exact and approximate matches for a query graph even if the query has exact matches. We shall elaborate on these procedures in Sections 5.4 and 5.5. Note that we do not discuss the action for modifying the query fragment (Lines 6–7) here as the procedure is same as the one described in Chapter 4. In summary, the main contributions of this chapter are as follows.

- We present a novel technique to instantiate our unified framework for blending visual subgraph query formulation and query processing on large networks.
- We demonstrate how the notion of frequent and infrequent fragments can be exploited to facilitate subgraph querying on large networks. Specifically, we present algorithms to decompose a network into a set of graphlets and supergraphlets and mine *approximate* frequent fragment set and SIFs as well as their occurrences from them. These fragments are

then used to construct action-aware indexes to support the new visual subgraph querying paradigm.

- We present a dynamic on-the-fly index structure called *graphlet-based spindle-shaped graph* (G-SPIG), which is a variant of the SPIG structure, to facilitate efficient pruning and retrieval of partial results during visual query formulation. Specifically, we describe how subgraph containment and similarity search can be performed by efficiently exploiting G-SPIGs and the latency offered by a visual querying environment.
- By applying our solution to large datasets, we show its effectiveness, significant improvement of the SRT over state-of-the-art methods based on the traditional paradigm, and ability to handle large networks for interactive subgraph querying.

Algorithm 5.13 ***An instantiation of the unified framework for large networks***

Input : GUI $action$, query q, candidate set R_q, subgraph distance threshold σ, supergraphlet set $\mathcal{D}_\Delta$, G-SPIG set $\mathbb{S}$.
Output : Query results *Results*

1: **if** *action* is `NewEdge` **then**
2: $q \leftarrow q + e_m$
3: $S_m \leftarrow$ GSpigConstruct(q, Q, e_m, $\mathbb{S}$) /*Algorithm 5.19*/
4: $R_q \leftarrow$ ExactCandidates($S_m.v_{target}$)
5: $(R_{free}, R_{ver}) \leftarrow$ SimilarCandidates(q, σ, $\mathbb{S}$)
6: **else if** *action* is `Modify` **then**
7: $q \leftarrow$QueryModification(q, R_q, $\mathbb{S}$, σ)
8: **else if** *action* is `Run` **then**
9: *Results* $\leftarrow$ ExactVerification(q, R_q, $\mathbb{S}$, $\mathcal{D}_\Delta$) /*Algorithm 5.20*/
10: *Results* $\leftarrow$ SimilarResultsGen(q, R_{free}, R_{ver}, *Results*, σ) /*Algorithm 5.21*/
11: **end if**

5.2 DECOMPOSITION OF A LARGE NETWORK

We now describe in detail how we decompose a large network into small pieces of graphlets and supergraphlets.

5.2.1 GRAPHLETS AND ADJACENT GRAPHLETS

As mentioned in the preceding section, we first partition a large network into a set of *partition graphs* and *bridges* using a cut-based graph partitioning algorithm (e.g., METIS [Karypis and

Kumar, 1999]). Note that the task of such graph partitioning algorithm is to assign a single partition number to each node of the input network based on the required number of nodes in one partition. Edges that connect nodes that have different partition numbers are "cut" away. The goal is to minimize edge-cut while trying to achieve the required number of nodes in a partition. After the partitioning, each node v is assigned a *partition number* $pid(v)$. Note that we are assuming an environment in which partitioning occurs once, while subgraph query processing can occur many times, therefore in the sequel we focus on the effect partitioning has on subgraph querying, not on the cost of partitioning itself. Additionally, although we exploit METIS, it is not tightly coupled to any specific graph partitioning technique. This enhances generality of the solution as it can be easily realized on top of any superior cut-based graph partitioning technique (by replacing METIS in Line 1 in Algorithm 5.12 with another partitioning technique).

Definition 5.2 [Partition Graph] *A **partition graph** $G_p = (V_p, E_p)$, with id p, is a subgraph of $G = (V, E)$ where $\forall v_i \in V$, $v_i \in V_p$ iff $pid(v_i) = p$.*

For example, the subgraphs with ids G_1 to G_4 in Figure 5.1 are partition graphs generated by a cut-based graph partitioning technique.

Definition 5.3 [Bridge] *Given two partition graphs G_{p_1} and G_{p_2} of $G = (V, E)$ where $p_1 \neq p_2$, a **bridge** of G_{p_1} and G_{p_2} is a graph $G_b = (V_b, E_b)$ that satisfies the following: (a) $\forall v_i \in V$, $v_i \in V_b$ iff $pid(v_i) = p_1$ and $\exists$ an edge (v_i, v_j) s.t. $v_j \in V$ and $pid(v_j) = p_2$; and (b) $\forall e = (v_1, v_2) \in E$, $e \in E_b$ iff $pid(v_1) = p_1$ and $pid(v_2) = p_2$.*

Informally, a bridge is constructed from cut edge(s) that link certain pair of partition graphs. For example, G_6 is a bridge of partition graphs G_2 and G_3. We refer to a partition graph or bridge collectively as *graphlet*. Each graphlet (denoted by G_ℓ) is identified by a unique identifier, denoted by $gid(G_\ell)$ (or gid_ℓ for brevity). Clearly, based on the above definitions, any edge in the original network can only belong to exactly one graphlet. A node in a graphlet G_ℓ is called a *boundary node iff* its degree in G_ℓ is less than its degree in the original network G. Since each edge of G belongs to exactly one graphlet, boundary nodes belong to more than one graphlet. Notice that all nodes of a bridge are boundary nodes. In the sequel, we denote the set of graphlets generated from G as $\mathcal{D}_\ell$.

Two graphlets $G_{\ell_1} = (V_{\ell_1}, E_{\ell_1})$ and $G_{\ell_2} = (V_{\ell_2}, E_{\ell_2})$ are *adjacent iff* $V_{\ell_1} \cap V_{\ell_2} \neq \varnothing$ and $E_{\ell_1} \cap E_{\ell_2} = \varnothing$. That is, adjacent graphlets share some common nodes in the original network but not edges. For example, in Figure 5.1 G_5 and G_1 are adjacent graphlets. Notice that adjacent graphlets are different from overlapping graphs where both nodes *and* edges must be shared. Clearly, two partition graphs (e.g., G_1 and G_2 in Figure 5.1) cannot be adjacent because each node has a single partition number and cannot belong to two different partition graphs at the same time. A set of graphlets is considered as an *adjacent set* (denoted as Δ) *iff* each graphlet is adjacent to at least one other graphlet in the set. For example, $\Delta = \{G_3, G_4, G_6, G_7\}$ because G_3 is adjacent to G_6 and G_7 while G_4 is adjacent to G_7.

Construction of graphlets. We now briefly describe the procedure to construct graphlets from the original network. We first obtain the partition graphs using METIS [Karypis and Kumar, 1999]. It takes as input the original network and a *partition threshold*, which specifies the target number of nodes in each partition graphs. METIS will then assign one partition number to each node in the original network. To construct a partition graph, nodes of the same partition are grouped together. If there is an edge between any two of these nodes in the original network, it will be added into the partition graph. To construct a bridge, for each pair of adjacent nodes in the original network, if their partition numbers are different, the two nodes and their edge will be added to the corresponding bridge. Observe that G is scanned only once and the time complexity of constructing the graphlet set is $O(|V|d_{\max})$ where $d_{\max}$ is the maximum degree of a node in G.

5.2.2 SUPERGRAPHLETS

A *supergraphlet* is a graph generated by *merging* a set of adjacent graphlets. Formally, let $\Delta = \{G_{\ell_1}, G_{\ell_2}, \ldots, G_{\ell_n}\}$ be an adjacent set and $n \geq 2$. Then a *supergraphlet* $G_\Delta = (V_\Delta, E_\Delta)$ of Δ is a graph satisfying the following: (a) $\forall\ v \in V_\Delta$, $v \in V_{\ell_i}$ where $0 < i \leq n$ and (b) $\forall\ e \in E_\Delta$, $e \in E_{\ell_j}$ where $0 < j \leq n$. For example, reconsider Figure 5.1. Let $\Delta = \{G_3, G_6\}$. Then the supergraphlet of Δ is G_8. In the sequel, we denote the set of all graphlets *and* all supergraphlets that can be constructed from the original network G as $\mathcal{D}_\Delta$.

Each supergraphlet G_Δ is assigned a *supergraphlet identifier*, denoted by $sgId(G_\Delta)$ ($sgId$ for brevity when the context is clear), which is generated based on the identifiers of the graphlets in Δ. A supergraphlet identifier is a concatenation of the identifiers of all graphlets in the adjacent set of a supergraphlet in ascending order. Formally, $sgId(G_\Delta)$ is $gid(G_{\ell_1}) - gid(G_{\ell_2}) - \ldots - gid(G_{\ell_n})$ where $\forall G_{\ell_i} \in \Delta$, $gid(G_{\ell_1}) < gid(G_{\ell_2}) < \ldots < gid(G_{\ell_n})$. For example, let $gid(G_3) = 3$ and $gid(G_6) = 6$ (Figure 5.1). Then the supergraphlet identifier of G_8 is `3-6`. We denote a $gid(G_{\ell_i})$ contained in an $sgId(G_\Delta)$ as $gid(G_{\ell_i}) \in sgId(G_\Delta)$. Observe that a gid can be considered as a special case of supergraphlet identifier containing only a single identifier. Hence, in the sequel we shall use the supergraphlet identifier to denote a graphlet identifier as well.

We define two operations, *union* and *intersection*, on supergraphlet identifiers. Given $sgId(G_{\Delta_1})$ and $sgId(G_{\Delta_2})$, the *union* of these two identifiers, denoted as $sgId(G_{\Delta_1} \cup G_{\Delta_2})$ ($sgId_{\Delta_1 \cup \Delta_2}$ for brevity), is a $sgId$ which contains all distinct graphlet identifiers of $sgId(G_{\Delta_1})$ and $sgId(G_{\Delta_2})$. On the other hand, the *intersection* of $sgId(G_{\Delta_1})$ and $sgId(G_{\Delta_2})$, denoted as $sgId(G_{\Delta_1} \cap G_{\Delta_2})$ ($sgId_{\Delta_1 \cap \Delta_2}$ for brevity), is a new $sgId$ which consists of graphlet identifiers that appear in both $sgId(G_{\Delta_1})$ and $sgId(G_{\Delta_2})$. For example, let $sgId(G_{\Delta_1}) =$ `3-7` and $sgId(G_{\Delta_2}) =$ `4-7`. Then, $sgId(G_{\Delta_1} \cup G_{\Delta_2}) =$ `3-4-7` and $sgId(G_{\Delta_1} \cap G_{\Delta_2}) =$ `7`.

Definition 5.4 [Maximal Cover Graph (MCG)] *Let G_Δ be the supergraphlet of the adjacent set $\Delta = \{G_{\ell_1}, G_{\ell_2}, \ldots, G_{\ell_n}\}$. A graph Q is called* ***maximal cover graph*** *(*MCG*) of G_Δ if Q is isomorphic to $G' = (V', E')$ where $G' \subseteq G_\Delta$ and $\forall G_{\ell_i} \in \Delta$, $\exists e \in E'$ s.t $e \in E_{\ell_i}$. Q is said to have a* ***cover match*** *in the original network G.*

Example 5.5 Consider the graph fragment g_1 in Figure 5.2a. There is a subgraph isomorphism from g_1 to the supergraphlet G_{Δ_1} where $\Delta_1 = \{G_3, G_6\}$. There are two edges in g_1 where the edge (v_7, v_2) belongs to the bridge G_6 and the edge (v_2, v_5) belongs to the partition graph G_3. Hence, g_1 is an MCG of G_{Δ_1}. Similarly, g_1 is also contained in the supergraphlet G_{Δ_2} where $\Delta_2 = \{G_4, G_7\}$. Specifically, edges (v_7, v_2) and (v_2, v_5) belong to G_7 and G_4, respectively. Hence, g_1 is also an MCG of G_{Δ_2}. We can also say that g_1 has two cover matches in the network. On the contrary, g_1 is not an MCG of G_{Δ_3} where $\Delta_3 = \{G_4, G_6, G_7\}$.

Construction of supergraphlets. We now present the procedure to construct a supergraphlet G_Δ from its adjacent set Δ. Note that supergraphlets are constructed only during index construction and when we need to perform a subgraph verification (i.e., to verify if a supergraphlet actually contains a (sub)graph). Algorithm 5.14 outlines the steps. It makes use of a map data structure, denoted by A, which maps a node v's identifier (*vid*) to the adjacent nodes (*vid*s) of v. It first loads all graphlets associated with the identifiers in the adjacent set Δ. For each node v in each graphlet, the algorithm adds v's adjacency list into the list mapped by v's identifier in A (Lines 2–8). Notice that v's id is unique in the original network and not just unique within a graphlet. Also, some node identifiers may exist in more than one graphlet as one node can belong to more than one graphlet.

After creating A, for each value *vid* in the *keyset* of A, a node with id equal to *vid* is constructed and added to G_Δ (Lines 9–12). After the addition of all nodes, we process the id lists in A. For each node id *vid*′ in each list, if it is greater than the associated key *vid*, then an edge connecting nodes with identifiers *vid* and *vid*′ is constructed in G_Δ (Lines 13–19). Notice that since the graph is undirected, a node identifier comparison is required to avoid duplicate construction of edges. Finally, the supergraphlet identifier of G_Δ is set accordingly (Line 20). The time complexity of constructing a supergraphlet is $O(Nd_{\max})$ where $d_{\max}$ is the maximum degree of a node in the adjacency set.

Example 5.6 Consider the adjacent set $\Delta = \{G_1, G_5\}$. For $gid(G_1) = 1$, for each node having *vid*s from 1–24 (in G_1), its adjacency list in G_1 is added into the node id list associated with the key *vid* in A. For instance, v_1 is adjacent to v_4 and v_5 in G_1. Hence, 4 and 5 are added into the adjacency list associated with key 1 in A. Similarly, for $gid(G_5) = 5$, v_{24} is adjacent to v_{25} in G_5. Hence, 25 and 24 are added into the lists associated with keys 24 and 25, respectively. After Line 10, A has a keyset of 25 values ranging from 1–25. Hence, 25 nodes with these ids are added to the supergraphlet. The corresponding 25 node id lists associated with A are then used to create the edges. For example, for the list with key 1, since $4 > 1$ and $5 > 1$, edges (v_1, v_4) and (v_1, v_5) are added into the supergraphlet. Finally, the supergraphlet identifier is set to `1-5`.

Algorithm 5.14 ***SupGrapheletConst***

Input : Adjacent set Δ, graphlet set $\mathcal{D}_\ell$.
Output : Supergraphlet G_Δ

```
1:  Initialize map A for representing adjacent nodes of each node
2:  for all gid_i of G_{ℓ_i} in Δ do
3:      Load graphlet G_{ℓ_i} with graphlet id gid_i from D_ℓ
4:      for all v ∈ G_{ℓ_i} do
5:          vid ← vertex id of v_i
6:          Add adjacency list of v in G_{ℓ_i} into A.vid
7:      end for
8:  end for
9:  for all vid in the keyset of A do
10:     Construct node v with id vid
11:     Add v into G_Δ
12: end for
13: for all vid in the keyset of A do
14:     for all vid' in A.vid do
15:         if vid < vid' then
16:             Connect (vid, vid') in G_Δ
17:         end if
18:     end for
19: end for
20: Set G_Δ's sgId
```

5.3 INDEXING FREQUENT AND INFREQUENT FRAGMENTS

In this section, we begin by defining the notion of frequent and infrequent fragments in the context of graphlets and supergraphlets. Next, we introduce the notion of *fragment join*, which we shall leverage for frequent and infrequent fragments generation. Then, we present algorithms for generating frequent and infrequent fragments from the decomposed network. Lastly, we briefly present how the action-aware indices introduced in the preceding chapter are adopted to index these fragments.

5.3.1 FREQUENT AND INFREQUENT FRAGMENTS

Let g be a subgraph of G_ℓ or G_Δ in $\mathcal{D}_\Delta$ and has at least one edge. Then, g is a *fragment* in $\mathcal{D}_\Delta$. Given a fragment $g \subseteq G_i$ and $G_i \in \mathcal{D}_\Delta$, G_i is called a *fragment support graph* (FSG) of g. Recall that each graphlet or supergraphlet can be identified by a supergraphlet identifier. Hence, we denote a set of supergraphlet identifiers of FSGs of g as $fsgId(g)$. Then the *support* of g, denoted as $sup(g)$, is the number of *graphlets* that are FSGs of g. Recall from Section 5.1.2, that we identify an initial set of frequent fragments by mining the graphlets using *gSpan* (Line 3 in Algorithm 5.12). Hence, the support of a fragment is defined based on the number of graphlets and not supergraphlets. For example, consider the fragment g_2 in Figure 5.2b. Since $fsgId(g_2) = \{3\text{-}7, 1, 6\}$, $sup(g_2) = 2$ as we only count the graphlets. Similarly, the support of g_1 in Figure 5.2a is 1 as $fsgId(g_1) = \{3\text{-}6, 4\text{-}7, 2\}$. Obviously, $sup(g) \leq |fsgId(g)|$.

A fragment g is *frequent* if $sup(g) \geq \alpha|\mathcal{D}_\ell|$ where α is the minimum support threshold, $0 < \alpha < 1$ and $\mathcal{D}_\ell \subseteq \mathcal{D}_\Delta$. We denote the set of frequent fragments in $\mathcal{D}_\ell$ as $\mathcal{F}$. Given a fragment g, if $sup(g) < \alpha|\mathcal{D}_\ell|$ then g is an *infrequent* fragment. Since the number of infrequent fragments can be large, it is not space-efficient to index all of them. Instead, we only index *small infrequent fragments* (SIFs). Given an infrequent fragment g, g is a SIF if (a) $|g| = 1$ or (b) $|g| = 2$ and g is an MCG of at least one adjacent set. For distinction, we refer to an infrequent fragment that is not a SIF as *non-small infrequent fragment* (N-SIF). From the first condition, we can infer that all size-one fragments that are not frequent are SIFs. We elaborate on the second condition by introducing the notion of *middle vertex* for size-two fragments. Note that as $sup(g) \leq |fsgId(g)|$, g may be a frequent subgraph in the original network but $g \notin \mathcal{F}$. In this case if $|g| \leq 2$ then it is classified as a SIF. Otherwise, it is a N-SIF.

If $|g| = 2$, then the *middle vertex* of g is one of its nodes that has a degree of two. Note that middle vertices exist in all two-sized fragments. Because g is connected, it can only have at most three nodes and one of them has a degree of two. If g has multiple edges between two nodes, then it may have two nodes and two edges between them. In this case, all nodes have degree of two. Consequently, any one of these nodes can be represented as a middle vertex.

Observe that a two-sized SIF ensures that no MCG is missed. We do not consider infrequent fragments with size greater than two as we shall see later SIFs of size up to two are sufficient to support efficient filtering during visual query processing. We denote the set of SIFs as $\mathcal{I}$.

Lemma 5.7 *If $|g| = 2$ is an* MCG *of an adjacent set Δ, then Δ has exactly two graphlets and the middle vertex of g is a boundary node of these graphlets.*

Proof. *(By contradiction)* The graph g has two edges. Since an edge can only belong to one graphlet, in order to satisfy the condition of MCG (Definition 5.4), the size of the adjacent set can only be at most two. Furthermore, from the definition of adjacent set, its size must be *at least* two. Therefore, the adjacent set size is exactly two.

Suppose there is an instance of g where the middle vertex is not a boundary node. Then, all adjacent nodes of this node belong to the same graphlet. Hence, the other nodes of g and the two edges of g belong to the same graphlet, and g is not an MCG. □

Consider the two-sized graph g_1 in Figure 5.2a and the network in Figure 5.1. Here $fsgId(g_1) = \{\texttt{4-7}, \texttt{3-6}, \texttt{2}\}$ and g_1 is an MCG of $\Delta = \{G_3, G_6\}$ (Example 5.5). Observe that $|\Delta| = 2$ and in both graphlets the middle vertex (vertex with label 2) of g_1 is the boundary node.

5.3.2 FRAGMENT JOIN

Recall that $fsgId(g)$ denotes a set of identifiers of supergraphlets or graphlets in $\mathcal{D}_\Delta$ containing a fragment g. Given two fragments g_1 and g_2, the *fsgId*s of these fragments (i.e., $fsgId(g_1)$ and $fsgId(g_2)$) may share common graphlets as some of the instances of these fragments may be contained in same graphlets (or supergraphlets). A *fragment join* operation enables us to identify these common graphlets in the *fsgId*s and "join" them to form new "joined" supergraphlets. As we shall see later, such operation is useful in facilitating index construction and query processing.

Definition 5.8 [Fragment Join] *Let g_1 and g_2 be two graph fragments. Then, the* ***fragment join*** *of g_1 and g_2, denoted by $g_1 \boxtimes g_2$, returns a set of supergraph identifiers $\mathbb{J}$ such that $\forall\ sgId(G_{\Delta_i}) \in fsgId(g_1),\ sgId(G_{\Delta_j}) \in fsgId(g_2),\ sgId(G_{\Delta_{ij}}) = sgId(G_{\Delta_i} \cup G_{\Delta_j}) \in \mathbb{J}$ iff $sgId(G_{\Delta_i} \cap G_{\Delta_j}) \neq \emptyset$.*

Example 5.9 Consider the fragments g_1 and g_2 in Figures 5.2a–b. Observe that $fsgId(g_1) = \{\texttt{4-7}, \texttt{3-6}, \texttt{2}\}$ and $fsgId(g_2) = \{\texttt{3-7}, \texttt{1}, \texttt{6}\}$. To compute the fragment join of g_1 and g_2, we select one of the FSG set, e.g., $fsgId(g_2)$, and "join" it with the other by considering each element. Observe that `3-7` has common identifiers with `4-7` and `3-6` in $fsgId(g_1)$. Hence, the former can be unioned with latter identifiers. Consequently, `3-4-7` and `3-6-7` are added as results of the fragment join. Now consider the supergraphlet identifier `1` in $fsgId(g_2)$. It has no common identifier with any element in $fsgId(g_1)$. Hence, no new *sgId* is added into the join results. Lastly, *sgId* `6` shares common identifier with `3-6`. Hence, it is unioned with `3-6` to produce `3-6`, which is added into the join results. So $g_1 \boxtimes g_2 = \{\texttt{3-4-7}, \texttt{3-6-7}, \texttt{3-6}\}$.

Obviously, if we compare every pair of identifiers in the FSG sets of g_1 and g_2 to compute $g_1 \boxtimes g_2$, it is expensive as the time complexity will be $\Theta(|fsgId(g_1)||fsgId(g_2)|)$. We resolve this issue by introducing an *identifier map* data structure that enables us to avoid comparing pairs of identifiers that cannot be unioned. Intuitively, an *identifier map* of $fsgId(g)$ is a map that maps graphlet identifiers to lists of supergraphlet identifiers such that for every $sgId(G_{\Delta_i}) \in fsgId(g)$, if $gid(G_{\ell_i}) \in sgId(G_{\Delta_i})$ then $sgId(G_{\Delta_i})$ is included in the list that is mapped by the key $gid(G_{\ell_i})$. For example, the identifier map for $fsgId(g_1)$ in the above example is a map consisting of five key

values representing the five graphlets in $fsgId(g_1)$: `2`, `3`, `4`, `6`, and `7`. The lists of *sgId*s associated with these keys are `(2)`, `(3-6)`, `(4-7)`, `(3-6)`, and `(4-7)`, respectively.

Algorithm 5.15 outlines the procedure to perform a fragment join by exploiting the identifier map. For each $sgId(G_{\Delta_i}) \in fsgId(g_1)$, it adds $sgId(G_{\Delta_i})$ to each list that is mapped by each key $gid(G_{\ell_j})$ in M, where $gid(G_{\ell_j})$ is contained in $sgId(G_{\Delta_i})$ (Lines 2–6). Next, each $sgId(G_{\Delta_i})$ in $fsgId(g_2)$ is considered (Line 7). For each $gid(G_{\ell_j})$ in $sgId(G_{\Delta_i})$, it loads the list mapped by $gid(G_{\ell_j})$ in the identifier map M (Lines 8–9). For each $sgId(G_{\Delta_k})$ in that list, if it has not joined with $sgId(G_{\Delta_i})$, $sgId(G_{\Delta_i} \cup G_{\Delta_k})$ is computed and added into $\mathbb{J}$ (Lines 10–16). Note that we can easily keep track of which supergraph identifiers have already been joined with $sgId(G_{\Delta_i})$ by maintaining a boolean array or a set of joined *sgId*s for each $sgId(G_{\Delta_i})$.

Algorithm 5.15 ***FragmentJoin***

Input : $fsgId(g_1), fsgId(g_2)$.
Output : $\mathbb{J} = g_1 \boxtimes g_2$

```
1:  Initialize identifier map M
2:  for all sgId(G_{Δ_i}) ∈ fsgId(g_1) do
3:     for all gid(G_{ℓ_j}) ∈ sgId(G_{Δ_i}) do
4:        M.gid_j ← M.gid(G_{ℓ_j}) ∪ sgId(G_{Δ_i})
5:     end for
6:  end for
7:  for all sgId(G_{Δ_i}) ∈ fsgId(g_2) do
8:     for all gid(G_{ℓ_j}) ∈ sgId(G_{Δ_i}) do
9:        for all sgId(G_{Δ_k}) ∈ M.gid(G_{ℓ_j}) do
10:          if sgId(G_{Δ_i}) has not joined with sgId(G_{Δ_k}) then
11:             sgId(G_ℓ) = sgId(G_{Δ_i} ∪ G_{Δ_k})
12:             Insert sgId(G_ℓ) into 𝕁
13:          end if
14:       end for
15:    end for
16: end for
```

Observe that for each $sgId(G_{\Delta_i}) \in fsgId(g_2)$, the number of matched $sgId(G_{\Delta_j})$ retrieved from M is bounded by $O(|fsgId(g_1)|)$. Hence, the worst case complexity of the aforementioned approach is still $O(|fsgId(g_1)||fsgId(g_2)|)$. However, we do not generate any pair that do not participate in a fragment join.

Next, we discuss certain characteristics of fragment join which we shall be exploiting subsequently.

Lemma 5.10 *Let $g = (V, E)$ be a graph where $|g| > 2$. Let $g_1 = (V_1, E_1)$, $g_2 = (V_2, E_2)$, and $g_1 \subset g$, $g_2 \subset g$ such that $V_1 \cap V_2 \neq \emptyset$ and $E_1 \cap E_2 \neq \emptyset$. If a graphlet or supergraphlet G_Δ contains g, then g_1 and g_2 overlap in G_Δ.*

Proof. Let s be a mapping of g in G_Δ. Since $g_1 \subset g$ and $g_2 \subset g$, there exist subgraphs of s that are mapping of g_1 and g_2 in G_Δ. Let us refer to these mappings as s_1 and s_2, respectively. Since $V_1 \cap V_2 \neq \emptyset$ and $E_1 \cap E_2 \neq \emptyset$, s_1 and s_2 also have common nodes and edges. Consequently, g_1 and g_2 overlap in G_Δ. □

To illustrate Lemma 5.10, reconsider Example 5.9 and Figure 5.2. The graph g_3 in Figure 5.2c has size greater than two and is contained in G_8 having *sgId* `3-6`. It also contains the graphs g_1 and g_2. Observe that g_1 and g_2 overlap in G_8.

Lemma 5.11 *If graphs g_1 and g_2 overlap in a graphlet or supergraphlet G_Δ, then $sgId(G_\Delta) \in \mathbb{J}$ where $\mathbb{J} = g_1 \boxtimes g_2$.*

Proof. Suppose $g_1 \subseteq G_1 \subseteq G_\Delta$ and $g_2 \subseteq G_2 \subseteq G_\Delta$, G_1 and G_2 are graphlets or supergraphlets and all nodes and edges of G_Δ belong to either G_1 or G_2 or both. This means that $sgId(G_\Delta) = sgId(G_{\Delta_1} \cup G_{\Delta_2})$. Based on the definition of FSG, we have $sgId(G_{\Delta_1}) \in fsgId(g_1)$ and $sgId(G_{\Delta_2}) \in fsgId(g_2)$.

Let s_1 be a mapping of G_1 in G_Δ and s_2 of G_2. We have s_1 is a subgraph of G_1 and s_2 is a subgraph of G_2. Based on the definition of overlap graph, s_1 and s_2 have some common nodes and edges. So G_1 and G_2 must have some common nodes and edges and hence they must contain some common graphlets. Hence, their supergraphlet identifiers also share common identifiers. That is, $sgId(G_{\Delta_1} \cap G_{\Delta_2}) \neq \emptyset$. Therefore, $sgId(G_{\Delta_1} \cup G_{\Delta_2})$, which is supergraphlet identifier of G_Δ, must be included in $\mathbb{J}$. □

Reconsider the above example. The graphs g_1 and g_2 in Figure 5.2 overlap in the supergraphlet with *sgId* `3-6`. They also overlap in the supergraphlet `3-4-7`. Observe that `3-6` and `3-4-7` both appear in $\mathbb{J}$ (Example 5.9).

Combining Lemmas 5.10 and 5.11, gives us the following.

Corollary 5.12 *If a graph G_Δ contains g where $|g| > 2$, then $sgId(G_\Delta) \in \mathbb{J}$ where $\mathbb{J} = g_1 \boxtimes g_2$, $g_1 = (V_1, E_1)$, $g_2 = (V_2, E_2)$, $g_1 \subset g$, $g_2 \subset g$ and $V_1 \cap V_2 \neq \emptyset$, $E_1 \cap E_2 \neq \emptyset$.*

In other words, the results of $g_1 \boxtimes g_2$ contain supergraphlet identifiers of all graphlets and supergraphlets that contain g. For example, the graph g_3 in Figure 5.2 is a supergraph of g_1 and g_2. From Example 5.9, we can see that supergraphlet identifiers of all graphs that contain

g_3 exist in the results of $g_1 \boxtimes g_2$. We shall be using this corollary later to generate candidate graphs during index construction and visual query processing.

5.3.3 GENERATION OF FREQUENT FRAGMENTS AND SIFS

We are now ready to present the steps for generating frequent fragments and SIFs from graphlets and supergraphlets. First, we use an existing frequent graph mining algorithm (in this work, we use *gSpan* [Huan et al., 2003]) to generate frequent fragments from the graphlet set (Line 3, Algorithm 5.12). Recall that each frequent fragment g is associated with a set of FSGs $fsgId(g)$. This step can identify *all* FSGs of size-one fragments as an edge can only belong to at most one graphlet. However, FSG sets of frequent fragments with size two or more are *incomplete* as a fragment cannot only be a subgraph of a graphlet but also a subgraph of a supergraphlet. Consequently, we need to devise a strategy to obtain *complete* sets of FSGs of frequent fragments as well as SIFs.

We take a *two-phase* approach to resolve the aforementioned issue. In the first phase, we identify all cover matches of all frequent fragments. It consists of two key steps. We identify all cover matches of frequent fragments having size equal to two. That is, we identify all supergraphlets containing size-two frequent fragments (Line 5, Algorithm 5.12). During this step, we also identify size-two SIFs and some of their cover matches in the supergraphlets. Next, we identify all cover matches for frequent fragments having size greater than two (Line 6). In the second phase, we complete identification of FSG sets of *all* SIFs (Line 7). We now elaborate on these phases in turn.

Phase 1. Recall that the frequent fragments generated by *gSpan* are grouped by their size (Line 4 in Algorithm 5.12) and parent graphs of each fragment are inspected. Grouping by size has a time complexity of $O(|\mathcal{F}| \log |\mathcal{F}|)$. To find parent graphs of a fragment of size k, the group containing fragments of size $(k-1)$ is inspected and the subgraph isomorphism test is performed.
Completion of the FSG *set of size-two frequent fragments.* Next, we proceed to complete the FSG set of frequent fragments of size two. Recall that if a size-two fragment g is not frequent and there exists cover matches to g, then it is a SIF. Hence, all fragments of size two that have cover matches need to be identified as they are either frequent fragments or SIFs. Based on Lemma 5.7, these fragments can only be MCGs of adjacent sets containing exactly two graphlets. Further, the middle vertex of these fragments must be a boundary node. Hence, we can exploit these two features of cover matches of size-two frequent fragments to identify them.

Algorithm 5.16 outlines the procedure to identify cover matches of size-two frequent fragments. For every boundary node v of each graphlet in $\mathcal{D}_\ell$, we construct all possible fragment $g = \{(v_1, v, v_2), (v_1, v), (v, v_2)\}$ where v_1 and v_2 are adjacent nodes of v in the original network G and $pid(v_1) \neq pid(v_2)$. If the partition number of v, $pid(v)$, is different from both $pid(v_1)$ and $pid(v_2)$, edges (v_1, v) and (v, v_2) belong to two different bridges. Otherwise, if $pid(v) = pid(v_1)$ or $pid(v) = pid(v_2)$, then one edge of g belongs to a partition graph and the other edge belongs to a bridge. In both cases, the way we choose v_1 and v_2 ensures that the two edges belong to

two different graphlets (denoted by G_1 and G_2). Obviously, G_1 and G_2 are adjacent because they share the common node v. Hence, we can conclude that the fragment g is an MCG of the adjacent set $\{G_1, G_2\}$. This also means that the supergraphlet of this adjacent set contains g. Lastly, the *sgId* of this supergraphlet is added into $fsgId(g)$ and $\mathcal{F}$ or $\mathcal{I}$ is updated. Observe that we do not construct the entire supergraphlet but only its identifier.

Algorithm 5.16 ***TwoEdgeFragment***

Input : $\mathcal{F}, \mathcal{I}, \mathcal{D}_\ell$.
Output : Updated $\mathcal{F}$ and $\mathcal{I}$

```
1:  for all v ∈ G_i, G_i ∈ D_ℓ do
2:     if v is a boundary node then
3:        Adj ← get v's adjacent node array in the original graph
4:        for all i = 0 to Adj.length − 1 do
5:           for all j = (i + 1) to Adj.length − 1 do
6:              if pid(Adj[i]) < pid(Adj[j]) then
7:                 gid(G_1) ← graph containing edge (v, Adj[i])'s sgId
8:                 gid(G_2) ← graph containing edge (v, Adj[j])'s sgId
9:                 sgId ← sgId(G_1 ∪ G_2)
10:                g = ({v, Adj[i], Adj[j]}, {(v, Adj[i]), (v, Adj[j])})
11:                fsgId(g) ← fsgId(g) ∪ sgId
12:             end if
13:          end for
14:       end for
15:    end if
16: end for
```

The number of boundary nodes is bounded by $|V|$. For each boundary node, the number of size-two fragments in which the boundary node is a middle vertex is bounded by the maximum degree ($d_{\max}$) of a node in G. Hence, the time complexity of the above step is $O(|V|d_{\max}^2)$.

Example 5.13 Consider the boundary node v with label `0` in G_1 in Figure 5.1. It has two neighbors, v_1 with label `23` and v_2 with label `4`. Since $(v, v_1) \in G_1$, $gid(G_1) = 1$ (Line 7). Also, $(v, v_2) \in G_5$, so $gid(G_2) = 5$ (Line 8). Hence, the *sgId* is set to `1-5` (Line 9). The fragment g represented by these two edges is `{(23, 0)(0, 4)}` (Line 10). Hence, *sgId* `1-5` is added into $fsgId(g)$ (Line 11).

Completion of the FSG *set of frequent fragments having size greater than two.* Algorithm 5.16 identifies *all* FSGs of two-sized frequent fragments. Next, we discuss how to complete FSG

sets of frequent fragments having size greater than two. We exploit Corollary 5.12 to identify candidate supergraphlets containing these frequent fragments. Algorithm 5.17 outlines this procedure. First, for each fragment g of size $i \geq 3$, we obtain all parent graphs of g having size $(i-1)$. We randomly choose two parent graphs of g, denoted as g_1 and g_2, and compute $\mathbb{J} = g_1 \boxtimes g_2$. Next, *sgId*s containing only one *gid* are removed from $\mathbb{J}$ because such graphlets have already been discovered by *gSpan*. Lastly, the subgraph isomorphism test is performed on each graph whose $sgId \in \mathbb{J}$ and *sgId*s of matched results are added to $fsgId(g)$. Specifically, it checks if a supergraphlet in $\mathcal{D}_\Delta$ actually contains a (sub)graph. Note that whenever a supergraphlet is constructed, we insert it into $\mathcal{D}_\Delta$ for subsequent reference as it may be a candidate for different graph fragments. If $\mathcal{D}_\Delta$ already contains this supergraphlet then it simply returns it. Otherwise, Algorithm 5.14 is invoked to construct it.

Lemma 5.14 *Given a fragment g, let g_1 and g_2 be two parent graphs of g. Then $g_1 \boxtimes g_2$ contains the supergraphlet identifiers of all supergraphlets that contain g.*

Proof. We have $|g| > 2$, $g_1 \subset g$, $g_2 \subset g$, and $|g_1| = |g_2| = |g| - 1$. Suppose g_1 and g_2 have no common edges and nodes, as $|g_1| = |g| - 1$, the maximum size of g_2 is $|g| - |g_1| = 1$. But $|g| > 2$ and $|g_2| = |g| - 1$. Hence, $|g_2| > 2 - 1$ (contradiction). Therefore, g_1 and g_2 have common edges and vertices. According to Corollary 5.12, $(g_1 \boxtimes g_2)$ contains supergraphlet identifiers of all supergraphlets containing g. □

The number of fragments we need to process is bounded by $O(|\mathcal{F}|)$. For each fragment, the fragment join is bounded by $O(T^2)$ where T is the maximum size of an *fsgId* set of a fragment in $\mathcal{F}$. Therefore, the time complexity of Algorithm 5.17 is $O(|\mathcal{F}|T^2)$. Note that after this step, FSG sets of all frequent fragments are complete, including all supergraphlets that contain frequent fragments.

Example 5.15 Consider the frequent fragments f_1, f_2, and f_3 in Figure 5.3a. Algorithm 5.16 computes the complete FSG sets of f_1 and f_2 ($fsgId(f_1) = \{4\text{-}7, 3\text{-}6, 2\}$ and $fsgId(f_2) = \{2, 3, 4\}$). *gSpan* computes $fsgId(f_3) = \{2\}$, which does not identify all occurrences of f_3 in the original network. Hence, Algorithm 5.17 is invoked to compute the complete FSG set of f_3. Observe that $|f_1| = |f_2| = 2$ and $|f_3| = 3$. Therefore, they belong to groups $\mathcal{F}_2$ and $\mathcal{F}_3$, respectively. Since f_1 and f_2 are subgraphs of f_3 (Line 5), their *fsgId*s are added into *candSets* (Line 6). As f_3 only has two parent graphs, $fsgId(f_1)$ and $fsgId(f_2)$ are selected and the fragment join of f_1 and f_2 is computed (Lines 9–10). This results in $\mathbb{J} = \{2, 3\text{-}6, 4\text{-}7\}$. Hence, 2 is removed from $\mathbb{J}$ (Line 11). Next, supergraphlets with *sgId*s 3-6 and 4-7 are loaded from $\mathcal{D}_\Delta$ (Lines 13–18). Because they both contain f_3, their *sgId*s are added into $fsgId(f_3)$ (Lines 19–21) and the algorithm terminates. Therefore, $fsgId(f_3) = \{2, 4\text{-}7, 3\text{-}6\}$.

Algorithm 5.17 ***CompleteFreqFrag***

Input : $\mathcal{F}$, $\mathcal{D}_\Delta$.
Output : Updated $\mathcal{F}$

```
1:  for all group F_i where i > 2 do
2:    for all fragment g ∈ F_i do
3:      Initialize candSets of g /* holds f's parents' fsgIds */
4:      for all fragment g_j ∈ F_{i-1} do
5:        if g_j ⊂ g then
6:          Add fsgId(g_j) to candSets
7:        end if
8:      end for
9:      Select fsdId(g_1) ∈ candSets and fsdId(g_2) ∈ candSets
10:     J ← g_1 ⊠ g_2 /* Algorithm 5.15 */
11:     Remove all single value sgIds from J
12:     for all sgId(G_{Δ_k}) ∈ J do
13:       if G_Δ with sgId(G_{Δ_k}) has been constructed in D_Δ then
14:         Load corresponding graph G_Δ from D_Δ
15:       else
16:         G_Δ ← SupGraphletConst(sgId(G_{Δ_k}), D_ℓ) /* Algorithm 5.14 */
17:         Add G_Δ into D_Δ
18:       end if
19:       if verify(f, G_Δ) then
20:         Add sgId(G_Δ) to fsgId(f)
21:       end if
22:     end for
23:   end for
24: end for
```

Phase 2. Notice that in Phase 1 we have also identified all SIFs of size two and their cover matches but their FSG sets are partially complete. In this phase, we shall identify size-one SIFs and complete FSG identifier sets of SIFs of all sizes.

Algorithm 5.18 outlines the procedure to identify FSG identifiers of SIFs of all sizes. Note that size-one SIFs can be retrieved from the set of graphlets ($\mathcal{D}_\ell$). For each edge, if it is a SIF (not in $\mathcal{F}$) then it is added to the SIF set $\mathcal{I}$ and *sgId* of the graphlet containing it is inserted into the corresponding *fsgId* set (Lines 1–5). The time complexity of this step is $O(|E|)$. Notice

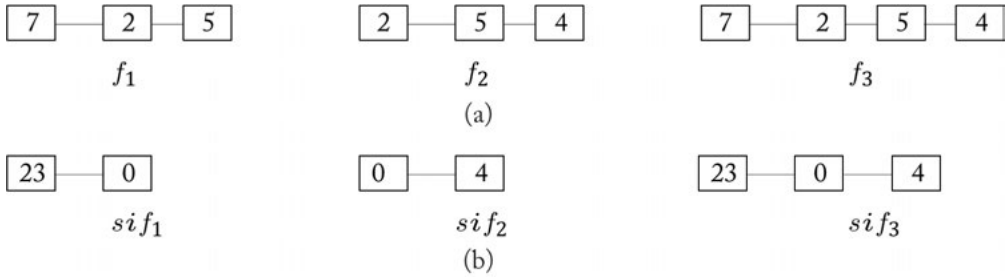

Figure 5.3: Frequent fragments and SIFs.

Algorithm 5.18 ***CompleteSIF***

Input : SIF set $\mathcal{I}$, $\mathcal{D}_\ell$.
Output : Updated $\mathcal{I}$

```
1:  for all edge g ∈ D_ℓ do
2:      if |fsgId(g)| < α|D_ℓ| then
3:          I ← (cam(g), fsgId(g))
4:      end if
5:  end for
6:  for all fragment g ∈ I where |g| = 2 do
7:      Identify g's two edges g_1, g_2
8:      Load fsgId(g_1), fsgId(g_2) from F or I
9:      candId(g) ← fsgId(g_1) ∩ fsgId(g_2)
10:     for all sgId(G_{Δ_k}) ∈ candId(g) do
11:         Load corresponding graph G_ℓ from D_ℓ
12:         if VERIFY(g, G_ℓ) then
13:             Add sgId(G_ℓ) to fsgId(g)
14:         end if
15:     end for
16: end for
```

that as a single edge cannot belong to two graphlets, *fsgId*s of size-one SIFs do not contain any supergraphlet identifier.

We have already identified size-two SIFs and their cover matches using Algorithm 5.16. However, they may also occur as subgraphs in graphlets. To identify these occurrences in order to complete the fragments' *fsgId* set, first the candidate graphlets are identified (Lines 6–9).

Specifically, the candidate set is generated as follows $candId(g) = fsgId(g_1) \cap fsgId(g_2)$ where g_1 and g_2 are size-one subgraphs of fragment g. Since a size-one fragment is either a frequent fragment or a SIF, the *fsgId* sets have already been constructed earlier. Hence, we do not need to scan the database again to obtain $candId(g)$. Next, the subgraph isomorphism test is performed on these candidate graphlets and identifiers of the matched results are added to $fsgId(g)$ (Lines 10–15).

Example 5.16 Consider the three SIFs in Figure 5.3b. Their *fsgId*s before invoking Algorithm 5.18 are $fsgId(sif_1) = \{1, 4\}$, $fsgId(sif_2) = \{2, 4, 5\}$, and $fsgId(sif_3) = \{1\text{-}5\}$. Since sif_1 and sif_2 are subgraphs of sif_3, $candId(sif_3)$ can be computed as $fsgId(sif_1) \cap fsgId(sif_2) = \{4\}$. Since G_4 contains sif_3, 4 is added to $fsgId(sif_3)$. Hence, the complete $fsgId(sif_3)$ is $\{1\text{-}5, 4\}$.

5.3.4 INDEX CONSTRUCTION

After generating complete sets of FSG identifiers, we can now adopt the action-aware indices of introduced in the preceding chapter for indexing the frequent fragments and SIFs. Specifically, the frequent fragments are indexed by A^2F-indexes. The A^2I-index indexes SIFs to prune the candidate space for infrequent queries. It consists of an array of SIFs arranged in ascending order of their sizes. Each entry stores the CAM code of a SIF g and a list of FSG identifiers of g.

5.4 GRAPHLET-BASED SPIG

In the preceding, we have seen how SPIGs have been successfully exploited in generating exact and similar subgraph candidates. Hence, we utilize this idea and create a variant of it called *graphlet-based* SPIG (G-SPIG) to suit the goal of finding exact and similar matches in the context of large graphs. We first briefly describe the structure of a G-SPIG and highlight its differences with SPIG. Next, we describe the algorithm for constructing a G-SPIG.

5.4.1 STRUCTURE OF G-SPIG

For each *new* edge e_m created by a user, we create a ***g**raphlet-based **spi**ndle-shaped **g**raph* (G-SPIG). Each edge is assigned a unique identifier according to its formulation sequence. That is, the *m-th* edge constructed by a user is denoted as e_m where m is the *label* of the edge. The edge with the *largest m* is referred to as *new edge*.

Similar to SPIG, a G-SPIG is also a directed graph $S_m = (V_m, E_m)$ where each vertex $v \in V_m$ represents a subgraph g of a query fragment containing the new edge e_m. In the sequel, we refer to a vertex v and its associated query fragment g interchangeably. There is a directed edge from a vertex v' to a vertex v if $g' \subset g$ and $|g| = |g'| + 1$. Hence, vertices that represent subgraphs of same size belong to the same level. The *source* vertex (vertex with no incoming edge) in the first level of S_m, denoted by $S_m.v_{source}$, represents e_m and the *target* vertex (vertex with no

outgoing edge) in the last level, denoted by $S_m.v_{target}$, represents the entire query fragment at a specific step.

The content of v in a G-SPIG is different from a SPIG. Specifically, each v is associated with the CAM code of the corresponding g (denoted by $cam(g)$), a list of labels of edges of g (denoted by $\mathcal{L}_E(g)$), a list of identifier set called *Indexed Fragments List*, denoted by $\mathcal{L}_{ind}(g)$, to capture information related to frequent or infrequent nature of g or its subgraphs, and a set of identifiers $\Omega(g)$ called *supergraphlet id set* to hold the *sgId*s of candidate graphlets and supergraphlets that may contain g, if g is not indexed by action-aware indices (i.e., g is a N-SIF).

An *Indexed Fragment List* $\mathcal{L}_{ind}(g) = (freqId(g), sifId(g))$ contains two attributes, namely *frequent id* ($freqId(g)$) and SIF *id* ($sifId(g)$). If g is in the A^2F-index or A^2I-index, then the identifier of the vertex or entry v representing g in the corresponding index is stored in the *frequent id* or SIF *id* attribute, respectively (i.e., $freqId(g) = a2fId(g)$ and $sifId(g) = \varnothing$ or $sifId(g) = a2iId(g)$ and $freqId(g) = \varnothing$, respectively).

If g is neither in the A^2F-index nor in the A^2I-index (i.e., $freqId(g) = sifId(g) = \varnothing$), then the *supergraphlet id set* $\Omega(g)$ stores $\mathbb{J} = g_{v1} \boxtimes g_{v2}$ where g_{v1} and g_{v2} are any two fragments associated with two different parents of v. If g_{v1} or g_{v2} is in the A^2F-index or A^2I-index, then their corresponding *fsgIds* are retrieved from these indices to compute $\mathbb{J}$. Otherwise, supergraphlet id sets of the two parents ($\Omega(g_{v1})$ and $\Omega(g_{v2})$) are used to compute $\Omega(g)$. Notice that we can always find two subgraphs to compute $\Omega(g)$ because all edges are either a frequent fragment or a SIF. Hence, a N-SIF has size of at least two and has at least two parent graphs.

Example 5.17 Suppose a user constructs the query graph in Figure 5.4a by following the sequence of actions (or steps) in Figure 5.4b. The numbers associated with edges in Figure 5.4b represent the query formulation sequence. Assume that f_1, f_2, and f_3 in Figure 5.4c are frequent fragments and sif_1 is a SIF. Let $fsgId(f_1) = \{27, 24, 11, 3, 2, 5, 30, 4, 6, 7\}$, $fsgId(f_2) = \{27, 24, 11, 9, 23, 21, 1, 16, 10\}$, $fsgId(f_3) = \{27, 24, 11, 18, 19, 14, 20, 8, 22, 15\}$, and $fsgId(sif_1) = \{27\text{–}33, 24, 11, 17, 32\text{–}35\}$. Let $a2fId(f_1) = 1$, $a2fId(f_2) = 2$, $a2fId(f_3) = 3$ and $sifId(dif_1) = 1$. Figure 5.5 depicts the G-SPIG S_4 constructed after the addition of the new edge e_4.

Each vertex represents a subgraph of the query containing e_4 and is identified by a pair of identifiers containing label of e_4 and the subgraph's breadth-first traversal order. That is, the vertex $v_{4,2}$ refers to the second vertex in S_4 that is visited during a breadth-first traversal. The identifier for each G-SPIG vertex is shown in a square bracket in Figure 5.5. A subgraph of the query associated with a vertex v is denoted as g_v. For example, the subgraph associated with vertex $v_{4,2}$ is denoted as $g_{v_{4,2}}$. Information associated with each vertex in S_4 is shown in Figure 5.6.

The first entry in the Indexed Fragment List refers to *freqId* while the other refers to *sifId* of the vertex. Since $v_{4,4}$ is a N-SIF, we need to calculate its supergraphlet identifier set. Specifically, we need to retrieve $fsgId(g_{v_{4,2}})$ and $fsgId(g_{v_{4,3}})$ and compute $g_{v_{4,2}} \boxtimes g_{v_{4,3}}$. Since $g_{v_{4,2}}$ and

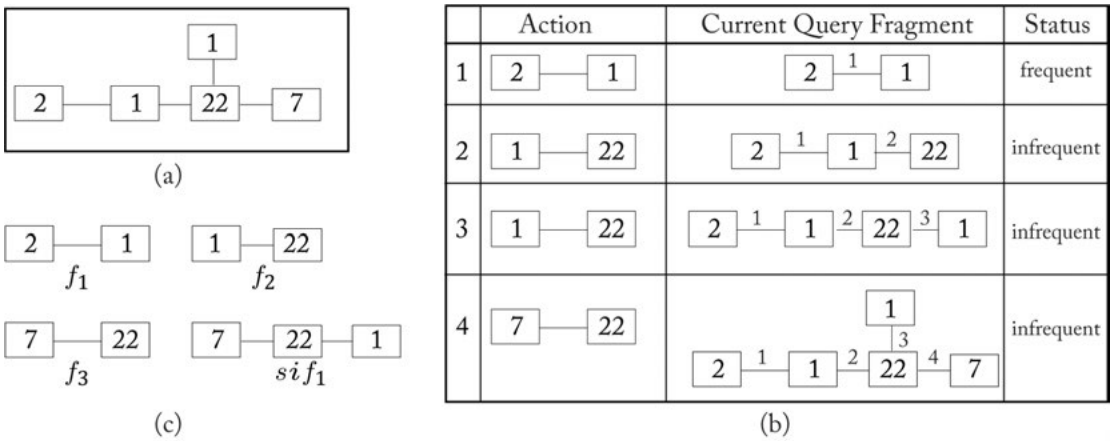

Figure 5.4: Visual query formulation.

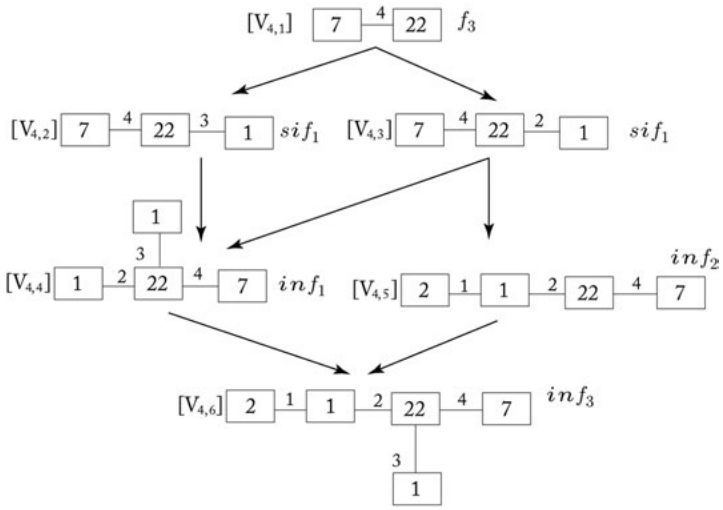

Figure 5.5: A G-SPIG.

$g_{v_{4,3}}$ are SIFs, $fsgId(g_{v_{4,2}}) = fsgId(g_{v_{4,3}}) = fsgId(sif_1)$ and is retrieved from the A^{2}I-index. Hence, $\Omega(v_{4,4}) = \{27\text{–}33, 24, 11, 17, 32\text{–}35\}$. Similarly, we can compute $\Omega(g_{v_{4,5}})$. Notice that $g_{v_{4,5}}$ has only one parent $g_{v_{4,3}}$. In this case, we use $v_{2,2}$ (of S_2) which is not a vertex of S_4 but is also associated with a parent graph of $g_{v_{4,5}}$ to compute $\Omega(g_{v_{4,5}})$ (for reasons discussed later). Since $g_{v_{2,2}}$ is a N-SIF, we use $\Omega(g_{v_{2,2}}) = g_{v_{1,1}} \boxtimes g_{v_{2,1}}$ to compute $g_{v_{2,2}} \boxtimes g_{v_{4,3}}$ where $g_{v_{1,1}}$ and $g_{v_{2,1}}$ represent f_1 and f_2, respectively. Hence, $\Omega(g_{v_{2,2}}) = \{27, 24, 11\}$. Since $g_{v_{4,3}}$ is sif_1, so $fsgId(sif_1)$ is used in computing this fragment join. Hence, $\Omega(g_{v_{4,5}}) = g_{v_{2,2}} \boxtimes g_{v_{4,3}} =$

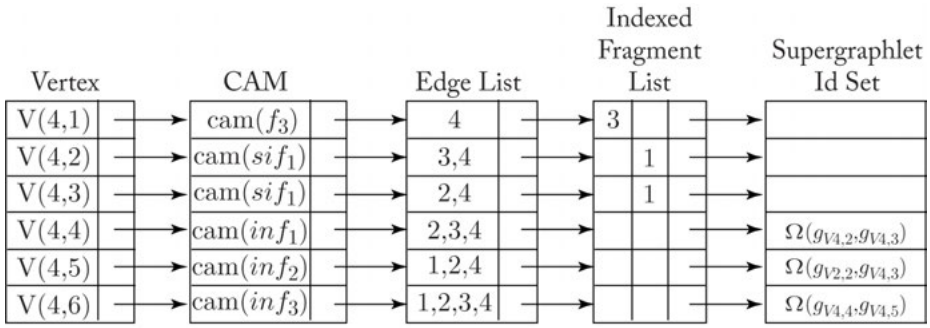

Figure 5.6: Information associated with the G-SPIG in Figure 5.5.

$\{27\text{-}33, 24, 11\}$. Lastly for $g_{v_{4,6}}$, since it is a N-SIF along with $g_{v_{4,4}}$ and $g_{v_{4,5}}$, $\Omega(g_{v_{4,4}})$ and $\Omega(g_{v_{4,5}})$ are used to calculate $\Omega(g_{v_{4,6}}) = g_{v_{4,4}} \boxtimes g_{v_{4,5}} = \{27\text{-}33, 24, 11\}$.

5.4.2 ALGORITHM

Algorithm 5.19 outlines the G-SPIG construction procedure. The building process starts from a new edge e_m (Lines 1–3). It first attaches the CAM code and edge label of e_m to the vertex $v_{m,1}$ of S_m and is enqueued in the vertex queue $\mathbb{Q}$. Let $v_{m,i}$ be a vertex dequeued from $\mathbb{Q}$ (Line 5). If the graph $g_{v_{m,i}}$ associated with $v_{m,i}$ is a frequent fragment or SIF, its frequent id or SIF id will be attached to $v_{m,i}$, respectively (Line 7). If $g_{v_{m,i}}$ is a N-SIF, the algorithm needs to find two vertices $v_{m,j1}, v_{m,j2}$ in S_m that are parents of $v_{m,i}$ (Line 9). Even though $g_{v_{m,i}}$ always has at least two subgraphs whose sizes are $|g_{v_{m,i}}| - 1$, not all of them contain the new edge e_m (e.g., $g_{v_{4,5}}$). Hence, their associated v may not belong to S_m. Consequently, if we can only find one parent of $v_{m,i}$ (there must be at least one incoming edge to $v_{m,i}$ because only the source vertex in S_m has no incoming edge), then first a subgraph g'_i can be constructed by removing e_m from $g_{v_{m,i}}$ (e.g., remove e_4 from $g_{v_{4,5}}$ in Example 5.17). Next, the algorithm seeks g'_i's associated vertex v in the $|g^i|$-th level of another G-SPIG in $\mathbb{S}$ as the second parent of $g_{v_{m,i}}$ (e.g., $g_{v_{2,2}}$ in S_2) (Lines 10–12). Note that vertices in a G-SPIG that represent subgraphs of the same size belong to the same level. Upon obtaining two parents, $g_{v_{m,j1}}$ and $g_{v_{m,j2}}$, the algorithm generates the supergraphlet id set of $g_{v_{m,i}}$ by computing fragment join of these parent graphs (Line 14). When $|g_{v_{m,i}}| = |q|$, the construction of S_m is completed and it is added to the the G-SPIG set $\mathbb{S}$ (Lines 16–17). Otherwise, the vertex $v_{m,j}$ is constructed as a child of $v_{m,i}$ in S_m. For each $g_{v_{m,j}} \supset g_{v_{m,i}}$ in q, if $v_{m,j}$ does not exist in $\mathbb{Q}$ then it attaches the CAM code and edge labels of g_j to $v_{m,j}$ and inserts the vertex into $\mathbb{Q}$. Lastly, it adds $v_{m,j}$ into S_m and constructs a directed edge from $v_{m,j}$ to $v_{m,i}$ (Lines 20–28). Figure 5.7 depicts the set of G-SPIGs created by this algorithm for the query graph in Figure 5.4a.

Algorithm 5.19 ***GSpigConstruct***

Input : Query q, Vertex queue $\mathbb{Q}$, new edge e_m, set of G-SPIGs $\mathbb{S}$.
Output : Graphlet-based spindle-shaped graph S_m

```
1:  v_{m,1} ← f(e_m)
2:  ENQUEUE(v_{m,1}, Q)
3:  INSERT(v_{m,1}, S_m)
4:  while Q ≠ ∅ do
5:      v_{m,i} ← DEQUEUE(Q)
6:      if g_{v_{m,i}} ∈ A^2F-index or A^2I-index then
7:          Attach v_{m,i} with sifId(g_{v_{m,i}}) or freqId(g_{v_{m,i}})
8:      else
9:          Find v_{m,j1}, v_{m,j2} in S_m that are parents of v_{m,i}
10:         if v_{m,j2} does not exist then
11:             g'_i ← g_{v_{m,i}} − e_m
12:             g_{v_{m,j2}} ← Search cam(g'_i) in the |g'_i|-th level of other S'_k ∈ S
13:         end if
14:         Ω(v_{m,i}) ← g_{v_{m,j1}} ⊠ g_{v_{m,j2}}
15:     end if
16:     if |g_{v_{m,i}}|=|q| then
17:         Add S_m in S
18:         return S_m
19:     else
20:         for all g_{v_{m,i}} ⊂ g_{v_{m,j}} ⊆ q and |g_{v_{m,j}}|=|g_{v_{m,i}}|+1 do
21:             if v_{m,j} ∉ Q then
22:                 v_{m,j} ← f(g_{v_{m,j}})
23:                 ENQUEUE(v_{m,j}, Q)
24:                 INSERT(v_{m,j}, S_m)
25:             end if
26:             Connect edge(v_{m,i}, v_{m,j})
27:         end for
28:     end if
29: end while
```

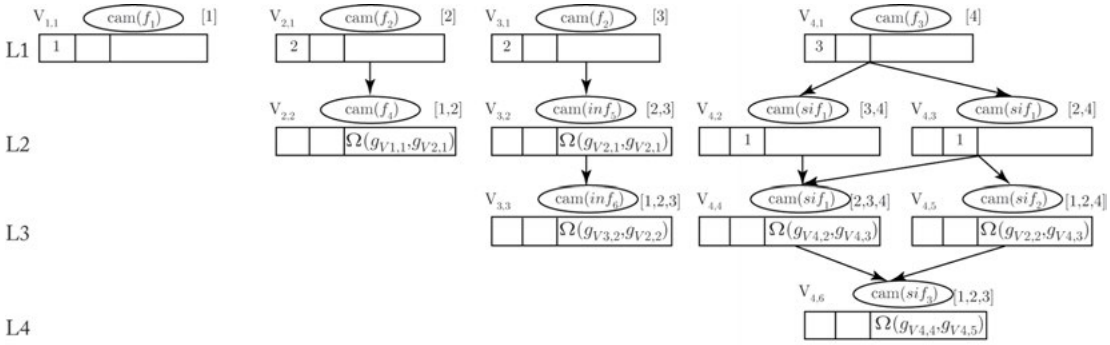

Figure 5.7: G-SPIG set (Edge Lists are in square brackets and Indexed Fragment Lists are shown in rectangular boxes).

Since, the number of vertices in each level in a G-SPIG is same as that of a SPIG, recall that the maximum number of vertices in the k-*th* level of S_m is C_{n-1}^{k-1} where n is the number of distinct edges of q. Consequently, the total number of vertices in the k-*th* levels of G-SPIGs in $\mathbb{S}$ is: $N(k) \leq C_n^k$. Note that in practice, often some nodes in q share the same vertex labels. For example, in the query in Figure 5.4(a) there are only three distinct edges (`(1,22)`, `(2,1)`, `(7,22)`). Consequently, the number of unique vertices in the k-*th* level of S_m is much less than the worst-case scenario.

Theorem 5.18 *Given a fragment $g_{v_{m,i}}$ associated with the vertex $v_{m,i}$ of a* G-SPIG, *if $g_{v_{m,i}}$ is a* N-SIF *then the supergraphlet identifiers of all graphlets and supergraphlets that contain $g_{v_{m,i}}$ are in $\Omega(g_{v_{m,i}})$.*

Proof. We have the supergraphlet id set of $g_{v_{m,i}} = g_{v_{m,j1}} \boxtimes g_{v_{m,j2}}$ where $g_{v_{m,j1}}$ and $g_{v_{m,j2}}$ are two parent vertices of $v_{m,i}$ or at the level that is right above level of $v_{m,i}$. The associated graphs of $v_{m,i}$, $v_{m,j1}$, and $v_{m,j2}$ are $g_{v_{m,i}}$, $g_{v_{m,j1}}$, and $g_{v_{m,j2}}$, respectively.

- $|g_{v_{m,i}}| > 1$ because all size-one fragments are either frequent or SIF, and hence, a N-SIF $g_{v_{m,i}}$ has a size of at least 2.
- If $|g_{v_{m,i}}| = 2$ is a N-SIF, then it does not have any cover match because if it does, it is already a SIF. As it is only contained by graphlets, all graphlets containing $g_{v_{m,i}}$ are included in $fsgId(g_{v_{m,j1}}) \cap fsgId(g_{v_{m,j2}})$ where $g_{v_{m,j1}}$ and $g_{v_{m,j2}}$ are two size-one subgraphs of $g_{v_{m,i}}$. Because any size-one fragments are either a frequent fragment or a SIF, their *fsgId* set can be retrieved from the action-aware indices. As $g_{v_{m,j1}}$ and $g_{v_{m,j2}}$ each has only one edge and an edge can only belong to one graphlet, all *sgId*s in $fsgId(g_{v_{m,j1}})$ and $fsgId(g_{v_{m,j2}})$ contain only one *gid*. Thus, for any pair $sgId_1 = \{gid_1\} \in fsgId(g_{v_{m,j1}})$ and $sgId_2 = \{gid_2\} \in fsgId(g_{v_{m,j2}})$, $sgId_1 \cap$

$sgId_2 \neq \emptyset$ *iff* $gid_1 = gid_2$. Therefore, $sgId_1 \cup sgId_2 \in g_{v_{m,j1}} \boxtimes g_{v_{m,j2}}$ *iff* they belong to $fsgId(g_{v_{m,j1}}) \cap fsgId(g_{v_{m,j2}})$. Therefore, $\Omega(g_{v_{m,i}}) = g_{v_{m,j1}} \boxtimes g_{v_{m,j2}} = fsgId(g_{v_{m,j1}}) \cap fsgId(g_{v_{m,j2}})$ also includes *sgId*s of all graphs that contain $g_{v_{m,i}}$ or $fsgId(g_{v_{m,i}}) \subset \Omega(g_{v_{m,i}})$.

- Suppose $fsgId(g_{v_{m,i}}) \subset \Omega(g_{v_{m,i}})$ is true until $|g_{v_{m,i}}| = k \geq 2$. When $|g_{v_{m,i}}| = (k+1) > 2$, $\Omega(g_{v_{m,i}}) = g_{v_{m,j1}} \boxtimes g_{v_{m,j2}}$. From Lines 9–12 of Algorithm 5.19 and based on the definition of G-SPIG, we have $|g_{v_{m,i}}| > 2$, $g_{v_{m,j1}} \subset g_{v_{m,i}}$, $g_{v_{m,j2}} \subset g_{v_{m,i}}$, and $|g_{v_{m,j1}}| = |g_{v_{m,j2}}| = |g_{v_{m,i}}| - 1 = k$.

  If $g_{v_{m,j1}}$ and $g_{v_{m,j2}}$ have no common edges and vertices, as $|g_{v_{m,j1}}| = |g_{v_{m,i}}| - 1$, the maximum size of $g_{v_{m,j2}}$ is $|g_{v_{m,i}}| - |g_{v_{m,j1}}| = 1$. But $|g_{v_{m,i}}| > 2$ and $g_{v_{m,j2}} = |g_{v_{m,i}}| - 1$. Hence, $g_{v_{m,j2}} > 2 - 1 = 1$ (contradiction).

  Therefore, $g_{v_{m,j1}}$ and $g_{v_{m,j2}}$ have common edges and vertices. According to Corollary 5.12, $\Omega(g_{v_{m,i}}) = g_{v_{m,j1}} \boxtimes g_{v_{m,j2}}$ includes *sgId*s of all graphlets and supergraphlets that contain $g_{v_{m,i}}$.

  If $g_{v_{m,j1}}$ or $g_{v_{m,j2}}$ is not frequent, $\Omega(g_{v_{m,i}})$ is used to calculate $g_{v_{m,j1}} \boxtimes g_{v_{m,j2}}$ instead. Because $|g_{v_{m,i}}| = k$, $fsgId(g_{v_{m,j}}) \subset \Omega(g_{v_{m,j}})$ is true. Thus, $\Omega(g_{v_{m,i}}) = g_{v_{m,j1}} \boxtimes g_{v_{m,j2}}$ still correctly includes *sgId*s of all graphlets and supergraphlets that contain $g_{v_{m,i}}$. Hence, $fsgId(g_{v_{m,i}}) \subset \Omega(g_{v_{m,i}})$ is true for $|g_i| = k + 1$.

Therefore, correctness of supergraphlet id set is ensured for all sizes of $g_{v_{m,i}}$. □

5.5 BLENDING VISUAL SUBGRAPH QUERY

We now have all the machinery in place to facilitate blending of query formulation and processing for subgraph search on large networks.

5.5.1 CANDIDATE DATA GRAPHS GENERATION

Exact candidate Set. The algorithm for generating candidate data graphs satisfying exact match is similar to Algorithm 4.8. Given the G-SPIG S_m of the latest added edge e_m and v is the target vertex of S_m, if v represents a frequent fragment, then it retrieves *fsgId* set of that fragment from the A^2F-index and use it as the candidate set. If v represents a SIF, then the algorithm retrieves *fsgId* set of the SIF from the A^2I-index and use it as the candidate set (replace *difId*(·) with *sifId*(·) in Lines 4–6). Otherwise, v represents a N-SIF. Hence, the supergraphlet id set of v, $\Omega(g_i)$, which is constructed during G-SPIG construction, is used as the candidate set (replace Lines 8–10 with $R_q \leftarrow \Omega(g_i)$).

Similar Candidate Set. The algorithm for generating similar subgraph matching candidates is similar to Algorithm 4.9. Given a subgraph distance threshold σ, the algorithm exploits the level structure of G-SPIG in $\mathbb{S}$ to identify relevant subgraphs of q that need to be matched for retrieving approximate candidate sets. Specifically, these subgraphs are graphs represented by

vertices at levels $|q|-1$ to $|q|-\sigma$ of all G-SPIGs in $\mathbb{S}$. Recall that the level of a vertex in a G-SPIG is the size of the query subgraph that its represent. Since the goal is to find graphlet or supergraphet that contains subgraph S_i of the query such that $dist_{S_i}(q, G) \leq \sigma$, i.e., $|S_i| \geq |q|-\sigma$, all identifiers of candidate graphs can be retrieved from vertices at levels $|q|-1$ to $|q|-\sigma$.

Let $R_{free}(i)$ and $R_{ver}(i)$ store the verification-free candidates and candidates that need verification in the i-th ($|q|$-$\sigma \leq i < |q|$) level of $\mathbb{S}$, respectively. For each vertex v_j in the i-th level, if it is a frequent fragment or SIF, then the algorithm retrieves the candidates satisfying v_j using the *ExactCandidates* procedure and combine them with $R_{free}(i)$. Otherwise, v_j is a N-SIF and requires verification. Consequently, $R_{ver}(i)$ is computed by combining $R_{ver}(i)$ with candidates returned by *ExactCandidates*. Next, it removes candidates that exist in both $R_{free}(i)$ and $R_{ver}(i)$ from $R_{ver}(i)$ as these are already identified as verification-free candidates. Finally, it adds $R_{ver}(i)$ and $R_{free}(i)$ in R_{ver} and R_{free}, respectively.

Example 5.19 Reconsider the formulation sequence of the query in Figure 5.4 and the corresponding G-SPIG set in Figure 5.6. Suppose $\sigma = 2$. When the first edge is added, G-SPIG S_1 is constructed (Line 3, Algorithm 5.13). The target vertex $v_{1,1}$ is passed as input to invoke the aforementioned algorithms. Since $freqId(v_{1,1}) = 1$, the identifier of fragment $g_{v_{1,1}}$ associated with $v_{1,1}$ is retrieved. Based on the identifier, the corresponding *fsgId* set is retrieved from the A^2F-index and assigned to R_q. Next, for similar candidate set generation, as $|q| = 1$, R_q remains unchanged.

When the second edge is added, S_2 is constructed and the target vertex is now $v_{2,2}$. Since $freqId(v_{2,2}) = sifId(v_{2,2}) = \varnothing$, $\Omega(g_{v_{2,2}}) = g_{v_{1,1}} \boxtimes g_{v_{1,2}}$, which has already been computed during the construction of S_2. Hence, it is assigned to R_q. For similar candidate set generation, as $|q| = 2$, vertices at level $i = |q| - 1 = 1$ of the G-SPIG set are considered (i.e., vertices $v_{1,1}$ and $v_{2,1}$). Since $freqId(v_{1,1}) = 1$, $fsgId(g_{v_{1,1}}) = \{27, 24, 11, 3, 2, 5, 30, 4, 6, 7\}$ (Example 5.17) is retrieved from the A^2F-index and added into $R_{free}(1)$. Similarly, $freqId(v_{2,1}) = 2$ and $fsgId(g_{v_{2,1}}) = \{27, 24, 11, 9, 23, 21, 1, 16, 10\}$ is retrieved and added into $R_{free}(1)$ set. After this step R_{ver} is empty and $R_{free}(1) = \{27, 24, 11, 3, 2, 5, 30, 4, 6, 7, 11, 9, 23, 21, 1, 16, 10\}$ is added into R_{free}.

S_3 is constructed with the addition of the third edge. Hence, $v_{3,3}$ is now the target vertex. Since $freqId(v_{3,3}) = sifId(v_{3,3}) = \varnothing$, $\Omega(g_{v_{3,3}}) = g_{v_{3,2}} \boxtimes g_{v_{2,2}}$, which has already been computed earlier, is assigned to R_q. As $|q| = 3$, vertices from level two to one (i.e., $v_{2,2}$, $v_{3,2}$, $v_{1,1}$, $v_{2,1}$ and $v_{3,1}$) of the G-SPIG set are considered. When level $i = 2$, $freqId(v_{2,2}) = sifId(v_{2,2}) = \varnothing$, $\Omega(g_{v_{2,2}}) = g_{v_{1,1}} \boxtimes g_{v_{2,1}} = \{27, 24, 11\}$ is added into $R_{ver}(2)$. Also, $freqId(v_{3,2}) = sifId(v_{3,2}) = \varnothing$, $\Omega(g_{v_{3,2}}) = g_{v_{2,1}} \boxtimes g_{v_{3,1}} = \{27, 24, 11, 9, 23, 21, 1, 16, 10\}$ is added into $R_{ver}(2)$, which is inserted into R_{ver} after executing Line 8. For $i = 1$, $fsgId(g_{v_{2,1}})$ and $fsgId(g_{v_{1,1}})$ are added into $R_{free}(1)$ in the same way as discussed in Step 2. Since $freqId(v_{3,1}) = 1$, $fsgId(g_{v_{3,1}}) = \{27, 24, 11, 9, 23, 21, 1, 16, 10\}$ is retrieved by probing the index and added into $R_{free}(1)$. Subsequently, $R_{free}(1) = \{27, 24, 11, 3, 2, 5, 30, 4, 6, 7, 11, 9, 23, 21, 1, 16, 10\}$ is added into R_{free}.

Finally, S_4 with the target vertex $v_{4,6}$ is constructed after the addition of the last edge. Since $freqId(v_{4,6}) = sifId(v_{4,6}) = \varnothing$, $\Omega(g_{v_{4,6}}) = g_{v_{4,4}} \boxtimes g_{v_{4,5}} = \{27\text{–}33, 24, 11\}$ is assigned to R_q. As $|q| = 4$, vertices from level $i = |q| - 1 = 3$ to $i = |q| - \sigma = 2$ of $\mathbb{S}$ are considered (i.e., $v_{3,3}$, $v_{4,4}$, $v_{4,5}$, $v_{2,2}$, $v_{3,2}$, $v_{4,2}$, and $v_{4,3}$). For $i = 3$, frequent ids and SIF ids of $v_{3,3}$, $v_{4,4}$, and $v_{4,5}$ are empty. Hence, their supergraphlet id sets are added into $R_{ver}(3)$. Consequently, $R_{ver}(3) = \{27\text{–}33, 24, 11, 17, 32\text{–}35, 27\}$ is added into R_{ver}. For $i = 2$, $sifId(v_{4,2}) = sifId(v_{4,3}) = 1$, so $fsgId(g_{v_{4,2}}) = fsgId(g_{v_{4,3}}) = fsgId(sif_1) = \{27\text{–}33, 24, 11, 17, 32\text{–}35\}$ are added into $R_{free}(2)$. Since frequent ids and SIF ids of both $v_{2,2}$ and $v_{3,2}$ are empty, $\Omega(g_{v_{2,2}}) = \{27, 24, 11\}$ and $\Omega(g_{v_{3,2}}) = \{27, 24, 11, 9, 23, 21, 1, 16, 10\}$ are added into $R_{ver}(2)$. Also, $R_{free}(2) = \{27\text{–}33, 24, 11, 17, 32\text{–}35\}$ and $R_{ver}(2) = \{27, 24, 11, 9, 23, 21, 1, 16, 10\}$. Hence, the common values $\{24, 11\}$ of $R_{free}(2)$ and $R_{ver}(2)$ are removed from $R_{ver}(2)$. Hence, finally, $R_{free}(2) = \{27\text{–}33, 24, 11, 17, 32\text{–}35\}$ and $R_{ver}(2) = \{27, 9, 23, 21, 1, 16, 10\}$ are added into R_{free} and R_{ver}, respectively.

5.5.2 GENERATION OF QUERY RESULTS

Exact subgraph matching results computation. The procedure for exact subgraph candidate verification is reported in Algorithm 5.20. If the final query is a frequent fragment or a SIF, the current exact subgraph candidate identifiers do not need verification and can be returned immediately as the final result (Lines 3–4). If the final query is a N-SIF, verification is required (Lines 6–16). Notice that if a *sgId* in the candidate set does not exist in $\mathcal{D}_\Delta$, Algorithm 5.14 will be invoked to construct the supergraphlet before verification is performed.

Similar subgraph matching results computation. Algorithm 5.21 outlines the procedure for generating ordered query results containing approximate matches. As the subgraph distance of candidate graphs associated with the *i-th* level of G-SPIGs in $\mathbb{S}$ is $|q| - i$, the higher level the candidate graph is in $\mathbb{S}$, the more similar it is to the query graph. At each level i from $|q| - 1$ to $|q| - \sigma$ in $\mathbb{S}$, verification-free candidates ($R_{free}(i)$) are added first in *Results* (Line 2). Then the elements in $R_{ver}(i)$ that already exist in *Results* are removed from $R_{ver}(i)$ (Line 3). This step is necessary because at lower levels, some data graph *sgId*s have already been verified and added to *Results* at previous higher levels. As such, they need to be removed from current $R_{ver}(i)$ to avoid duplicate verification. Next, each candidate element in $R_{ver}(i)$ is verified by *SimVerify* procedure to obtain the result set to be added into *Results* (Lines 5-6). Note that *SimVerify* is similar to the one employed in the preceding chapter and exploits the G-SPIGs to handle similarity verification [Hung et al., 2014].

Computation of all matches in a given supergraphlet. In our visual paradigm a user may select a result (super)graphlet G_Δ to initiate viewing all matches (both exact and approximate) to the query graph in G_Δ. We again exploit the G-SPIG of q and invoke the VF2 algorithm [Cordella et al., 2001] multiple times for each subgraph of q whose distance is not larger than σ. Since this process can produce some redundant matches, we filter them using a subset inclusion test.

Algorithm 5.20 ***ExactVerification***

Input : Query q, R_q, G-SPIG set $\mathbb{S}$, $\mathcal{D}_\Delta$.
Output : Results

```
1:  S_m ← the newest G-SPIG in 𝕊
2:  v ← S_{m,v_target}
3:  if freqId ≠ ∅ or sifId ≠ ∅ then
4:     Results ← R_q
5:  else
6:     for all sgId_i in R_q do
7:        if graph G_Δ with sgId_i already been constructed in D_Δ then
8:           Load G_Δ from D_Δ
9:        else
10:          G_Δ ← SupGraphletConst(sgId_i, D_ℓ) /* Algorithm 5.14 */
11:          Add G_Δ into D_Δ
12:       end if
13:       if verify(q, G_Δ) then
14:          Add sgId(G_Δ) into Results
15:       end if
16:    end for
17: end if
```

Note that we do not need to find approximate matches starting from $\sigma = 0$ as we can leverage previously-computed states of VF2 at termination (by the *SimVerify* procedure).

Remark 5.20 Note that our focus here is not to develop an efficient similar subgraph verification technique for large networks. In fact, similar to our approach in Chapter 4, we can easily replace the implementation of *SimVerify* with a more efficient one without modifying our framework. Similarly, since the size of a graphlet (controlled by the partition threshold) is significantly smaller than the original network to facilitate effective result visualization, we use the aforementioned simple strategy to effectively compute all matches.

5.6 PERFORMANCE STUDY

The visual subgraph querying framework discussed in this chapter is implemented in Java JDK 1.7. In this section, we refer to it is QUBLE (denoted by QUB). We run all experiments on an Intel Xeon X5570 2.93 GHz machine running on Windows 7 × 64 with 12 GB RAM. Note

Algorithm 5.21 ***SimilarResultsGen***

Input : q, R_{free}, R_{ver} and σ, G-SPIG set $\mathbb{S}$.
Output : Ordered result set *Results*

```
1: for i=|q|-σ to |q|-1 do
2:    Results ← Results ∪ R_free(i)
3:    R_ver(i) ← R_ver(i) ∩ Results
4:    for all candidate graphlet c in R_ver(i) do
5:       if SimVerify(q, 𝕊, c, |q| − i) is True then
6:          Results ← Results ∪ {c}
7:       end if
8:    end for
9: end for
```

that there is no existing system that realizes this new visual subgraph query paradigm on large networks. Nevertheless, since one of our goal is to demonstrate that our proposed paradigm does not sacrifice system response time compared to the traditional paradigm, we confine ourselves to compare it against SAPPER (denoted by SAP) for *selective*[1] experiments. We obtain the binary code of SAP from the authors of Zhang et al. [2010]. In this section, we report key performance results. For detailed results, the reader may refer to Hung et al. [2014].

5.6.1 EXPERIMENTAL SETUP

Datasets. We use subsets of *modified* DBLP co-authorship dataset modeled as a graph where a node and an edge represent an author and co-authorship, respectively. Since most of the nodes are distinct author names, there are no frequent fragments. Hence, in order to investigate the effect of frequent as well as infrequent fragments, we map each node (author) to its degree and use it as its label. This generates at most 300 distinct node labels. The characteristics of these datasets are reported in Table 5.2.

Querysets. Recall that since queries are formulated by end users using an "edge-at-a-time" visual interface, it is not realistic to expect a user to formulate large queries visually. Therefore, we chose query graphs having sizes of 4, 8, 12, 16, and 20. Figure 5.8 depicts the queryset. All these queries are chosen as N-SIFs as such type of queries demonstrate "worst" case performance of QUB. Recall that frequent and SIF queries do not need expensive candidate verification as the results can be efficiently computed directly from action-aware indexes. On the other hand,

[1]SAPPER [Zhang et al., 2010] only retrieves similarity matches containing *same* number of nodes as the query graph. In contrast, QUB allows retrieval of similar subgraphs that do not necessarily have the same number of nodes as the query. Hence, candidate data graphs as well as result sets generated by these approaches are different and incomparable.

Table 5.2: Datasets

ID	Number of Nodes	Number of Edges	Average Degree of Nodes
D1K	1,108	2,977	5.38
D5K	5,535	14,202	5.14
D10K	11,070	28,012	5.07
D20K	21,535	55,160	5.13
D50K	53,837	127,278	4.73
D100K	107,674	255,523	4.75

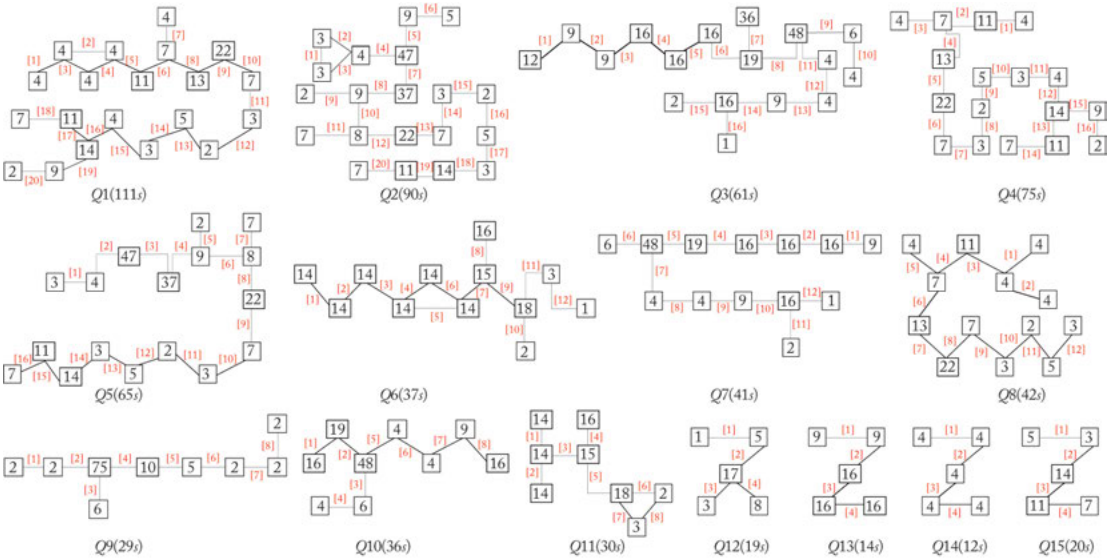

Figure 5.8: Queryset (DBLP).

NIF queries are not indexed by our indexing schemes and require candidate verification. Unless mentioned otherwise, we shall be using the default sequence for formulating a particular query.

Participants profile. Eight unpaid male volunteers (ages from 21–27) participated in the experiments. None of them are familiar with any graph query languages. They were first trained to use the GUI of QUB. For every query, participants were given some time to determine the steps that are needed to formulate it visually. This is to ensure that the effect of thinking time is minimized during query formulation. Note that faster a user formulates a query, the lesser time QUB has for G-SPIG construction. Each query was formulated five times by each participant

and reading of the first formulation was ignored. The average query formulation time (QFT) for a query by all participants is given in Figure 5.8.

Default parameter settings. We set $\sigma = 4$, $\beta = 3$, $p = 50$ for QUB and $\theta = 3$ for SAP unless specified otherwise. We chose different values for α for different datasets (Table 5.3). This is because the number of graphlets varies greatly for different datasets. For example, there are only 69 graphlets in D1K but 17716 in D100K dataset under default parameter settings. Therefore in order to have a reasonable set of frequent fragments, we choose different α values. If the same α is used for all datasets, the number of frequent fragments will be unnecessarily too high in small datasets or too small in large datasets to be meaningful. The default α value is set to α_1 in Table 5.3.

Table 5.3: α values

ID	α_1	α_2	α_3	α_4
D1K	0.02	0.03	0.04	0.05
D5K	0.02	0.03	0.04	0.05
D10K	0.01	0.015	0.02	0.025
D20K	0.01	0.015	0.02	0.025
D50K	0.005	0.01	0.015	0.02
D100K	0.005	0.01	0.015	0.02

5.6.2 SYSTEM RESPONSE TIME (SRT)

We first investigate SRTs of QUB for evaluating subgraph similarity queries by varying different parameters and compare it with SAP. The *average* SRT is computed by taking the average of SRTs of all participants (last four formulations). In the sequel, the SRT of QUB refers to this average SRT unless specified otherwise. Furthermore, we group the query set according to their sizes and SRTs of queries in each group are averaged and recorded as the SRT of that group size.

Figure 5.9 reports the SRTs for different query size. In SAP, the SRT refers to the query execution time. Each query was executed five times and results from the first run were always discarded. Observe that QUB performs significantly better than SAP consistently across different query size for different datasets. This is mainly because of QUB's approach of blending query formulation with query processing. As SAP failed to construct index for D100K in 3 h, we do not compare it with QUB for this dataset.

Effect of subgraph distance threshold. The definition of *edge edit distance* θ in SAP is not compatible with our subgraph distance threshold σ. While QUB allows both edges and nodes to be missed (i.e., the node set of a match can be a subset of the query graph's node set), edge

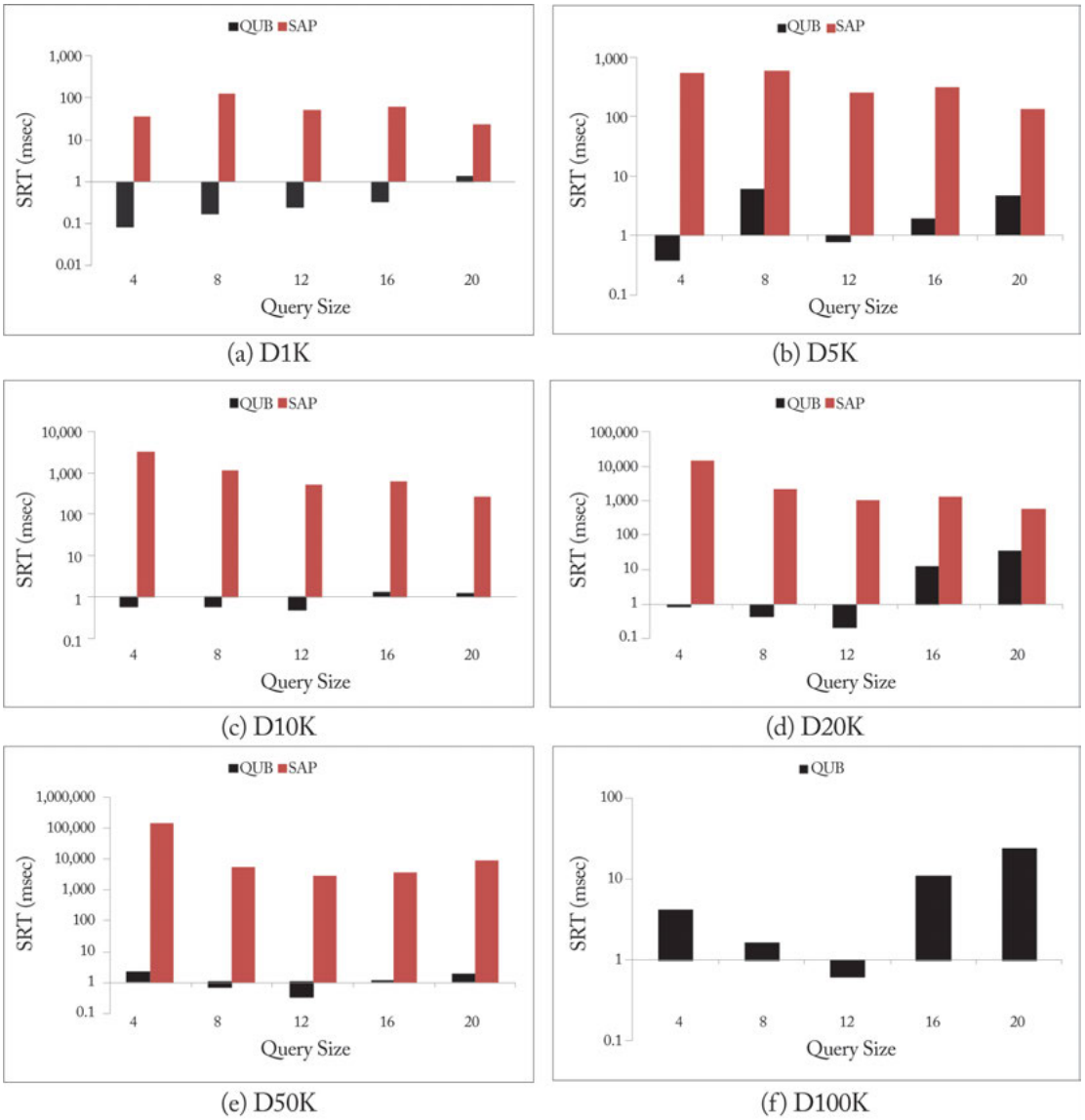

Figure 5.9: SRTs of QUBLE and SAPPER (in log scale).

edit distance only allows edges to be missed (i.e., the node set of a match must be exactly the same as that of the query graph). Besides, for an edge edit distance θ, if there is no subgraph of the query that has exactly θ edges less than the query and the same node set, SAP immediately terminates its search. So a higher θ value may not return any result for some queries. Hence,

we do not compare the SRTs of QUB with SAP under different subgraph distance threshold values. Figure 5.10 plots SRTs of QUB for different σ values. Generally, SRT increases with σ as more candidate graphs are generated when σ is relaxed, allowing more approximate matches of the query. However, in some cases it decreases with increase in σ as it is influenced by the processing order of vertices during subgraph verification. Importantly, the SRTs are cognitively negligible as typically it takes less than a second to view query results.

Effect of partition threshold. In Figure 5.11, we study the effect of partition threshold p on the SRT. In most cases, we observe an increasing trend in SRT. This is because when p increases, the size of candidate data graphs increases. Thus, it takes more time to perform subgraph isomorphism verification on these candidate graphs. In some cases, we also observe decreasing trend, due to decrease in the number of candidate graphs. For a given dataset, when p increases, the number of graphlets decreases. This sometimes leads to a decrease in the number of candidate graphs.

Effect of the number of verification candidates. Figure 5.12 depicts the number of verification candidates in QUB and corresponding SRTs of representative queries for different σ values. We use the D100K dataset as it produces the highest number of candidate graphs. We can make the following observations. In most cases, the SRT is correlated to the number of candidate graphs that require verification. This is because in QUB when a user clicks `Run`, the candidate graph set has already been generated and only verification needs to be performed. Observe that the SRT is not strictly proportional to the number of verification candidates. This is because the verification algorithm employs state space pruning, which in some cases reduces state space faster than in other cases. Lastly, observe the sharp increase in the SRT for $Q2$. This is primarily due to the simple subgraph verification method we have used rather than QUB's candidates pruning ability. We expect QUB's performance to significantly improve if a superior candidate verification algorithm is adopted.

Cost of all-matches computation. We now measure the time taken by QUB to compute all matches when a user clicks on a result graphlet. Specifically, given a query graph, we measure the *maximal computation time* (MCT), which is the highest time taken on any graphlet in the result set. That is, the MCT represents the "worst case" performance of a query. Figure 5.13 depicts the MCTs of six representative queries across different values of σ. Clearly, the MCT increases when σ increases since larger σ implies that QUB generates more approximate matches. Importantly, although we use a simple strategy for computing all matches, the MCT is still below one second which is cognitively negligible.

5.6.3 INDEX SIZE

We now investigate the index size in QUB. Note that the publicly-available executable file of SAP does not store indices back to the disk after construction and terminates right after processing the query input. Hence, we are unable to compare the index size of QUB with SAP.

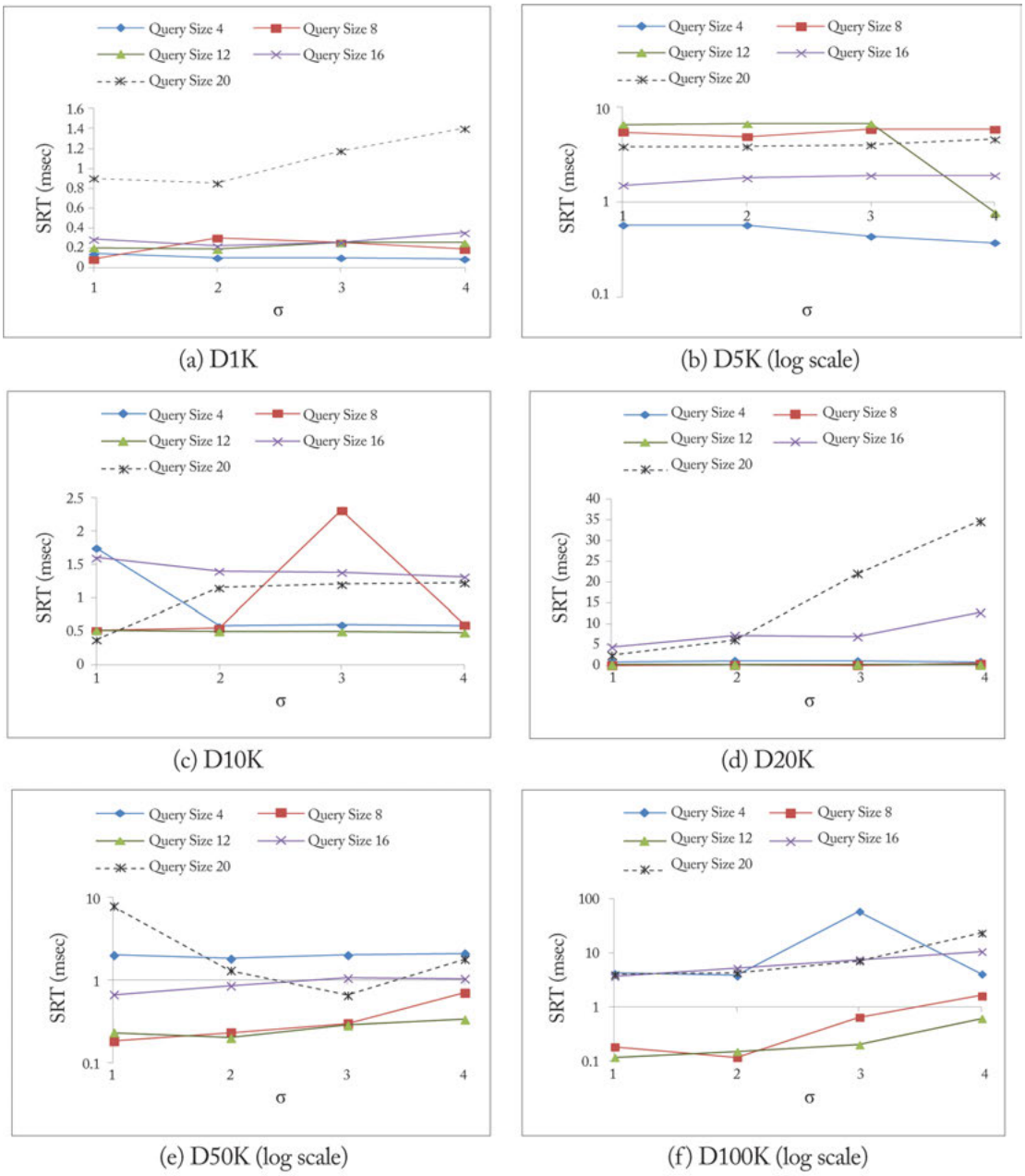

Figure 5.10: Effect of σ.

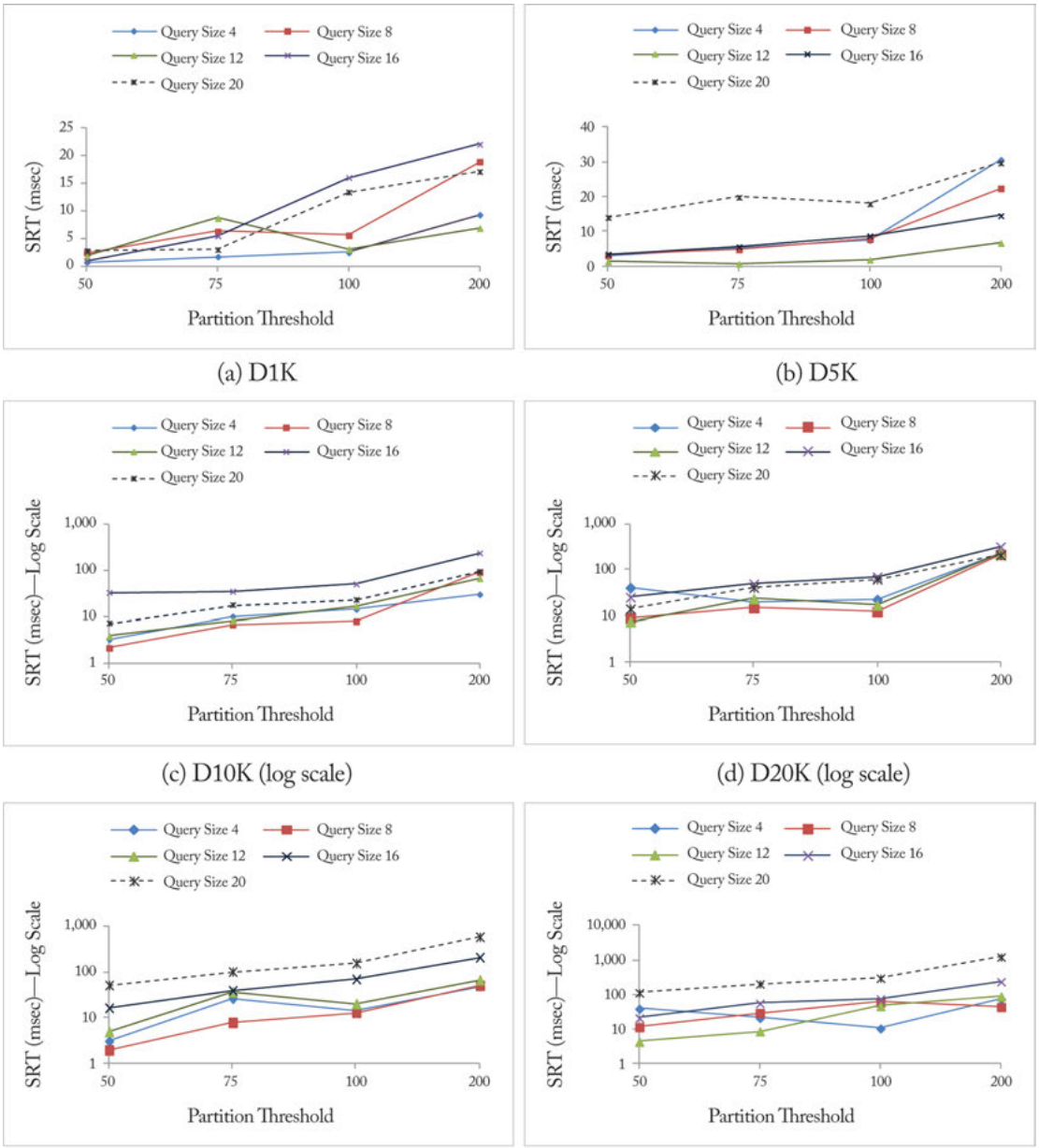

Figure 5.11: Effect of partition threshold on SRT.

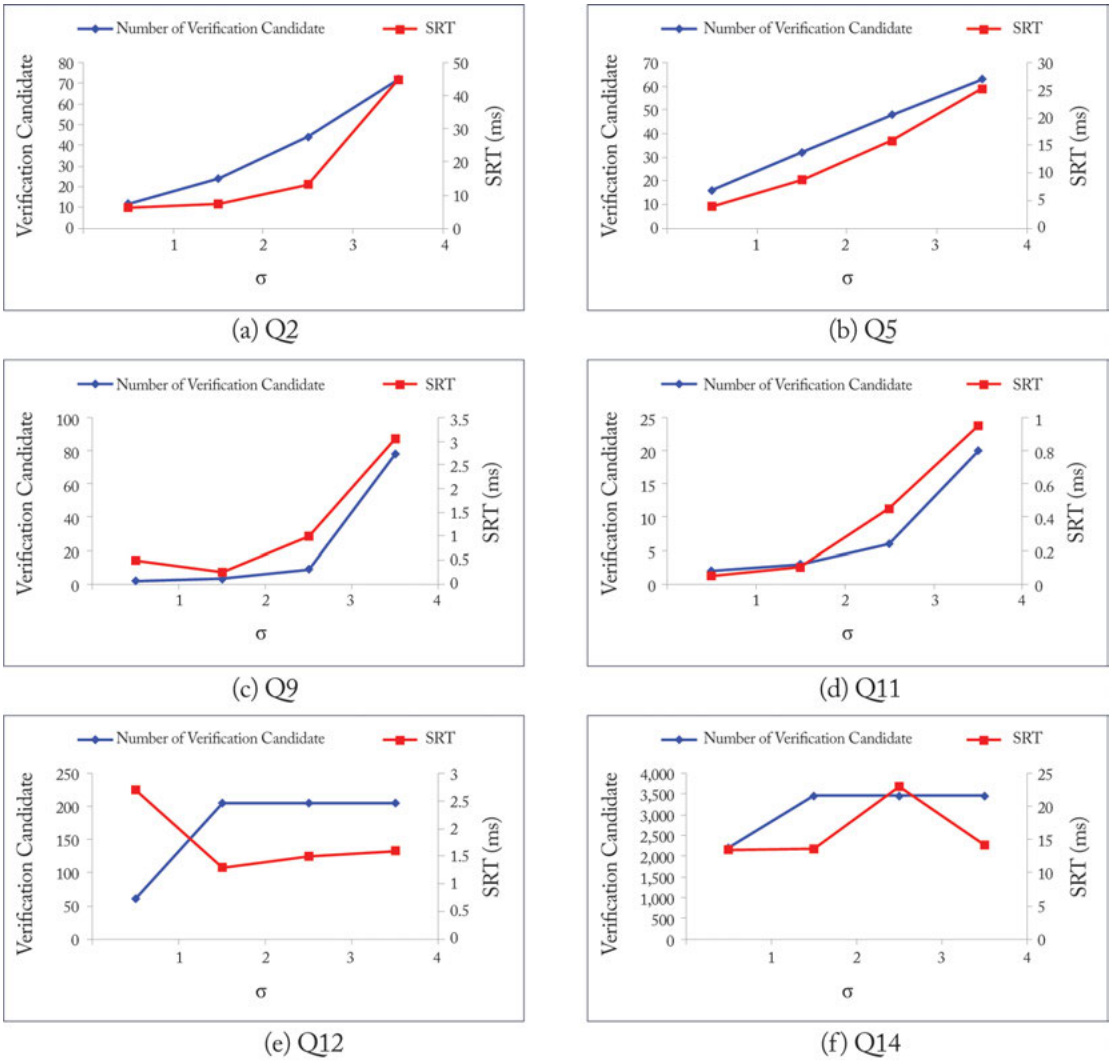

Figure 5.12: Number of verifications.

Table 5.4 reports effect of the number of nodes on index size. Observe that our index size is acceptable as most modern machines nowadays have large memory. Results related to effects of p and α on the index size and index construction cost are reported in Hung et al. [2014].

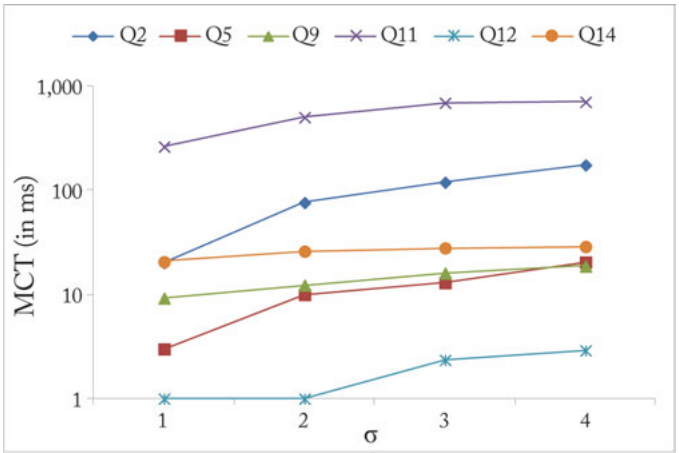

Figure 5.13: Performance of computing all matches on DBLP.

Table 5.4: Index size (MB)

Dataset	D1K	D5K	D10K	D20K	D50K	D100K
QUB	56	262	441	589	781	1167

5.6.4 PREFETCHING TIME

Recall that every time a new edge is added during visual query formulation, a G-SPIG is constructed and candidate sets are computed. During construction of a G-SPIG, we need to perform fragment join to compute the candidate set if a fragment is a N-SIF. In this set of experiments, we investigate the computation time of fragment join operation during query formulation as well as *prefetching time* (time to execute Lines 2–5 of Algorithm 5.13). Figure 5.14 reports the prefetching time and fragment join cost. Observe that all prefetching times and fragment join execution times take less than 50 msec. Hence, both these operations are efficiently supported in QUBLE and takes cognitively negligible time. Notice that the fragment join cost may be zero for the first two steps in some queries as these edges are not N-SIF and can be directly retrieved using A^2F and A^2I-indexes. More importantly, since the time taken to construct an edge in QUB typically is at least 2 sec, prefetching step can easily be completed by exploiting the GUI latency.

In Hung et al. [2014], we have reported the impact of different QFS on the prefetching time and SRT. Our results show that the prefetching operation for different sequences at each step is very efficient and takes negligible time. It is significantly lower (almost an order of mag-

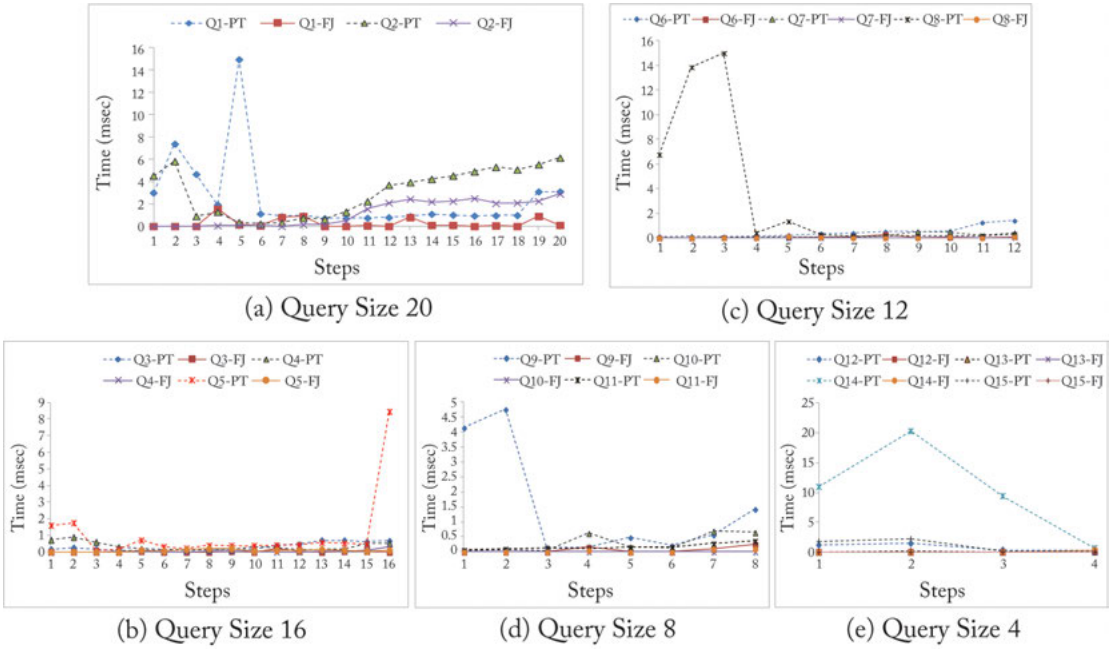

Figure 5.14: Prefetching time and cost of fragment join (in msec).

nitude) than the available GUI latency of at least 2 sec. Importantly, the formulation sequences only have minor effect on the prefetching time and SRTs.

5.6.5 PERFORMANCE ON A MILLION-NODES NETWORK

In the preceding section, we have empirically demonstrated the superiority of QUBLE in processing subgraph queries on networks containing at most 100 K nodes. Although we use a network that is an order of magnitude larger than those used in recent studies [Zhang et al., 2010], today's networks may have millions of nodes (e.g., social networks). Can QUBLE support subgraph similarity queries on such massive networks? In this section, we provide answer to this question affirmatively. Specifically, we use a *YouTube* social network graph[2] where each node represents a user and each edge represents the friendship relation between two users. This network contains 1,157,827 nodes and 2,987,624 edges. Similar to the DBLP dataset, we label each node using its degree which yields 979 unique labels. We chose 17 N-SIF queries of sizes 4, 8, 12, 16, and 20 (Figure 5.15). Unless specified otherwise, we set $\alpha = 0.002$, $\beta = 3$, $\sigma = 4$, and $p = 300$. Using this setting, we get 733,188 graphlets, 106 frequent fragments and 41,280,795 SIFs. Since the number of SIFs is very large, the A^2I-index is stored in a RDBMS (MySQL

[2]Downloaded from `http://snap.stanford.edu/data/com-Youtube.html`.

Community Server 5.5) as it is too large to fit in the main memory. Specifically, the sizes of A^2F and A^2I indexes are 1.93 MB and 23.86 GB, respectively.

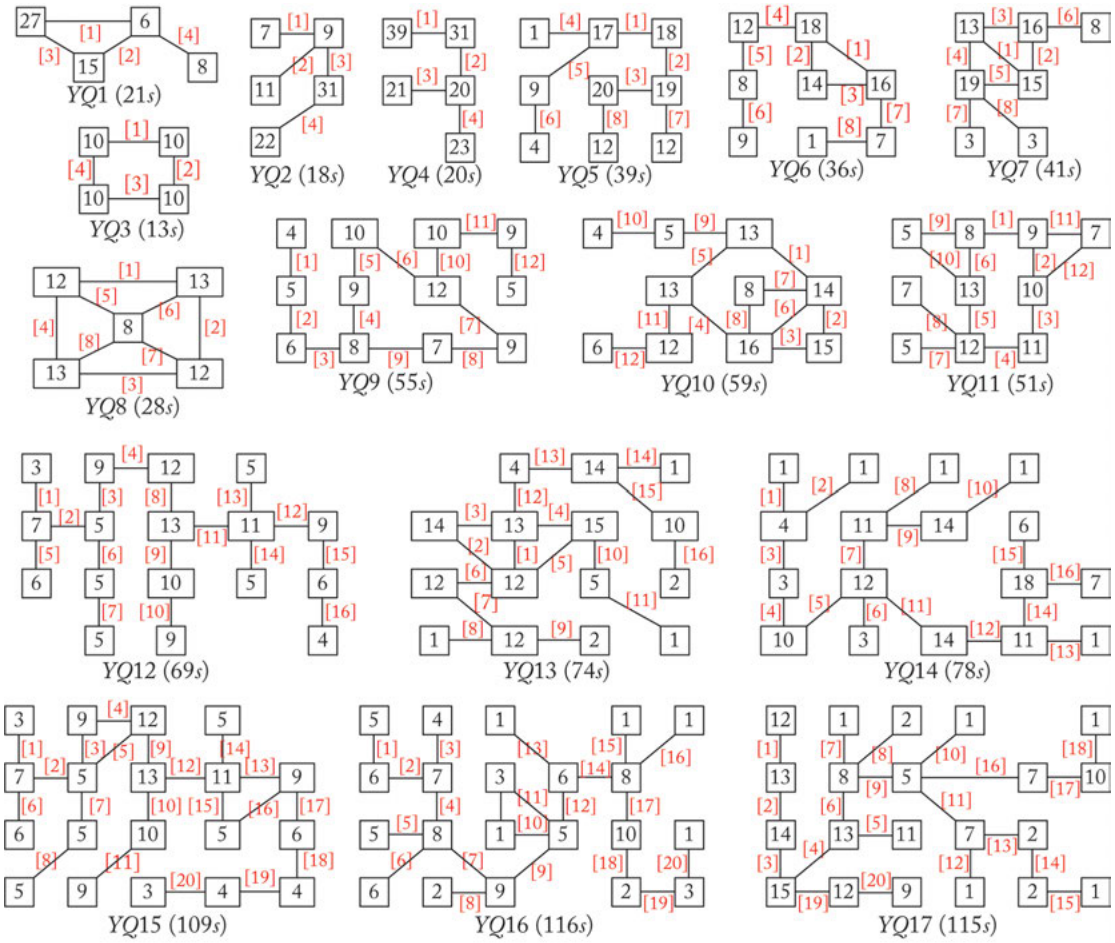

Figure 5.15: Query set (YouTube).

Figure 5.16a reports the SRT of QUB for different query size. Expectedly, it increases with query size. Figure 5.16b depicts the SRT across different values of σ. Since larger σ leads to more approximate results which demands more verifications, the SRT increases with σ. Importantly, the SRT is still below 5 sec highlighting the effectiveness of QUB to handle million-nodes networks.

Figures 5.16c–g plot the prefetching time and fragment join cost of QUB. Observe that larger queries and latter steps tend to take more prefetching time. It is expected as the core cost is the construction time of G-SPIG which grows for larger queries. Nevertheless, even for the

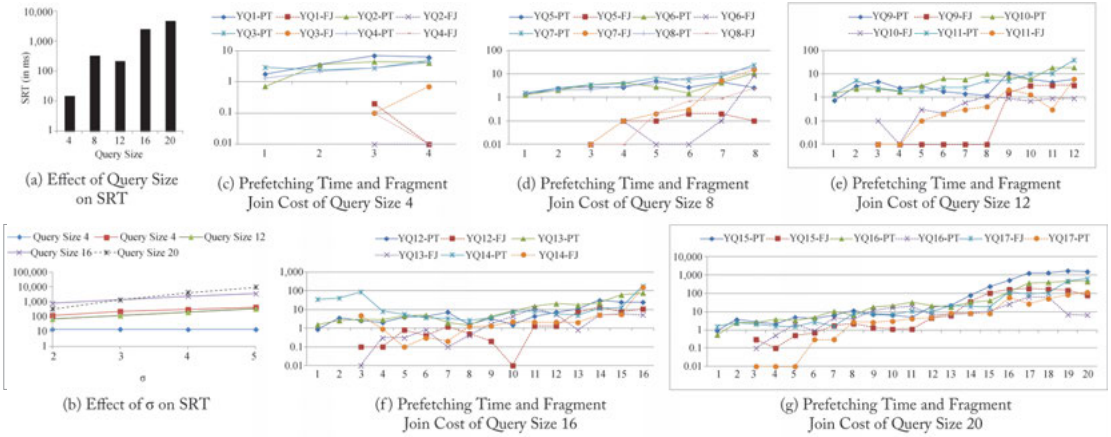

Figure 5.16: SRTs, prefetching times, and cost of fragment join for the *YouTube* network (in msec).

last step of a 20-edge query, the prefetching time is still within the latency offered by the GUI (at least 2 sec). Additionally, similar to the DBLP dataset, the fragment join operation is very efficient.

Our study also showed that despite the increase in network size, the impact of query formulation sequence on SRTs is cognitively negligible for these queries as reported in Hung et al. [2014]. Additionally, even for a million-nodes network we observed that our simple all-matches computation strategy takes only few seconds which we believe is acceptable in a visual querying environment.

5.7 BIBLIOGRAPHIC NOTES

The visual subgraph querying framework for large networks discussed in this chapter appeared in Hung et al. [2014]. There have been a number of studies to speed up evaluation of subgraph queries over large networks [Khan and Elnikety, 2014, Xie and Yu, 2011, Zhang et al., 2010, Zhu et al., 2012]. In contrast to our framework, none of these strategies address the subgraph query problem by partitioning a large network into small pieces. The indexing strategies do not exploit frequent and infrequent patterns for candidate pruning. More importantly, all these efforts follow the conventional query processing paradigm where the formulation of an entire query graph *precedes* its evaluation. Specifically, the indexing and query processing techniques are designed based on the assumption that the complete query graph is known ahead of time.

In Chapter 4, we described our visual querying paradigm to support substructure search queries on a large set of small- or medium-sized graphs. First, we focus on querying large networks here instead of a large set of small- or medium-sized graphs. Second, we use SIFs instead

of DIFs as representative infrequent fragments. Note that a DIF is either an infrequent fragment of size one or it is a smallest infrequent subgraph of an infrequent fragment. On the other hand, a SIF is either an infrequent fragment of size one or it is a two-sized infrequent fragment and an MCG of at least one adjacent set. Third, the generation of frequent fragments and SIFs are much more involved here. In the preceding chapter, we observed that frequent fragments and DIFs can be directly generated by using an existing frequent subgraph mining algorithm. In contrast, in this framework the network needs to be decomposed into graphlets and supergraphlets and then they need to be carefully processed to create approximate frequent fragment set and SIFs. Fourthly, although the topological structure of a SPIG and a G-SPIG is identical, the vertex content is different. In a G-SPIG, each vertex stores a set of (super)graphlet identifiers of ($\Omega(g)$) and a SIF id among other features, which are irrelevant in a SPIG. Consequently, the construction algorithm of G-SPIG differs from the SPIG construction. Due to these differences, the candidate generation process during query formulation also differs.

5.8 CONCLUSIONS

In this chapter, we presented another instantiation of Algorithm 4.3—a visual framework that supports processing of subgraph queries on large networks by blending their evaluation with visual query formulation. First, it decomposes a large network into a set of graphlets and supergraphlets using a minimum cut-based graph partitioning technique. Next, it mines approximate frequent fragments and SIFs from them and identifies their occurrences in the data graph. Once these fragments are generated, it extends the indexing framework described in the preceding chapter to support efficient blending of visual subgraph query formulation and query processing. Exhaustive experimental studies on large networks again validated the merit of our paradigm of blending human interactions with query processing.

CHAPTER 6

Human Interaction with Query Results

The preceding chapters have described innovative frameworks that exploit human interactions with a visual graph query interface during query formulation to process subgraph queries iteratively. Naturally, it is imperative for such visual subgraph querying frameworks to enable efficient exploration and visualization of result matches of such subgraph queries. This is a challenging problem as it requires effective summarization and visualization of the content and structure of the matching subgraphs involved in a potentially large collection of query results. In this chapter, we describe efforts reported in recent literature to this end.

Surprisingly, although there have been several work in the literature on interactive data exploration [Idreos et al., 2015], very few have focused on rich and interactive exploration of graph query results. Specifically, we can categorize visual exploration of graph query results into two types: (a) visual exploration of query results in a large collection of small- and medium-sized graphs [Huang et al., 2017, Jin et al., 2010, 2012] and (b) visual exploration of result matches in a large network [Hung et al., 2014, Pienta et al., 2016, 2018]. We elaborate on them in turn.

Despite the fact that exploration of query results is the first step toward sensemaking for a user, scant attention has been paid on this problem by both academic and commercial graph data management communities.

6.1 RESULTS EXPLORATION FOR SMALL- OR MEDIUM-SIZED DATA GRAPHS

Early efforts for exploration of query results [Jin et al., 2010, 2012] simply displayed result matches in a list, without revealing connections among results. A matching subgraph in each data graph is then highlighted with different color to identify the component of the data graph that matches a visual query. Figure 1.2 depicts an example of visualizing a result match in a data graph [Jin et al., 2010, 2012] where a result match is colored in cyan and results are presented in a scrollable list format. These results may be further sorted by various measures such as subgraph distance (for approximate match).

The aforementioned efforts do not provide additional capabilities for richer exploration and analysis of visual subgraph query results. For instance, in Jin et al. [2010, 2012] a user can only iteratively scroll through each result data graph to view query results. Clearly, this is tedious even for modest sized query results and cannot easily reveal patterns and relationships between matching subgraphs. In fact, even in traditional graph querying framework where a subgraph query is textually formulated, majority of the research efforts do not focus on efficient exploration of large query results. More importantly, a visual graph querying framework paves the way for *exploratory search* [White and Roth, 2009] that demands *multi-stream* visualization and exploration of search results. Such richer search and exploration have not been considered in these early efforts. Note that *exploratory search* represents a class of search activities that go beyond lookup retrieval and typically involves users who may not be familiar with the underlying data in a specific domain [White and Roth, 2009]. Such search process is considered dynamic. As the underlying data space is unfamiliar to the user, the query formulation evolves iteratively or progressively as the user becomes more familiar with the content. Hence, exploratory search activities have no predetermined goals and are considered as open-ended [White and Roth, 2009]. For instance, due to the topological complexity of data graphs, end users may not possess precise knowledge of the subgraph structures in a graph repository. Also, they may not always have a clear search intent leading to difficulty in formulating precise query graphs. Consequently, support for exploratory search on graph data is paramount. However, scant attention has been paid to build such a framework. Particularly, exploratory search on a graph database must confront the following challenges: ways to formulate, reformulate, and process the query graph where multiple and iterative (i.e., progressive) query formulation is necessary; cues to guide users to learn about the underlying graph data and identify possible search directions beyond the starting point specified by the initial query graph through results exploration.

6.1.1 PICASSO

PICASSO (**P**rogress**I**ve **C**onnected Subgr**A**ph **S**ubstructure **S**earch T**O**ol) [Huang et al., 2017] is a system that utilizes the query processing engine described in Chapter 4 to support visual exploratory search. It accomodates the results of the initial and reformulated query graphs to be juxtaposed in the form of *parallel search streams* (i.e., parallel query-results pairs) that facilitates exploration of the underlying data and possible identification of new search directions. Furthermore, it provides a framework to further search and analyze various features of the search results during the exploration process to facilitate understanding of the data. We now elaborate on these novel results exploration features in detail.

Recall from Chapter 4, PRAGUE utilizes the latency offered by the GUI actions to retrieve candidate data graphs matching a partial query fragment during visual query formulation. In an exploratory subgraph search, a user may iteratively reformulate and re-execute a query fragment by adding new query fragments or by deleting existing ones (an update is a sequence of deletion and insertion). Hence, we can leverage on the indexing and query processing schemes

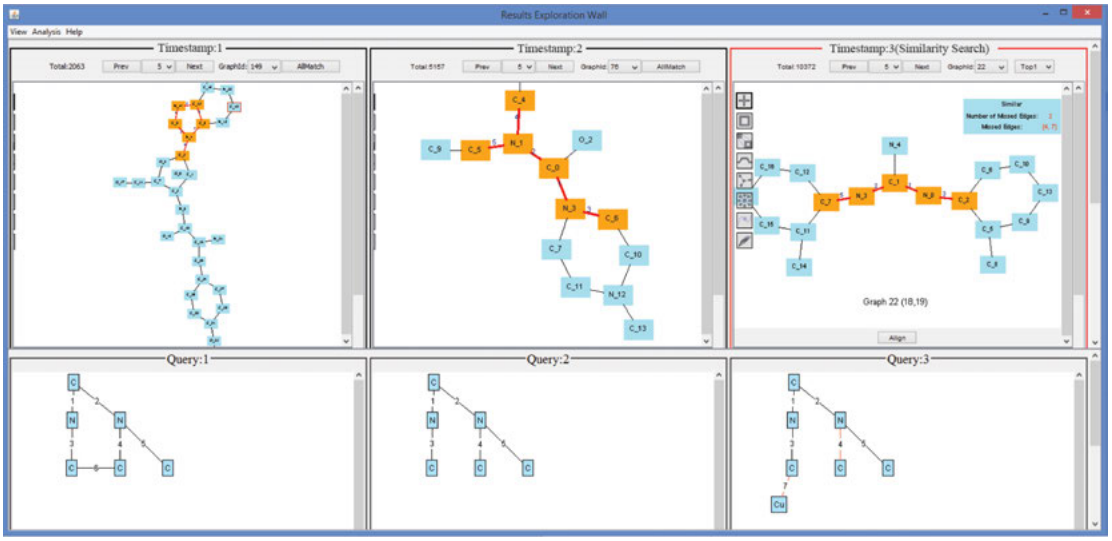

Figure 6.1: Multi-stream results viewer.

of PRAGUE to realize it. Specifically, whenever the user clicks the `Run` button, the current query fragment is processed to retrieve result matches using the PRAGUE framework. Since in an exploratory search a query may be executed several times, for each run the query-results pair is stored to generate a *search stream*.

Intuitively, *search streams* are interactive structure for viewing the query graphs and related results in order to foster analysis of former and current results during an exploratory search in a user-friendly manner. Figure 6.1 shows an example of the *multi-stream results view* to enable viewing results in the form of parallel multiple search streams. Each search stream is divided into two areas: the *query graph view* at the bottom and the *results view* on top. Each execution of a query graph fragment creates an instance of the query graph view and results view. Specifically, Figure 6.1 shows a scenario where the query graph was iteratively formulated and executed three times: (1) initial query fragment, (2) after deleting the edge `6` from the query fragment, and (3) after adding edge `7`. Hence, there are three parallel search streams. Each result data graph in a stream displays an exact or similar match to the corresponding query graph by highlighting a matching subgraph with different colored nodes and edges. The user can browse through results in each stream by clicking on the `Prev` or `Next` button at the top. She may also browse a specific result data graph by selecting its `GraphId` using the drop-down menu. Note that in the current instantiation (Figure 6.1), the query fragments at timestamps *1* and *2* have exact matches in the database. However, the reformulated query fragment at timestamp *3* does not have any exact match and hence similarity search is invoked. Consequently, in the rightmost stream the results are ranked based on the number of missing edges in ascending order. The number of missing

edges and their details are displayed in each data graph and the query fragment. For example, in Figure 6.1, edges 4 and 7 in the query (highlighted in red) are missing in the above data graph.

For visual clarity, PICASSO shows one representative matching subgraph in each data graph. If one wishes to see all matches in a data graph, she may simply click on the `AllMatch` button to view them. It implements an extended version of VF2 [Cordella et al., 2001] to find all exact or similar matches in a data graph. Each match is shown separately in the data graph by scrolling the *results view* panel of a stream vertically. For example, consider the search stream at timestamp *2* in Figure 6.1. There are four isomorphic subgraphs to the query graph in the data graph (id 76). We can retrieve all of them by clicking on the `AllMatch` button. Figure 6.2 shows two such matching subgraphs in this data graph.

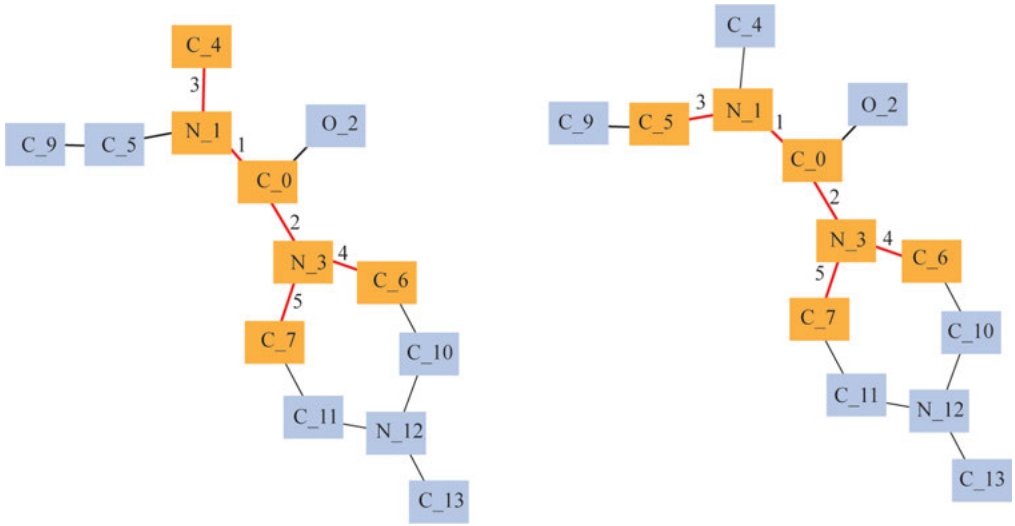

Figure 6.2: Different matches in a data graph.

PICASSO also enables us to gain insights of the results space by further searching it to track data graphs in the search streams of an exploratory search. Figure 6.3 depicts an instantiation of this feature. Currently, it supports two binary operators, namely `"In"` and `"Not In"`. The former retrieves data graphs that appear in both input search streams (identified by timestamp) whereas the latter retrieves those that exist in the left operand search stream but not in the right operand. For instance, in Figure 6.3, results that are in timestamp *2* but not in *1* are retrieved. There are 3094 data graphs in the results that satisfy this condition. One of them (graph 76) is shown in Figure 6.3. Consequently, this feature enables a user to keep track of data graphs that appear or do not appear in the results of a reformulated query graph and choose her search direction accordingly.

Results in different search streams may have different topological and statistical properties. PICASSO aids a user to view and compare them. For instance, it shows the number of results

in each search stream and type of matches (exact vs. similar) using a bar chart. Figure 6.4 (left) depicts an example for the three search streams in Figure 6.1. Clicking on a bar of a search stream enables us to view the size distribution of the data graphs in it. Figure 6.4 (right) shows the size distribution when the first bar (stream at timestamp *1*) is clicked. This module also supports comparison of these statistics across different search streams. It also visually displays several topological properties of the data graphs in a search stream (e.g., average degree, k-core, label distribution) and how they evolve with every reformulation of a query.

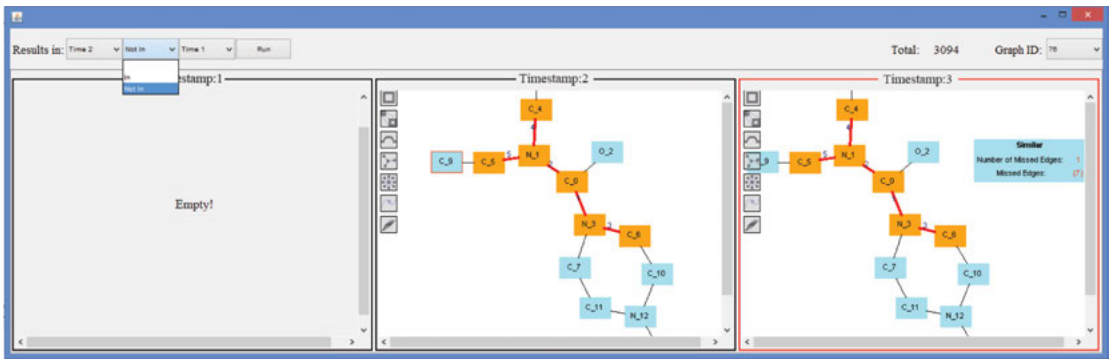

Figure 6.3: Exploration history analysis in PICASSO.

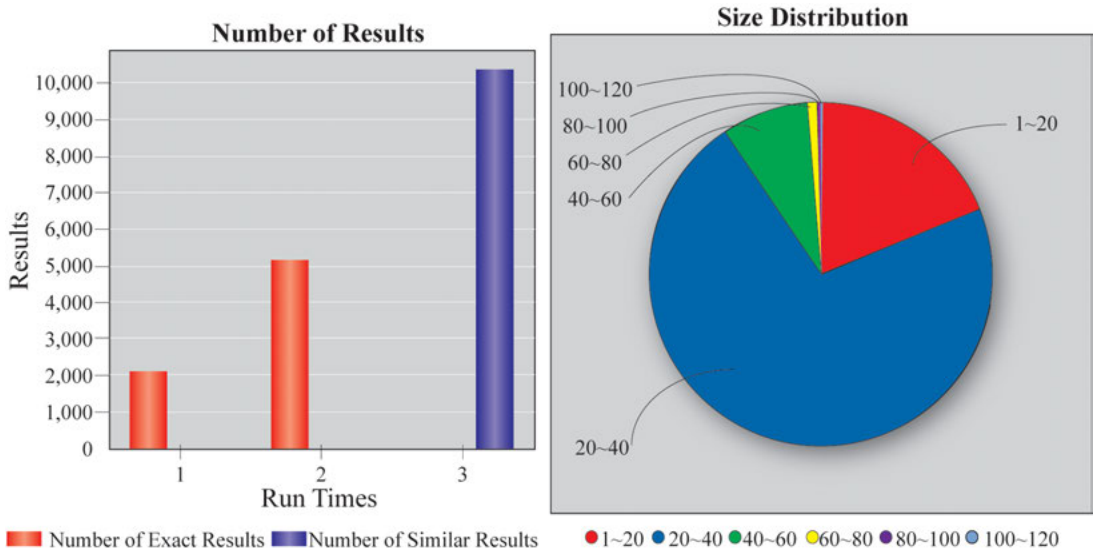

Figure 6.4: Results statistics.

In summary, PICASSO facilitates users to formulate queries toward their search goals. Specifically, search streams foster to explore the underlying data without fear of losing current work. It allows users to keep track of former query fragments, revisit them if necessary, while branching out to reformulate them to reach their search goals. Furthermore, features to support results comparison can potentially facilitate quick instantiation of new queries and query reformulation.

6.2 RESULTS EXPLORATION ON LARGE NETWORKS

It is well known that visualizing large graphs containing thousands of vertices and edges is cognitively challenging. In fact, Ware and Mitchell [2008] reported that it is visually challenging to comprehend graphs containing tens of vertices in 2D. Consequently, the aforementioned approach of highlighting the results matches of a visual subgraph query by color coding them in the original data graph is ineffective in the context of large networks as it is not only challenging to locate query results in a giant "hairball" but also it is extremely difficult to comprehend the structural relationships among the vertices in a matching subgraph overlaid on a large network. Despite these challenges, there are very few work on query results exploration on large networks. Similar to Jin et al. [2010, 2012], early efforts such as VISAGE [Pienta et al., 2016] simply displays the results in form of a list. Recently, there have been increasing attention to develop advanced techniques for query results exploration for large networks. These work can be broadly classified into three types, namely *region-based*, *exemplar-based*, and *feature-based exploration*.

6.2.1 REGION-BASED EXPLORATION

Intuitively, a *region-based exploration* scheme iteratively displays a small region of the underlying network containing a result match of a subgraph query. By showing only a fragment of the original network one at a time, it alleviates the cognitive overhead associated with the visualization of all query results on the original network. The approach in Hung et al. [2014] (detailed in Chapter 5) adopts this strategy. Specifically, it takes a two-level approach to explore and visualize query results. In the first level, it shows a set of small "pieces" of the original network where each piece contains one match (exact or approximate) of the query graph. Recall that in this approach a large network is decomposed into a set of (super)graphlets. Given a query graph, only those (super)graphlets that contain at least one result match are returned. Hence, it leverages on these decomposed pieces to display result matches. Specifically, results are viewed in "supergraphlet-at-a-time" mode where one supergraphlet or graphlet containing result matches is displayed on the results screen one at a time. Observe that such supergraphlet-driven view enables a user to clearly locate a matched result and understand its relationship with neighboring nodes. Figure 6.5 depicts the results visualization of a subgraph query (result match is colored in red). Panel 2 shows the set of (super)graphlet identifiers containing at least one matches. Each (super)graphlet occupies a section of the panel and its matches are listed underneath. By default, only matches with minimum subgraph distance to the query is displayed. For

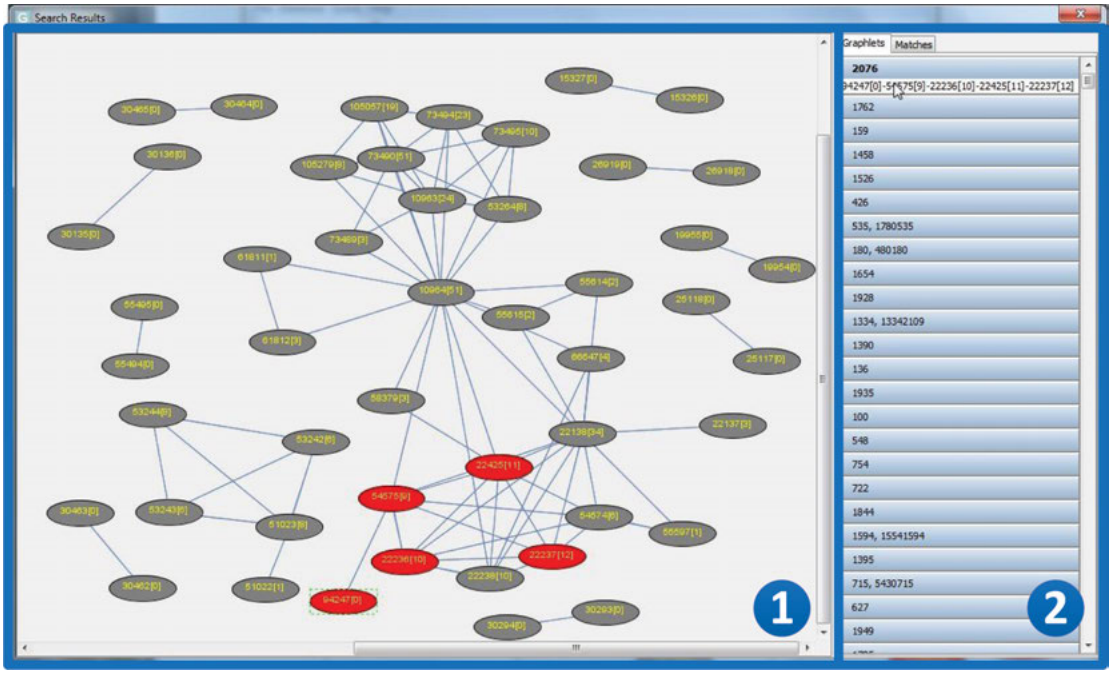

Figure 6.5: Region-based results exploration and visualization.

example, the graphlet with *sgId 2076* has a single exact match *94247[0]-54575[9]-22236[10]-22425[11]-22237[12]*.[1] Panel 1 displays the corresponding (super)graphlet (e.g., *2076*). When the mouse is hovered on a match in Panel 2, the corresponding subgraph is highlighted in Panel 1 in red color. Moreover, when a vertex is double-clicked in Panel 1, neighbors of this vertex are automatically displayed, further facilitating graph exploration.

Given a (super)graphlet containing at least one result match, the second level exploration enables a use to view *all* matches to the query graph in the supergraphlet by identifying them using VF2 [Cordella et al., 2001]. Note that since the size of a (super)graphlet is significantly smaller than the original network, the remaining matches to the query graph can be quickly computed. For instance, in the above example a user can right-click on the (super)graphlet identifier in Panel 2 to initiate computation of all matches. For example, the list associated with the graphlet *2076* in Panel 2 in Figure 6.6 shows all matches in the graphlet. Clicking on a match in Panel 2, presents a "localized" view of the matching subgraph in the network in Panel 1 (Figure 6.7). Additionally, one can also view the results sorted in ascending order of subgraph distance by clicking on the tab `Matches` in Panel 2. An interesting byproduct of this second level exploration strategy is that it is not necessary to find *all* matching subgraphs as soon as the query

[1]Here each node is represented as $id[label]$ where id is a globally unique identifier and $label$ is the node's label which is also used in the query.

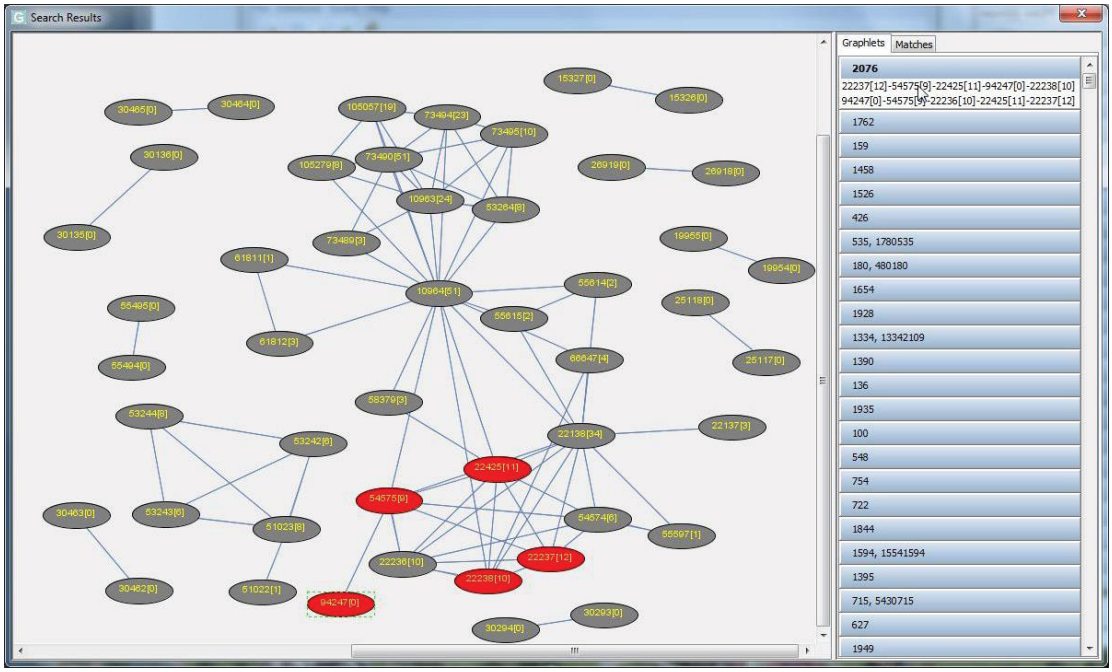

Figure 6.6: Region-based exploration of approximate matches.

is executed (i.e., the `Run` icon is clicked). Instead, some of these can be computed *on demand* as an end user explores a specific region of the underlying network containing these matches.

> *Visual exploration of subgraph query results opens up the opportunity to defer the retrieval of some matching subgraphs on-demand during results exploration instead of finding all matches during query execution, thereby improving the performance of graph querying systems.*

6.2.2 EXEMPLAR-BASED EXPLORATION

In an *exemplar-based exploration* scheme, a user can select a specific query result (i.e., an exemplar) and relax its constraints to retrieve other similar results. A user can also start exploration by specifying only the topology (without constraints on node values) and iteratively add constraints to narrow in on specific results. Such approach is adopted by VIGOR [Pienta et al., 2018]. Figure 6.8 depicts the visual interface of VIGOR for exploration of exact subgraph query results. Consider a query on DBLP that aims to find an author who has published two papers with a

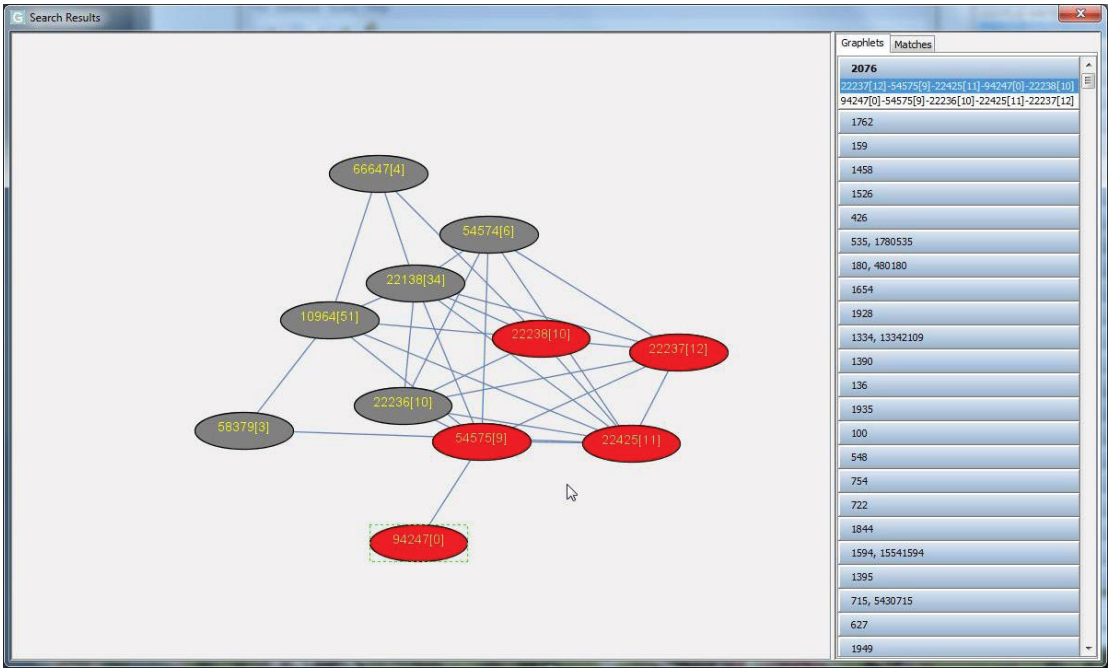

Figure 6.7: Drill-down "localized" view.

co-author, where the papers were published to `VAST` and another conference. In exemplar-based exploration, a user may start only with the structure of this graph query (Figure 6.9a) and then incrementally add value constraints on nodes to select specific results from the query result set. For instance, she may chose the other conference to be `KDD` (Figure 6.9b) and select the author to be `Shixia Liu` (Figure 6.9c), to find related results.

6.2.3 FEATURE-BASED EXPLORATION

The exemplar-based exploration takes a bottom-up approach where one starts with a specific result or topology. *Feature-based exploration* schemes, on the other hand, takes a top-down approach by generating a high-level overview of all the query results. Specifically, it groups the query results based on the structural features and embed them in a low-dimensional representation. VIGOR [Pienta et al., 2018] also supports such feature-based exploration (Figure 6.8C). Note that the number of subgraphs can be large in the query results and they typically share nodes and edges. Furthermore, nodes and edges in different subgraphs may have often have very different features. Specifically, given a set of k results, VIGOR first extracts individual *node features* and some *topological features* (e.g., degree, egonet features, clustering coefficient) from the neighborhood around each result in the underlying network. It also supports a *Feature Explorer*

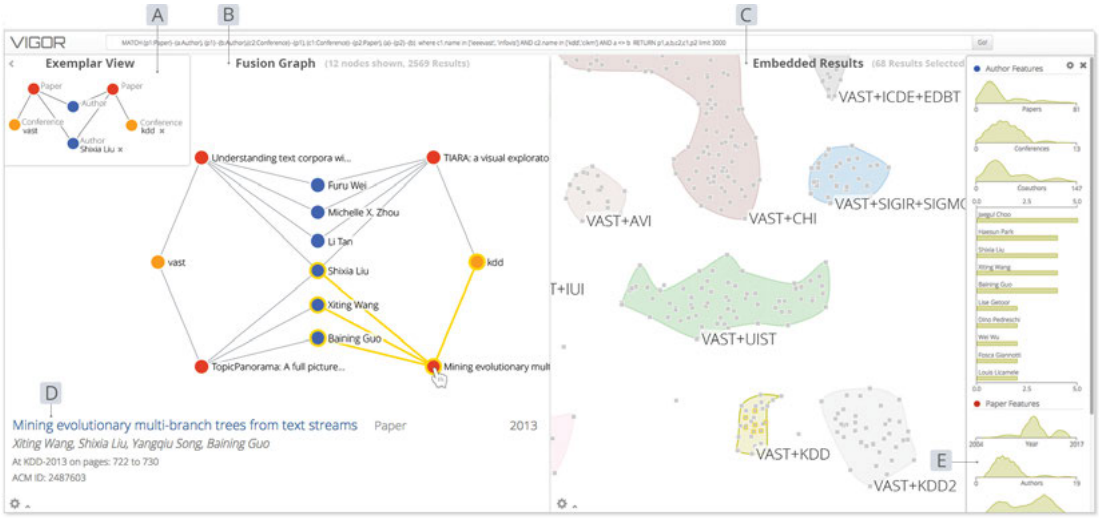

Figure 6.8: Visual interface of VIGOR [Pienta et al., 2018].

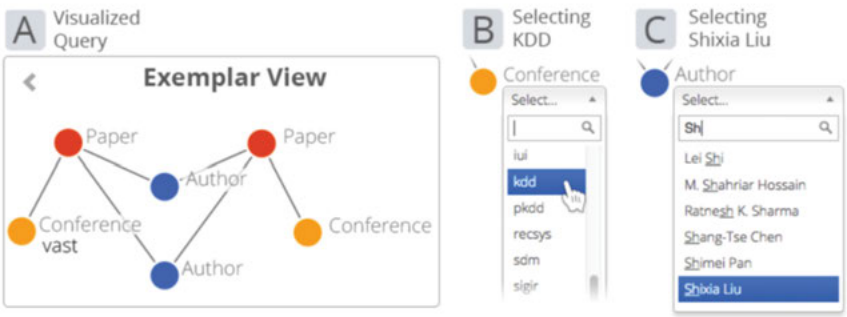

Figure 6.9: An example of exemplar-based exploration [Pienta et al., 2018].

that shows common node values and feature distributions for each node type (Figure 6.8E). These features are combined and the values are rearranged by feature. It then collapses the feature sub-vectors with different lengths into a uniform-length vectors by computing the moment of distribution functions (mean, variance, skewness, and kurtosis). Next, it performs dimensionality reduction using Principle Component Analysis (PCA) to reduce the dimensions to two. Lastly, these dimensionally reduced points are clustered using a density-based clustering technique and presented to the user. On the visual interface, each query result is represented as a square glyph. Clicking on a square allows for the selection of an exemplar result for bottom-up exploration.

6.3 BIBLIOGRAPHIC NOTES

The initial efforts to display and explore subgraph query results in list format appeared in Jin et al. [2010, 2012], and Pienta et al. [2016]. More advanced exploratory search on a collection of small- or medium-sized data graphs was recently demonstrated in Huang et al. [2017]. The research on interaction with query results from large networks primarily appeared in Hung et al. [2014] and Pienta et al. [2018]. Specifically, the region-based exploration was first introduced in Hung et al. [2014]. Exemplar-based and feature-based explorations are key features of the VIGOR framework described in Pienta et al. [2018].

6.4 CONCLUSIONS

Exploration and visualization of subgraph query results is a challenging problem that have received scant attention in the literature. A large number of contemporary graph querying systems simply showcase the results in form of a list, which is tedious to explore or comprehend. However, exploration of query results is only the beginning of the sensemaking process for an end user. In this chapter, we introduce several state of the art frameworks that are part of a seminal collection of work for facilitating exploration and visualization of subgraph query results. These efforts enable us to explore the results space in a top-down or bottom-up manner as well as analyze the various features associated with the contents of matching subgraphs in multi-faceted way.

CHAPTER 7

Simulation of Visual Subgraph Query Formulation

User studies are the *sine qua non* for evaluating the performance and effectiveness of the aforementioned visual graph querying techniques. For example, consider the visual subgraph querying paradigm introduced in Chapter 4. In contrast to the traditional query processing paradigm where the runtime performance of a large number of subgraph queries can be easily measured by automatically extracting a random collection of subgraphs from the underlying data and executing them [Katsarou et al., 2015], each visual query graph must be formulated by a set of users. This is because in this paradigm the availability of the GUI latency at each formulation step is exploited to prefetch and refine candidate matches. For example, a user with extensive experience with the GUI may move to the direction of the target item in Panel 2 (Figure 1.2) rapidly whereas a novice user may need to visually inspect each item in the panel to find the target item. Similarly, suppose a user selects a label `C` in Panel 2 which is identical to the one selected in the preceding step to create an edge `C-C` in Panel 3. Then the time taken to perform the latter task is typically lesser than the former due to retention of `C`'s location in the user's memory. Similarly, the GUI latency at each formulation step is also exploited to provide timely suggestions of new edges and nodes that can be added to a partially constructed query graph (Chapter 3).

Observe that the common thread in the aforementioned issues is the necessity to formulate visual query graphs by human users. Recall that each query in such a visual querying environment may follow many different query formulation sequences (QFSs) and the GUI latency at each step may vary across different users and different tasks. For instance, reconsider the query in Figure 1.2 which depicts one of many QFSs. Different users may formulate it by starting from different nodes and following different sequences of edge construction. For example, Figure 2.5 shows two QFSs for formulating this query graph. As the size of the query graph grows, the number of QFSs grows exponentially!

Nevertheless, comprehensive understanding of the performance behavior of aforementioned visual query formulation-based techniques demands "exhaustive" study of their performances from a variety of aspects such as (a) impact of different characteristics of query graphs (e.g., size, frequent or infrequent queries) formulated by users as well as (b) impact of different sequence of query formulation by novice and experienced users. Although subjective assessments, such as user studies, are useful for obtaining quick feedback on the performance of

a proposed technique, they are prohibitively expensive when used for such exhaustive performance evaluation. Specifically, it is expensive and time-consuming to engage a large number of users of different types (novice and experienced) to formulate many visual queries of different characteristics as well as different QFSs. This is especially true for small companies, start-ups, small academic research groups, and individual developers and researchers. In fact, our experience suggests that having to formulate many queries with different QFSs strongly deters users from participating in an empirical study. Furthermore, user studies are susceptible to a variety of evaluation conditions (e.g., cognitive load) that can impart variations in the outcomes of the study. Thus, we emphasize the need for objective and automated techniques to realistically simulate the subgraph query formulation process, which can support an exhaustive performance study of a variety of visual query formulation-based schemes. Such an automated technique can complement traditional subjective usability assessment methods.

> *Comprehensive user studies to exhaustively and systematically evaluate performance of visual graph querying techniques and applications are expensive and time consuming.*

In this chapter, we present a novel *synthetic visual subgraph query simulator* to address the aforementioned problem. Specifically, we present an algorithm called ViSual (**VI**sual **S**ubgraph Q**U**ery Formul**A**tion Simu**L**ator) [Bhowmick et al., 2017], which focuses on simulating subgraph query construction on a database containing a large number of small- or medium-sized graphs.[1] Using ViSual, one can automatically generate many *test* subgraph queries having different *user-specified characteristics* (e.g., frequent, infrequent) and simulate their formulation based on different QFSs —without requiring human users, thus addressing the aforementioned challenge of exhaustive performance study. In summary, this chapter makes the following contributions.

- In contrast to the popular strategy of randomly extracting subgraphs of specific sizes from the underlying graph repository as test queries, we present an *index-driven* approach to automatically generate different *types* of test queries satisfying *user-specified constraints* by leveraging underlying indexing schemes designed originally for processing subgraph queries.
- We present a *HCI-inspired quantitative model* for estimating the GUI latency in each formulation step to realistically simulate the visual query construction process without requiring human users. Specifically, it takes into consideration the time taken to undertake various actions on the GUI in order to construct a visual query fragment.

[1] Our approach can easily be extended to massive graphs as the process of subgraph query formulation is orthogonal to the nature of underlying data graphs.

- We propose a novel and generic algorithm, grounded on our HCI-inspired quantitative model, to automatically simulate visual query construction of different test subgraph queries with different QFSs.
- We conduct an extensive experimental study to demonstrate the effectiveness and accuracy of ViSual.

The rest of the chapter is organized as follows. We present an overview of the ViSual framework in the next section. We present the index-driven algorithm to automatically generate test queries in Section 7.2. Our HCI-inspired quantitative model to estimate the subgraph query formulation time is described in Section 7.3. We present the ViSual algorithm to simulate visual subgraph query formulation in Section 7.4. Experimental results are presented in Section 7.5. We review related research in Section 7.6. The last section concludes the chapter. The set of key symbols used in this chapter is given in Table 7.1.

Table 7.1: Key symbols used in this chapter

Symbol	Definition
T_m	Movement time
T_s	Selection time
T_d	Drag time
T_e	Edge construction time
$\mathbb{S} = \{S_1, S_2, ..., S_n\}$	A set of QFSs
$M = \{V_M, E_M, \tau\}$	QFS graph
p	Target distance in the *Label Panel*
$a, b, \eta, m_1, m_2, n_1, n_2$	Parameters of the query formulation time model

7.1 OVERVIEW OF ViSual

The overall framework of ViSual is depicted in Figure 7.1. It consists of three key components, namely, an *index-based technique* to generate test subgraph queries having different user-specified characteristics (C1), an HCI-inspired *quantitative model* to estimate the time for subgraph query formulations (C2), and an algorithm to simulate visual query formulation (C3) of the generated test queries by leveraging the quantitative model. Notably, the *quantitative model* leverages principles from HCI on visual task completion to estimate the time required to construct an edge in a query graph (i.e., GUI latency) *without* requiring human users. For instance, assume that the query in Figure 1.2 is a test query. Then our proposed framework can accurately simulate its formulation on the GUI by following different QFSs. Indeed, a key goal of our work is to

advance the role of quantitative models as a method for offering more *systematic*, *rigorous* and *general* explanations of visual query formulation. It is *rigorous*, because our model simulates the query construction process and thereby reduces the potential for ambiguity and miscalculation. It is *systematic* as otherwise disparate concepts (HCI-based visual task modeling and subgraph query formulation) are brought together in a single framework. Lastly, it is *general*, because it enables us to provide a single explanation regarding the formulation of a variety of subgraph queries on different visual query interfaces. Furthermore, our proposed model is *extensible* as it can be extended to support more complex graph queries, different visual query interfaces, as well as query formulation errors.

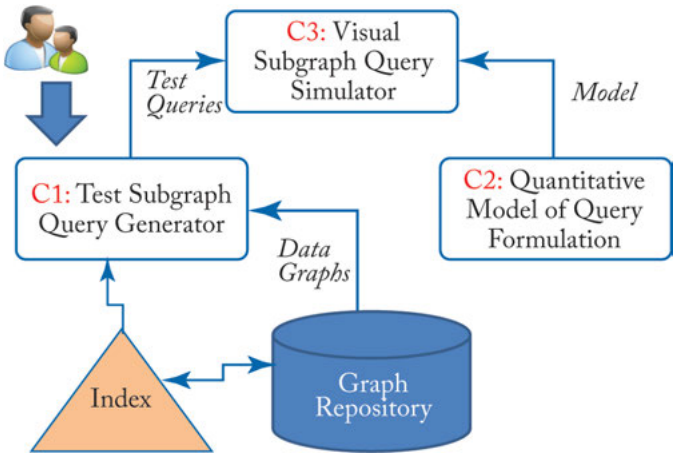

Figure 7.1: Framework of ViSual.

Our quantitative model paves the way for the ViSual algorithm to automatically simulate the visual subgraph query formulation process without the need for a user study. As remarked earlier, this is fundamental for automating the evaluation of the performance of visual query formulation-based techniques such as incremental query processing and query suggestions. We demonstrate the effectiveness of ViSual by empirically investigating it using real-world datasets and human users.

7.2 INDEX-BASED GENERATION OF SUBGRAPH QUERIES

In this section, we discuss how to generate different *types* of test queries in ViSual by utilizing the action-aware indexes. Intuitively, a user may generate three types of subgraph queries in ViSual, namely, *frequent*, *DIF*, and *NIF* queries. A frequent subgraph query is a frequent fragment in $\mathcal{D}$ whereas DIF and NIF queries are infrequent fragments (recall from Chapter 2). Specifically, a DIF and a NIF query is a DIF and a NIF in $\mathcal{D}$, respectively. A user may specify as input the

distribution (proportion of the total number of queries) of these three types of queries that she wishes to generate for simulation.

Given the query size s, the total number of size-s query graphs $\mathcal{N}$ that needs to be generated, and the distributions of query types, our algorithm aims to generate the appropriate number of frequent and infrequent test subgraph queries. A key feature of our generation scheme is that it leverages the action-aware indexes designed originally for query processing (described in Chapter 4) to generate these queries instead of randomly extracting them from $\mathcal{D}$. An advantage of utilizing such indexes is that we do not need to create yet another specialized index for this task. Furthermore, as we shall see in Section 7.5, such We now elaborate on the generation processes of these three types of queries in turn.

Index-based strategy for test query generation have superior quality and generation time compared to traditional random sampling-based approach.

Algorithm for frequent query generation. Algorithm 7.22 outlines the procedure for generating frequent queries of size s by leveraging the A^2F-index. If the specified size s is not larger than the maximum size of a frequent subgraph in $\mathcal{D}$ (denoted by $f_{\max}$) then the A^2F-index is traversed to the level containing frequent subgraphs of size $s-1$ and counts the total number of children (denoted by c) the vertices in level $s-1$ (Lines 3–4) have. If c is greater than the desired number of frequent queries n_ω, then for each vertex in level $s-1$ it randomly selects the CAM codes of an appropriate number of child vertices (Lines 6–8). Otherwise, as the desired number of frequent queries is larger than the number of frequent subgraphs in level s, the CAM codes of all these subgraphs are selected (Line 11). These codes are then transformed to frequent subgraph queries (Line 13). Observe that instead of simply choosing randomly from all vertices at level s (Lines 6–8), the algorithm "distributes" the selection across the children of a maximal number of vertices at level $s-1$ based on n_ω and s in order to inject greater degree of structural variability to the test queries. The worst-case time complexity of Algorithm 7.22 is $O(n_\omega \times |cam(Q_F)|)$ where $|cam(Q_F)|$ is the length of the CAM code Q_F.

Algorithm for DIF query generation. Algorithm 7.23 outlines the procedure for generating size-s DIF queries by leveraging the A^2I-index. Similar to the above procedure, if s is not larger than the maximum size of a DIF fragment in $\mathcal{D}$ (denoted by $d_{\max}$) then the A^2I-index is scanned to randomly select n_δ CAM codes of size-s DIFs, which are then transformed to DIF queries. Note that if n_δ is equal to or larger than the total number of size-s DIFs in the index then all of them are selected. The worst-case time complexity of Algorithm 7.23 is $O(n_\delta \times |cam(Q_D)|)$.

Algorithm for NIF query generation. Lastly, the generation of NIF queries in Algorithm 7.24 is relatively more involved as this type of infrequent subgraphs are not indexed by the infrequent index. However, recall that a NIF contains at least one DIF. Hence, the A^2I-index is scanned

Algorithm 7.22 GenerateFreqQueries

Input : Query size s, number of queries n_ω
Output : Set of size-s frequent queries Q_F

```
1:  Read A²F-index
2:  if s ≤ |f_max| then
3:     Traverse to level s − 1 of A²F-index
4:     c ← CountChildren(s − 1)
5:     if c > n_ω then
6:        for v in level s − 1 do
7:           Q_{F_v} ← SelectChildren(s, v)
8:           Q_F ← Q_F ∪ Q_{F_v}
9:        end for
10:    else
11:       Q_F ← Select CAM codes of all nodes at level s
12:    end if
13:    Q_F.CamToGraph()
14: end if
```

first to iteratively choose the CAM codes of DIFs having size less than s (Line 9). For each DIF c, it retrieves the $fsgIds(c)$ of data graphs in $\mathcal{D}$ whose sizes are at least s (Lines 11–12). If the set of $fsgId$ $|F|$ is larger than n_η, an $fsgId$, f, is randomly selected from F (Line 17). This $fsgId$ f is then used to retrieve the corresponding data graph $g_f \in \mathcal{D}$. We expand the subgraph c to g_s by traversing breadth-first and iteratively adding a new edge to c, terminating when $|g_s| = s$ (Lines 18–19).[2] Since g_s is a size-s NIF, we add it in Q_N (Line 20). The above steps are repeated iteratively until the specified number of NIF queries are generated. In the case when $|F| \leq n_\eta$, we iterate through the entire set of F to retrieve the corresponding data graph (Line 24), expand the subgraph and add the subgraph to Q_N (Lines 25–26). The complexity of Algorithm 7.24 is $O(k \times |DIF_k| + n_\eta \times s)$ where DIF_k is the number of DIFs having size k.

Example 7.1 Reconsider the examples of action-aware indexes in Section 4.3. Suppose we wish to generate two size-3 frequent queries on a chemical compound database. Then, Algorithm 7.22 is invoked and the MF-index in Figure 4.3 is traversed to level 2. It then counts the number of children of the vertices in this level (i.e., 2). Since the condition in Line 5 is not satisfied, it selects all the vertices in level 3 (Line 10) and converts the corresponding CAM codes to query graphs. Similarly, suppose we wish to generate five size-3 NIF queries. Then Algorithm 7.24

[2]Note that it is possible to expand the subgraph in a depth-first manner.

Algorithm 7.23 GenerateDIFQueries

Input : Query size s, number of queries n_δ
Output : Set of size-s DIF queries Q_D

```
1:  Read A²I-index
2:  if s ≤ |d_max| then
3:      p_s ← Retrieve start index of size-s DIFs in A²I-index
4:      p_e ← Retrieve end index of size-s DIFs in A²I-index
5:      if n_δ < (p_e − p_s) + 1 then
6:          Q_D ← Randomly select n_δ CAM codes of size s-DIFs
7:      else
8:          Q_D ← Select CAM codes of all size s-DIFs
9:      end if
10:     Q_D.CamToGraph()
11: end if
```

is invoked. Here $k = 3 - 1 = 2$ (Line 4). Hence, the A^2I-index in Figure 4.3 is traversed to iteratively retrieve DIFs up to size 2 and the corresponding *fsgIds* are retrieved (Lines 9–12) to find relevant data graphs which are then traversed to extract size-3 NIF (Lines 17–26).

Remark 7.2 We emphasize that the above test subgraph query generation framework is not tightly coupled to the action-aware indexes. For instance, we can easily replace it by a framework based on any existing frequent fragment-based indexes and augmenting it by the A^2I-index (to generate DIF and NIF queries). In the case, no indexes are available then the action-aware indexes can be generated from the data graphs. On the other hand, if the underlying graph database utilizes a path-based or tree-based index instead of a frequent graph-based index then the subgraph query generation framework can be seemlessly replaced by one that generates various types of test queries using these existing indexes without impacting the ViSual framework. This is because, as we shall see later, our HCI-inspired quantitative model and visual subgraph simulation technique (`C2` and `C3` in Figure 7.1) are not influenced by any specific test query generation scheme.

7.3 QUANTITATIVE MODELING OF VISUAL QUERY FORMULATION

In this section, we present an HCI-inspired, extensible, quantitative model of visual subgraph query formulation. To facilitate our exposition, we utilize the generic structure of a visual graph

Algorithm 7.24 GenerateNIFQueries

Input : Query size s, number of queries n_η
Output : Set of size-s NIF queries Q_N

```
1:  Read A²I-index 𝕀
2:  Q_N = ∅
3:  if s ≤ |d_max| then
4:      k ← s − 1
5:  else
6:      k ← s
7:  end if
8:  for j = k to j = 1 do
9:      C ← RetrieveDIFs(j, 𝕀)
10:     for all c ∈ C do
11:         F_c ← getFSGIds(c, s)
12:         F ← F ∪ F_c
13:     end for
14: end for
15: if |F| > n_η then
16:     while |Q_N| ≠ n_η do
17:         f ← SelectFSGIds(F)
18:         Retrieve graph g_f from 𝒟
19:         g_s ← Extract(c, s)
20:         Q_N.Add(g_s)
21:     end while
22: else
23:     for all f ∈ F do
24:         Retrieve graph g_f from 𝒟
25:         g_s ← Extract(c, s)
26:         Q_N.Add(g_s)
27:     end for
28: end if
```

query interface and the visual query formulation process introduced in Chapter 2. In the subsequent sections, we shall present algorithms that leverage on our proposed model to realistically simulate visual subgraph query formulation.

7.3.1 MODELING QUERY FORMULATION TIME

It is paramount to simulate the time taken by the query formulation steps accurately during visual subgraph query formulation in order to enable automated performance evaluation of a variety of visual query formulation-based techniques that we have introduced earlier. However, the key challenge to simulate the aforementioned steps is that we cannot assume all users take same amount of time to complete each step. For instance, time taken to move the mouse to the *Label Panel* or to select a label may differ for different users and for different organizations of the label items. Hence, instead of assuming any arbitrary time to finish each of the aforementioned steps, we resort to a systematic and rigorous approach that is drawn upon the literature in HCI to quantitatively model the time available to perform each step.

For clarity, we reiterate the steps introduced in Chapter 2 for visual graph query formulation.

1. Move the mouse cursor to the *Label Panel*.
2. Scan and select a label.
3. Drag the selected item to the *Query Panel* and drop it. Each such action represents formulation of a query node (denoted by u) with the specified label.
4. Repeat Steps 1–3 for constructing another node v.
5. Construct an edge between u and v by clicking on them, respectively.
6. Repeat Steps 4 and 5 until the complete query is formulated.

Observe that Steps 1–5 contribute to the GUI latency during visual query formulation. Let us refer to the times taken to complete Steps 1, 2, 3, and 5 as *movement time* (denoted by T_m), *selection time* (T_s), *drag time* (T_d), and *edge construction time* (T_e), respectively. We now discuss how to quantify them.

> *The latency of user interaction with the visual graph query interface during query formulation can be quantitatively modeled by leveraging task modeling techniques in HCI.*

Estimating movement time T_m. Reconsider Step 1. It involves acquisition of a target in the *Label Panel* at a distance D from the mouse cursor which is in the *Query Panel*. Note that typically the *Label Panel* is a rectangular two-dimensional target. Consequently, the item selection is constrained by the width and height of the panel and the cursor must travel along a two-dimensional vector to it. Hence, we adopt the model in Accot and Zhai [2003] that focuses on acquiring targets having rectangular, square, or circular shapes. The movement time T_m is

quantified as follows:

$$T_m = a + b \log_2 \left(\sqrt{\left(\frac{D}{W}\right)^2 + \eta \left(\frac{D}{H}\right)^2} + 1 \right), \tag{7.1}$$

where D is distance of the desired item (in the *Label Panel*) to the cursor, H and W denote the *Label Panel*'s height and width, respectively. The parameters a, b, and η are empirically determined constants. Note that η allows the model to weight the effect of the height differently from the effect of the width. Observe that D depends on the relative position of the desired item within the "visible" window of the *Label Panel*. If the desired item is currently visible on the *Label Panel*, the user would move the mouse cursor to that item. Otherwise, the user has to move the mouse cursor to the *Label Panel* with the intention to navigate the panel by scrolling in order to locate the desired item. Note that navigation can start at any position within the *Label Panel*.

Let p_c be the position of the cursor in the *Label Panel*. Then, we employ the following two rules to compute p_c.

1. If the desired item is currently visible on the *Label Panel*, p_c is set to the position of this item within the *Label Panel*. For instance, p_c=2 if the desired item is the second item visible on the *Label Panel*.
2. Otherwise, the relative position is any random position in the *Label Panel*. That is, $0 < p_c \leq x$ where x is the number of visible items on the *Label Panel*.

For example, consider Figure 7.2 with a visible window of six items and having a list containing nine items. Suppose a user wishes to select `Cu`. Since it is visible in the *Label Panel*, $p_c = 3$. On the other hand, if she wishes to select `P` then $0 < p_c < 6$ as it is not visible in the *Label Panel*. Observe that the value of p_c enables us to compute D accurately.

Estimating selection time T_s. The *Label Panel* can be considered as an item list and one needs to select a target item (label) during query formulation by navigating the cursor through it using vertical movements, possibly with scrolling, to select the desired label. We assume that the labels are organized vertically and hence ignore horizontal movement in this panel as the horizontal width is negligible.

The user's general task is to scroll the list, if necessary, to bring the desired item into view, and then select it with a pointing device. Hence, the following steps occur from the user's perspective.

1. Determine where the desired item is likely to be, relative to the current location of the cursor in the list.
2. Determine what navigation action to take next and execute it.
3. If the desired item is still not visible then go back to Step 1.
4. If visible, move the cursor to the item and select it.

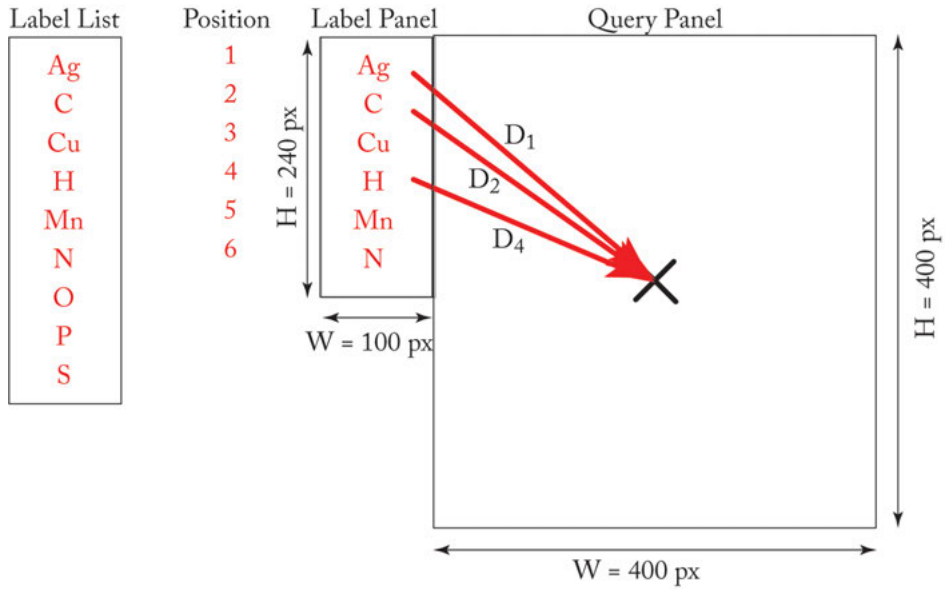

Figure 7.2: Position of labels and its influence on D.

Observe that the first step has a large impact on the number of times that the first two steps must be repeated. This is because Step 1 involves comparing the target item with those in the display, and determining the appropriate direction and distance of movement. Clearly, for a user to successfully determine the direction of movement, she must have a *mental model* of the contents of the *Label Panel*. Although different users may have different models, essentially it depends on how well a user knows the data.

- *Scenario 1:* If she has extensive knowledge of the data, either through prior experience or because the items are organized in a standard fashion (e.g., lexicographically ordered), then she will be able to move to the direction of the target item rapidly.
- *Scenario 2:* If a user is unfamiliar with the items (e.g., the items are random, the organization strategy of the *Label Panel* is not yet known), then she has to leverage on visual search and inspect each item in the panel to find the desired item.

For example, when searching an alphabetically sorted *Label Panel* for `N`, a user can rapidly scroll through many items before zooming in to list contents near this item. On the other hand, if she is unfamiliar with the *Label Panel* then she has to visually search the list by comparing `N` with each item in the panel.

We adopt the following quantitative model proposed by Cockburn and Gutwin [2009] for modeling selection time of an item, which realizes the aforementioned intuition. Specifically, it quantitatively describes a list selection performance that follows a *linear model* when the user's

task is dominated by visual inspection (Scenario 2), and a *logarithmic model* that arises from predictable list items (Scenario 1).

The *logarithmic* model is given in Equation (7.2):

$$T_s = m_1 + n_1 \times (\log_2(p+1)) \,. \tag{7.2}$$

In this equation, p is the *target distance* of the target label, and m_1 and n_1 are empirically determined constants through regression analysis for a particular type of scrolling task. Note that the above logarithmic model enables us to model the relationship between movement time and target distance as human pointing to visual targets allow a large proportion of the distance toward the target to be rapidly completed without attending to sensory feedback.[3] In our model, we compute p by using the node selected in the preceding step as reference. For example, assume that the target node is labeled with `Cu` in a list that is alphabetically sorted. Let `C` be the node label selected by the preceding step. Then p is computed as the total number of nodes below or above (e.g., 2) the node selected in the preceding step (i.e., `C`).

The *linear* model, on the other hand, is given by the following equation where the acquisition time increases linearly with distance. Recall that in this model a user must compare each item with the intended target[4]:

$$T_s = m_2 + n_2 \times p. \tag{7.3}$$

In this equation, m_2 and n_2 are empirically determined constants.

Lemma 7.3 *Let a graph database $\mathcal{D}$ contains K distinct labels. Then for a visual query Q on $\mathcal{D}$, the selection time T_s of each label for visually constructing a query node in Q is bounded as follows: $m \leq T_s \leq m_2 + n_2 \times (K-1)$ where $m \in \{m_1, m_2\}$.*

Proof. The proof follows from the definition of T_s in Equations (7.2) and (7.3). The minimum value of T_s is $m = m_i$ when $p = 0$ where $i \in \{m_1, m_2\}$. This happens when we select a label which is identical to the one selected in the preceding step (e.g., consecutive selection of the `C` label to create an edge `C-C`). On the other hand, T_s is maximum in the linear model when the last (resp. first) node in Panel 2 is selected for the first time after the selection of the first (resp. last) node in the preceding step. In this case, $p = K - 1$. Hence, from Equation (7.3), $T_s = m_2 + n_2 \times (K-1)$. □

Estimating drag time T_d. The *drag time* can be modeled using Equation (7.1) as the cursor is now moving from the *Label Panel* to the *Query Panel*. Specifically, in this case D is the *Query Panel*'s distance to the cursor (we assume that a label is dropped in mid region of the panel), H

[3] In control-theory and in human physiology research this process is described by the term "open-loop" or "ballistic action". In our framework, an example of a ballistic action includes searching for an item in the *Label Panel* that is sorted (alphabetically, numerically, etc.) and locating an item that resides at a known location within the list.

[4] This process is referred to as "closed-loop" in the literature.

and W denote *Query Panel*'s height and width, respectively. Similar to the movement time, D is affected by the relative position of the desired item in the visible window of the *Label Panel*.[5] Hence, we utilize the following rules to determine the relative position rp_i of a desired item i in the *Label Panel*.

1. If the desired item is currently visible on the *Label Panel*, rp_i is set to the position of the desired item in the *Label Panel*.
2. Otherwise, the relative position is any random position in the *Label Panel* and is computed as follows. Let x be the exact position of i in the item list, y is the number of visible items, z is the number of items in the item list, and $a = x - (z - y)$. Then,

$$\begin{cases} 1 \leq rp_i \leq x & \text{if } x \leq y \\ a \leq rp_i \leq y & \text{if } x > z - y \\ 1 \leq rp_i \leq y & \text{otherwise} \end{cases}. \tag{7.4}$$

For example, reconsider the *Label Panel* in Figure 7.2. Then, $1 \leq rp_{\texttt{Cu}} \leq 3$ and $5 \leq rp_{\texttt{P}} \leq 6$ for dragging `Cu` and `P`, respectively.

Estimating edge construction time T_e. Lastly, T_e models Step 5 of the query formulation process comprising of (a) clicking on the node v (assuming the last node dropped on the panel is v); (b) moving the cursor from v to u; and (c) clicking on the node u. Let t_c be the time to click a node, n_c be the number of clicks on a node (in our case $n_c = 1$), and T_m be the movement time from v to u. Then,

$$T_e = 2n_c \times t_c + T_m. \tag{7.5}$$

We assume $t_c = 80$ ms [Card et al., 1980]. Note that T_m can be computed using Equation (7.1) where H and W now denote the height and width of a node (constant for all nodes in a given GUI),[6] respectively, and D is the distance between u and v.

Note that thinking time for users during edge construction is negligible because when a user drags and drops a specific label ℓ from the *Label Panel* to the *Query Panel*, usually she has already mentally determined how this ℓ-labeled node will be connected with rest of the query graph.

Remark 7.4 A keen reader may observe that for a given query Q with a set of QFSs $\mathbb{S}$, the numbers of nodes and edges in $\mathbb{S}$ remain constant. Hence, the variability in query formulation

[5] A shorter drag time is needed if the desired item (in the visible window of the *Label Panel*) is located nearer to the drag destination (marked as "X" in Figure 7.2) in the *Query Panel*. Note that the positioning of the desired item in the visible window is actually dependent on the ordering of items in the entire item list. For example, if the *Label Panel* can contain only six visible items, an item `C` that is ordered as the second item in the entire item list can only have relative positions as either the first or the second item in the visible window.

[6] Accot and Zhai [2003] can model square-, rectangular-, or circular-shaped query nodes.

time is due to T_s, T_d, and T_m, that may vary for different QFSs as they are dependent on the order in which nodes are constructed. To elaborate further, for a given QFS $S_i \in \mathbb{S}$, suppose the current and next nodes to be constructed are u and v, respectively. Assume that u and v are located far apart in the label list such that when u is visible in the *Label Panel*, v is not and vice versa (e.g., `C` and `P` in Figure 7.2). Then, after dragging u to the *Query Panel*, we need to move the cursor back to the *Label Panel*. However, since the next node v (e.g., `P` in Figure 7.2) is not currently visible in the *Label Panel*, we need to move the cursor to an arbitrary position in the *Label Panel* and navigate it using the scrollbar to locate v. The movement time T_m is then the time taken to move the cursor from the *Query Panel* to an arbitrary position in the *Label Panel*. The selection time T_s is the time taken to navigate the label list to locate v. Suppose after scrolling, v is found at the i-th position of the *Label Panel*, then T_d is the time taken to drag v (from its i-th position) to the *Query Panel*.

Now consider another scenario where u and v are located near each other and both are visible on the *Label Panel* (e.g., `C` and `Cu`). Then, after constructing u, T_m is the time taken to move the cursor from the *Query Panel* to v on the *Label Panel*. Since v is close to u, the selection time T_s is shorter than the previous scenario.

7.3.2 MODEL EXTENSIBILITY

Supporting richer visual subgraph queries. An obvious question is whether the aforementioned model can be easily extended to a richer variety of subgraph queries? Typically, in complex subgraph queries one may add additional predicates on nodes and edges (e.g., ontology-based queries [Wu et al., 2013], bounds on distance between a pair of nodes [Lin et al., 2012]). Our model does support richer visual subgraph queries. Observe that T_m, T_s, and T_d in our model represent the Steps 1, 2, and 3, respectively, during query formulation. Specifically, these steps are used to construct a node in the *Query Panel*. The visual construction of additional node predicates will often take place after Step 3 (after the creation of a node in the GUI) and typically consists of keyboard entries (for predicate specification) and mouse clicks. Similarly, any additional predicates on edges will be typically specified after Step 5 and will also consist of keyboard entries and mouse clicks. Hence, these steps can easily be modeled by a variant of Equation (7.5).

Supporting canned patterns for visual query formulation. Recall that in this chapter we assume a visual query interface where queries are formulated only in edge-at-a-time mode. The availability of canned patterns enables users to quickly construct larger query graphs with fewer clicks. Such patterns are typically laid out in a separate panel in the GUI in several recent visual graph querying systems, cf. Yang et al. [2014]. Hence, it is important for our model to be extensible to handle such interfaces.

Obviously, an additional visual task that needs to be modeled in this context is the selection of a canned pattern during query construction and then dragging and dropping it into the *Query Panel*. Hence, the time cost of completing such a visual task comprises movement time to access

a specific pattern and drag time to drop it in the *Query Panel*. Although the drag time can be modeled using our aforementioned model, unfortunately, the computation of the movement time is challenging as it depends on the layout of the patterns. Specifically, if each pattern is enclosed in a circular, rectangular, or square-shaped object then Equation (7.1) can be adopted to model the movement time cost. However, if the canned patterns are of *arbitrary* shape then the model proposed by Grossman et al. [2007], which quantifies the movement time for an arbitrary shape, needs to be utilized. Specifically, it is given by extending Fitts' Law as follows:

$$T_m = a + b \times F\left(\int_R \int bndf(X', Y'), dY'dX'\right), \tag{7.6}$$

where R is the target's region, X' is the error parallel to the line of movement and Y' is the error perpendicular to the line of movement. The function *bndf* denotes bivariate normal density function and is given as:

$$bndf(X', Y') = \frac{1}{cA\sqrt{2\pi}} e^{-\left(\frac{X'^2}{2(cA^2)}\right)} \frac{1}{dA\sqrt{2\pi}} e^{-\left(\frac{Y'^2}{2(dA^2)}\right)},$$

where c and d are empirically determined constants. Observe that the movement time is quantified using a probabilistic model which maps the probability of hitting a target, $P(hit)$, with an "open-loop" movement. Importantly, unlike Equation (7.1) it does not require the target to have a well defined height and width.

Supporting different visual query interface designs. Observe that a visual graph query interface can be designed in different ways and consequently a user may formulate a visual query differently. We advocate that our proposed query formulation model provides adequate flexibility to handle a variety of query interfaces. For instance, consider a visual query interface where one can select a node in the *Label Panel* and click on it to add it to the *Query Panel* instead of explicitly dragging it. In this case, we do not need to consider T_d. Instead, we can add the *click time* to T_s. Specifically, the click time can be modeled using t_c in Equation (7.5). Similarly, if the *Label Panel* pops out automatically when a mouse button is clicked on the *Query Panel*, then in this interface the movement time to the *Label Panel* can be ignored. Instead, the model will include t_c and the selection time T_s for node construction.

Supporting query formulation errors. Lastly, our model can also be extended to incorporate query formulation errors. For example, suppose we intend to delete a node that has been constructed earlier. This may involve the following actions: (a) move the cursor to the specified node; (b) right-click on the node to invoke a list of actions; and (c) select the "delete" action from the menu item to delete the node. Observe that these three actions can be modeled using T_m, t_c, and T_s, respectively. Similarly, if we intend to delete a previously constructed edge we can achieve it by double clicking on the edge or adjacent nodes, which can be modeled utilizing t_c and T_m (similar to Equation (7.5)).

7.4 SIMULATION OF VISUAL SUBGRAPH QUERY CONSTRUCTION

We now have all the machinery in place to facilitate simulation of visual subgraph query construction. In this section, we present the `ViSual` algorithm toward this goal, which leverages our proposed HCI-inspired query formulation model to realize realistic simulation of the subgraph query construction activity. We begin by describing the graph representation of the query formulation process.

7.4.1 GRAPH REPRESENTATION OF QUERY FORMULATION

Recall that a query Q is associated with a set of QFSs (denoted by $\mathbb{S}$). The process of generating all possible QFSs can be modeled as a formal language where the words in the language represent the QFS. The language generated by the grammar is a form of regular language as it follows the type-3 grammar rules in Chomsky Hierarchy. Hence, $\mathbb{S}$ can be represented as a state machine.

Formally, we model $\mathbb{S}$ of a query Q as a directed *QFS graph* $M = (V_M, E_M, \tau)$ by representing it as a state machine. Each node $n \in V_M$ represents a state and is denoted by 2-tuple $n = (label(n), k_n)$ where $label(n)$ is the label of n and k_n is the position of n in the *Label Panel*. For instance, a node `(C,1)` has label `C` and occupies the first position in the *Label Panel* (e.g., Figure 1.2). Each edge $(u, v) \in E_M$ is labeled with the *state transition time* T from node u to node v where $T = (T_m + T_s + T_d + T_e)$. Specifically, each edge is associated with two state transition times, namely *minimum transition time* and *maximum transition time*, corresponding to the node selection (T_s) based on the logarithmic (Equation (7.2)) and linear (Equation (7.3)) models, respectively. M also contains a special node τ where $label(\tau) =$ `"Start"` and $k_\tau = \emptyset$, which models the time taken to construct the first node in a QFS S by following an edge from τ to the first node. Figure 7.4 depicts the QFS graph of the query in Figure 7.3 where the alphabet set is {`O`,`C`,`H`} and the language is specified by the grammar derived from all possible unique sequences of the query.

Observe that formulation of a query $Q = (V, E)$ by following a specific QFS $S = [v_1, v_2, \ldots, v_{|V|}]$ is equivalent to traversing a path $\beta = (\tau.n_1.n_2 \ldots n_{|V|})$ in M where $label(n_i) = v_i$ for $1 \leq i \leq |V|$. Hence, the *query formulation time* (QFT) for S can be computed as aggregation of the minimum or maximum state transition times of all edges in β. For example, the path consisting of bold edges in Figure 7.4 represents the first QFS. The minimum and maximum QFT are 2413.1 msec and 4009.1 msec, respectively.

7.4.2 THE ViSual ALGORITHM

Algorithm 7.25 outlines the `ViSual` algorithm. Given a set of test queries $\mathbb{Q}$ and whether the selection time of items in the *Label Panel* should follow a logarithmic or linear model (specified by the variable $LabelPanelModel$), it first initializes a parameter set $\mathcal{W}$ associated with our proposed query formulation model. Some parameters in $\mathcal{W}$ are automatically computed based

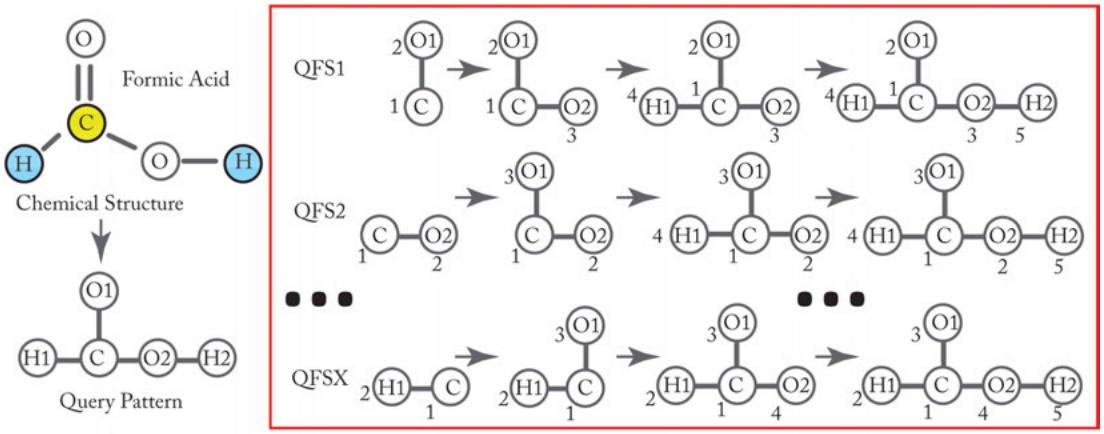

Figure 7.3: A subgraph query and its query formulation sequences (QFSs). The numbering next to each node is the order that the node is added to the query.

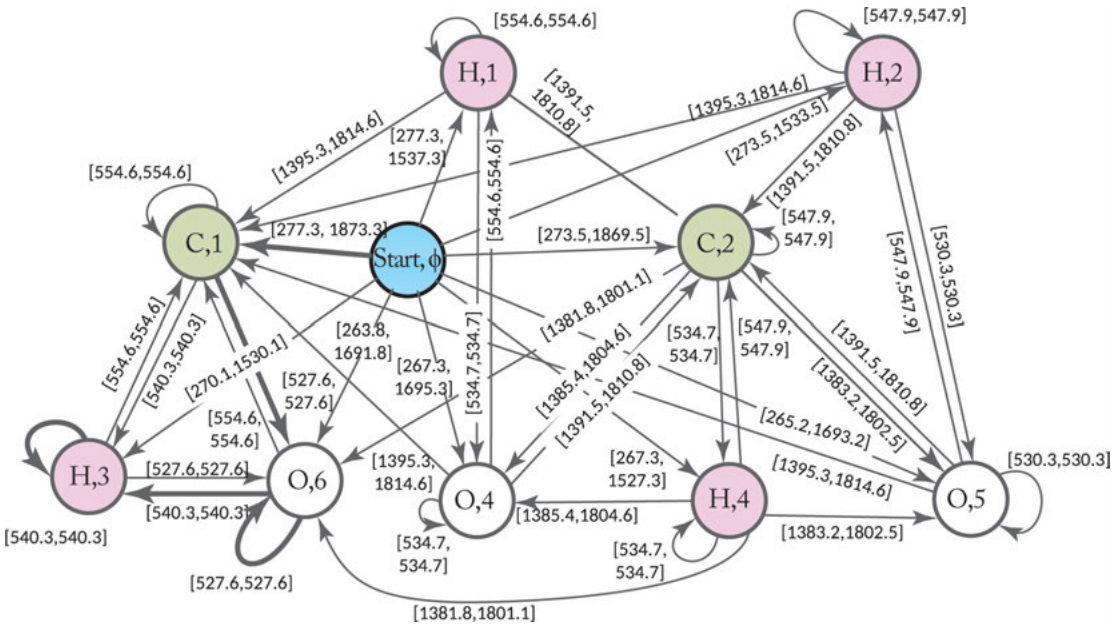

Figure 7.4: QFS graph of the query in Figure 7.3. Each edge is labeled with (minimum transition time, maximum transition time).

on the GUI structure (e.g., heights and widths of *Label* and *Query* Panels, and query nodes). Other parameters such as a, b are empirically determined as we shall see in Section 7.5. Next, it computes the number of visible labels n in the *Label Panel* (Line 2), obtains the list of items l in it (Line 3) and the query formulation model $\mathcal{F}$ to compute T_m, T_s, T_d, and T_e (Line 4) as described in the preceding section.

Algorithm 7.25 The `ViSual` Algorithm

Input : Set of test subgraph queries $\mathbb{Q}$, *LabelPanelModel*

1: $\mathcal{W} \leftarrow$ InitializeParam(*LabelPanelModel*)
2: $n \leftarrow$ getNumVisibleLabel($\mathcal{W}$)
3: $l \leftarrow$ getLabelList($\mathcal{W}$)
4: $\mathcal{F} \leftarrow$ getQFTModel($\mathcal{W}$)
5: **for all** $Q \in \mathbb{Q}$ **do**
6: $\quad M \leftarrow$ generateQFSGraph($Q, n, l, \mathcal{F}$)
7: $\quad \mathbb{S}_{unique} \leftarrow$ generateAllPermutation(Q)
8: $\quad \mathbb{S} \leftarrow$ checkQFS(QFS_{unique})
9: $\quad$ **for all** $S \in \mathbb{S}$ **do**
10: $\quad\quad t \leftarrow$ simulate($S, \mathcal{W}, M$) /**Algorithm 7.26**/
11: $\quad$ **end for**
12: **end for**

Then, it constructs the QFS graph M of Q by utilizing $\mathcal{F}$. Next, for each query $Q \in \mathbb{Q}$, it generates all query formulation sequences using a state-of-the-art permutation algorithm [Bassil, 2012] (Line 7). Then, for each permutation, the sequence is checked to ensure that a partially formulated query graph is not disconnected (Line 8). Permutations that result in such a sequence are discarded. Finally, for each "valid" QFS S, it invokes the *Simulate* algorithm (Line 10) as described below. The complexity of *generateQFSGraph* is $O(n^2|Q|^2)$ where $|Q|$ is the length of query pattern Q and n is the number of visible labels in the *Label Panel*.

The *Simulate* algorithm. Given a specific query formulation sequence S of Q, a set of parameters $\mathcal{W}$, and the QFS graph M, Algorithm 7.26 outlines the procedure to simulate the construction process of S. First, it retrieves the node label of the first node v_i in S and the position of its label in the *Label Panel* (Lines 1–3). Next, the algorithm computes the time t taken to construct v_i by utilizing the state transition time of the edge (τ, v_i) in M (Line 4). The algorithm waits for t time (Line 5) to simulate the latency incurred due to the construction of v_i. It marks v_i in Q to symbolize completion of its formulation and then moves to the formulation of the next node v_j in S (Line 7) and its edge. Similarly, the formulation time of v_j and its corresponding

edge is computed by utilizing the parameter set $\mathcal{W}$ and traversing the edge (v_i, v_j) in M. The algorithm waits for t_j time (Line 12) to simulate the latency incurred due to the construction of (v_i, v_j). Lastly, it marks the edge (v_i, v_j) after the simulation of its formulation is completed. The algorithm terminates when all nodes and edges in S have been marked as constructed.

Algorithm 7.26 The SIMULATE Algorithm

Input : Query formulation sequence S of query Q, parameter set $\mathcal{W}$, QFS graph M

1: v_i ←GETFIRSTNODE(S)
2: ℓ_i ←GETNODELABEL(v_i)
3: k_i ←GETNODELABELPOSITION(ℓ_i,M)
4: t ←COMPUTESTATETRANSITIONTIME($\mathcal{W}$, M, ℓ_i, k_i)
5: WAIT(t)
6: Mark v_i as constructed
7: v_j ←GETNEXTNODE(v_i, S)
8: **while** $v_j \neq$ null **do**
9: ℓ_j ←GETNODELABEL(v_j)
10: k_j ←GETNODELABELPOSITION(ℓ_j,M)
11: t_j ←COMPUTESTATETRANSITIONTIME($\mathcal{W}$, M, ℓ_j, k_j)
12: WAIT(t_j)
13: Mark (v_i, v_j) as constructed
14: $v_i \leftarrow v_j$
15: v_j ←GETNEXTNODE(v_i, S)
16: **end while**

Remark 7.5 The `ViSual` algorithm is not tightly coupled to any specific visual query application, and therefore can provide a consistent means for performance analysis across a variety of applications. Specifically, in Line 5 or 12 (Algorithm 7.26) a separate thread can be invoked to evaluate performances of algorithms designed for visual query suggestion, empty result feedback, or incremental visual query processing. An example in the context visual subgraph query processing is demonstrated in Bhowmick et al. [2015]. Furthermore, it is not tightly coupled to any specific test subgraph query generation framework as it can ingest any subgraph query for simulation. Consequently, the `ViSual` algorithm can also run on a different test query generator.

7.4.3 FINDING MINIMAL AND MAXIMAL QFS

Given a visual query, the aforementioned `ViSual` algorithm enables us to exhaustively simulate *all* possible QFSs. However, in some scenarios a user may simply wish to simulate a QFS with the largest or smallest formulation time in order to comprehend the upper and lower bounds of QFT. This will enable her to quickly evaluate the performance of her proposed visual querying application for the "best" and "worst" case scenarios. For instance, one may wish to quickly gauge the performance of the query suggestion schemes discussed in Chapter 3 under these two scenarios. To this end, we need to find such QFSs from the QFS graph efficiently.

Given a query Q with a set of QFSs $\mathbb{S}$, the *minimal QFS* $S_{\min} \in \mathbb{S}$ and *maximal QFS* $S_{\max} \in \mathbb{S}$ are the QFSs that consume the minimum and maximum query formulation times, respectively. Since the QFS graph M is based on modeling the query formulation sequences in form of a state machine, the problem of searching for $S_{\min}$ and $S_{\max}$ can be reduced to finding *L-constrained shortest* and *longest paths*, respectively, in M. Given a finite alphabet $\sum$, a directed graph $G = (V, E, c, l)$ with cost function $c : E \rightarrow \Re_+$ and label function $l : E \rightarrow \sum$, and a language $L \subseteq \sum^*$, the *L-constrained shortest path problem* finds a path $\rho = (e_1, e_2, \cdots, e_k)$ in G where $e_i \in E$ such that $c(\rho) < c(\rho')$ for all paths ρ' and $l(\rho), l(\rho') \in L$ where $l(\rho)$ is the concatenation $l(e_1) \cdot l(e_2) \cdots l(e_k)$ of the labels of ρ's edges. In particular, Barrett [2000] have developed a polynomial time algorithm to address the L-constrained shortest path problem for context-free and regular languages. Similarly, the *L-constrained longest path problem* aims to identify a path $\rho = (e_1, e_2, \cdots, e_k)$ in G such that $c(\rho) > c(\rho')$. Note that unlike the shortest path problem, the longest path problem is known to be NP-complete [Pardalos and Migdalas, 2004]. Fortunately, the longest path problem in a directed acyclic graph G can be solved in $O(|V| + |E|)$ time using a single breath-first search [Raje and Sarrafzadeh, 1995]. Note that since the edges of M are annotated with minimum and maximum transition times, computation of $S_{\min}$ and $S_{\max}$ only need to consider the minimum and maximum transition times, respectively. The formal algorithms for computing them and performance results are given in Bhowmick et al. [2017].

7.5 PERFORMANCE STUDY

`ViSual` is implemented in Java JDK 1.7.0. The experiments were conducted on a 64-bit Intel(R) Xeon(R) 3.60 GHz processor and 8 GB RAM. The operating system was Windows 7 Professional. A Logitech Mouse was used and its acceleration was set to its default settings (acceleration: 2/1).

7.5.1 PERFORMANCE OF TEST SUBGRAPH QUERY GENERATION

First, we investigate the test subgraph query generation time (Algorithms 7.22–7.24) by varying the number of frequent, DIFs and NIFs queries, and their sizes. We use the real-world AIDS dataset as summarized in Table 7.2 to this end. The average size of a graph is 25 nodes and 27 edges. The maximum size of a graph is 222 nodes and 251 edges. There are 63 different types

of atoms (labels). We also use the synthetic graph dataset generator of FG-Index [Cheng et al., 2007] (*Graphgen*) to generate four datasets with sizes from 20–80K. We set the number of distinct labels to 10. The average number of graph edges in each dataset is set to 30 and the average graph density is 0.1. We use the action-aware indexes and their default settings as described in Chapter 4 for these datasets. As a baseline, we use random query sampling. The steps for generating the random query samples follow that in Katsarou et al. [2015]. Observe that in contrast to our approach, random sampling-based strategy does not classify a query as frequent or infrequent. Hence, we are confined to select queries with specific size and number using this technique.

Table 7.2: Datasets

ID	Dataset	Size (K)	Support (α)
0	AIDS	10	0.1
1	AIDS	40	0.05
2	AIDS	40	0.1
3	Synthetic	20	0.05
4	Synthetic	40	0.05
5	Synthetic	60	0.05
6	Synthetic	80	0.05

The query sizes are varied from 3–11. Note that since we assume the "edge-at-a-time" visual query formulation strategy in this chapter, users typically do not construct large visual queries using this scheme as it can be tedious and time consuming. Since the number of frequent or DIF queries is typically fewer than NIF queries, we generate 10–50 frequent and DIF queries and 10–150 NIF queries. Note that the total numbers of frequent and DIF queries in these datasets decrease as the query size increases. Figure 7.5 plots the query generation times. We observe that in general, the query generation time using our index-based approach is faster (especially for frequent and DIF queries) compared to the baseline, especially for the AIDS dataset (Figure 7.5, IDs 0-3). Second, for frequent and DIF queries, the performance of our approach remains relatively constant regardless of the query size and number of queries as they can be directly generated using the indexes. On the other hand, the generation time for NIF queries increases with the number of queries or query size and is comparatively more expensive as underlying data graphs need to be traversed.

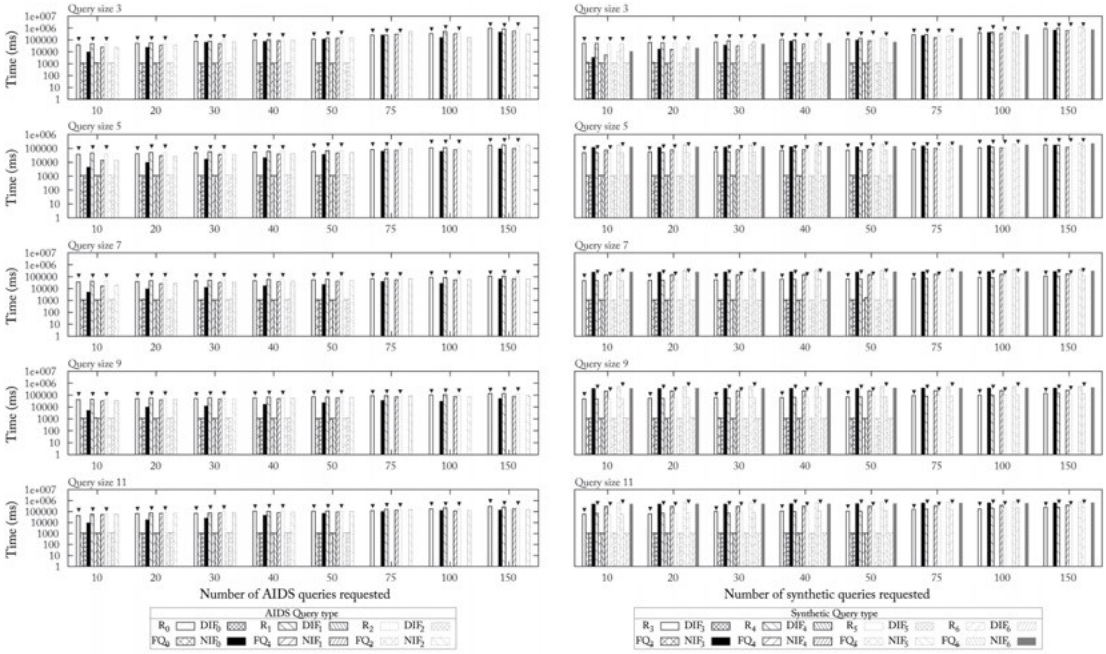

Figure 7.5: Query generation times for datasets. L_x and R_x denote L-type and random queries for dataset ID x in Table 7.2, respectively. Query generation times by random query sampling are marked with ▼.

7.5.2 PERFORMANCE OF THE QUERY FORMULATION MODEL AND ViSual

Next, we evaluate the accuracy of the query formulation model (Section 7.3) and the `ViSual` algorithm (Algorithm 7.25). We use the visual graph query interface described in Chapter 4 to formulate a set of benchmark queries shown in Figure 7.6. The queries are selected based on several features such as size (3–10), topology (path, tree, graph, and cycle), and label variety. We use two variants of this interface, one where the labels are sorted in lexicographic order and another where they are in random order, to simulate the logarithmic and linear models, respectively (Equations (7.2) and (7.3)). In subsequent experiments, we use the *Dataset* 2 (Table 7.2) as these experiments are on query formulation and do not depend on the dataset size.

Participants profile. Ten unpaid volunteers participate in query formulation to help us to empirically determine the various parameter values of our model and to validate the accuracy of our model. The participants were given training to familiarize themselves with the GUI. For every query, the participants were given some time to determine the steps that are needed to formulate the query visually. They were given one query at a time in random order. They were

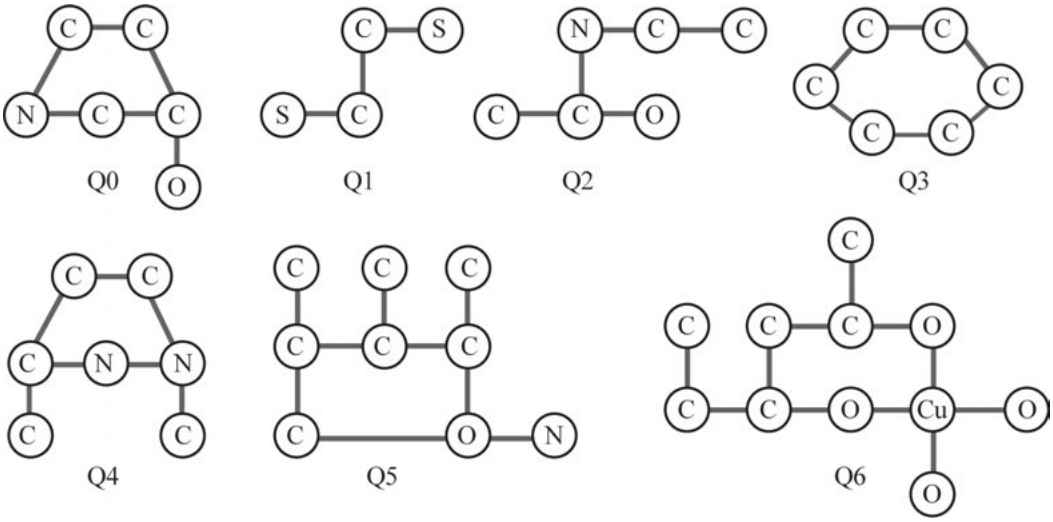

Figure 7.6: Queries performed by participants.

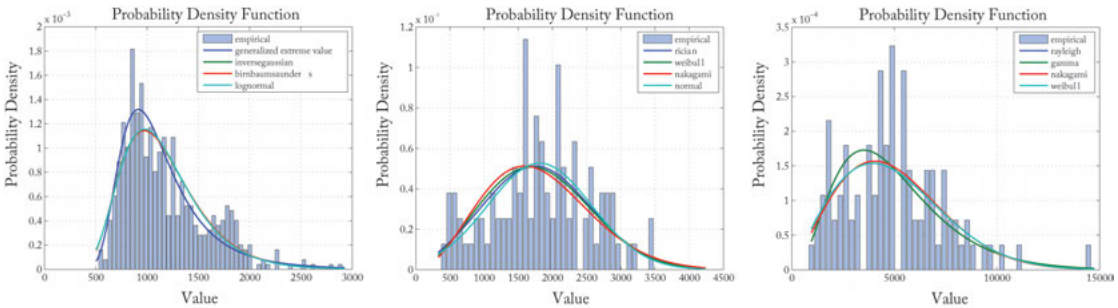

Figure 7.7: Selection of model parameters: distribution of T_m (left); distribution of T_s (Log) (middle); distribution of T_s (Linear)(right).

free to formulate them by following any QFS. If an error was committed during the formulation process, that particular formulation effort is ignored and the process has to start afresh since in our quantitative model we ignore query formulation errors.

Values of model parameters. The participants were tasked to formulate three queries ($Q0$-$Q2$), three times by each participant using each of the two variants of the query interface. The movement and selection times (T_m and T_s) are recorded for all participants and all trials. Then, these values are used to learn the distribution of the model parameters using MatLab (Figure 7.7). Note that several different distributions may fit these movement and selection times. We select the best fitting one. Using these distributions, T_m and T_s are randomly generated and suit-

able parameter ranges are then derived by solving Equations (7.1)–(7.3). Specifically, assuming $\eta = 0.33$ [Accot and Zhai, 2003] for Equation (7.1) yielded the ranges [280–579] and [229–790] for a and b, respectively. Similarly, for Equation (7.2) (selection time for the logarithmic model), the ranges of m_1 and n_1 are [294–1011] and [304–1250], respectively. For Equation (7.3) (selection time for the linear model), the ranges of m_2 and n_2 are [816–3500] and [200–1030], respectively. Note that high variability of the model parameters of T_s is expected as some users may be unfamiliar with the organization of the labels and might navigate up or down the *Label Panel* several times before selecting a label. Finally, a parameter value is selected randomly from its respective range for simulating the query formulation.

> *`ViSual` can accurately simulate subgraph query formulation behavior of different types of users, paving the way for automated performance benchmarking of a wide variety of applications involving visual interaction of users with graphs.*

Accuracy of the query formulation model. Next, we proceed to determine the accuracy of our model in terms of T_m, T_s, T_e, T_d for different queries and different user types using the empirically determined parameters. We simulate the four queries ($Q3 - Q6$) using `ViSual` and record T_m, T_s, T_e, T_d and the total QFT for all queries and compare them against those recorded for all participants (average time). Each query is now formulated five times by each participant. Figure 7.8 plots the results. Observe that our model in general predicts these times with high accuracy. For each query formulated by each participant, we compute the *symmetric mean absolute percentage error* (SMAPE) using the following equation:

$$\text{SMAPE} = \frac{100}{n} \sum_{t=1}^{n} \frac{|F_t - A_t|}{|A_t| + |F_t|},$$

where A_t is the actual value, F_t is the forecast value, and n is the number of fitted points. Note that this version of SMAPE computation is easier to interpret than that in Hyndman et al. [2006] as it yields an error in the range [0–100%] instead of [0–200%]. The average SMAPE for the timings for each query vary in the range [18.2–20.5%]. T_e estimated by `ViSual` is always higher as in our model we assume a fixed pixel distance ($D = 125.4px$, which is 1/5-th the width of the *Query Panel*) between nodes. However, some users may draw nodes closer to one another (i.e., D is smaller).

Cost of QFS graph construction. Lastly, we report the cost of QFS graph construction (Line 6 in Algorithm 7.25). We randomly select 10 frequent, 10 DIF, and 10 NIF queries for each query size and generate their QFS graphs. Note that for query size 11, no frequent and DIF queries are found. Figure 7.9 shows the results. Observe that QFS graph construction is efficient and the construction time is relatively stable as query size increases. This is because it depends on

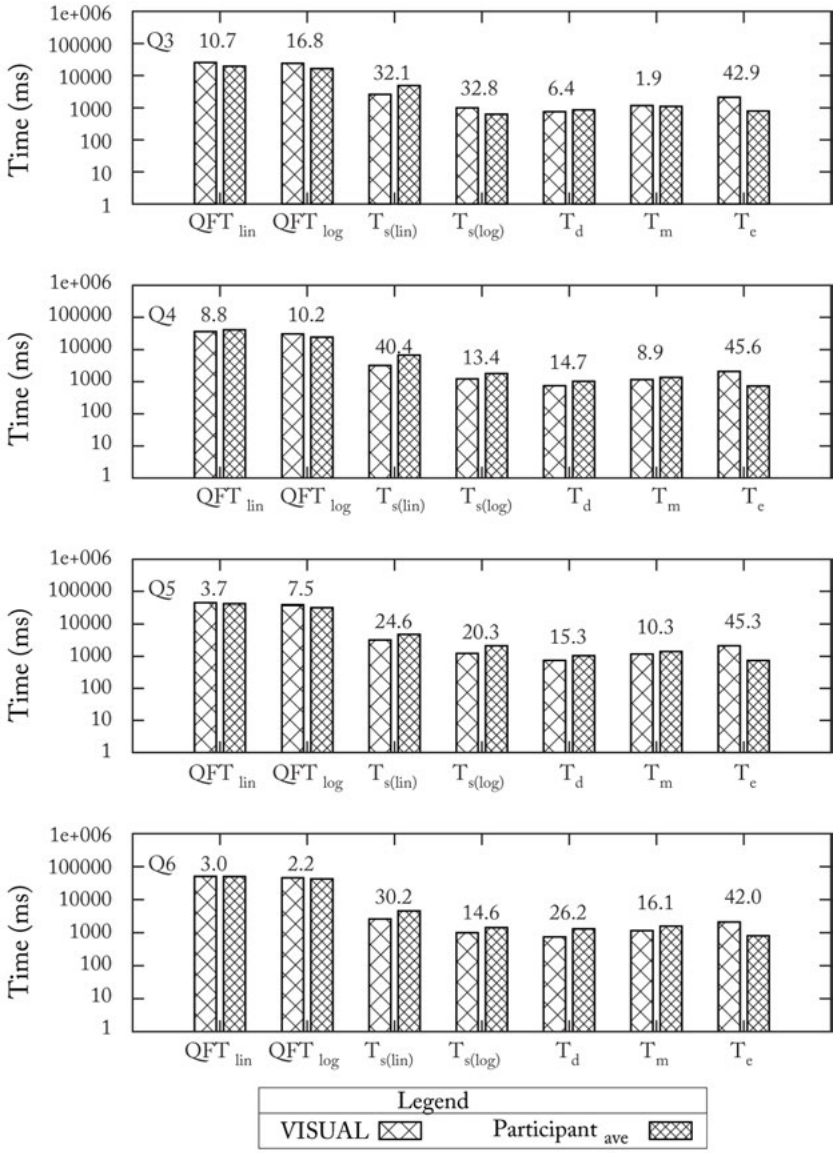

Figure 7.8: Query formulation model performance (*lin*=linear, log=logarithm). The numbers above the bars are the average SMAPE values.

the number of unique labels in a given query and not on its size. In our benchmark datasets, the former is typically less than the latter.

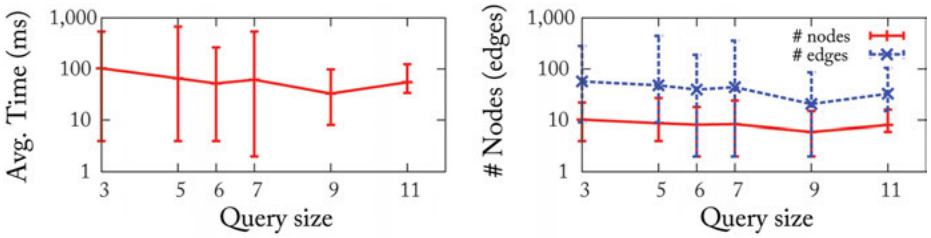

Figure 7.9: Construction time and size of QFS graphs.

7.5.3 APPLICATION OF ViSual

Lastly, we show how ViSual can be utilized to automatically evaluate the performance of PRAGUE (Chapter 4) on larger number of small-sized substructure search queries. Specifically, we integrated ViSual on top of it and randomly generated 100 queries (50 frequent and 50 NIFs) of 5–8 edges, with different topologies (path, tree, graph, and cycle), and simulated their formulation for all possible QFSs. We use the AIDS 40 K dataset ($\alpha = 0.1$). We use the frequent and NIF queries for investigating the SRT for subgraph containment search and substructure similarity search, respectively. Figure 7.10a shows the average SRT for all possible QFS. Figure 7.10b plots the average processing time of *each* edge in 5-edge to 8-edge query graph sets. Note that this time is used to construct SPIGs. In summary, it shows the effectiveness of ViSual in undertaking large-scale performance evaluation of visual subgraph querying frameworks.

7.6 BIBLIOGRAPHIC NOTES

Several work in the literature have focused on synthetic generation of datasets for relational [Gray et al., 1994, Lo et al., 2014], XML [Schmidt et al., 2002, Yao et al., 2004], and graph [Iosup et al., 2016] data for performance benchmarking. However, they neither focus on data-driven generation of test queries nor simulation of the query formulation process in a visual environment. `gMark` [Bagan et al., 2016] is a domain- and query language-independent framework for synthetic graph and query workload generation. In contrast to ViSual, query generation here is via user-defined schema and it does not focus on simulation of visual query formulation.

Quantitative modeling of visual tasks has appeared in many HCI studies [Accot and Zhai, 2003, Ahlstrom, 2005, Card et al., 1980, Cockburn and Gutwin, 2009]. These approaches focus on visual tasks such as document editing, diagram editing, menu selection, etc. However, prior to our work, it has not been studied in the context of visual query simulation.

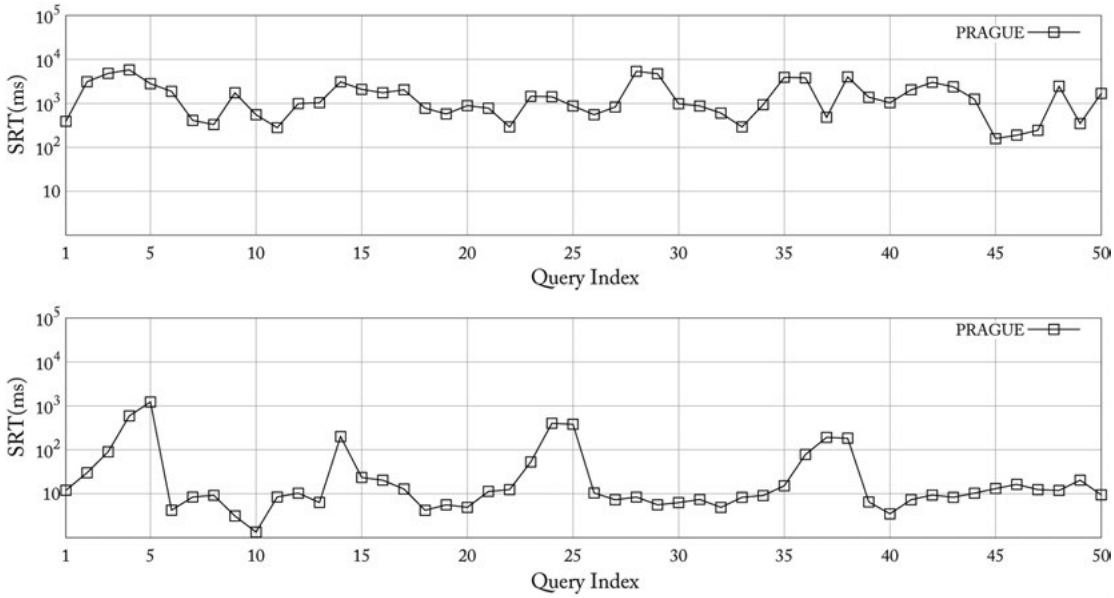

Figure 7.10: Performance study of PRAGUE using ViSual.

7.7 CONCLUSIONS

A fundamental challenge in interactive visual querying schemes is the heavy reliance on expensive and time-consuming user studies for evaluating performances of a variety of GUI-based applications. In this chapter, we address this challenge by presenting a framework called ViSual for simulating visual subgraph query construction. Building this framework entails a multidisciplinary effort drawing upon the literature in HCI and graph data management. Specifically, we leveraged work in HCI to quantitatively model the visual subgraph query formulation time, which is subsequently exploited by the `ViSual` algorithm to realistically simulate the query construction process. Our experimental study demonstrates the effectiveness and accuracy of our framework. Although this book is about graph querying, it is easy to see that the proposed paradigm can be extended to other visual querying environments.

CHAPTER 8

The Road Ahead

In this chapter, we summarize the contributions of this book and establish several lines of inquiry associated with human interaction with graphs for future research.

8.1 SUMMARY

The contributions of this book are summarized as follows.

- In Chapter 3, we present recent approaches that aid a user to formulate visual subgraph queries. Specifically, we present deep integration-based techniques for suggestion of subgraph patterns that can be added to a visual query during its formulation by utilizing the knowledge of partially formulated query fragments. To this end, this chapter has presented query decomposition, candidate suggestion generations, and ranking for subgraph query autocompletion.

- In a traditional visual query processing paradigm, query evaluation can be performed in two key steps. First, the visual query is transformed into its textual or algebraic form. Second, the transformed query is evaluated using an existing state-of-the-art graph query processing method. Chapters 4 and 5 introduce recent deep integration-based visual querying frameworks that depart from this traditional paradigm by exploring a graph query processing paradigm that blends query construction and query processing to prune false results and prefetch partial query results in a single-user environment by exploiting the availability of GUI latency during visual query formulation. These frameworks construct offline and online indexes based on underlying features of data graphs. When a user adds a new edge to a visual subgraph query fragment, they construct and maintain an adaptive online index for the edge to facilitate blending of visual query formulation and processing. In particular, these frameworks allow a user to execute a query fragment any time during query formulation and not wait until the entire query is visually formulated.

- Scant attention has been paid in the literature on the problem of query results visualization and exploration. In Chapter 6, we summarize early efforts on visual exploration of graph query results. First, we present a deep integration-based results exploration system for a large collection of small- or medium-sized graphs that utilizes the query processing engine described in Chapter 4 to support visual exploratory search. Second, we introduce different types of schemes to visualize and explore query results in large networks. In particular,

a region-based exploration scheme iteratively displays a small region of the underlying network containing a result match of a subgraph query. The exemplar-based exploration scheme adopts a bottom-up approach by allowing a user to select a specific query result (i.e., an exemplar) and relax its constraints to retrieve other similar results. Feature-based exploration schemes, on the other hand, takes a top-down approach by generating a high-level overview of all the query results by grouping them based on structural features and embed them in a low dimensional representation.

- Comprehensive user studies are needed to exhaustively and systematically evaluate performance of the aforementioned visual action-aware query processing techniques, but, unfortunately, they are expensive and time consuming. To reduce the cost and time needed, Chapter 7 introduced a recent technique that automatically generates many test subgraph queries having different user-specified characteristics (e.g., frequent, infrequent) using indexes and simulate their formulation based on different query formulation sequences without requiring human users. Specifically, it employs an HCI-inspired *quantitative model* to estimate the time for subgraph query formulation and an algorithm to simulate visual query formulation of the generated test queries by leveraging the quantitative model. This framework can then be used to automate exhaustive evaluation of performances of various visual action-aware query processing techniques.

8.2 FUTURE RESEARCH

While good progress has already been made, research on bridging graph querying and HCI opens up many opportunities for continued research. The final part of this chapter presents open problems in this area. Some of these topics were introduced by recent vision papers [Bhowmick, 2014, Bhowmick et al., 2016]. Our grand vision is a pervasive desire to continue stimulating shift in our traditional thinking by bringing together HCI and graph data management to work together.

Data-driven visual graph query interface construction. Visual graph interfaces are fundamental to promoting human interactions with graph-structured data. In this book, we focused on GUIs that are constructed and maintained manually. It is interesting to explore if GUI construction and maintenance can be *data-driven*. That is, the content of various panels of a GUI are automatically constructed and maintained as underlying data evolves. Such a data-driven paradigm has several benefits such as superior support for visual subgraph query construction, significant reduction in the manual cost of maintaining an interface for any graph query-based application, and portability of the interface across diverse variety of graph querying applications. DAVINCI [Zhang et al., 2015] is a recent effort toward this goal.

Guidance on large networks. Existing efforts on guidance during visual query formulation (e.g., subgraph suggestions) primarily focused on a large collection of small- or medium-sized graph. Scant attention has been devoted to similar techniques on massive networks. Hence, it is

interesting to explore novel techniques that can guide users to construct visual queries on such large networks.

Visual querying paradigm on massive graphs. All research related to guidance for visual query formulation, visual action-aware query processing, and exploration and visualization of query results have focused either on a large set of small- or medium-sized data graphs or on networks with millions of nodes. A natural extension to this paradigm is to support similar problems on massive graphs (comprising hundreds to billions of nodes), which may demand a distributed framework and novel algorithms built on top of it.

Efficient processing of complex graph queries. Current research demonstrates the viability of blending visual formulation and processing of subgraph containment and subgraph similarity search queries. It is an open problem to enhance the expressiveness of such visual querying framework to handle more complex subgraph queries such as homomorphism-based subgraph queries [Fan et al., 2010] and regular path queries [Libkin and Vrgoc, 2012]. BOOMER [Song et al., 2018] is a recent such effort toward supporting homomorphism-based subgraph queries on large networks.

Aesthetics-aware* GUI *design. An issue that is paramount to an end user but widely ignored by the data management community is the aesthetics of the layout of the GUI. Many HCI studies have asserted a strong link between visual complexity and aesthetics of web pages [Tuch et al., 2012] have attempted to measure their aesthetics automatically by analyzing HTML sources and screenshots of web pages [Reinecke et al., 2013]. In particular, GUI screenshot-based measure is considered superior to other methods as it better represents what a user sees [Reinecke et al., 2013]. The work in Michailidou et al. [2008] and Miniukovich and De Angeli [2014, 2015] proposed an array of *aesthetics metrics* to quantify visual complexity such as visual clutter, color variability, contour congestion, and layout quality.

Existing approaches to make a GUI stand out is to work out all of the aforementioned details of visual design manually. *How can we quantify visual complexity and aesthetics of a graph query interface?* The answer to this question shall not only enable us to compare the aesthetics of set of existing visual query interfaces automatically but also guide us to iteratively redesign these interfaces in order to increase their aesthetics appeal.

Multi-faceted exploration and visualization of query results. As remarked earlier, techniques that enable rich, interactive exploration and visualization of graph query results are still in its infancy. *How can we easily explore and visualize results of a variety of subgraph queries to gain better understanding of it?* This is an especially challenging problem when the underlying graph is massive as the entire graph looks like a giant hairball and the subgraphs that are returned as results to a query are lost in the visual maze. Furthermore, it is interesting to explore additional insights that we may attach to the matched results that may enable end users for further exploration.

Deep integration paradigm for query-by-example. In this book, we focus on the deep integration paradigm for visual query-based human interaction with graph data. As remarked in Sec-

tion 2, there are recent efforts on query-by-example for graph-structured data. However, these techniques are designed for the shallow integration model. Since visual formulation of example queries also gives rise to GUI latency, it opens up opportunities for realizing the deep integration paradigm by blending visual formulation of example queries and their processing. We believe that the techniques discussed in this book can enable exploration in this direction.

Beyond visual graph querying. Although in this book we have focused our attention on visual graph querying, it is not difficult to realize that investigation of human interaction with graphs is also pivotal for graph analytics and machine learning systems. This is especially so when highly popular deep learning models can be represented as a computation graph. Recent work on integrating human interactions with graph analytics have focused on the shallow integration paradigm [Chau, 2012]. It is interesting to explore how blending of graph analytics and machine learning frameworks with HCI (i.e., deep integration) can be crystallized in order to realize the broader vision of human interaction with graph-structured data.

Bibliography

S. Abiteboul, Y. Amsterdamer, T. Milo, and P. Senellart. Auto-completion learning for XML. In *SIGMOD*, 2012. DOI: 10.1145/2213836.2213928. 49

J. Accot and S. Zhai. Refining Fitts' law models for bivariate pointing. In *ACM SIGCHI*, 2003. DOI: 10.1145/642611.642646. 153, 157, 168, 170

D. Ahlstrom. Modeling and improving selection in cascading pull-down menus using Fitt's law, the steering law, and force fields. In *SIGCHI*, 2005. DOI: 10.1145/1054972.1054982. 170

G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. Generating flexible workloads for graph databases. In *VLDB*, 2016. DOI: 10.14778/3007263.3007283. 170

C. Barrett. Formal-language-constrained path problems. *SIAM Journal on Computing*, 30(3):809–837, 2000. DOI: 10.1137/s0097539798337716. 164

Y. Bassil. A comparative study on the performance of permutation algorithms. *Journal on Computer Science and Research*, 1(1):7–19, 2012. 162

S. S. Bhowmick and S. Prakash. Every click you make, I will be fetching it: Efficient XML query processing in RDBMS using GUI-driven prefetching. In *ICDE*, 2006. DOI: 10.1109/icde.2006.64. 6, 86, 87

S. S. Bhowmick. DB ⋈ HCI: Towards bridging the chasm between graph data management and HCI. In *DEXA*, 2014. DOI: 10.1007/978-3-319-10073-9_1. 174

S. S. Bhowmick, B. Choi, and S. Zhou. VOGUE: Towards a visual interaction-aware graph query processing framework. In *CIDR*, 2013. 5

S. S. Bhowmick, H.-E. Chua, B. Thian, and B. Choi. ViSual: An HCI-inspired simulator of blending visual subgraph query construction and processing. In *ICDE*, 2015. DOI: 10.1109/icde.2015.7113406. 163

S. S. Bhowmick, B. Choi, and C. E. Dyreson. Data-driven visual graph query interface construction and maintenance: Challenges and opportunities. *PVLDB*, 9(12), 2016. DOI: 10.14778/2994509.2994517. 174

S. S. Bhowmick, H.-E. Chua, B. Choi, and C. Dyreson. `ViSual`: Simulation of visual subgraph query formulation to enable automated performance benchmarking. *IEEE Transactions on Knowledge and Data Engineering*, 29(8): 1765–1778, 2017. DOI: 10.1109/tkde.2017.2690392. 146, 164

S. S. Bhowmick, C. E. Dyreson, B. Choi, and M.-H. Ang. Interruption-sensitive empty result feedback: Rethinking the visual query feedback paradigm for semistructured data. In *CIKM*, 2015. DOI: 10.1145/2806416.2806432. 5

A. Bonifati, W. Martens, and T. Timm. An analytical study of large SPARQL query logs. In *VLDB*, 2017. DOI: 10.14778/3149193.3149196. 58

A. Borodin, H. C. Lee, and Y. Ye. Max-sum diversification, monotone submodular functions and dynamic updates. In *PODS*, 2012. DOI: 10.1145/2213556.2213580. 37

D. Braga, A. Campi, and S. Ceri. XQBE (XQuery by example): A visual interface to the standard XML query language. In *TODS*, 2005. DOI: 10.1145/1071610.1071613. 4, 49

A. Z. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences*, 1997. DOI: 10.1109/sequen.1997.666900. 35

H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph. *Pattern Recognition Letters*, 19(3):255–259, 1998. DOI: 10.1016/s0167-8655(97)00179-7. 32

S. Card, T. P. Moran, and A. Newell. The keystroke-level model for user performance time with interactive systems. *CACM*, 23(7), 1980. DOI: 10.1145/358886.358895. 157, 170

T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini. Visual query systems for databases: A survey. *Journal of Visual Languages and Computing*, 8(2), 1997. DOI: 10.1006/jvlc.1997.0037. 6

D. H. Chau. *Data Mining Meets HCI: Making Sense of Large Graphs*. Ph.D. Dissertation. `https://www.cc.gatech.edu/~dchau/papers/polo-chau-thesis.pdf`, 2012. DOI: 10.21236/ada566568. 176

D. H. Chau, C. Faloutsos, H. Tong, J. I. Hong, B. Gallagher, and T. Eliassi-Rad. GRAPHITE: A visual query system for large graphs. In *ICDM Workshop*, 2008. DOI: 10.1109/icdmw.2008.99. 13

J. Cheng, Y. Ke, W. Ng, and A. Lu. FG-Index: Towards verification-free query processing on graph databases. In *SIGMOD*, 2007. DOI: 10.1145/1247480.1247574. 7, 10, 22, 28, 61, 62, 79, 165

A. Cockburn and C. Gutwin. A predictive model of human performance with scrolling and hierarchical lists. *Human-Computer Interaction*, 24(3):273–314 (2009). DOI: 10.1080/07370020902990402. 155, 170

S. Comai, E. Damiani, and P. Fraternali. Computing graphical queries over XML data. In *TOIS*, 2001. DOI: 10.1145/502795.502797. 4, 49

L. P. Cordella, P. Foggia, et al. An improved algorithm for matching large graphs. *Proc. of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, 2001. 75, 118, 136, 139

Z. Fan, Y. Peng, B. Choi, J. Xu, and S. S. Bhowmick. Towards efficient authenticated subgraph query service in outsourced graph databases. In *TSC*, 2014. DOI: 10.1109/tsc.2013.42. 22

W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu. Graph homomorphism revisited for graph matching. In *PVLDB*, 2010 DOI: 10.14778/1920841.1920986. 10, 175

J. Feng and G. Li. Efficient fuzzy type-ahead search in XML data. *TKDE*, pages 882–895, 2012. DOI: 10.1109/tkde.2010.264. 20, 48

M. A. Gallego, J. D. Fernandez, M. A. Martinez-Prieto, and P. de la Fuente. An empirical study of real-world SPARQL queries. In *USEWOD Workshop*, 2011. 58

X. Gao, B. Xiao, D. Tao, and X. Li. A survey of graph edit distance. In *Pattern Analysis and Applications*, 13(1), 2010. DOI: 10.1007/s10044-008-0141-y. 53

S. Gollapudi and A. Sharma. An axiomatic approach for result diversification. In *WWW*, 2009. DOI: 10.1145/1526709.1526761. 32

J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD*, 1994. DOI: 10.1145/191843.191886. 170

T. Grossman, N. Kong, and R. Balakrishnan. Modeling pointing at targets of arbitrary shapes. In *ACM SIGCHI*, 2007. DOI: 10.1145/1240624.1240700. 159

W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu. iGraph: A framework for comparisons of disk-based graph indexing techniques. In *PVLDB*, 2010. DOI: 10.14778/1920841.1920901. 43, 44

W.-S. Han, J. Lee, and J.-H. Lee. TurboISO: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*, 2013. DOI: 10.1145/2463676.2465300. 7, 10, 75

F. Haag, S. Lohmann, S. Bold, and T. Ertl. Visual SPARQL querying based on extended filter/flow graphs. In *AVI*, 2014. DOI: 10.1145/2598153.2598185. 13

M. Herschel, Y. Tzitzikas, K. S. Candan, and A. Marian. Exploratory search: New name for an old hat? `http://wp.sigmod.org/?p=1183`, 2014.

J. P. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraph in the presence of isomorphism. In *ICDM*, 2003. DOI: 10.1109/icdm.2003.1250974. 10, 62, 76, 93, 105

K. Huang, S. S. Bhowmick, S. Zhou, and B. Choi. PICASSO: Exploratory search of connected subgraph substructures in graph databases. *PVLDB*, 10(12):1861–1864, 2017. DOI: 10.14778/3137765.3137794. 5, 57, 133, 134, 143

H. H. Hung, S. S. Bhowmick, B. Q. Truong, B. Choi, and S. Zhou. QUBLE: Towards blending interactive visual subgraph search queries on large networks. *VLDB Journal*, 23(3), 2014. DOI: 10.1007/s00778-013-0322-1. 5, 118, 120, 127, 128, 131, 133, 138, 143

R. J. Hyndman, et al. Another look at measures of forecast accuracy. *International Journal Forecasting*, 22(4):679–688, 2006. DOI: 10.1016/j.ijforecast.2006.03.001. 168

S. Idreos, O. Papaemmanouil, and S. Chaudhuri. Overview of data exploration techniques. In *SIGMOD*, 2015. DOI: 10.1145/2723372.2731084. 133

A. Iosup, T. Hegeman, W. L. Ngai, S. Heldens, A. Prat-Pérez, T. Manhardt, H. Chafi, M. Capota, N. Sundaram, M. J. Anderson, I. G. l. Tanase, Y. Xia, L. Nai, and P. A. Boncz. LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms, a Technical Report. In *VLDB*, 2016. DOI: 10.14778/3007263.3007270. 170

M. James, C. Michael, B. Brad, et al. Big data: The next frontier for innovation, competition, and productivity. *The McKinsey Global Institute*, 2011.

N. Jayaram, A. Khan, C. Li, X. Yan, and R. Elmasri. Querying knowledge graphs by example entity tuples. In *ICDE (Demo)*, 2014. DOI: 10.1109/icde.2016.7498391. 15, 16

N. Jayaram, S. Goyal, and C. Li. VIIQ: Auto-suggestion enabled visual interface for interactive graph query formulation. *PVLDB*, pages 1940–1951, 2015. DOI: 10.14778/2824032.2824106. 24, 47, 48, 49

C. Jin, S. S. Bhowmick, X. Xiao, J. Cheng, and B. Choi. Gblender: Towards blending visual query formulation and query processing in graph databases. In *SIGMOD*, 2010. DOI: 10.1145/1807167.1807182. 11, 52, 80, 86, 133, 134, 138, 143

C. Jin, S. S. Bhowmick, B. Choi, and S. Zhou. Prague: A practical framework for blending visual subgraph query formulation and query processing. In *ICDE*, 2012. 3, 5, 11, 13, 52, 53, 86, 133, 134, 138, 143

G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1), 1999. DOI: 10.1137/s1064827595287997. 92, 96, 98

F. Katsarou, N. Ntarmos, and P. Triantafillou. Performance and scalability of indexed subgraph query processing methods. In *VLDB*, 2015. DOI: 10.14778/2824032.2824054. 145, 165

A. Khan and S. Elnikety. Systems for big graphs. In *VLDB*, 2014. DOI: 10.14778/2733004.2733067. 10, 131

N. Kriege, P. Mutzel, and T. Schäfer. Practical SAHN clustering for very large data sets and expensive distance metrics. *Journal of Graph Algorithms Applications*, 2014. DOI: 10.7155/jgaa.00338. 32

U. Leser. A query language for biological networks. In *Bioinformatics*, 21:ii33–ii39, 2005. DOI: 10.1093/bioinformatics/bti1105. 1

Y. Li, C. Yu, and H. V. Jagadish. Enabling schema-free xquery with meaningful query focus. In *VLDB Journal*, 2008. DOI: 10.1007/s00778-006-0003-4. 48

L. Libkin and D. Vrgoc. Regular path queries on graphs with data. In *ICDT*, 2012. DOI: 10.1145/2274576.2274585. 175

W. Lin, X. Xiao, J. Cheng, and S. S. Bhowmick. Efficient algorithms for generalized subgraph query processing. In *CIKM*, 2012. DOI: 10.1145/2396761.2396805. 10, 158

C. Lin, J. Lu, T. W. Ling, and B. Cautis. LotusX: A position-aware XML graphical search system with auto-completion. In *ICDE*, 2012. DOI: 10.1109/icde.2012.123. 49

S. Liu, J. P. Cedeno, K. S. Candan, M. L. Sapino, S. Huang, and X. Li. R2DB: A system for querying and visualizing weighted RDF graphs. In *ICDE*, 2012. DOI: 10.1109/icde.2012.134.

E. Lo, N. Cheng, W. Lin, W.-K. Hon, and B. Choi. MyBenchmark: Generating databases for query workloads. *VLDB Journal*, 23(6), 2014. DOI: 10.1007/s00778-014-0354-1. 170

G. Marchionini. Exploratory search: From finding to understanding. *Communications of the ACM*, pages 41–46, 2006. DOI: 10.1145/1121949.1121979.

J. J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software Practice and Experience*, 12(1), pages 23–34, 1982. DOI: 10.1002/spe.4380120103. 42

E. Michailidou, S. Harper, and S. Bechhofer. Visual complexity and aesthetic perception of web pages. In *Proc. of ACM International Conference on Design of Communication*, 2008. DOI: 10.1145/1456536.1456581. 175

A. Miniukovich and A. De Angeli. Quantification of interface visual complexity. In *Working Conference on Advanced Visual Interfaces*, 2014. DOI: 10.1145/2598153.2598173. 175

A. Miniukovich and A. De Angeli. Computation of interface aesthetics. In *SIGCHI*, 2015. DOI: 10.1145/2702123.2702575. 175

M. Mongiovi, R. Di Natale, R. Giugno, A. Pulvirenti, A. Ferro, and R. Sharan. SIGMA: A set-cover-based inexact graph matching algorithm. In *Journal of Bioinformatics and Computer Biology*, 2010 DOI: 10.1142/s021972001000477x. 10, 79, 80

D. Mottin, F. Bonchi, and F. Gullo. Graph query reformulation with diversity. In *KDD*, 2015. DOI: 10.1145/2783258.2783343.

A. Nandi and H. V. Jagadish. Effective phrase prediction. In *VLDB*, 2007.

C. Nevill-Manning and I. W. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

S. Pandey and K. Punera. Unsupervised extraction of template structure in web search queries. In *WWW*, pages 409–418, 2012. DOI: 10.1145/2187836.2187892. 28

Y. Papakonstantinou, M. Petropoulos, and V. Vassalos. QURSED: Querying and reporting semistructured data. In *SIGMOD*, 2002. DOI: 10.1145/564712.564714. 49

P. Pardalos and A. Migdalas. A note on the complexity of longest path problems related to graph coloring. *Applied Mathematics Letters*, 17(1):13–15, 2004. DOI: 10.1016/s0893-9659(04)90003-1. 164

R. Pienta, A. Tamersoy, A. Endert, S. Navathe, H. Tong, and D. H. Chau. VISAGE: Interactive visual graph querying. In *AVI*, 2016. DOI: 10.1145/2909132.2909246. 13, 133, 138, 143

R. Pienta, F. Hohman, A. Endert, A. Tamersoy, K. A. Roundy, C. S. Gates, S. B. Navathe, and D. H. Chau. VIGOR: Interactive visual exploration of graph query results. In *IEEE Transactions on Visualization and Computer Graphics*, 2018. DOI: 10.1109/tvcg.2017.2744898. 133, 140, 141, 142, 143

L. Qin, J. Xu Yu, and L. Chang. Diversifying top-k results. *CoRR*, abs/1208.0076, 2012. DOI: 10.14778/2350229.2350233. 32

S. Raje and M. Sarrafzadeh. Variable voltage scheduling. In *ISLPED*, 1995. DOI: 10.1145/224081.224084. 164

K. Reinecke, T. Yeh, L. Miratrix, R. Mardiko, Y. Zhao, J. Liu, and K. Z. Gajos. Predicting users' first impressions of website aesthetics with a quantification of perceived visual complexity and colorfulness. In *SIGCHI*, 2013. DOI: 10.1145/2470654.2481281. 175

A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB*, 2002. 170

H. Shang, X. Lin, Y. Zhang, J. X. Yu, and W. Wang. Connected substructure similarity search. In *SIGMOD*, 2010. DOI: 10.1145/1807167.1807264. 10, 53, 79, 80

Y. Song, H. E. Chua, S. S. Bhowmick, B. Choi, and S. Zhou. BOOMER: Blending visual formulation and processing of p-homomorphic queries on large networks. In *Proc. of 44th ACM SIGMOD International Conference on Management of Data (ACM SIGMOD)*, ACM Press, Houston, June 2018. DOI: 10.1145/3183713.3196902. 175

B. Q. Truong and S. S. Bhowmick. MustBlend: Blending visual multi-source twig query formulation and query processing in RDBMS. In *DASFAA*, 2013. DOI: 10.1007/978-3-642-37450-0_17. 6, 87

A. N. Tuch, E. E. Presslaber, M. Stöcklina, K. Opwis, and J. A. Bargas-Avila. The role of visual complexity and prototypicality regarding first impression of websites: Working towards understanding aesthetic judgements. *International Journal of Human-Computer Studies*, 70, 2012. DOI: 10.1016/j.ijhcs.2012.06.003. 175

M. L. Fernández Venero and G. Valiente. A graph distance metric combining maximum common subgraph and minimum common supergraph. *Pattern Recognition Letters*, 22:753–758, 2001. DOI: 10.1016/S0167-8655(01)00017-4. 32

M. R. Vieira, H. L. Razente, M. C. N. Barioni, M. Hadjieleftheriou, D. Srivastava, C. Traina, and V. J. Tsotras. On query result diversification. In *ICDE*, 2011. DOI: 10.1109/icde.2011.5767846. 32

W. D. Wallis, P. Shoubridge, M. Kraetzl, and D. Ray. Graph distances using graph union. *Pattern Recognition Letters*, 22:701–704, 2001. DOI: 10.1016/s0167-8655(01)00022-8. 32

C. Ware and P. Mitchell. Visualizing graphs in three dimensions. *ACM Transactions on Applied Perception (TAP)*, 5(1), 2008. DOI: 10.1145/1279640.1279642. 138

R. W. White and R. A. Roth. Exploratory search: Beyond the query-response paradigm. *Synthesis Lectures on Information Concepts, Retrieval, and Services 1*, 1, 2009. DOI: 10.2200/s00174ed1v01y200901icr003. 134

Y. Wu, S. Yang, and X. Yan. Ontology-based subgraph querying. In *ICDE*, 2013. DOI: 10.1109/icde.2013.6544867. 10, 158

Y. Xie and P. S. Yu. CP-Index: On the efficient indexing of large graphs. In *CIKM*, 2011. DOI: 10.1145/2063576.2063835. 10, 89, 91, 131

X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *ICDM*, 2002. DOI: 10.1109/icdm.2002.1184038. 25, 26, 29, 42, 57, 60

X. Yan, P. S. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *SIGMOD*, 2004. DOI: 10.1145/1007568.1007607. 1, 10, 22, 25, 28, 35, 51

X. Yan, P. S. Yu, and J. Han. Substructure similarity search in graph databases. In *ACM SIGMOD*, 2005. DOI: 10.1145/1066157.1066244. 7, 10, 51, 53, 79, 80

H. Yang and X. Xu. Bias towards regular configuration in 2D pointing. In *SIGCHI*, 2010. DOI: 10.1145/1753326.1753536.

S. Yang, Y. Xie, Y. Wu, T. Wu, H. Sun, J. Wu, and X. Yan. SLQ: A user-friendly graph querying system. In *SIGMOD*, 2014. DOI: 10.1145/2588555.2594516. 7, 13, 158

B. Yao, M. T. Özsu, and N. Khandelwal. XBench benchmark and performance testing of XML DBMSs. In *ICDE*, 2004. DOI: 10.1109/icde.2004.1320032. 170

P. Yi, B. Choi, S. S. Bhowmick, and J. Xu. AutoG: A visual query autocompletion framework for graph databases (demo). *PVLDB*, 9:1505–1508, 2016. DOI: 10.14778/3007263.3007295. 21, 48

P. Yi, B. Choi, S. S. Bhowmick, and J. Xu. AutoG: A visual query autocompletion framework for graph databases. *VLDB Journal*, 26(3):347–372, 2017. DOI: 10.14778/3007263.3007295. 5, 21, 22, 31, 41, 48

P. Yi, B. Choi, S. S. Bhowmick, and J. Xu. AutoG: A visual query autocompletion framework for graph databases. `https://goo.gl/Xr9MRY`, 2017. DOI: 10.14778/3007263.3007295. 45

S. Zhang, J. Yang, and W. Jin. SAPPER: Subgraph indexing and approximate matching in large graphs. In *VLDB*, 2010. DOI: 10.14778/1920841.1920988. 10, 120, 129, 131

J. Zhang, S. S. Bhowmick, H. H. Nguyen, B. Choi, and F. Zhu. DAVINCI: Data-driven visual interface construction for subgraph search in graph databases. In *IEEE ICDE*, 2015. DOI: 10.1109/icde.2015.7113411. 174

Y. Zhou, S. S. Bhowmick, E. Leonardi, and K. Widjanarko. XBlend: Visual XML query formulation meets query processing. In *ICDE*, 2009. DOI: 10.1109/icde.2009.57. 6

G. Zhu, X. Lin, K. Zhu, W. Zhang, and J. Xu Yu. TreeSpan: Efficiently computing similarity all-matching. In *SIGMOD*, 2012. DOI: 10.1145/2213836.2213896. 7, 10, 131

Authors' Biographies

SOURAV S. BHOWMICK

Sourav S. Bhowmick is an Associate Professor in the School of Computer Science and Engineering (SCSE) at Nanyang Technological University. He leads the data management research group (DANTe) in SCSE. His research has appeared in top-tier venues in data management and analytics such as *SIGMOD, VLDB, VLDB Journal, TKDE, WWW*, and *KDD*. Sourav has been keynote and tutorial speaker for several international conferences including SIGMOD and VLDB. He has received *Best Paper Awards* at ACM CIKM 2004 and ACM BCB 2011 for papers related to evolution mining and biological network summarization, respectively. His work on influence maximization was nominated for the best paper award in ACM SIGMOD 2015. Sourav has served as a PC member of premium data management and data mining conferences (e.g., SIGMOD, VLDB, KDD) and a reviewer for various premium journals (e.g., *VLDB Journal*).

BYRON CHOI

Byron Choi is an Associate Professor in the Department of Computer Science at Hong Kong Baptist University (HKBU). He obtained his Ph.D in Computer and Information Science from the University of Pennsylvania in 2006. His research interests include graph-structured databases, database usability, and database security. Byron's publications have appeared in premium venues such as *TKDE, VLDBJ, SIGMOD*, and *VLDB*. He has served as a program committee member or reviewer of premium conferences and journals including *PVLDB, VLDBJ, TKDE* and *TOIS*. He has served as the director of a Croucher Foundation Advanced Study Institute (ASI), titled "Frontiers in Big Data Graph Research" in 2015. He was a recipient of the HKBU President's Award for Outstanding Young Researcher in 2016.

CHENGKAI LI

Chengkai Li is an Associate Professor in the Department of Computer Science and Engineering at the University of Texas at Arlington. He received his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 2007. Chengkai's research interests are in database, data mining, web data management, and natural language processing. He is conducting research on computational journalism, crowdsourcing and human computation, data exploration by ranking (top-k), skyline and preference queries, database testing, entity query,

and usability challenges in querying graph data. Chengkai's papers have appeared in prestigious database, data mining, and web conferences (e.g., SIGMOD, VLDB, CIDR, KDD, WWW, WSDM) and journals (e.g., *TODS, TKDD, TKDE*). He has served as General Co-Chair and Program Co-Chair of IEEE IPCCC, and he has also served on the organizing committee of SIGMOD. He served on the program committees of premier conferences (e.g., SIGMOD, VLDB, KDD, WWW, IJCAI). He has also been a reviewer for prestigious journals (e.g., *TODS, TOIS, TKDE, VLDB Journal*). Chengkai is a recipient of the 2011 and 2012 HP Labs Innovation Research Award.