Secure subgraph query services

Zhe Fan

Hong Kong Baptist University
Secure Subgraph Query Services

FAN Zhe

A thesis submitted in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Principal Supervisor: Associate Professor CHOI Byron

Hong Kong Baptist University

August 2015
Declaration

I declare that this thesis has been composed by myself under the guidance of my principal supervisor professor Byron CHOI and co-supervisor professor Jianliang XU. The thesis has not previously included in any thesis, dissertation or report submitted to any institution for a degree, diploma or other qualification. All sources of information have been acknowledged by means of references to the relevant publications.

Signature:____________________

August 2015
Abstract

Graphs are powerful tools for a wide range of real applications, from Biological and Chemical Databases, Social Networks, Citation Networks to Knowledge Bases. Large graph data repositories have been consistently found in recent applications. Due to the high complexity of graph queries, e.g., NP-Completeness of subgraph query, and the lack of IT expertise, hosting efficient graph query services for the owners of graph data has been a technically challenging task. And hence, they may prefer to outsource their services to third-party service providers (SPs) for scalability, elasticity and efficiency.

Unfortunately, SPs may not always be trusted. Security, typically the integrity and confidentiality, of the data, has been recognized as one of the critical attributes of Quality of Services (QoS). This directly influences the willingness of both data owners and query clients to use SP’s services. To address these concerns, this thesis proposes novel techniques to solve both authentication-aware and privacy-aware subgraph query.

Firstly, we study authenticated subgraph query services (Chapter 3). To support the service, we propose Merkle IFTree (MIFTree) where Merkle hash trees are applied into our Intersection-aware Feature-subgraph Tree (IFTree). IFTree aims to minimize I/O in a well-received subgraph query paradigm namely the filtering-and-verification framework. The structures required to be introduced to verification objects (VOs) and the authentication time are minimized. Subsequently, the overall response time is minimized. For optimizations, we propose an enhanced authentication method on MIFTree.

Secondly, we propose structure-preserving subgraph query services (Chapter 4). A crucial step of this part is to transform the seminal subgraph isomorphism algorithm (the Ullmann’s algorithm) into a series of matrix operations. We propose a novel cyclic group based encryption (CGBE) method for private matrix operations. We propose a protocol that involves the query client and static indexes for optimizations. We prove that the
structural information of both query graph and data graph are preserved under CGBE and analyze the privacy preservation in the presence of the optimizations.

Thirdly, we propose asymmetric structure-preserving subgraph query processing (Chapter 5), where the data graph is publicly known and the query structure/topology is kept secret. Unlike other previous methods for subgraph queries, this part proposes a series of novel optimizations that only exploit graph structures, not the queries. Further, we propose a robust query encoding and adopt our proposed cyclic group based encryption method, so that the query processing can be transformed into a series of private matrix operations and performed securely.

The effectiveness and efficiency of all the techniques presented in this thesis are experimentally evaluated with both real-world and synthetic dataset.

**Keywords:** Outsourced Databases, Graph Databases, Subgraph Query, Authenticated Query Processing, Privacy-preserving Query Processing.
Acknowledgments

I would like to express my deep gratitude to my supervisor, Dr. Byron CHOI, for his great patience, inspiring guidance and constructive suggestions in my studies and research works in these years. He has brought me into this challenging research area and shared insightful experiences with me. I would also like to thank my co-supervisor, Dr. Jian-liang XU, for his continuous encouragement and help. Moreover, I would like to thank Prof. Wenfei FAN, for his great support and advice when I was visiting the University of Edinburgh.

I would like to thank my colleagues and friends for their direct and indirect help. In particular, I should mention Dr. Yun PENG, Mr. Qian CHEN, Mr. Peipei YI, Mr. Lei CHEN, Mr. Chuojie XIAN, Mr. Xiaoqing XIE, Mr. Shuxiang YIN, Dr. Xin LIN, Dr. Dingming WU, Mr. Zhuo CHEN, Mr. Jingjing CHEN, Mr. Jintian DENG, Mr. Chao TIAN, Mr. Ruizhe HUANG, Mr. Shen GAO, Dr. Xin LIU, among many others.

Finally, I take this special occasion to thank my father Xiaohui FAN and my mother Lihui WANG for raising me and supporting me for so many years. Without them, I would never go so far.
# Table of Contents

Declaration ............................................. i

Abstract .............................................. ii

Acknowledgments ........................................ iv

Table of Contents ........................................ v

List of Figures ........................................... ix

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

1.1 Graph Databases ........................................ 1

1.1.1 Modest Size Graphs .................................. 1

1.1.2 Large Graph Repositories ............................. 2

1.1.3 Graph Queries ....................................... 3

1.2 Outsourced Graph Databases and Their Security .................. 3

1.3 Authenticated Subgraph Query Services ........................... 5

1.4 Structure-preserving Subgraph Query Services ....................... 7

1.5 Asymmetric Structure-preserving Subgraph Query Service for Large Graphs 11

1.6 Thesis Organization ...................................... 13

## Chapter 2  Related Work .................................. 14

2.1 Security-aware Graph Queries and Management ...................... 14

2.1.1 Authenticated Graph Query ............................. 14

2.1.2 Privacy-preserving Graph Query .......................... 15
Chapter 4  Structure-Preserving Subgraph Query Services  64

4.1  Problem Formulation  64
4.2  Backgrounds and Preliminaries  66
  4.2.1  Data Model  66
  4.2.2  Revised Ullmann’s Algorithm  66
4.3  subIso with Matrix Operations  69
4.4  Structure-Preserving subIso  73
  4.4.1  Structure-Preserving Matching  73
  4.4.2  Structure-Preserving Enumeration  79
  4.4.3  Structure-Preserving Refinement  82
4.5  Privacy Analysis  86
  4.5.1  Privacy of the Encryption Method  86
  4.5.2  Privacy of SPsubIso  88
4.6  Experimental Evaluation  92
  4.6.1  Experimental Setup  92
  4.6.2  Experiments on Real Datasets  93
  4.6.3  Experiments on Synthetic Datasets  98

Chapter 5  Asymmetric Structure-Preserving Subgraph Queries for Large Graphs 100

5.1  Problem Formulation  100
5.2  Preliminaries and Overview  102
  5.2.1  Data Model  102
  5.2.2  Overview of Our Approach  102
5.3  Query Preprocessing at the Client  104
  5.3.1  Retrieving Parameters for Optimization  104
  5.3.2  Query Encoding  105
List of Figures

1.1 Overview of the system model for authenticated subgraph query service over transactional graph databases. ................................. 4
1.2 Overview of the system model for structure-preserving subgraph query service over transactional graph databases. ......................... 8
1.3 Overview of the system model for asymmetric structure-preserving subgraph query service over large graphs. .............................. 11

3.1 Illustration of gIndex and its query processing ................................. 21
3.2 Two kinds of Merkle Hash Trees used ........................................... 24
3.3 Overview of efficient authentication method ..................................... 31
3.4 Illustration of higher-order features .............................................. 33
3.5 Illustration of IFTree (partial) .................................................... 34
3.6 Subgraph query processing on IFTree ............................................. 38
3.7 Illustration of MHTs of node(p2) of an MIFTree (partial) .................... 40
3.8 VO for basic authentication ....................................................... 44
3.9 VO for enhanced authentication ................................................... 48
3.10 I/O cost and query performance ................................................ 55
3.11 Basic authentication method .................................................... 56
3.12 Enhanced authentication method ................................................ 58
3.13 Authentication performance on synthetic dataset ............................ 60
3.14 VO size vs. authentication time of various query sizes of the basic authentication method on the AIDS dataset. ......................... 62

4.1 Overview of our techniques. ....................................................... 65
4.2 Adjacency matrices of $Q$ and $G$; Two possible mappings ($M_1$ and $M_2$) and a violation in $C_2$ (Formula 4.2.1). ........................................ 69
4.3 (a) The truth table of $M_Q \overline{C}_i$; (b) Illustration of $M_Q$ and $M_{\overline{C}_i}$; and (c) TMatch. ......................................................... 72
4.4 (a) The encoding of the truth table shown in Fig. 4.3(a); (b) Encryption by CGBE; and (c) Illustration of SPMatch with $M_1$ and $M_2$. ............. 74
4.5 Illustration of mapping enumerations and SPEnum. ................................. 80
4.6 (a) Illustration of the $h$-hop vertices and static index; and (b) an refinement by the index .............................................................. 86
4.7 Performance on varying query sizes. ..................................................... 94
4.8 Effectiveness of the starting pruning depth $d$. ....................................... 96
4.9 Effectiveness of SI. ................................. ................................. 97
4.10 Performance on synthetic dataset. ......................................................... 99

5.1 Overview of the system model. ......................................................... 101
5.2 Overview of our approach. ......................................................... 101
5.3 (a) Illustration of the preprocessing at the client; (b) Construction of candidate subgraphs; and (c) Minimization of $CS_s$ by NC. ................. 102
5.4 (a) Construction of candidate matchings; and (b) Enumeration of possible mappings. .............................................................. 109
5.5 SPVerify (and SPVerify*) between $Q_k$ and $CM_{s1}$. ............................ 115
5.6 Average preprocessing time at the client. ......................................... 126
5.7 Average received encrypted message size at the client. ......................... 127
5.8 Average decryption time at the client. .............................................. 128
5.9 Average total running time at the $SP$. ............................................. 129
5.10 Average % of reduced vertices in $CS_s$ by NC. ................................ 129
5.11 Average % of the pruned redundant $CM_s$ by Cache. ......................... 130
5.12 Average Cache size at $SP$. ......................................................... 131
5.13 Average % of the pruned redundant mappings by NEC. .................... 132
5.14 Average size of messages $R_k$s when $c = 11$. ................................. 132
Chapter 1

Introduction

1.1 Graph Databases

Graph databases have been widely used in various range of emerging applications, including social network (e.g., Facebook [2], Google+ [4] and Twitter [80]), chemi-informatics (e.g., PubChem [77] and AIDS [79]), knowledge base (e.g., Freebase [3], WikiPedia [10] and DBpedia [1]), computational biology [84, 91] and data exchange [16], whose data are modeled as graphs. In this section, we will first categorize the graph databases into two types. For each type, we discuss its main characteristics and some important applications. We then introduce some widely used graph queries on top of those graph data. Note that this thesis focuses on static graph data and dynamic graphs are beyond of this thesis.

1.1.1 Modest Size Graphs

The first type of graph data considers a graph database with a large collection of modest size graphs [22, 42, 60, 94, 95, 106, 116, 122]. Graph databases of such data are often called transactional graph databases, which have been widely used in many applications.

In computational biology analysis, as of 2012, it is reported in [32] that there are more than 1,500 online molecular biology databases in the literature. For example, in protein motif discovery [48], proteins are usually modeled as graphs, on which nodes represent amino acids and edges represent the distance relationships between amino acids.
Various graph mining techniques can be used to find the motifs in the proteins. In addition, people also model many other biological structures as graphs. The genetic regulatory network [56] contains the gene expression information in cells, the signal transduction network [54] models the signal transmission between cells and the metabolic network [25] models the metabolism in various organisms. An important research problem in this area is the pathway analysis (e.g., [84, 91]).

Furthermore, in chemi-informatics, people also represent chemical molecules (or compounds) as graphs and store them in transactional graph databases. For instance, PubChem [77], a free chemical molecule database offered by NCBI, provides public access to numerous chemical molecules. Users can search molecules of their interests using PubChem. They may draw a query graph and query the molecules in the database that contain the query graph or similar structure. Moreover, there are many other public chemical molecular databases, such as ZINC [8] and eMolecules [5]. As reported in [107, 118], such chemical databases are used in the pharmaceuticals industry and can shorten the drug development cycle.

1.1.2 Large Graph Repositories

The second type of graph data handles a graph database with a very large graph [30, 39, 90, 96, 104, 119]. Similarly, graph databases with such data have also been used in many real world applications.

In social networks like Twitter [80], the number of users has exceeded 500 millions and the total number of tweets sent has exceeded 170 billions. Facebook [2], another popular social network service, the number of users has exceeded 1.15 billions and the number of friend connections has exceeded 150 billions. Large number of graph analysis techniques are proposed on top of such network [78, 89, 98, 102]. Besides, various network evolved from social network have also emerged, e.g., collaboration network, email communication network and network of co-authorship.

Moreover, in knowledge bases, ontology are modeled as graphs, in particular, nodes are used to represent entities and edges to represent the relationships between entities.
Such knowledge bases find applications in entity linking [29] and keywords search and interpretation [97], etc. For example, there are more than 4.3 millions entities and 40.3 millions connections in DBpedia [1], and the number of entities has even exceeded 23 millions in Freebase [3].

1.1.3 Graph Queries

Various graph structural queries, e.g., reachability query (e.g., [26, 43, 55, 92, 112, 113]), shortest distance (or path) query (e.g., [11, 21, 26, 33, 75, 114]), subgraph similarity query (e.g., [45, 93, 108, 117]), graph simulation query (e.g., [31, 44]) and subgraph isomorphism query (e.g., [22, 42, 60, 94, 95, 101, 106, 116, 122]), have been proposed over both of the above graph databases. In particular, subgraph isomorphism query (in short subgraph query) has become one of the fundamental and popular queries [12], which finds various applications in multiple domains, e.g., social network matching [96, 119], knowledge base cleaning [30], ontology-based matching [104] and graph structure mining [105], despite its NP-Completeness [34].

In this thesis, we mainly focus on subgraph query, for graph databases with graphs of both modest and large size. Specifically, given a query graph $Q$, on one hand, the subgraph query over a transactional graph database is to retrieve all graphs in the database that contain $Q$ as a subgraph; and on the other hand, the subgraph query over a large graph $G$ is to check if $Q$ is a subgraph of $G$.

1.2 Outsourced Graph Databases and Their Security

Due to the high complexity of the query evaluation, in particular, NP-Completeness of subgraph query, and the cost of hosting the explosive volume of data and performing large-scale computations, the owners of graph databases may not always have the necessary IT infra-structure and expertise to provide the best usage of their data. An appealing solution to address this issue of managing voluminous data is to outsource the owners’ data to a third-party Service Provider $SP$ (e.g., Amazon EC2 and Google Cloud Service).
Then, the \( SP \) provides query services on the data owners’ behalf. Such \( SP \)s are often equipped with high performance computing utilities (e.g., a cloud) that offer better scalability, elasticity and IT management [38]. For instance, according to [77], PubChem has managed 19 million unique compound structures. PubChem allows laboratories to submit their data [76]; and PubChem manages the data on the laboratories’ behalf. In addition to PubChem, in drug engineering, many commercial service providers (e.g., [6, 7, 53]) support outsourcing of pharma databases owned by laboratories. Laboratories then focus on the curation of their data.

Unfortunately, however, \( SP \)s may not always be trusted. Security, typically the \textit{integrity} and \textit{confidentiality}, of the data, has been recognized as one of the critical attributes of Quality of Services (QoS). This directly influences the willingness of both data owners and query clients to use \( SP \)’s services.

In this thesis, we propose both authentication-aware and privacy-aware subgraph query service for outsourced transactional graph databases, which will be discussed in details in Chapters 3 and 4, respectively. In Chapter 5, we further extend the system setting and study the privacy-ware subgraph query for graph databases with large graphs, where \( SP \) itself is the owner of the data that can be found in many real world applications. And for the rest of this chapter, we will take a look at the overview of these three parts.
1.3 Authenticated Subgraph Query Services

As motivated above, \( SP \) may be untrusted and/or compromised to attacks and clients may receive tampered results, which may influence the practicality of outsourcing graph databases. Hence, in order to guarantee the data integrity of the subgraph query service, this thesis first proposes an efficient query authentication framework to support the subgraph query services. Note that this part focuses on the graph database with a collection of graphs with modest size, i.e. transactional graph databases.

**Motivating example.** Fig. 1.1 shows an example of a graph database \( G \) and a query graph \( q \). Suppose the service provider stores \( G \) and its index and the client retrieves graphs that contain \( q \). Suppose \( g_4 \) is the answer graph. However, the service provider might return incorrect results, e.g., \( g_1 \), simply abort the computation or return partial answers as some queries may in fact take long to evaluate. In this scenario, the owner/client may never be sure whether the data was outsourced correctly. In practice, the query can be some sensitive chemical compound such as benzopyrene, a carcinogenic substance recently found in some ramen. A compromised service provider might collaborate with some ramen companies and conveniently skip their ingredients that contain benzopyrene.

Majority of existing querying or indexing algorithms for subgraph queries adopt a filtering-and-verification framework \([22, 60, 94, 95, 106, 116]\) consisting of two key steps. (1) In the filtering phase, the query is decomposed into a set of individual features and an index is searched with those features. The search of each individual feature yields a set of graphs (represented by graph IDs) containing this searched feature. The sets of graphs are intersected to form a candidate set (a superset of answers). (2) In the verification phase, each graph in the candidate set is checked by an exact subgraph isomorphic algorithm to compute the final result set. However, to the best of our knowledge, none of the existing subgraph querying works addresses authentication of such a framework.

In a typical query authentication system \([37]\), a data owner publishes his database and signature; A service provider processes queries from a client and transmits to the client both the answer and a verification object \((V\mathcal{O})\) which stores the processing traces such as
index traversals and; By using the answer and \( \mathcal{VO} \), the client synthesizes the digest of the database/index and compares it with the data owner’s signature to verify the authenticity of the answer.

As the filtering-and-verification framework is not specially designed for authentication, we note at least three problems that may cause large \( \mathcal{VO} \) to be transmitted to clients and inefficient authentication at clients. Firstly, query features must be authenticated to ensure the correct graph IDs are fetched and intersected. The more the query features, the larger the \( \mathcal{VO} \). Unfortunately, none of the previous work minimizes the number of query features used in query processing. Secondly, all graph IDs involved in the intersections must be represented in the \( \mathcal{VO} \) so that the client can efficiently and correctly verify the intersections. Thirdly, the answer graphs do not generally form a range. In the worst case, each answer graph is authenticated separately. This makes direct applications of classical techniques (e.g., MHT [73] or signature chaining [81]) inefficient. Observe that both the query features and their graph IDs (described in the first two problems) dominate the I/O of the filtering phase and therefore, the problem of minimizing \( \mathcal{VOs} \) is directly related to minimizing I/O of the filtering-and-verification framework.

To solve it, we propose a novel authentication-friendly index called Intersection-aware Feature-subgraph Tree (IFTree) to address the aforementioned technical challenges. We then apply MHTs to IFTree called Merkle IFTree for efficient authentication. Specifically, for the first problem, in order to minimize the number of features used in the filtering phase, we propose a novel higher-order feature called Partially Overlapping Features (POF) which are themselves features composed by individual features. We propose to decompose a query into an optimal POF set such that fewest POFs (i.e., fewest intersections) are used in querying time and meanwhile, more individual features are implicitly used in the filtering phase. As a result, fewer graph IDs are fetched while the candidate set is minimized. As we shall see later, the number of fetched graph IDs in query processing on IFTree is 5 times smaller than that of a baseline. Moreover, the size of candidate set using IFTree is around 25% smaller than that of a baseline. Consequently, the \( \mathcal{VO} \) size and authentication time are reduced by a factor of 3.6 and 3.3, respectively. For the
second problem, we propose a compact matrix representation of intersection of graph IDs on MIFTree to form an enhanced authentication. Our experiments show that the compact representation improves the $\mathcal{VO}$ size and the authentication time by a factor around 2.5 and 3.4 (respectively). For the last problem, we determine the optimal ordering of graphs that are “intersect-able”. Our empirical study demonstrates that graphs needed to be authenticated form the fewest number of ranges and the corresponding $\mathcal{VO}$ size is reduced by around 40%. We observe that the overall improvement of the response time over the baseline is often more than an order of magnitude. We show that the energy saving on smartphone by using our proposed techniques is about 27% over the baseline.

In summary, the contributions of this part are listed as follows.

- We propose a novel higher-order feature, called partially overlapping feature for indexing graphs. We leverage these features to propose a novel index, namely Intersection-aware Feature-subgraph Tree (IFTree). For basic authentication, we apply MHTs to various structures of IFTree called MIFTree.

- We propose a novel matrix representation of intersection of graph IDs for enhanced authentication.

- We cluster the graphs that are “intersect-able” by adopting approximation algorithms.

- We conduct extensive experiments with real and synthetic datasets to demonstrate the effectiveness and superiority of our proposed methods.

1.4 Structure-preserving Subgraph Query Services

In some applications [14, 15, 47, 58], the $\mathcal{SP}$ may be honest-but-curious. That means the $\mathcal{SP}$ performs the query evaluation correctly (data integrity is warranted), but it may be interested in some private information of the data instead. Therefore, the confidentiality of data in $\mathcal{SP}$ may be seriously threatened. We note that there is a bloom on the research on query processing with privacy preservation for protecting the confidentiality of the data.
in the past decade. For example, in the literature of relational databases [13, 46], spatial databases [18,47] and graph databases [15]. However, up to date, private subgraph query has not yet been studied before. In this part, we aim to propose a privacy aware subgraph query service, which preserve the structure (i.e. topological information) of both graph data and query graph. Note that this part still focuses on transactional graph databases.

**Motivating example.** Consider a pharmaceutical company with revenue that depends mostly on the invention of health care products. The company may have discovered new compounds for a new product. To save laboratory work, it may query the compounds from proprietary biological pathway networks to check whether it is possible for the ingredient compounds to form other compounds via certain chemical reactions (a structural pattern from the network). However, on the one hand, the company is reluctant to expose the queries (the ingredients) to the SP, as it may apply for patents for the synthesis. On the other hand, the owner of the pathway networks may not only lack the expertise to host query services but may also be reluctant to release the networks to the public. The owner is willing to release it to paid users only. Hence, it is crucial to protect both the queries and the network from the SP. Such privacy concerns also arise from social networks and biological networks, among many other applications.

![Diagram](image)

**Figure 1.2:** Overview of the system model for structure-preserving subgraph query service over transactional graph databases.

In this thesis, we investigate that the query client may prefer not to expose the structure of query graphs to the SP, and meanwhile, the data owner may not want the SP to be able to infer the structure of their graph data. The fundamental problem being studied is to **evaluate subgraph query at the SP with a preservation of the structures of both the**
query graphs and graph data in the paradigm of the query services. In particular, this part aims to protect the adjacency matrices of the data graph and queries from the \( \mathcal{SP} \). To our knowledge, such a problem has never been addressed before.

The intrinsic difficulty of this problem is that the \( \mathcal{SP} \) cannot optimize query processing by directly using the structures of the graph, since such information cannot be exposed. However, most of the existing subgraph isomorphism algorithms (e.g., VF2 [100], QuickSI [94] and Turbo\_iso [39]) for the query services must traverse the graph, which by definition leaks structural information. A naïve method is to transfer the entire database to the client for query processing. However, it is inefficient when the database is large.

Our techniques for a structure-preserving sublso (denoted as SPsublso) are derived from the Ullmann’s algorithm [99], a seminal algorithm for subgraph isomorphism. We revise the Ullmann’s algorithm into three steps that form the foundation of our techniques. (1) Enum enumerates all possible subgraph isomorphism mappings \( M_i \)s from query graph \( Q \) to data graph \( G \); (2) Match verifies if the mapping \( M_i \) is valid or not; and (3) Refine reduces the search space of \( M_i \)s by degree and neighborhood constraints. The benefits of adopting the Ullmann’s algorithm are twofold: (1) the query evaluation between \( Q \) and \( G \) is mostly a series of matrix operations between their adjacency matrices \( M_Q \) and \( M_G \). It does not require traversals on structures; and (2) its query evaluation requires simple structures. This makes the privacy analysis simpler.

Specifically, to facilitate structure-preserving computations, we first transform sublso into a series of mathematical computations, denoted as Tsublso. Tsublso comprises three steps, corresponding to sublso: (1) TEnum enumerates all \( M_i \)s; (2) TMatch verifies the validity of \( M_i \) by additions and multiplications using \( M_Q \) and \( M_G \), where \( M_G \) is the complement of \( M_G \); and (3) TRefine reduces the search space of \( M_i \)s by inner products on our proposed static indexes \( SI_Q \) and \( SI_G \) of \( Q \) and \( G \), where \( SI_Q \) (\( SI_G \)) is an ensemble of \( h \)-hop information of each vertex of \( Q \) (\( SI_G \)) represented by a bit vector.

The major benefit of these three steps of Tsublso is that only mathematical operations are involved, which allows an adoption of private computations in encrypted domains. Based on Tsublso, we present our novel structure-preserving sublso (SPsublso).
In particular, we first propose a new private-key encryption scheme, namely cyclic group based encryption scheme (CGBE), to encrypt $M_Q$ and $M_G$ as $M_{Q_k}$ and $M_{G_k}$. Then, we propose SPMatch involving the additions and multiplications under CGBE to check the validity of each mapping $M_i$, with negligible false positives. Further, the computation results under CGBE can be aggregated to reduce communication overheads between the client and the $SP$. We prove that CGBE is perfectly secure under chosen plaintext attack and the $SP$ cannot learn any structures from SPMatch.

Next, we propose SPEnum which optimizes the mapping enumeration by introducing a protocol that involves the client’s participation, who informs the $SP$ useless enumerations. In addition, to optimize SPSublo, we develop SPRefine which exploits private inner products on the static indexes to derive a refinement that reduces the number of possible mappings. The indexes of the graphs are computed and encrypted offline, whereas those of the queries are computed once by the clients online. We analyze the effects of these optimizations on the probabilities that the $SP$ may correctly determine graph structures. Therefore, the clients may tune the trade-off between performances and privacy requirements.

To summarize, the contributions of this part are as follows:

- We transform the Ullmann’s algorithm $sublo$ as $Tsublo$. It only involves a few mathematical computations, such that its private version can be proposed and analyzed;

- We propose a structure-preserving $sublo$ (SPsublo) based on $Tsublo$, consisting of SPMatch, SPEnum and SPRefine. Specifically, we propose CGBE for SPMatch, which supports efficient encryption and decryption, partial additions and multiplications, and aggregation of computation results. We propose a protocol for SPEnum that involves the client to eliminate useless mappings. We propose SPRefine that exploits private inner products of static indexes to further optimization;

- We analyze the privacies of SPMatch, SPEnum and SPRefine; and

- We conduct detailed experiments to verify that SPsublo is efficient and our opti-
mizations are effective.

1.5 Asymmetric Structure-preserving Subgraph Query Service for Large Graphs

As motivated by Sec. 1.4, SPs may be curious about the queries’ privacy. Subgraph queries that preserve the query structure over large graphs has to be studied. We motivate the problem with an application scenario, which does not require domain knowledge. Others can be found in network medicine \(^1\) and patterns in communication networks.

![Figure 1.3: Overview of the system model for asymmetric structure-preserving subgraph query service over large graphs.](image)

**Motivating example.** Law enforcement agencies are increasingly using social media to solve crimes. According to a recent survey\(^2\) of 1,221 federal, state and local law enforcement who use social media, four out of five officials used social media to solve crimes. Suppose a law enforcement agency is investigating a set of suspicious individuals over a public social network (e.g., Cloob, which is a Persian-language social networking website, mainly popular in Iran) held in a third party SP. In order to monitor the online activities of these individuals with one another, the agency wishes to glean information related to interactions between them on the network by issuing a subgraph query representing the relationships between the individuals. Unfortunately, it is possible that the SP may have been infiltrated by friends or sympathisers of these individuals. Hence, in order to protect the privacy of the intent of the agency from the SP, the agency cannot expose the subgraph query directly, especially the query structure (i.e., specific relationship pattern between the individuals). How can the agency glean relevant information using a


subgraph query while preserving its topological privacy?

It is worth highlighting that the intrinsic technical challenge of this research direction is that although the data graph is available to the SP, the SP cannot optimize the queries by directly exploiting the structure of the query graphs. In comparison, recent subgraph isomorphism algorithms (e.g., VF2 [100], Turboiso [39] and QuickSI [94]) intensively utilize the query graphs for optimization, which by definition, leaks their structural information. More recently, the work reported in [109] supports “structureless” graph queries. However, the structure is automatically generated by a ranking model and the SP is aware of the queries.

One may also attempt to solve the problem with a naive solution in which the SP exhaustively traverses all of the data graph to enumerate all candidate mappings (i.e., possible mappings) between the query and the graph and return them to the client for verification. The intuition is that since the query structure is not exploited, its privacy is preserved. However, this is infeasible because the number of candidate mappings is exponential to the graph size in the worst case.

The first challenge of this research is then “how to reduce a large data graph and subsequently the number of candidate mappings for verification, without exposing the query structure?”. Our first idea is to determine the minimized candidate subgraphs that contain at least a candidate mapping. Then candidate mappings are enumerated from those subgraphs instead of the original graph. In particular, we propose optimizations that use novel neighborhood containment of data vertices to minimize the subgraphs. Second, we determine subgraphs (called candidate matchings) from a candidate subgraph, where candidate mappings are enumerated. In comparison, in previous work [39,94,100] where privacy is not a concern, the matching (i.e., the query graph) is known. We propose a subgraph cache and use neighborhood equivalent classes to further minimize the number of matchings and mappings.

The second challenge is “how to verify if a candidate mapping is a subgraph isomorphism mapping without leaking the query structure?”. We propose a query encoding scheme and adopt an encryption scheme for query graphs. With these, we derive a basic
structure-preserving verification method that consists of a series of private matrix operations. Moreover, to minimize communication overheads, we propose to use the complement of the encoding for an enhanced verification method for queries of bounded sizes.

The contributions of this part are summarized as follows:

- At query time, we first propose a new candidate subgraph exploration in the absence of query structure, to reduce a large data graph for query processing. We propose further reducing the size of candidate subgraphs by using neighborhood containment.

- Since candidate matchings are determined from candidate subgraphs, we propose a subgraph cache to prune the candidate matchings that are enumerated.

- We propose a robust encoding scheme and its verification method. We propose a model for the client to determine a proper encoding for his/her query.

- We conduct extensive experiments with real datasets to investigate the effectiveness and efficiency of our proposed methods.

1.6 Thesis Organization

The rest of this thesis is organized as follows. In Chapter 2, we present the related works. We study the authenticated subgraph query service over transactional graph databases in Chapter 3. In Chapter 4, we propose the structure-preserving subgraph query service for transactional graph database. Chapter 5 presents the asymmetric structure-preserving subgraph query service for large graphs. Chapter 6 concludes this thesis and discusses directions for future work.
Chapter 2

Related Work

In the previous chapter, we have introduced the problems that will be studied in this thesis. In this chapter, we present a survey of some recent works that are relevant to those problems.

2.1 Security-aware Graph Queries and Management

2.1.1 Authenticated Graph Query

Although there are several efforts in the literature on query authentication for relational and range queries [19, 66, 81], stream queries [83, 111], spatial queries [110], XML queries [28], text search [82], and multi-dimensional queries [23], very few work focus on authentication of graph query processing.

Yiu et al. [114] propose authentication of shortest path queries on road networks. However, the ordering of objects in road networks can be determined offline, e.g., by network-based distance. Such ordering is absent in graph databases in general and it is not clear how to adopt this work to subgraph queries. Kundu et al. [61–64] propose a series of methods for a closely related problem. They verify the authenticity of a given portion of data (subtree/subgraph that users’ have the right to access to) without any leakage of extraneous information of the data (tree/graph/forest). They optimize the signature needed and recently propose a scheme that uses one signature [61, 62]. However, in our problem setting, the portion of the data retrieved is the answer of a client’s query, which
is yet to be processed by an untrusted service provider. Therefore, the client is required to authenticate both the soundness and completeness (see SubSec. 3.2.1) of the portion of retrieved data. Search DAGs (Directed Acyclic Graph) [72] is a generalized model for authenticating a large class of data structures, e.g., binary trees, multi-dimensional range trees and tries. However, subgraph query processing can hardly be efficiently cast into a DAG search. Moreover, Peng et al. [86] propose authenticated subgraph similarity search over outsourced graph data.

2.1.2 Privacy-preserving Graph Query

There is a bloom on the research on query processing with privacy preservation for protecting the confidentiality of the data in the past decade. For example, in the literature of relational databases [13, 46], spatial databases [18, 47]. However, very few work focus on privacy-preserving graph databases.

Cao et al. [15] propose to support subgraph query over an encrypted database with a number of small graphs. Their work protects the privacy of the query, index and data features. This work does not solve the problem of the subgraph isomorphism testings of the candidate graphs. Such testings are required to be performed at the client side. Cao et al. [14] study tree pattern queries over encrypted XML documents. The traversal order for each query is predetermined. In the context of graphs, the order cannot be predetermined. Firstly, they predetermine the traversal order of both queries and XML documents and encode them as bit strings by such ordering. Secondly, they transform the pattern query processing between each query and XML document into the inner product on the encoding. Thirdly, they adopt the private inner product from [15, 103] to encrypt the encoding of of queries and XML documents. Thereby, the structure are then preserved from the SP. However, in the context of subgraph isomorphism, the traversal order of the queries and graph data are not given. We cannot apply their techniques directly. Fan et al. [112, 113] propose privacy-preserving reachability query services over encrypted index and data to preserve both of the query and graph structure, under the same system model of this work. He et al. [43] analyze the vertex reachability of the graph data,
with the preservation of edge privacy. Kundu et al. [61] propose a series of methods to verify the authenticity of a *given portion of data* without any leakage of extraneous information about the data (tree/graph/forest). Gao et al. [33] propose neighborhood-privacy protected shortest distance in the paradigm of cloud computing. This method aims to preserve all of the neighborhood connections and the shortest distances between each two vertices in outsourced graph data. However, it allows some connection and distance information between vertices to be exposed. Mouratidis et al. [75] propose the shortest path computation with no information leakage by using the PIR [24] protocol. The high computational cost of PIR is known to be a concern. Karwa et al. [57] present some novel algorithms for releasing *subgraph counts* of a graph by satisfying the differential privacy of edges, which requires that the presence or absence of a certain edge be hidden.

### 2.1.3 Privacy-preserving Graph Publication

There are many studies on privacy preserving graph publication recently. Most of them mainly focus on certain structure anonymization of the graph data, e.g., [20, 68, 120, 121]. The basic idea for most of their techniques are based on the modification of the original graph in order to make it satisfy the predetermined security requirement, as well as to maintain some utilities of graphs (*e.g.*, the distribution of degree, lengths of shortest path and transitivity). Unfortunately however, such utilities captured from those anonymized graphs cannot be used for subgraph queries analysis. For example, [20] transforms the original graph into $k$ disjoint isomorphic subgraphs, where the original structure of the graph has already been changed and can hardly be used to support subgraph query.

### 2.2 Graph Query Processing and Indexing

#### 2.2.1 Indexing Techniques for Graph Query Processing on Transactional Graph Databases

To evaluate subgraph queries on transactional graph databases, a large number of indexing techniques have been proposed in the literature. Those efforts can be roughly
classified into two approaches, namely feature-based approaches [22, 60, 94, 95, 106, 116] and non-feature-based approaches [42, 122]. Recently, iGraph [40] implemented these techniques on a common platform and reported that former approaches outperform latter approaches in most cases. Moreover, gIndex [106] performs the best among all the existing approaches in most of the test settings. Therefore, we adopt the feature-based approach in Chapter 3.

In particular, majority of feature-based techniques adopt a filtering-and-verification framework [22, 60, 94, 95, 106, 116] consisting of two key steps. (1) In the filtering phase, the query is decomposed into a set of individual features and an index is searched with those features. Examples of features are subgraphs, which can be generated using mining tools such as gSpan [105] and CAM code [49]. The search of each individual feature yields a set of graphs (represented by graph IDs) containing this searched feature. The sets of graphs are intersected to form a candidate set (a superset of answers). (2) In the verification phase, each graph in the candidate set is checked by an exact subgraph isomorphic algorithm to compute the final result set.

2.2.2 Subgraph Isomorphism Algorithm

As subgraph isomorphism algorithm is the crucial component for subgraph query services, in this part, we will review numbers of classical algorithms.

Ullmann [99] is a seminal algorithm for subgraph isomorphism. Its basic idea is the search with backtracking with respect to the matrix that represents possible isomorphic relationships. In the recent decade, several algorithms (e.g., VF2 [100], QuickSI [94], Turbo_{iso} [39] and BoostIso [90]) have been proposed to enhance the Ullmann’s algorithm. They all require to traverse the query on graph data. For instance, VF2 [100] relies on a set of state transitions and traversals on the graph and query. QuickSI [94] optimizes the ordering in traversals of graphs. Turbo_{iso} [39] exploits neighborhood information and local regions of vertices to further optimize query performance. Turbo_{iso} involves determining an optimal traversal in query processing. However, the traversals themselves carry structural information, which makes privacy preservation complicated if it is possi-
ble at all. BoostIso [90] exploits vertex relationship in speeding up the performance. Wu et al. [109] supports structureless graph queries, as the query structure is automatically formulated. However, the queries are known to the $\mathcal{SP}$. 
Chapter 3

Towards Efficient Authenticated Subgraph Query Service in Outsourced Graph Databases

Data integrity plays an important role in Quality of Services as motivated in Chapter 1. In this chapter, we propose authenticated subgraph query services that preserve the data integrity in outsourced graph databases.

3.1 Backgrounds and Preliminaries

In this section, we first discuss the background to subgraph query processing and query authentication. We then formulate the problem studied. A baseline approach and an overview of our solution are discussed.

3.1.1 Data Model

This part assumes undirected labeled connected graphs. For simplicity, we may use the term graphs to refer to them. A graph is denoted as \( g = (V, E, \Sigma, l) \), where \( V(g) \), \( E(g) \), \( \Sigma \) and \( l \) are the set of vertices, the set of edges, the set of labels of vertices and edges and the function that maps a vertex or edge to a label, respectively. We use \( |g| \) to denote the
size of graph $g$, where $|g| = |E(g)|$. Following the literature of a popular stream of graph databases [22,42,60,94,95,106,116,122], we consider graphs of modest sizes.

### 3.1.2 Subgraph Query

**Definition 3.1.1.** Given two graphs $g = (V, E, \Sigma, l)$ and $g' = (V', E', \Sigma', l')$, a subgraph isomorphism from $g$ to $g'$ is an injective function $\varphi : V(g) \rightarrow V(g')$ such that

- $\forall u \in V(g), \varphi(u) \in V(g'), l(u) = l'(\varphi(u))$; and
- $\forall (u, v) \in E(g), (\varphi(u), \varphi(v)) \in E(g'), l(u, v) = l'(\varphi(u), \varphi(v))$.

Subgraph query can be formally defined in Def. 3.1.1. We say a graph $g$ is a subgraph of another graph $g'$ if there exists a subgraph isomorphism from $g$ to $g'$, denoted as $g \subseteq g'$ or $\text{subIso}(g, g') = \text{true}$. It is known that deciding whether $g$ is the subgraph of $g'$ is NP-hard. Subgraph query processing can be described as follows.

**Definition 3.1.2.** Given a graph database $G = \{g_1, g_2, ..., g_n\}$ and a query graph $q$, we want to determine the query answers $R_q = \{g_i | \text{subIso}(q, g_i), g_i \in G\}$.

**Subgraph query paradigms.** Two query paradigms for subgraph queries have been proposed: feature-based (e.g., [22,60,94,95,106,116]) and non-feature-based (e.g., [42,122]) indexes. From Sec. 2 above, iGraph [40] concludes that the former often outperforms the latter. This work contributes to the feature-based approaches.

Feature-based approaches index graphs by their individual features. The term *individual feature* is used to refer to those proposed previously, as the one we put forward comprise individual features that form “higher-order” features. An index of this approach uses these features as the search keys for the graphs that contain them.

A well-received query paradigm for feature-based approaches is the filtering-and-verification framework [22,60,94,95,106,116]. Early work on subgraph query processing such as Shasha et al. [95] proposes *filtering graphs via paths* and then *verifying* the remaining graphs through $\text{subIso}$. Some later works [22,60,94,95,106,116] proposed innovative solutions that follow such a framework. To illustrate the filtering-and-verification
framework, we present a seminal index called gIndex [106] which is shown efficient in many cases [40]. gIndex proposes discriminative frequent features as individual features, denoted as $F$, for indexing. A discriminative frequent feature $f, f \in F$, is

- a subgraph whose size is smaller than or equal to $\text{maxL}$, where $\text{maxL}$ is a user-defined maximum feature size;

- a frequent feature that $|D_f| \geq \text{SISF}(|f|)$, where $D_f = \{f \mid f \subseteq g, g \in G\}$, $|D_f|$ is called the support of $f$ and SISF is a user-defined Size-Increasing-Support-Function of each $f$; and

- discriminative, s.t., $\frac{|\bigcap_{f' \in F \wedge f' \subseteq f} D_{f'}|}{|D_f|} \geq \text{dr}$, where dr is a user-defined discriminative ratio.

The function SISF returns a support that increases with the input feature size. gIndex sets SISF$(1) = 1$ by default. SISF gives the flexibility to allow indexing with infrequent features.

The individual features are represented by a canonical string called minimum DFS code [105] and gIndex is a prefix tree of the minimum DFS codes. gIndex processes queries in two phases. (1) Filtering: enumerate the maximum individual feature set $F_q$ from $q$, where $F_q = \{f \mid f \subseteq q, f \in F, \exists f', s.t., f \subseteq f', f' \subseteq q\}$, and filter out the graphs that do not contain a feature in $F_q$ to obtain the candidate set $C_q$ by performing the following intersections:

$$C_q = \bigcap_{f \in F_q, F_q \subseteq F} D_f$$

(3.1.1)
(2) Verification: determine the query answers $R_q$ from the candidate set by invoking $\text{subIso}$, where $R_q = \{g|q \subseteq g, g \in C_q\}$.

It is worth noting that the intersections in the filtering phase are performed on graph IDs whereas $\text{subIso}$ in the verification phase is invoked with graph data. Therefore, all previous indexes (see [22, 60, 94, 95, 106, 116]) propose innovative ideas to filter more non-answer graphs that aim to minimize the candidate set $C_q$.

Example 3.1.1. We illustrate the filtering-and-verification framework with an example in Fig. 3.1. The upper half of Fig. 3.1 shows the gIndex constructed from a set of individual features mined from $G$ in Fig. 1.1, $F = \{f_1, f_2, \ldots, f_8\}$, where $\text{SISF}(1) = 1$, $\text{SISF}(2) = 2$, $\text{maxL}$ and $\text{dr}$ are set to 2 and 0.1, respectively. $f_r$ is an artificial root node. The lower half of Fig. 3.1 shows its query processing: Given a query graph $q$, the filtering phase first enumerates all the maximum individual features $F_q = \{f_3, f_7\}$ of $q$ and performs intersections of the graphs (via IDs) containing the individual feature(s) ($D_{f_3}$ and $D_{f_7}$) to compute the candidate set $C_q = D_{f_3} \cap D_{f_7} = \{g_1, g_4\}$. The verification phase invokes $\text{subIso}$ on each graph in $C_q$, and computes the answers $R_q = \{g_4\}$.

3.1.3 Query Authentication

Cryptographic primitives. Similar to other works on authentication, we assume a one-way collision-resistant hash function (e.g., SHA and MD5) is denoted as $h(x)$, where $x$ is a data value to be hashed and the hash value $h(x)$ is often referred to as the digest of $x$. It is infeasible to determine the preimage of a digest. We assume a public-key digital signature scheme, such as RSA, that guarantees the authenticity of a message or value. The signer has a private key ($SK$) and can produce a signed message $y = \text{sign}(x, SK)$. Any public user has a public key ($PK$) and can verify the message by decryption.

Merkle Hash Tree (MHT). The Merkle Hash Tree [73] is a classical authentication technique. The main idea of MHT is illustrated with an example shown in Fig. 3.2(a). It is a classical MHT built on data values $\{x_1, \ldots, x_4\}$. Each leaf node is associated with the digest (hash) of its data value, e.g., $H_{x_1} = h(x_1)$. Each internal node contains the digest of the concatenation of the digest of its child nodes, e.g., $H_{x_1,x_2} = h(H_{x_1} | H_{x_2})$. A data owner
signs the digest of the root node.

To authenticate a data value, e.g., \( x_2 \), the service provider sends to the client \( x_2 \) and a \( \mathcal{VO} \) that consists of the digests \( H_{x_1} \) and \( H_{x_3,x_4} \) and the signed root digest of \( H_r \). The client computes from the \( \mathcal{VO} \), \( H_{x_2} = h(x_2) \), \( H_{x_1,x_2} = h(H_{x_1}|H_{x_2}) \), and finally the root digest \( H_{x_1,x_4} = h(H_{x_1,x_2}|H_{x_3,x_4}) \). The client uses the data owner’s public key to compare \( H_{x_1,x_4} \) and the signed root digest. If they agree, \( x_2 \) has not been tampered with. \( \text{MHT} \) can be extended to authenticate a set of data values.

\( \text{MHT} \) has been generalized to a multi-way index (such as Merkle B-tree [66]) for database applications. Moreover, it has been embedded into index nodes (see the Embedded Merkle B-tree (EMB-tree) [66]) to minimize \( \mathcal{VO} \) sizes. Fig. 3.2(b) shows a search tree embedded with an \( \text{MHT} \). The data in the \( \text{MHT} \) are \( \{x_1, \ldots, x_4\} \), the search keys are \( \{1, 2, 3, 4\} \).

- Each leaf node is associated with the search key and the digest (hash) of its data value, e.g., \( (1, H_{x_1}) \) where \( 1 \) is the search key of \( x_1 \); and

- Each internal node contains the search key and the digest of the concatenation of the digest of its child nodes, e.g., \( (2, H_{1,2}) \) where \( H_{1,2} = h(h(1)|H_{x_1})|h(h(2)|H_{x_2}) \).

Suppose that the search of the key 2 needs to authenticate. The \( \mathcal{VO} \) contains \( (1, H_{x_1}) \) and \( (4, H_{3,4}) \) and the data owner’s signature on the root digest \( H_r \). The client computes \( H_{x_2} = h(x_2) \), \( H_{1,2} = h(h(1)|H_{x_1})|h(h(2)|H_{x_2}) \), \( H_{1,4} = h(h(2)|H_{1,2})|h(h(4)|H_{3,4}) \), and finally the root digest \( H_r = h(h(4)|H_{1,4}) \). Similarly, by comparing the synthesized root digest and the data owner’s signature, the client verifies the authenticity of the data from the service provider. From the boundaries (i.e., 1 and 4) of the search, the client verifies that the search is correct.

In this work, we apply both kinds of \( \text{MHTs} \) (Figs. 3.2 (a) and (b)) to various structures of our index to minimize \( \mathcal{VO} \).
3.2 Problem Formulation and Overview

3.2.1 Problem Formulation

System Model. The system model follows the existing authentication framework, that comprises three parties — (i) data owner DO, (ii) service provider SP and (iii) querying client.

(i) The DO owns a graph database $G$. The DO or SP first generates an index to support subgraph query processing. Then, DO signs the root digest of the index. (ii) The SP receives a query $q$ from a client, processes it on behalf of DO and returns the answer graphs $R_q$ to the client. Since $SP$ may not be trusted, it is required to return not only $R_q$ but also a VO and the DO’s signature to the client. (iii) Upon receiving the VO, the client verifies the $R_q$ the SP returns. We assume the client has the public key of the DO for authentication. In particular, the client verifies the following:

- Soundness: all graphs in $R_q$ are answers and they are not tampered with, i.e., $\forall g \in R_q, g \in G \land q \subseteq g$; and

- Completeness: there is no graph that is not in $R_q$ but is an answer, i.e., $\nexists g \notin R_q, g \in G \land q \subseteq g$.

Threat Model. In our system model, the SP may not always be trusted. It may be a potential adversary or have been tampered with by attackers. In either case, we assume that the SP may alter the graph data or the index structure, introduce wrong answers, skip certain answers or abort the computation. An authentication framework is considered
secure if attacking it under this threat model is as hard as inverting a one-way collision-resistant hash function.

Given the above preliminaries, we are ready to formally present the problem statement.

**Problem statement.** *Given the above system and threat models, we seek an efficient authentication framework where the client may submit a subgraph query and verify the soundness and completeness of the answers returned by the service provider.*

### 3.2.2 Baseline Authentication — MgIndex

In this subsection, we derive a baseline technique from gIndex. We sketch the main ideas of this naïve authentication approach and discuss the drawbacks of such an approach. For a concise exposition, we present the details in set semantics, unless otherwise specified.

With reference to Formula 3.1.1 in Sec. 3.1.2, in order to authenticate the answer of the query \( q \), the client must authenticate the correctness of (i) the query features \( F_q \) and (ii) their graph IDs \( D_f \) (for all \( f \in F_q \)) in order to verify the authenticity of the candidate set \( C_q \). Therefore, the client can examine \( C_q \) to obtain the answer \( R_q \).

The baseline approach called MgIndex simply applies MHT to (i) the children of each index node of gIndex; and (ii) the graphs (with IDs) of \( D_f \) of each feature \( f \), respectively. The query processing of MgIndex is similar to that of gIndex but incorporates with \( \mathcal{VO} \) construction. More specifically, the \( \mathcal{VO} \) of MgIndex consists of three main parts:

\[
\mathcal{VO} = \mathcal{VO}_{\text{index}} \cup \mathcal{VO}_{C_q} \cup \psi_F
\]

1. \( \mathcal{VO}_{\text{index}} \) contains the digests that record the search of each individual feature \( f \in F_q \) during query processing and all the graph IDs (and the graphs’ hash values if the graphs are not present in \( R_q \)) of \( D_f \) for all \( f \in F_q \);

2. \( \mathcal{VO}_{C_q} \) contains the non-answer graphs in the candidate set, i.e., \( \mathcal{VO}_{C_q} = C_q - R_q \), denoted as \( C_q^{\neg R_q} \); and

3. \( \psi_F \) is simply the signature of the data owner.
Example 3.2.1. We use Example 3.1.1 to illustrate the V\(O\).

1. \(V\!O_{\text{index}}\) contains the digests that record the search of \(F_q = \{f_3, f_7\}\). Suppose the search locates \(f_7\) first. The \(V\!O_{\text{index}}\) includes the digests of the nodes \(f_1, f_3, f_4\) and \(f_5\). The digest of node \(f_2\) is computed by the client. When the search locates \(f_3\), the digest of \(f_3\) in \(V\!O_{\text{index}}\) is replaced by the actual content of the node \(f_3\). Thus, the client can verify \(f_3\). The graph IDs for each graph in \(D_{f_7}\) and \(D_{f_3}\) (i.e., \(\{1,2,4\}\) and \(\{1,4\}\) respectively), and the hash value of \(g_2\) are added to \(V\!O_{\text{index}}\); and

2. \(V\!O_{C_q}\) contains the non-answer graphs in the candidate set, i.e., \(V\!O_{C_q} = C_q - R_q = \{g_1\}\), where \(C_q = \{g_1, g_4\}\) and \(R_q = \{g_4\}\).

Regarding the authentication at the client side, firstly, the client rebuilds the root digest of \(MgIndex\) using \(V\!O_{\text{index}}\) and \(V\!O_{C_q}\) to verify that \(F_q\) and \(C_q\) are not tampered with. Secondly, it enumerates the query again to verify that \(F_q\) is exactly \(f_3\) and \(f_7\) by using \(V\!O_{\text{index}}\). Thirdly, the client performs intersections on \(\{1,4\}\) and \(\{1,2,4\}\) to verify the correctness of \(C_q\). Finally, the client performs the \text{subIso} tests to verify \(g_4\) is the answer but not \(g_1\).

Below, we provide the details of the baseline approach for authenticated subgraph query processing algorithm (Alg. 1). These details supplement the verbose pseudo-code, that is used to construct the running example in Example 3.2.1.

The overall authentication algorithm (\text{auth}_{\text{MgIndex}}). The overall algorithm can be described as follows. The inputs of Alg. 1 are the query graph \(q\) and \(T_F\), where \(T_F\) is the prefix tree of features \(F\). The outputs are the query result \(R_q\) and their \(V\!O\). It first finds all the maximal features \(F_q\) by using \text{find\_maxfeatures} (Line 2). \text{find\_maxfeatures} (to be elaborated next) is a traversal on the search tree \(T_F\) that constructs \(V\!O_{\text{index}}\). After computing \(F_q\), it then computes the candidate set \(C_q\) (Lines 3-4) by intersections. In Lines 5-10, \(V\!O_{\text{index}}\) is modified according to the candidate answer determined in Lines 3-4. The construction of \(V\!O_{C_q}\) is presented in Lines 11-15. Finally, Alg. 1 generates the total \(V\!O\) and returns with \(R_q\) (Lines 16-17).
Algorithm 1 auth_MgIndex(q, T_F)

Require: q is a query graph, the prefix tree T_F of features F

Ensure: R_q, VO

1: Initialize R_q = {}, C_q = G, VO_index = [], VO_C_q = {}
2: F_q = find_maxfeatures(q, T_F, VO_index)
   /* compute C_q */
3: for each f ∈ F_q
4:   C_q = C_q ∩ D_f
   /* construct VO_index */
5: for each f ∈ F_q
6:   Initialize a list L = []
7: for each g_j ∈ D_f
8:   if g_j ∈ C_q then L = L ⊕ j /* append ID */
9:   else L = L ⊕ (j, H_{g_j}) /* append ID and digest */
10: VO_index[f] = (f, L)
   /* construct R_q and VO_C_q */
11: for each g ∈ C_q
12:   if subIso(q, g) = true
13:      R_q = R_q ∪ g
14:   else /* construct VO for non-answer */
15:      VO_C_q = VO_C_q ∪ g
16: VO = (VO_index, VO_C_q, ψ_F)
17: return R_q and VO

Enumeration of features (find_maxfeatures). Alg. 2 presents the algorithm for find_maxfeatures.

The algorithm is presented in the style of gIndex [106], which is a depth first search of minimum DFS order [106] to enumerate all the maximal features F_q of q. In Alg. 2, the only difference from [106] is that it needs to record the VO_index while searching for the features, which will be used in Alg. 1. In Line 1, VO_index′ is the VO_index at the boundary of the search; S is a set of features enumerated so far; F_q is the maximal features of q; and U is the edges of q covered by S. The algorithm first sorts all the (individual) edges in q ordered by minimum DFS order (Line 2). It records the feature and the associated digest for each child node of the root of MgIndex (Lines 3-5) in VO_index, as they are at the boundary of the current search. Alg. 2 then invokes the traversal algorithm auth_DFS.
(Lines 6-8) to enumerate the features of \( q \). After generating all the features of \( q \), the algorithm then computes and returns the maximal features \( F_q \) (Lines 9-10).

The traversal pseudo-code of a prefix search tree \( \text{auth}\_\text{DFS} \). The Procedure 2.1 is a depth first search procedure with generating the features of \( q \) and the \( \mathcal{V}\mathcal{O}_{\text{index}} \). At each traversal step, if the current feature \( s \) is not the minimum [106], then the current recursion is terminated (Lines 11-12). If \( s \) is in \( T_F \) (Line 13), then \( s \) is a feature of \( q \) (Line 13). The edge \( e \) is covered by \( s \) and \( s \) is added to \( S \) (Lines 14-15). The feature and the associated digest for each child node of \( s \) is involved in \( \mathcal{V}\mathcal{O}_{\text{index}} \) (Lines 16-18). Then, \( \text{auth}\_\text{DFS} \) proceeds to each child \( c \) of \( s \) in \( q \) (Lines 20-22), again in the minimum DFS order.

Example 3.2.2. We use Example 3.1.1 to illustrate the search of the query features in Alg. 2. The search \( \text{find}\_\text{maxfeatures} \) starts at the artificial root node \( f_r \). The child nodes of \( f_r \) are the current boundary nodes and they are recorded in \( \mathcal{V}\mathcal{O}_{\text{index}}' \) (Lines 3-5). The search proceeds to the child nodes of \( f_r \) with the minimum DFS order \( S^1 \) (Lines 6-8).

According to the example, the first edge \( e \) in \( S^1 \) is \((C,O)\). In the sub-procedure \( \text{auth}\_\text{DFS} \), since \( s \) is a minimum (Lines 11-12) and \( s \) is exactly \( f_2 \) (Line 13), where \( s = e \), \( e \) is added in \( U \) as covered (Line 14) and \( s \) is added in \( S \) as a feature of \( q \) (Line 15). \( s \) (i.e., \( f_2 \)) is the current visited node, and the child nodes of \( f_2 \) are the boundary nodes which are recorded in \( \mathcal{V}\mathcal{O}_{\text{index}} \) (Lines 16-19). The \( \text{auth}\_\text{DFS} \) then recursively expands the search to the child of \( s \) in \( q \) (Lines 20-22), i.e., expands \((O,H)\) and \((O,O)\) respectively. However, \( s \) expanded with \((O,H)\) is not a feature of \( q \) (Line 13). The traversal then proceeds to \( s \) expanded with \((O,O)\) (i.e., \( f_7 \)) and modifies \( \mathcal{V}\mathcal{O}_{\text{index}}' \).

After \( \text{auth}\_\text{DFS} \) finishes traversing the search tree rooted at \( f_2 \), the only edge of \( q \) not covered by \( S \) is \((O,H)\). \( \text{find}\_\text{maxfeatures} \) then proceeds to search \((O,H)\) as it is the next edge in \( S^1 \) (Line 6). Similarly, \( \text{auth}\_\text{DFS} \) traverses the subtree rooted at \( f_3 \) (Lines 7-8). After the traversal, \( S \) covered \( q \) (i.e., \( U = q \)) and the traversal terminates (Line 10).

Finally, \( \text{find}\_\text{maxfeatures} \) determines \( F_q = \{f_7,f_3\} \) from \( S \) and returns \( F_q \) (Lines 9-10).

The sketch of the baseline approach reveals the performance bottlenecks of subgraph
Algorithm 2 \text{find}\_\text{maxfeatures} (q, T_F, V_O\text{index})

\textbf{Require:} \( q \) is a query graph, the prefix tree \( T_F \) of features \( F \), \( V_O\text{index} \) is the \( V_O \) records the search

\textbf{Ensure:} \( F_q \)

1: Initialize \( V_O\text{index}' = [ ] \), \( S = \{ \} \), \( F_q = \{ \} \), \( U = \{ \} \)

2: \( S^1 \) is a list of edges \( e \) of \( q \) ordered by the minimum DFS order

3: \textbf{for each} child \( f \) of \( f_r \) in \( T_F \) /* \( f_r \) is the root node of \( T_F \) */

4: \( V_O\text{index}'[f] = (f, \mathcal{H}_f) \) /* boundary nodes */

5: \( V_O\text{index}'[f_r] = (f_r, V_O\text{index}' \) /* visited nodes */

6: \textbf{for each} edge \( e \) in \( S^1 \), \( e \notin U \land q \neq U \)

7: \( s = e \)

8: \text{auth}\_DFS (\( e, s, S, V_O\text{index}' \), \( q, T_F, U \))

9: compute \( F_q \subseteq S \), s.t., \( F_q \) is a set of maximal features

10: return \( F_q \)

Procedure 2.1 \text{auth}\_DFS (\( e, s, S, V_O\text{index}' \), \( q, T_F, U \))

11: if \( s \neq \text{mindfs}(s) \)

12: \textbf{return}

13: if \( V_O\text{index}[s] \neq \emptyset \) /* \( s \) is a feature in \( T_F \) */

14: \( U = U \cup e \) /* mark \( e \) as covered */

15: \( S = S \cup s \)

16: initialize \( V_O\text{index}' = [ ] \)

17: \textbf{for each} child \( f \) of \( s \) in \( T_F \)

18: \( V_O\text{index}'[f] = (f, \mathcal{H}_f) \) /* boundary nodes */

19: \( V_O\text{index}'[s] = (s, V_O\text{index}' \) /* visited nodes */

20: \textbf{for each} child \( c \) of \( s \) in \( q \land q \neq U \)

21: \( e' = c - s \)

22: \text{auth}\_DFS (\( e', c, S, V_O\text{index}' \), \( q, T_F, U \))

query authentication. The more features (i.e., more intersections) are used to determine \( C_q \) (Formula 3.1.1), the more \( V_O\text{index} \) is needed to authenticate \( F_q \) and the more graph IDs of \( D_f \) are introduced. This not only leads to large \( V_O \), but also requires high time costs to authenticate them. Similar to query processing, query authentication also requires to minimize \( C_q \) as the non-answer graphs (not the IDs) are included in \( V_O\text{C}_q \).
### Table 3.1: Frequently Used Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(q, g, G)</td>
<td>a query graph, a graph data and a graph database</td>
</tr>
<tr>
<td>(f)</td>
<td>an individual feature</td>
</tr>
<tr>
<td>(F)</td>
<td>a set of all individual features in (G) or the prefix tree that indexes all features of (G)</td>
</tr>
<tr>
<td>(p, P)</td>
<td>a partially overlapping feature POF and a set of all POFs in (G)</td>
</tr>
<tr>
<td>(D_p)</td>
<td>a set of graphs that each of which contains (p)</td>
</tr>
<tr>
<td>(ID(D_p))</td>
<td>a list of IDs of the graphs in (D_p)</td>
</tr>
<tr>
<td>(P_q, P_{q^{opt}})</td>
<td>the POFs of (q) and the optimal decomposition of POFs</td>
</tr>
<tr>
<td>(C_q, R_q, C^R_q)</td>
<td>the candidate set, the answer set and the non-answer candidate set</td>
</tr>
<tr>
<td>(\mathcal{H}_g, \mathcal{H}_p)</td>
<td>a digest of graph (g) and node((p)) of MIFTree</td>
</tr>
<tr>
<td>(\mathcal{H}_{D_p})</td>
<td>a root digest of the classical MHT on all graphs in (D_p)</td>
</tr>
<tr>
<td>(\mathcal{H}_p^r)</td>
<td>a root digest of the embedded MHT on node((p))’s child nodes</td>
</tr>
<tr>
<td>(\mathcal{VO}_{\text{index}})</td>
<td>the digests that record the search of features</td>
</tr>
<tr>
<td>(\mathcal{VO}_{C_q})</td>
<td>the non-answer graphs in the candidate set</td>
</tr>
<tr>
<td>(N_I, N_F)</td>
<td>the digest of MIFTree nodes and the digest of prefix (F) nodes</td>
</tr>
<tr>
<td>(I_M)</td>
<td>a set of subgraph isomorphism mappings from (q) to (g \in R_q)</td>
</tr>
<tr>
<td>(\psi_I, \psi_F)</td>
<td>a signature of MgIndex and MIFTree from (DO)</td>
</tr>
<tr>
<td>(p_{\text{min}})</td>
<td>the minimal POF in (P_{q^{opt}})</td>
</tr>
<tr>
<td>(M_p)</td>
<td>a binary matrix of (ID(D_p)) of (p) (for minimizing I/O)</td>
</tr>
<tr>
<td>(\mathcal{H}_{M_p})</td>
<td>a root digest of the classical MHT on (ID(D_p))</td>
</tr>
</tbody>
</table>

### 3.2.3 Overview of our Approach

In response to the drawbacks of the baseline approach, we propose more efficient authentication techniques. The frequently used symbols of our discussions are listed in Table 3.1.

To minimize \(\mathcal{VO}_{\text{index}}\) while keeping \(C_q\) small, we propose essentially to precompute some intersections offline, such that fewer intersections are involved at query time and hence need to be authenticated by clients. In particular, we propose higher-order features (Partially Overlapping Features POFs). In a nutshell, a POF consists of a set of overlapping individual features. If a data graph contains a POF, this implies it also contains those individual features in the POF. Hence, POFs are more selective than individual feature,
We propose the Intersection-aware Feature-subgraph Tree (IFTree) to index a graph database by POFs $P_q$. Merkle IFTree (MIFTree) is proposed by adopting MHTs on IFTree for basic authentication.

The overview of our solution is depicted in Fig. 3.3. ① The client issues the query graph $q$ to the $SP$. The $SP$ first enumerates all the POFs $P_q$ of $q$. We then study how to decompose $q$ into an optimal set $P_{opt}^q$ which has the fewest number of intersections and smallest $C_q$. ② $P_{opt}^q$ is then searched on MIFTree to obtain all the graph IDs of $D_p$, denoted as $\text{ID}(D_p)$, where $p \in P_{opt}^q$ and $D_p$ is a set of graphs that contain $p$. The candidate set $C_q$ is determined by intersecting $\text{ID}(D_p)$ as shown in Formula 3.1.1. ③ We derive a basic method to derive $\mathcal{VO}_{index}$ from MIFTree which is similar to MgIndex. ④ In addition, as the query graph size increases, so does the number of intersections. It is inefficient to include all $\text{ID}(D_p)$s, $p \in P_{opt}^q$ in a $\mathcal{VO}$. Hence, to minimize the $\mathcal{VO}$ needed to authenticate intersections, we propose an enhanced method that uses a compact representation $M_p$ for each $D_p$ of $p$. We only include the single smallest $M_p$, namely $M_{p_{min}}$ to $\mathcal{VO}$. $M_{p_{min}}$ itself must be authenticated by the client but the answer graphs indicated by $M_{p_{min}}$ may not fall into a range. We therefore analyze $M_p$ offline to cluster the “intersect-able” graphs in each $D_p$, $p \in P$, for an optimal ordering of the graphs stored in $D_p$. ⑤ For $\mathcal{VO}_{C_q}$, we include the non-answer candidate $C_q^{R_q}$ and the mappings $I_M$ between the query and its answers. ⑥ $DO$’s signature $\psi$ is added to $\mathcal{VO}$. The client and result in smaller candidate sets.
finally receives the $\mathcal{VO}$ to authenticate the answer.

3.3 Partially Overlapping Features

In this section, we derive the partially overlapping features (POFs) that aim to minimize the number of intersections involved in query time. The benefits are threefold. Fewer intersections are computed in query time; fewer graph IDs are fetched; and more individual features are implicitly involved and often lead to small candidate sets.

3.3.1 Types of Overlapping Features

Features can be composed in various ways. We derive POFs and call them higher-order features as they themselves are features and composed by individual features.

To describe POFs, we first present a few notations needed: Individual features $F$ can be features proposed by any existing works. We adopt discriminative frequent feature [106] as the individual feature in this work. We use $g$ and $F_g$ to denote a graph and its individual features. We call the subgraph of $g$ that is isomorphic to $f$ as an instance of $f$, i.e., $g \in D_f$. With these notations, we derive POFs. We start with the feature of multiple individual features.

**Definition 3.3.1.** A feature $\{f_1, \cdots, f_n\}$ is a co-existing feature of $g$ if $g$ contains an instance of $f_i$ for all $i \in \{1, \cdots, n\}$, where $\{f_1, \cdots, f_n\} \subseteq F_g$.

The definition above can be trivially extended to a database $G$. Let $\{f_1, \cdots, f_n\}$ be a co-existing feature of $G$, a graph $g$ in $G$ contains it iff $g \in D_{f_1} \cap \cdots \cap D_{f_n}$.

The next feature, namely overlapping feature, concerns not only the existence of features but also the overlapping of features.

**Definition 3.3.2.** $\{f_1, \cdots, f_n\}$ is an overlapping feature of $g$ if it is a co-existing feature of $g$ and there is a set $S:\{s_1, \cdots, s_n\}$ in $g$, where $s_i \in S$ is an instance of $f_i$, and $S$ forms a connected graph.

We remark that singleton sets $\{f_1\}$ (i.e., $n = 1$) are considered as “overlapping” features since each of their instance definitely forms a connected graph.
Example 3.3.1. Fig. 3.4 illustrates Defs. 3.3.1 and 3.3.2. In Fig. 3.4(a), \( \{f_4, f_7\} \) is a co-existing feature of \( g_1 \). In Fig. 3.4(b), \( \{f_2, f_7\} \) is an overlapping feature of \( g_1 \), as the instances of \( f_2 \) and \( f_7 \) not only exist but also overlap.

One may be tempted to derive more sophisticated features, e.g., by exploiting the topology graph of an overlapping feature. However, such features may introduce a high complexity in query processing. In this work, we adopt overlapping features. Moreover, consider overlapping features e.g., in Fig.3.4(b). The instances of \( f_2 \) and \( f_7 \) are completely overlapped. In practice, \( D_{f_7} \) is often a subset of \( D_{f_2} \). Indexing graphs with both \( f_2 \) and \( f_7 \) are often redundant. Hence, we propose partially overlapping features defined in Def. 3.3.3. An example is shown in Fig. 3.4(c).

**Definition 3.3.3.** \( p: \{f_1, \ldots, f_n\} \) is a partially overlapping feature (POF) of \( g \), if (1) it is an overlapping feature of \( g \) and (2) there does not exist \( f_i, f_j \in p \), s.t., for each instance \( s_i \) of \( f_i \) and \( s_j \) of \( f_j \), \( s_i \) is completely overlapping with (i.e., contained in) \( s_j \).

Singleton sets are considered POFs since (1) they are special cases of overlapping features and (2) no two features whose instances are completely overlapping. This subtle case has a practical implication: Clients may issue queries with exactly one feature and it may be indexed.

To specify the desired POFs for indexing, we define a user-specified constraint. In particular, POFs should be small in size and have certain minimum support from a database.

**Definition 3.3.4.** The constraint of POFs \( P \) is \((\text{maxSize}, \text{minSup})\), where maxSize and minSup are the maximum size and the minimum support of \( P \) in a database \( G \), i.e., \( \forall p \in P, |p| \leq \text{maxSize} \) and \( |D_p| \geq \text{minSup} \).

The number of all POFs of a database \( G \) is exponential to the number of features in
worst case. In practice, many POFs do not have sufficient support. We adopt an enumeration algorithm to compute all POFs that satisfy the user-defined \((\text{maxSize}, \text{minSup})\).

It is worth mentioning that the graphs indexed by a POF \(p : \{f_1, \ldots, f_n\}\) (denoted as \(D_p\)) are a proper subset of the graphs in \(D_{f_1} \cap \cdots \cap D_{f_n}\). Indexing with \(p\) may be viewed as precomputing the intersections. In the rest of this work, we use the term \textit{features} \(P\) to refer to POFs, whereas \(f_1, \ldots, f_n\) are referred to \textit{individual features}.

### 3.4 Intersection-aware Feature-subgraph Tree (IFTree)

In this section, we present IFTree that indexes a graph database \(G\) with all POFs that satisfy \text{maxSize} and \text{minSup}. We present the querying processing of IFTree, which is authenticated in Sec. 3.5.

#### 3.4.1 IFTree

IFTree is a prefix tree on POFs where each node represents a POF and points to a list of graph IDs. Recall from Def. 3.3.2 that each POF is a set of individual features. The subset operator \(\subset\) over all the POFs is a partially ordered set. To derive a search tree on the set, we assume that each individual feature has an ID and a POF \(p\) is represented by a \textit{string of IDs} of its individual features sorted in ascending order. We use \(\text{str}(p)\) to denote the string of \(p\). For example, let \(p = \{f_1, f_2\}\). \(\text{str}(p) = "1.2"\). We say \(p_i\) precedes \(p_j\), denoted as \(p_i \prec p_j\), \textit{iff} \(\text{str}(p_i)\) is a prefix of \(\text{str}(p_j)\). With such a representation of POFs, we define a prefix search tree called IFTree.
**Definition 3.4.1.** Intersection-aware Feature-subgraph Tree (IFTree) is a prefix search tree of POFs $P$ on a graph database $G$, denoted as $T_P : (\text{str}, \text{node}, V, E, \text{ID}, p_r)$, where

- $\text{str}$ is a function that $\text{str}(p)$ returns the string of $p$;
- $\text{node}$ takes a POF $p$ and returns the node of $p$ in $T_P$;
- $V = \{\text{node}(p_i) | p_i \in P\}$;
- $E = \{(\text{node}(p_i), \text{node}(p_j)) | p_i \prec p_j \land (\not\exists p'_i, p_i \prec p'_i \land p'_i \prec p_j)\}$. The children of a node($p_i$) are sorted in lexicographical order w.r.t $\text{str}$;
- $\text{ID}$ is a function that $\text{ID}(D_p)$ returns the list of IDs of the graphs in $D_p$; and
- $p_r$ is an empty POF $\emptyset$ and $\text{node}(p_r)$ is an artificial root node of the IFTree.

**Example 3.4.1.** Fig. 3.5 shows the IFTree of the POFs $P$ of $G$ of Fig. 1.1. Due to space constraints, we skip the enumeration that yields $P: \{p_1 \ldots p_{15}\}$. Each box of the tree represents a POF. The constraint of POF $\text{maxSize, minSup}$ is $(2, 2)$. Consider $p_9$. The IFTree has an edge between $p_1$ and $p_9$ but not $p_2$ and $p_9$ as $\text{str}(p_2) = "2", \text{str}(p_9) = "1.2"$ and therefore $p_2 \not\prec p_9$. To illustrate the processing of existing indexes and IFTree, let's assume that a query contains two individual features $f_2$ and $f_7$. $gIndex$ retrieves and intersects $D_{f_2}$ and $D_{f_7}$ whereas IFTree simply retrieves $D_{p_{11}}$.

### 3.4.2 Query Processing on IFTree

The query processing on IFTree is detailed in Alg. 3.\(^1\). It takes a query graph $q$, a graph database $G$, the prefix tree $T_F$ of features $F$ and the IFTree $T_P$ of $G$ as input. It determines all maximum individual features $F_q$ that fully cover $q$ (Line 2). From $F_q$, it computes all possible POFs $P_q$ from $F_q$ (Line 3) and determines the optimal POFs $P_{q}^{\text{opt}}$ from $P_q$ (Line 4), which shall be discussed shortly. For each POF $p$ in $P_{q}^{\text{opt}}$, the graphs of $D_p$ are retrieved by searching IFTree and maintained in a candidate set $C_q$ (Lines 5-6).

\(^1\)Some pseudocode in Alg. 3, e.g., Lines 1 and 2, are straightforward but verbose. Hence, for concise presentation, we present their main ideas in text.
For each graph in $C_q$, the algorithm verifies if it is in fact an answer (Lines 7-8). Following up Example 3.1.1, we use Fig. 3.6 to illustrate the query processing on IFTree (shown in Fig. 3.5) in the following discussion.

Algorithm 3 Query Processing ($q$, $G$, $T_F$, $T_P$)

**Require:** A query graph $q$, a graph database $G$, the prefix tree $T_F$ of features $F$ and the IFTree $T_P$ of $G$

**Ensure:** the answer set of $q$ $R_q$

1: Initialize $R_q$ to $\emptyset$ and $C_q$ to $G$
2: $F_q = \text{find max features}(q, T_F)$ // $F_q$ fully cover $q$
3: $P_q = \text{find POF}(q, F_q, T_P)$ // Enumeration
4: $P_{q_{\text{opt}}} = \text{opt POF MWSC}(F_q, P_q)$
5: for each $p \in P_{q_{\text{opt}}}$
6: $D_p = \text{search}(p, T_P)$; $C_q = C_q \cap D_p$
7: for each $g \in C_q$
8: if $\text{subIso}(q, g) = \text{true}$ then $R_q = R_q \cup g$
9: return $R_q$

It is worth noting that Alg. 3 involves two optimizations. The first one is similar to an existing work [106] — the query $q$ is decomposed into maximum individual features $F_q$ by using an enumeration method. $f$ is maximum in terms of $q$ if and only if there does not exist a larger $f'$ such that $\text{subIso}(f, f') = \text{true}$ and $\text{subIso}(f', q) = \text{true}$. Unlike previous work, we determine $F_q$ that fully covers $q$. When compared to non-covers, a cover $F_q$ is expected to be more selective and yields a small candidate set in the filtering phase. $F_q$ is then used to enumerate POFs, as indicated in the RHS of Fig. 3.6. For example, as in Example 3.1.1, gIndex computes $F_q$ as $\{f_3, f_7\}$. However, Alg. 3 determines $F_q$ as $\{f_2, f_3, f_7\}$ (in Line 2). Without $f_2$, $F_q$ does not fully cover $q$.

The second optimization is that an optimal decomposition $P_{q_{\text{opt}}}$ is determined from $F_q$. In the filtering-and-verification framework (e.g., Fig. 3.6), graph data are fetched from disk mainly in two steps: (i) when graph IDs of $D_p$’s are fetched from disk for performing intersections; and (ii) when candidate graphs are fetched for subiso tests. This leads to two competing objectives in computing $P_{q_{\text{opt}}}$. (i) On one hand, fewer intersections (i.e., fewer POFs in $P_{q_{\text{opt}}}$) on ID($D_p$) ($p \in P_{q_{\text{opt}}}$) are desirable to minimize I/O due to graph IDs of $D_p$’s. (ii) On the other hand, larger POFs (i.e., more individual features) in
$P^\text{opt}$ cover the query more and reduce the size of the candidate set $C_q$ and the I/O for fetching it. The objectives can be illustrated with an example. Suppose Alg. 3 (\texttt{find.POF} in Line 3) determines $P_q = \{p_2, p_3, p_7, p_{11}\}$. Two possible decompositions are $P_1^q = \{p_2, p_3, p_7\}$ and $P_2^q = \{p_3, p_{11}\}$. We may choose $P_2^q$ since the number of intersections and the candidate set are 2 and $\{g_4\}$, respectively. In contrast, those of $P_1^q$ are 3 and $\{g_1, g_4\}$, respectively. $P_2^q$ is in fact the optimal decomposition of $q$.

**Minimization of I/O by using $P^\text{opt}_q$.** The problem discussed above can be formulated as an optimization where both $|P^\text{opt}_q|$ and I/O are minimized. To present the problem, we define a binary matrix $M_q$ where each row $i$ represents the POF $p_i$ from all possible POFs $P_q$ of $q$, each column $j$ represents an individual feature $f_j \in F_q$ and each entry $M_q(i, j)$ is 1 if $f_j$ is in $p_i$, otherwise 0. The weight of each row $i$ of $M_q$ is $w_i = \frac{1}{\text{hamming}(M_q(i, *))}$, where $\text{hamming}(M_q(i, *))$ is the hamming weight that returns the number of 1s in the row $i$ of $M_q$. For instance, consider Fig. 3.6. $w_{11} = \frac{1}{\text{hamming}(M_q(11, *))} = 1/2$ as $p_{11} = \{f_2, f_7\}$.

**Definition 3.4.2.** Given a weight value $w_i$ to each row $i$ of $M_q$, $w_i = \frac{1}{\text{hamming}(M_q(i, *))}$, the problem of optimal decomposition of a query $q$ from $P_q$ is to determine $P^\text{opt}_q$, where $P^\text{opt}_q \subseteq P_q$ and $P^\text{opt}_q$ fully covers $F_q$ s.t. $\sum_{p_i \in P^\text{opt}_q} w_i$ is minimized.

The optimal decomposition addresses the above two objectives. (i) To minimize $\sum_{p_i \in P^\text{opt}_q} w_i$, fewer terms are included in the sum, which not only indicates fewer intersections in query processing, but also minimizes I/O due to graph IDs. (ii) For each $p_i$, the more 1s in $M_q(i, *)$, the more individual features it contains, the smaller $w_i$ and $D_{p_i}$. Therefore, using $p_i$ leads to a smaller candidate set.

**Proposition 3.4.3.** The problem of optimal decomposition of a query $q$ from $P_q$ is NP-hard.

**Proof.** The proof is established from a simple reduction from minimum weighted set cover problem. For the given query graph $q$, the universe set $U$ contains all the individual features in $F_q$, i.e., $U = F_q$. The set of subsets of $U$ is denoted as $S$, in our cases, $S = P_q$. For each $p_i$ in $S$, its weight is $w(p_i) = \frac{1}{|p_i|}$. A collection $S'$ of sets from $S$, which covers all the individual features in $U$ and minimizes the $\sum_{p_i \in S'} w(p_i)$, is the optimal
maximum features & fully cover $F_q = \{f_2, f_3, f_7\}$

$q$ enumerate

enumerate via IFTree $P_q = \{p_2, p_3, p_7, p_{11}\}$

$p_{11} = \{f_2, f_7\}$

$p_3 = \{f_3\}$

$P_q^{opt} = \{p_3, p_{11}\}$

filter via intersections

$C_q = D_{p_3} \cap D_{p_{11}} = \{g_4\}$

verify

$g_4$ enumerate

$R_q = \{g_4\}$

Figure 3.6: Subgraph query processing on IFTree

decomposition of $q$ from $P_q$, i.e., $S' = P_q^{opt}$. That means finding such collection $S'$, which is the optimal decomposition $P_q^{opt}$ of $q$ from $P_q$, is finding the minimum weighted set cover of $S$. Therefore, the problem of finding the optimal decomposition of $q$ from $P_q$ is NP-hard.

We adopt a classical heuristic algorithm for MWSC to solve the problem. The idea is simple: it iteratively chooses the POF with the smallest weight (covering the most number of uncovered features in $F_q$) and removes the covered features from $F_q$. It terminates when $F_q$ is empty (fully covered). This heuristic can be exemplified by the example $M_q$ shown in Fig. 3.6. Initially, $w_2 = w_3 = w_7 = 1$ as the hamming weights of $M_q(2, \ast)$ $M_q(3, \ast)$ and $M_q(7, \ast)$ are 1. $w_{11} = 1/2$ as $\text{hamming}(M_q(11, \ast)) = 2$. In the first iteration, $p_{11}$ is chosen. Since $f_2$ and $f_7$ are covered by $p_{11}$, they are removed from $F_q$ and the weights $w_2$, $w_3$ and $w_7$ are updated accordingly. In the second iteration, $p_3$ is chosen. All features in $F_q$ are covered and the algorithm terminates. $P_q^{opt}$ is $\{p_3, p_{11}\}$.

3.5 Merkle IFTree (MIFTree)

Thanks to the minimization of I/O by using $P_q^{opt}$, the query processing trace needed to be included in VOs is reduced when IFTree is adopted for query authentication. To facilitate efficient authentication, we propose to apply MHTs to IFTree to obtain Merkle IFTree (MIFTree). Recall that IFTree is a prefix tree for the string representations of POFs. The index nodes near the root of IFTree often have large fanouts, as those POFs may overlap with many other individual features to form larger POFs. Therefore, an MHT
is embedded to the children of each index node to minimize $\forall \mathcal{O}$. In addition, in practice, some POFs may index a large number of graphs. For instance, in the dataset AIDS, the number of graphs containing the POF of an index node near the root of the IFTree is 12% of the total number of graphs. When some of these graphs are selected into the candidate set in the filtering phase, a classical MHT is needed to efficiently authenticate these graphs. Hence, we propose the Merkle IFTree (MIFTree) as follows.

**Definition 3.5.1.** MIFTree is an IFTree extended with two kinds of MHTs: (i) An MHT is embedded to the child nodes of each node $(p)$ of MIFTree; and (ii) A classical MHT is built on top of all graphs (with graph IDs) in $D_p$ for each node $(p)$.

The rest of this section describes the signing of MIFTree in detail and a basic authentication of MIFTree.

### 3.5.1 Signing MIFTree

Similar to the majority of search trees for query authentication, we associate hash values/digests to the nodes of IFTree. The data owner $DO$ signs the root of the digest of MIFTree. Specifically, we formalize the digests and signatures of MIFTree below.

**Definition 3.5.2.** The digest of a data graph $g_i$ is defined as $\mathcal{H}_{g_i} = h(\text{mindfs}(g_i))$.

Graphs are cast into some (publicly known) canonical representation before their digests are computed. In this work, we adopt the minimum DFS code [105], denoted as mindfs, but other representations may also be adopted.

**Definition 3.5.3.** The digest of a node node$(p)$ of MIFTree is $\mathcal{H}_p = h(h(\text{str}(p))|\mathcal{H}_{D_p}|\mathcal{H}_p^r)$, where

- $\text{str}(p)$ is the string of $p$;
- $\mathcal{H}_{D_p}$ is the root digest of the classical MHT of ID$(D_p): [j_1, \cdots , j_m]$. The data in the MHT are $\{(j_1,g_{j_1}), \cdots , (j_m,g_{j_m})\}$; and

---

2Due to space constraints, we have to omit the details of mindfs. As an example, mindfs$(g_3)$= ((1,2,C,C), (2,3,C,C), (2,4,C,O)). The first two digits are the DFS sequence of the vertices of a graph. The following characters are vertices’ labels.
\[ H_{p_2} = h(h(\lambda(p_2))|H_{D_{p_2}}|H_{p_2}^r) \]

**Example 3.5.1.** With reference to Fig. 3.7, we present an example of the digest of node\((p_2)\), denoted as \(H_{p_2} = h(h(\text{str}(p_2))|H_{D_{p_2}}|H_{p_2}^r)\). (1) is the sketch of MIFTree. \(H_{D_{p_2}}\) is the root digest of (2) the classical MHT of ID\((D_{p_2})\), which is built on top of the data \(\{1, g_1, \cdots, 4, g_4\}\). \(H_{p_2}^r\) is the root digest of (3) the embedded MHT of node\((p_2)\)’s children, which are node\((p_{10})\), node\((p_{11})\) and node\((p_{12})\). The data it embeds are \{node\((p_{10})\), \cdots, node\((p_{12})\)\}, while the search keys are \{p_{10}, p_{11}, p_{12}\}.

**Definition 3.5.4.** The signature of the root node\((p_r)\) of MIFTree is \(\psi_I = \text{sign}(h(h(\text{str}(p_r))|H_{p_r}), SK)\), where SK is the private key of the DO.

It should be remarked that the individual features \(F_q\) must be authenticated in order to verify the correctness of POFs. We organize all features \(F\) of \(G\) with a prefix tree \(T_F\) similar to MIFTree. The authentication process of \(F_q\) is simpler than that of \(P_{q_{\text{opt}}}^r\).

### 3.5.2 Basic Authentication Method

In this subsection, we present the constitution of VO and a basic authentication method. For a concise exposition, we present the details in *set semantics*, unless otherwise specified.

**Verification object.** The overview of the constitution of VO can be given as follows. VO
consists of the VO for recording the searches of \(P_{q}^{\text{opt}}\) on MIFTree (\(VO_{\text{index}}\)) and the VO for the candidate set (\(VO_{C_{q}}\)). Informally, \(VO_{\text{index}}\) includes the visited nodes in searching POFs (denoted as \(N_{I}^{v}\)) and some boundaries of the search paths of POFs (denoted as \(N_{I}^{b}\)). These are necessary to reconstruct the digest of the root of MIFTree. Moreover, the graphs (not only their IDs) in the candidate set are included in \(VO_{C_{q}}\) for client’s verification. While the query answers \(R_{q}\) must be returned, the non-answers in the candidate set \(C_{q}\) must also be included in \(VO_{C_{q}}\), denoted as \(C_{q} \bar{R}_{q}\), where \(C_{q} \bar{R}_{q} = C_{q} - R_{q}\), to verify that no graph in \(C_{q} \bar{R}_{q}\) is an answer. For verification efficiency, the mappings between the query and the answers are included in \(VO_{C_{q}}\). To sum up, we define the constitution of \(VO\), presented in Def. 3.5.6 which consists of the structures and auxiliary structures discussed above.

As discussed in Def. 3.5.1, we have applied MHTs in MIFTree for small \(VO\). The description of \(VO\) of an MHT is well-known but verbose, which includes the answers, the boundaries and the search keys of search paths. For succinct presentation, we define a term “\(VO\) of MHT” to leverage on the known results from MHT.

**Definition 3.5.5.** Suppose an MHT is built on a set of objects \(O : \{o_1, \cdots, o_n\}\) and the corresponding search keys are \(\{k_1, \cdots, k_n\}\) (if has). Given a set of objects \(O'\), \(O' \subseteq O\), the \(VO\) of the MHT of \(O\) is the \(VO\) needed to authenticate \(O'\).

For example, recall from Sec. 3.1 that Fig. 3.2(b) shows an embedded MHT where \(\{x_1, x_2, x_3, x_4\}\) are data values and \(\{1,2,3,4\}\) are the search keys. The search of the key is 2 and the answer is \(x_2\). The \(VO\) of the MHT are \((1, H_{x_1})\) and \((4, H_{3,4})\), with which \(H_{r}\) can be synthesized.

**Definition 3.5.6.** The \(VO\) constitution of basic authentication for subgraph query is a tuple \((VO_{\text{index}}, VO_{C_{q}})\), where

\[
VO_{\text{index}} = (N_{I}, N_{F}, \psi):
\]

- \(N_{I} = (N_{I}^{v}, N_{I}^{b})\) is the digest of MIFTree nodes, where
  - \(N_{I}^{v} : \{n_r, n_1, \cdots, n_m\}\), where \(P_q = \{p_1, \cdots, p_m\}\) and \(p_r\) is the root of MIFTree.
  - **Case 1:** \(p_i \notin P_{q}^{\text{opt}}, n_i = (p_i, H_{D_{p_i}})\).
Algorithm 4 Auth_Query_Processing \((q, G, T_F, T_P, \psi)\)

**Require:** A query graph \(q\), a graph database \(G\), the prefix tree \(T_F\) of features \(F\), the MIFTree \(T_P\) of \(G\) and \(\psi\).

**Ensure:** the answer set of \(q\) \(R_q\) and verification object \(VO\).

1: Initialize \(R_q\) and the structures in \(VO\) to \(\emptyset\) and \(C_q\) to \(G\)
2: \(F_q = \text{find\_maxfeatures}(q, T_F)\) //\(F_q\) fully cover \(q\)
3: \(P_q = \text{find\_POF}(q, F_q, T_P)\) //Enumeration
4: \(P_{opt} = \text{opt\_POF\_MWSC}(F_q, P_q)\)
   */ construct \(VO\) of Case 1 of \(p\) of \(N_I\) */
5: \(\text{for each } p_i \notin P_{opt} \land p_i \in P_q \cup \{p_r\}\)
6: \(N_I^v = N_I^v \cup (p_i, H_{D_p_i})\)
7: \(N_I^b = N_I^b \cup b_i\) /* the \(VO\) of \(MHT\) of node \(p_i\)'s children */
8: \(\text{for each } p_i \in P_{opt}\)
9: \(D_{p_i} = \text{search}(p_i, T_P);\ C_q = C_q \cap D_{p_i}\)
   */ construct \(VO\) of Case 2 of \(p\) of \(N_I\) */
10: \(\text{for each } p_i \in P_{opt}\)
11: \(L_{p_i} = []\)
12: \(\text{for each } g_j \in D_{p_i}\)
13: \(\text{if } g_j \in C_q \text{ then } L_{p_i} = L_{p_i} \oplus j\) /* append ID */
14: \(\text{else } L_{p_i} = L_{p_i} \oplus (j, H_{g_j})\) /* append ID and digest */
15: \(N_I^v = N_I^v \cup (p_i, L_{p_i})\)
16: \(N_I^b = N_I^b \cup b_i\) /* the \(VO\) of \(MHT\) of node \(p_i\)'s children */
   */ construct \(VO\) for features \(F_q\) */
17: \(N_F = \text{construct\_NF}(F_q)\)
   */ construct \(VO\) \(C_q\) */
18: \(\text{for each } g \in C_q\)
19: \(\text{if subIso}(q, g) = \text{true}\)
20: \(R_q = R_q \cup g\)
   /* construct \(VO\) for answer */
21: \(I_M = I_M \cup m\), where \(m\) is the mapping from \(q\) to \(g\).
22: \(\text{else} \) /* construct \(VO\) for non-answer */
23: \(C_{q^R} = C_{q^R} \cup g\)
24: \(VO = ((N_I, N_F, \psi), (I_M, C_{q^R}))\)
25: \text{return } R_q \text{ and } VO
Case 2: \( p_i \in P_{opt}^q \): \( n_i = (p_i, L_{p_i}), L_{p_i} : [l_1, \ldots, l_k] \), where ID(\( D_{p_i} \)) : [1, \ldots, k], and \( l_j = j \), if \( g_j \in C_q \); otherwise, \( l_j = (j, H_{g_j}) \).

- \( N_I^b : \{ b_r, b_1, \ldots, b_m \} \), where \( b_i \) is the \( \mathcal{VO} \) of \( \text{MHT of node}(p_i) \)'s children, \( p_i \in P_q \cup \{ p_r \} \);

- \( N_F = (N_F^w, N_F^b) \) is similar to \( N_I \), as \( F \) is also organized in a prefix tree \( T_F \) ordered by the \( \text{mindfs} \) order. The only difference from \( \text{MIFTree} \) is that each node of the \( T_F \) points a feature but not a list of graph IDs; and

- \( \psi = \{ \psi_F, \psi_I \} \) is the signature of the \( \mathcal{DO} \).

\( \mathcal{VO}_{C_q} = (I_M, C_q^{R_q}) \):

- \( I_M : \{ m_1, \ldots, m_n \} \) is a set of subgraph isomorphism mappings from \( q \) to \( R_q : \{ g_1, \ldots, g_n \} \); and

- \( C_q^{R_q} \) are non-answer graphs in the candidate set \( C_q \).

**\( \mathcal{VO} \) construction.** The \( \mathcal{VO} \) of a query is constructed by Alg. 4 at the \( \mathcal{SP} \) side. Alg. 4 is Alg. 3 extended with \( \mathcal{VO} \) construction: Lines 5-7 and 10-16 for \( N_I \), Line 17 for \( N_F \), and Lines 21-23 for \( \mathcal{VO}_{C_q} \). The extension of \( \text{find\_maxfeatures} \) with \( \mathcal{VO} \) construction is presented in Algo. 2 and that of \( \text{find\_POF} \) is similar (Lines 2-3). As in Alg. 3, to evaluate \( q \), Alg. 4 determines \( P_{opt}^q \) from \( q \) and \( T_F \) (Lines 2-4). In Lines 5-7, for each \( p_i \) in \( P_q \) or \( p_r \) but not in \( P_{opt}^q \), it includes \( (p_i, H_{D_{p_i}}) \) in \( N_I^w \) (Case 1 of Def. 3.5.6) and \( b_i \) in \( N_I^b \), where \( b_i \) is the \( \mathcal{VO} \) of \( \text{MHT of node}(p_i) \)'s children. A subtle remark is that \( \text{node}(p_r) \) is the root of \( \text{MIFTree} \) and it is always visited and considered in \( N_I^w \). \( C_q \) is computed in Lines 8-9 (same as Lines 5-6 in Alg. 3). Then, in Lines 10-14, for each \( p_i \) in \( P_{opt}^q \) (Case 2 of Def. 3.5.6), and for each \( g_j \) in \( D_{p_i} \), if \( g_j \) is in \( C_q \), it adds \( j \) to \( L_{p_i} \); otherwise, \( (j, H_{g_j}) \) to \( L_{p_i} \). The \( \mathcal{VO} \) for \( p_i \) is added to \( N_I^w \) (Line 15). The construction of \( N_I^b \) in Line 16 is the same as that of in Line 7. In Line 17, \( N_F \) for \( F_q \) is constructed similar to \( N_I \), as both \( F \) and \( \text{POFs} \) are indexed by prefix trees. Regarding \( \mathcal{VO}_{C_q} \), in Lines 18-23, if a graph \( g \) in \( C_q \) is an answer, its mapping between the query is added to \( I_M \); otherwise, \( g \) is added to \( C_q^{R_q} \).

The overall \( \mathcal{VO} \) is constructed and returned to the client (Lines 24-25).
Example 3.5.2. Following up the query processing shown in Fig. 3.6, Fig. 3.8 shows the VO determined by Alg. 4. Recall that $P_q = \{p_2, p_3, p_7, p_{11}\}$ and $P_q^{opt} = \{p_3, p_{11}\}$. Regarding $VO_{index}$, $N^V_I = \{n_r, n_2, n_3, n_7, n_{11}\}$. $n_r = (p_r, \mathcal{H}_{D_p})$. $n_2 = (p_2, \mathcal{H}_{D_p})$ and $n_7 = (p_7, \mathcal{H}_{D_p})$ since $p_2, p_7 \notin P_q^{opt}$ (Case 1 of Def. 3.5.6). Since $p_3$ and $p_{11}$ are in $P_q^{opt}$, $n_3 = (p_3, L_{p_3})$ and $n_{11} = (p_{11}, L_{p_{11}})$ (Case 2 of Def. 3.5.6). We note that $ID(D_3) = [1, 4]$, $ID(D_{11}) = [2, 4]$ and $g_4 \in C_q$. Then, $L_{p_3} = [(1, \mathcal{H}_{g_1}), 4]$ and $L_{p_{11}} = [(2, \mathcal{H}_{g_2}), 4]$. Since $g_1, g_2 \notin C_q$, only their IDs are needed. Due to space issues, the $N^V_I$ shown is partial. The RHS of Fig. 3.8 shows the (partial) MHTs of the children of node($p_r$) and node($p_2$). The white boxes indicate the VO derived from MHTs and they are parts of $b_r$ and $b_2$ in $N^V_I$. The $I_M$ in $VO_{C_q}$ is the subgraph isomorphism mapping from $q$ to $g_4$. Since $C_q = R_q = \{g_4\}$, $C_q^{\bar{R}_q}$ is empty.

Authentication at client. When the client receives $R_q$ and $VO$, he/she verifies the correctness of $R_q$. Since the process is similar to Alg. 4 and existing authentication works, we only give an example and highlight the major steps and elaborate Step 4) below, which is unique in MIFTree:

1. compute $F_q$ and verify $F_q$ is the maximum individual fully cover features of $q$ by
using \( q, N_F \) and \( \psi_F \); 

2. compute \( P_q \) and verify \( P_q \) is consistent to those in \( N_I \) by using \( q, F_q \) and \( N_I \); 

3. determine \( P_q^{opt} \) by using \( F_q \) and \( P_q \); 

4. synthesize \( H_{p_i} \) by using \( P_q^{opt} \) and the \( \mathcal{VO} \); 

5. verify the \( H_{p_i} \) with the signature \( \psi_I \) and the public key; 

6. determine \( C_q \) by intersecting the \( L_p \)s from \( N_I^p \), where \( p \in P_q^{opt} \); and 

7. verify \( R_q \) by using \( I_M \); and if \( I_M \) is not correct, invokes \( \text{subIso} \); and verify \( C\bar{R}_q \) by invoking \( \text{subIso} \).

In Step 4), the root digest \( H_{p_i} \) is synthesized bottom-up: We start the synthesis from the \( p \) in \( P_q \) that do not have a \( p' \in P_q \) s.t. \( p \prec p' \). At each synthesis step, \( p_i \) can only be in one of the two cases: Case 1 \( p_i \) is in \( P_q \) but not in \( P_q^{opt} \). \( n_i \) of \( p_i \) is \( (p_i, H_{D_{p_i}}) \). Case 2 \( p_i \) is in \( P_q^{opt} \). \( n_i \) of \( p_i \) is \( (p_i, L_{p_i}) \). \( H_{D_{p_i}} \) is determined from \( L_{p_i} \), \( C\bar{R}_q \) and \( R_q \), which contains the IDs, digests and the graphs of \( D_{p_i} \). The remaining part to-be-determined is \( H_{p_i} \). In both cases, \( H_{p_i} \) is determined from the \( \mathcal{VO} \) of \( \text{MHT} \) of \( \text{node}(p_i) \)’s children, in \( N_I^b \).

The synthesis must have computed the digest of \( \text{node}(p_i) \)’s children (if it is not already in \( \mathcal{VO} \)), as the synthesis is defined bottom-up. With \( p_i, H_{D_{p_i}} \) and \( H_{p_i}^r \) (Def. 3.5.3), the client can recompute \( H_{p_i} \). Then, \( p_i \) is removed from \( P_q \). In the recursive step, the synthesis proceeds to another \( p \) in \( P_q \) with no \( p' \in P_q \) s.t. \( p \prec p' \). With the \( H_p \) for all \( p \in P_q \), \( H_{p_i} \) is synthesized. We present the formal proofs of the soundness and completeness of the basic authentication as follows.

**Theorem 3.5.7.** The basic authentication method is sound and complete.

**Proof.** We establish our theorem with reference to the authentication steps presented in the end of Sec. 3.5.2. In order to prove the soundness and completeness of the query

---

\(^3\)As \( F \) is organized in a prefix tree \( T_F \), \( F_q \) can be verified by using \( q, N_F \) and signature \( \psi_F \) in a similar way.
answers of the basic authentication, we first comment how we establish the correctness of $F_q$, $P_q$ and $P_q^{opt}$.

**Proof of correctness of $F_q$.** $F$ is organized in a prefix tree. We can establish the correctness of $F$ from the authentication of prefix trees, studied in [72]. More specifically, in Step 1 of the basic authentication $F_q$ is computed by the client from $q$ and $N_F$. (Recall that $N_F$ contains the digests of the nodes for the prefix tree of $F$). The client has $q$. The client can verify the features in $N_F$ are sound and complete with respect to the prefix tree of $F$ by synthesizing the root digest of the prefix tree and the data owner’s signature $\psi_F$ [72]. After verifying $N_F$ is not forged, the client computes if $F_q$ is the maximum individual fully cover features of $q$.

**Proof of correctness of $P_q$ and $P_q^{opt}$.** $P_q$ and $P_q^{opt}$ are computed from $q$ and $F_q$ (in Step 2 and 3) of the authentication).

The MIFTree is a prefix tree of the string representations of POFs. The soundness of $N_I$ can be established by the correctness of the authentication of prefix trees [72]. The client can compute $P_q$ by applying find_POF on $q$, $F_q$ and $N_I$. If $N_I$ is not consistent to the $P_q$ computed, then the client can be alerted that $N_I$ is tampered with. The client can compute the correct $P_q^{opt}$ using opt_POF_MWSC.

With the correct of $F_q$, $P_q$ and $P_q^{opt}$, we can then analyze the soundness and completeness of the query answers.

**Proof of soundness of $R_q$.** Assume that a graph $g$ in $R_q$ is modified or bogus. There are only two possible cases:

- **$\text{subIso}(q, g) = \text{false}$:** this is detected when the client performs $\text{subIso}$ by using the isomorphic mapping $I_M$ (in Step 7) of authentication); or

- **$\text{subIso}(q, g) = \text{true}$:** since $g$ is a bogus, the digest of the node that $g$ belongs to cannot be synthesized because $h$ is a one-way collision-resistant function. Subsequently the digest $H_{pv}$ is not generated correctly and the client can detect this with the signature $\psi_I$ and the public key (in Steps 4) and 5) of authentication).
Proof of completeness of \( R_q \). Assume a graph \( g \) is an answer but not in \( R_q \). There are two possible cases:

- \( g \in C_q^{R_q} \): This is detected when the client performs subIso on all graphs in \( C_q^{R_q} \) (in Step 7) of authentication; or

- \( g \notin C_q^{R_q} \): Denote \( P_q^{\text{opt}} \) as \( \{p_1, \ldots, p_m\} \). Since \( P_q^{\text{opt}} \) is correct and \( g \) is an answer, \( g \) contains an instance of \( P_q^{\text{opt}} \). There are only two cases that lead to \( g \notin C_q^{R_q} \wedge g \notin R_q \):

  - \( \exists p_i, g \notin D_{p_i} \): In this case, \( \mathcal{SP} \) had modified \( D_{p_i} \). Hence, \( H_{p_i} \) and \( H_{p_r} \) cannot be synthesized correctly, and this will be detected when comparing the signature \( \psi_I \); or

  - \( \forall p_i, g \in D_{p_i} \): While the \( \mathcal{SP} \) may perform the intersections of all \( D_{p_i} \)s incorrectly, the client will perform the intersections on all \( L_{p_i}, p_i \in P_q^{\text{opt}} \) (in Step 6) of authentication). Hence, the client will be able to detect that \( g \) is an answer.

\[ \Box \]

Example 3.5.3. To illustrate Step 4), we present the major steps of the synthesis of \( H_{p_r} \) of Fig. 3.8. To compute \( H_{p_r} \), bottom-up, we may start the synthesis from \( p_{11} \) since \( p_{11} \in P_q^{\text{opt}} \) and \( \exists p' \in P_q, s.t. p_{11} \prec p' \). We may start at \( p_{3} \) for a similar reason. Let’s start at \( p_{11} \). \( n_{11} = (p_{11}, L_{p_{11}}) \). \( H_{D_{p_{11}}} \) can be computed from \( L_{p_{11}} \) and \( R_q \). The root digest \( H^{\text{r}_{p_{11}}} \) of the MHT of node(\( p_{11} \))’s children can be computed since the MHT is empty. \( H_{p_{11}} \) can then be determined from \( p_{11}, H_{D_{p_{11}}} \) and \( H^{\text{r}_{p_{11}}} \). \( p_{11} \) is removed from \( P_q \). After that, we may proceed to \( p_{2} \), since \( p_{2} \in P_q \wedge p_{2} \notin P_q^{\text{opt}} \). \( n_{2} = (p_{2}, H_{D_{p_{2}}}) \). We determine \( H^{\text{r}_{p_{2}}} \) from the computed \( (p_{11}, H_{p_{11}}) \) and the VO of MHT of node(\( p_{2} \))’s children such as \( b_{2} \). In this case, \( H_{p_{2}} \) is obtained and \( p_{2} \) is removed from the \( P_q \). We then proceed to \( p_{3} \). \( H_{p_{3}} \) is obtained, similar to the synthesis of \( H_{p_{11}} \). \( H_{p_r} \) is synthesized similar to \( H_{p_{2}} \). With the same logic, \( n_r = (p_{r}, H_{D_{p_r}}) \). With \( H_{p_{2}}, H_{p_{3}}, H_{p_r} \) and the VO of MHT of node(\( p_r \))’s children, \( H_{p_r} \) is synthesized.
3.6 Enhanced Authentication

While the basic method presented in Sec. 3.5 is natural to authenticate the filtering-and-verification framework of subgraph query, \( \mathcal{V} \mathcal{O} \) sometimes contains excessive graph IDs. In this section, we propose two enhancements on the basic method.

Firstly, all graph IDs of each feature \( p \in P_q^{opt} \) are returned and in Step 6) of authentication, intersected at the client side to ensure the correctness of \( C_q \). To optimize this, we propose a compact representation of graph IDs. Secondly, graph IDs are needed to synthesize the digests of MIFTree nodes, as elaborated in Step 4) of authentication. As motivated in Sec. 1, graph IDs of \( C_q \) do not fall into a range in general which may lead to large \( \mathcal{V} \mathcal{O} \)s when classical authentication techniques are adopted. Hence, we propose to cluster graphs with similar feature sets offline. As a result, when a query is retrieved by using a set of features, the IDs of \( C_q \) may be clustered and represented by a smaller \( \mathcal{V} \mathcal{O} \).

### 3.6.1 Compact Representation of Graph IDs

The main idea to reduce the excessive graph IDs for verifying the intersections is to encode all the features of each graph in a \( D_p \) in a binary matrix \( M_p \). The data owner signs the matrix. Hence, the client requires one ID \( (D_p) \) and \( M_p \) to verify the intersections.
Definition 3.6.1. For each node node(p), the matrix representation $M_p$ of $\text{ID}(D_p)$ is a $m \times n$ binary matrix, where $n = |D_p|$, $m = |P|$. $M_p(i, j) = 1$ if $g_j \in D_p$, and $M_p(i, j) = 0$, otherwise.

Next, we build a classical MHT to each $M_p$ (defined with $\mathcal{H}_{M_p}$ in Def. 3.6.2). The authentication process can then be described as follows. Consider $P_q^{\text{opt}} = \{p_1, \ldots, p_m\}$. To authenticate $C_q = D_{p_1} \cap \cdots \cap D_{p_m}$, instead of using $L_p$ in $N^v_i$ of $\mathcal{VO}$ for all $p$ in $P_q^{\text{opt}}$ (Case 2 of $N^v_i$), we use only $M_{p_{\text{min}}}$, where $p_{\text{min}} \in P_q^{\text{opt}}$ and $|D_{p_{\text{min}}}|$ is the smallest among all $|D_p|$, $p \in P_q^{\text{opt}}$. The digest $\mathcal{H}_p$ of each node node(p) in IFTree includes $\mathcal{H}_{M_p}$ and the $\mathcal{VO}$ includes only the graph IDs of $p_{\text{min}}$.

Definition 3.6.2. The digest of a node node(p) is

$$h(h(id) | h(\text{str}(p)) | \mathcal{H}_{D_p}) | \mathcal{H}_{|D_p|} | \mathcal{H}_p | \mathcal{H}_{M_p})$$, where

- $id$ is the ID of $p$, $\text{str}(p)$, $\mathcal{H}_{D_p}$ and $\mathcal{H}_p$ are the same as in Def. 3.5.3;
- $\mathcal{H}_{|D_p|}$ is the digest of the size of $D_p$; and
- $\mathcal{H}_{M_p}$ is the root digest of the classical MHT of $M_p$. The data in the MHT are \{(1, s_i), \ldots, (|P|, s_{|P|})\}, where for all $i, s_i = M_p(i, *)$.

The modifications on $\mathcal{VO}$ constitution are then described as follows. Other parts of $\mathcal{VO}$ are identical to those in Def. 3.5.6.

- In Case 2 of $N^v_i$ of $\mathcal{VO}$, for $p_i \in P_q^{\text{opt}}$ but $p_i \neq p_{\text{min}}$, we include only $n_i = (i, p_i, \mathcal{H}_{D_{p_i}}, |D_{p_i}|, \mathcal{H}_{M_{p_i}})$ in $\mathcal{VO}$, where $|D_{p_i}|$ is used to verify $p_{\text{min}}$ in $P_q^{\text{opt}}$ at the client side.
- For $p_{\text{min}}, n_{\text{min}} = (\text{min}, p_{\text{min}}, L_{p_{\text{min}}}, B_{p_{\text{min}}})$, where (i) $\text{min}$ is the ID of $p_{\text{min}}$; (ii) $p_{\text{min}}$ is the POF itself; (iii) $L_{p_{\text{min}}}$ contains the IDs of graphs in $C_q$ and the $\mathcal{VO}$ of the MHT of $\text{ID}(D_{p_{\text{min}}})$; and (iv) $B_{p_{\text{min}}}$ is a set of $(i, s_{p_i})$, where $(i, s_{p_i}) \in B_{p_{\text{min}}}$ if $p_i \in P_q^{\text{opt}} \wedge p_i \neq p_{\text{min}}$, and the $\mathcal{VO}$ of MHT of $M_{p_{\text{min}}}$.

We remark that $B_{p_{\text{min}}}$ records the bit strings of $s_i$ of $M_{p_{\text{min}}}$ where $i \neq \text{min}$. $s_{\text{min}}$ is not needed as $s_{\text{min}}$ can be derived from $L_{p_{\text{min}}}$. Finally, determining $D_{p_1} \cap \cdots \cap D_{p_m}$ is
equivalent to computing $s_1 \land \cdots \land s_m$ which very often requires smaller $\mathcal{VO}$. We provide the formal proofs of soundness and completeness of the enhanced method as follows.

**Theorem 3.6.3.** The enhanced authentication method is sound and complete.

**Proof.** The proof of correctness of $F_q$, $P_q$ and $P_q^{\text{opt}}$ is the same as the one presented in Theorem 3.5.7. The proof of the soundness of $R_q$ is the same as that of Theorem 3.5.7. Here, we focus on the proof of the completeness.

**Proof of completeness of $R_q$.** Assume a graph is an answer but not in $R_q$. There are two possible cases:

- $g \in C_q^{\overline{R_q}}$: This is detected when the client performs $\text{subIso}$ on all graphs in $C_q^{\overline{R_q}}$; or

- $g \notin C_q^{\overline{R_q}}$: Denote $P_q^{\text{opt}}$ as \{p_1, \ldots, p_m\}. $g$ contains instances of $P_q^{\text{opt}}$ as $g$ is an answer. There are again two cases that lead to $g \notin C_q^{\overline{R_q}} \land g \notin R_q$:
  
  - $\exists p_i$, $g \notin D_{p_i}$. In this case, $\mathcal{SP}$ had modified $D_{p_i}$, i.e., $M_{p_i}$ had been modified. Hence, $H_{M_{p_i}}$ cannot be synthesized correctly which leads to wrong $H_{p_i}$, and the client will be alerted when comparing the $\mathcal{DO}$ signature; or
  
  - $\forall p_i$, $g \in D_{p_i}$. The $\mathcal{SP}$ has performed the intersection incorrectly. However, this is detected when the client performs the conjunctions on all $s_i, p_i \in P_q^{\text{opt}}$. 

$\square$

**Example 3.6.1.** Following up Example 3.5.2, Fig. 3.9 shows the major parts of the $\mathcal{VO}$ determined by the enhanced method. The differences of $\mathcal{VO}$ from the Example 3.5.2 are localized in $N^f_v$. Foremost, $P_q = \{p_2, p_3, p_7, p_{11}\}$ and $P_q^{\text{opt}} = \{p_3, p_{11}\}$. Since $|D_{p_3}| = |D_{p_{11}}| = 2$, we just choose $p_3$ as the $p_{\text{min}}$. We show (1) the (partial) matrix $M_{p_3}$ in the LHS of the figure. The bit strings are shown next to the matrix. On the RHS of $M_{p_3}$ is its (2) (partial) MHT. Regarding the $\mathcal{VO}$, we first discuss $p_{11}$. Since $p_{11}$ is in $P_q^{\text{opt}}$ but $p_{11} \neq p_{\text{min}}$. Thus, $n_{11} = (11, p_{11}, H_{D_{p_{11}}}, 2, H_{M_{p_{11}}})$. Next, for $p_{\text{min}}$ (i.e., $p_3$), $n_3$ of $N^f_v$ is $(3, p_3, L_{p_3}, B_{p_3})$. From previous examples, we have $D_{p_3} = \{g_1, g_4\}$ and $C_q = \{g_4\}$. 

50
\[ L_{p3} = [H_{(1,g_1)}, 4] \], where 4 is the graph ID in \( C_q \) and \( H_{(1,g_1)} \) is the VO of the MHT of \( \text{ID}(D_{p3}) \). Regarding \( B_{p3} \), only \( p_{11} \) is in \( P_{q}^{\text{opt}} \) but \( p_{11} \neq p_{\text{min}} \). Thus, \((11, s_{11})\) is included in \( B_{p3} \). Finally, the VO of MHT of \( M_{p3} \) is included in \( B_{p3} \). We remark that the ID of \( g_1 \) is not needed in \( L_{p3} \), since \( s_{11}[1] = 0 \) and \( s_{11} \) will be authenticated in \( B_{p3} \). Thus, \( g_1 \) is certainly not in \( C_q \).

### 3.6.2 Clustering Intersect-able Graphs

The matrix \( M_p \) (defined in Def. 3.6.1) not only minimizes the number of graph IDs by using \( M_{p_{\text{min}}} \), but also indicates how much VO is needed for authenticating the candidate set. In particular, let \( \text{intv}(M_p, i) \) denote the number of intervals in the row of \( p_i \), where all entries in each interval are 1s. The 1s in \( M_p(i, \ast) \) correspond to the graphs in \( D_p \cap D_{p_i} \) and \( \text{intv}(M_p, i) \) is the number of ranges needed to be authenticated. To authenticate a range, the upper and lower bounds of the range are needed in VO. This argument can be generalized to the intersections of multiple sets.

In this subsection, we define the problem of optimal permutation (of columns) of \( M_p \). The ordering of graphs in \( \text{ID}(D_p) \) is optimal when intersecting the graphs of other POFs, the number of the intervals is minimized. We remark that the ordering is optimal in the absence of queries.

**Definition 3.6.4.** Given a \( m \times n \) binary matrix \( M_p \) for node \( p \), the optimal permutation for \( M_p \) (OPM) is to transform \( M_p \) into \( M'_p \) by column permutation s.t. \( \text{cost}(M'_p) = \sum_{i=1}^{m} \text{intv}(M_p, i) \) is minimized, where \( |P| = m \) and \( |D_p| = n \).

Finding the optimal ordering of graphs of \( \text{ID}(D_p) \) is to determine the optimal column permutation of \( M_p \).

**Proposition 3.6.5.** The problem of OPM is NP-hard.

**Proof.** The proof is established from a reduction from Shortest Hamiltonian Path (SHP) problem. Let \( K = (V, E, W) \) be an undirected weighted complete graph with \( n \) vertices. \( V, E, W \) are the set of vertices, edges and weight values, respectively. In particular, \( w(v_i, v_j) \in W \) where \( v_i, v_j \in V \) and \( e_{i,j} \in E \). \( M \) is a \( m \times n \) binary matrix generated from
where the \( j \)-th column of \( M \) represents \( v_j, v_j \in V \), i.e., the permutation of columns of \( M \) is \( p = (v_1, v_2, \cdots, v_{n-1}, v_n) \) and \( n = |V| \). For all \( i, j \leq n \), \( \text{dist}(M, i, j) = w(v_i, v_j) \), where \( \text{dist}(M, i, j) = |M(*, i) \oplus M(*, j)| \). Specially, \( M(*, 1) = M(*, n) = 0 \). Fix the 1-st and \( n \)-th column of \( M \), \( M' \) is the OPM for \( M \) by columns permutation s.t. for all possible matrices permuted from \( M \) \( \text{cost}(M') \) is the minimized. Thus the permutation of columns of \( M' \) is denoted as \( p' = (v_1, v_2, \cdots, v_{n-1}, v_n) \). Then we can get a path \( \mathcal{P} \) on \( K \) in terms of \( p' \), where \( \mathcal{P} = (v_1, v_2', \cdots, v_{n-1}', v_n) \). Since \( \text{cost}(M') \) is the minimized, and \( \text{cost}(M') = \frac{1}{2} \sum_{i=1}^{n-1} \text{dist}(M', i, i+1) = \frac{1}{2} \sum_{v'_i \in p', i=1}^{n-1} w(v'_i, v'_{i+1}). \) Therefore \( \sum_{v'_i \in p', i=1}^{n-1} w(v'_i, v'_{i+1}) \) is minimized. In this case, \( \mathcal{P} \) is exactly the SHP of \( K \). Because finding a SHP from a weighted complete graph is NP-hard, thus the problem of OPM is NP-hard.

The OPM problem can be solved by heuristics of SHP. We cast an instance of OPM into that of SHP. Specifically, given an instance of OPM \( M_p \), we generate a complete graph in terms of \( M_p \). Each column (graph ID in \( \text{ID}(D_p) \)) of \( M_p \) is a vertex and the weight of the edge between two vertices is the total number of different 1s between the two respective columns. The difference of the row of \( p_i \) states that one graph has \( p_i \) but the other does not. That is, one graph appears in \( D_p \cap D_{p_i} \) and the other does not. A final trick is to add an artificial node \( s_0 \) as the source and sink of the graph being constructed. We extend \( M_p \) with a column of zeros for \( s_0 \). The SHP of such a complete graph encodes a permutation of columns of \( M_p \). We have proved that the total sum of the weight of the optimal SHP is twice of the number of intervals in \( M_p \) after the optimal permutation. One of the most efficient approximation algorithms for SHP LKH-2 [69] is adopted. The algorithm is \( K\)-opt and the approximation ratio is preserved under the above conversion.

**Example 3.6.2.** To illustrate the effect of the permutation, we create a small artificial example. Suppose that \( D_q^{\text{opt}} \) is \( \{p_i, p_j\} \), \( \text{ID}(D_{p_i}) = [1, 3, 5, 7, 9] \) and \( \text{ID}(D_{p_j}) = [2, 3, 8, 9] \). Assume further \( p_i \) and \( p_j \) are the only POFs of the database. Then, \( C_q = \{g_3, g_9\} \). \( p_{\text{min}} \) is \( p_j \) as \( |D_{p_j}| = 4 \) and \( |D_{p_i}| = 5 \). Before permutation, the \( L_{p_{\text{min}}} \) in \( \mathcal{V}O \) is \([H_{(2,g_2)}, 3, H_{(8,g_8)}, 9]\). In contrast, after the permutation, \( \text{ID}(D_{p_{\text{min}}}) = [2, 8, 3, 9] \). The \( L_{p_{\text{min}}} \) contains \([H_{(2,g_2), (8,g_8)}], 3, 9\).
3.7 Experimental Evaluation

In this section, we present a detailed experimental evaluation that verifies the performance of our proposed techniques and the effectiveness of our optimizations.

3.7.1 Experimental Setup

**Running Platform.** We conducted all our experiments on a machine with an Intel Core 2 Quad 2.4GHz CPU and 4 GB memory running Windows 7 OS. All our techniques were implemented using C++. We implemented our algorithms on top of iGraph [40]. SHA and RSA were used as our cryptographic signing schemes.

**Dataset.** Following previous experiments of iGraph, we used the same real-world and synthetic datasets in our experimental evaluation. The real-world dataset consists of 10,000 graphs, all of which are drawn from a real AIDS Antiviral dataset (hereafter denoted as AIDS) [79]. AIDS has been used in many studies of subgraph queries [22, 42, 60, 94, 95, 106, 116, 122]. On average, AIDS has 25.42 vertices and 27.40 edges. The number of distinct vertex labels and distinct edge labels are 51 and 4.

For the synthetic dataset, we used SYN.10K.E30.D3.L50 (denoted as SYN). It contains 10,000 graphs of which the average size (the number of edges) is 30; the average density is 0.3; and the number of distinct vertex/edge labels is 50.

We used gSpan [105] with the default settings [106] on the above two datasets to obtain a set of *discriminative frequent features*, which are served as *individual features* for our experiment.

**Query sets.** For both AIDS and SYN, the query sets (denoted as $Q_n$) used have been benchmarked in previous works [22, 42, 60, 94, 106, 116]. Each $Q_n$ contains 1000 graphs with size (the number of edges) of $n$, e.g., $Q_4$ represents 1000 graphs sized 4.

**I/O cost and query time comparison.** We used two representative indexes, namely gIndex [106] and FGIndex [22], to compare the I/O cost (number of graph IDs and graph data fetched) and query time of IFTree. We used the same settings for gIndex and FGIndex as in previous experiment [40]. We note that gIndex often outperformed FGIndex except for small queries and hence we concentrated on comparisons using gIndex.
Baseline comparison. Since there is no existing work on subgraph query authentication, we implemented the authentication on gIndex [106] as a baseline, denoted as MgIndex (see SubSec. 3.2.2). For MgIndex, we also used the same settings as gIndex. Since it is known that binary MHTs yield smaller \( \mathcal{V}/\mathcal{O} \), in our implementation, the MHTs used are binary MHTs.

Offline computation and memory overhead. The offline computation mainly involves (1) the selection of individual features, which takes around 0.5min and 1min for AIDS and SYN, respectively; (2) the selection of POFs, which takes around 30min and 1min for each of the dataset; and (3) the clustering of the intersect-able graphs, which takes around 24h for each of the dataset. For both basic authentication and enhanced authentication, the memory consumptions at the server side and the client side are always smaller than 300MB and 8MB, respectively.

3.7.2 Experiments on AIDS

Effects of maxSize and minSup of POF. Fig. 3.10(a) reports the effects of the maximal size (maxSize) and the minimum support (minSup) of POFs by varying maxSize and minSup for Q8 queries. The x-axis is (maxSize, minSup), e.g., (4, 0.5) represents maxSize = 4 and maxSup = 500. The trends were that when minSup increased or maxSize decreased, the number of POFs of the IFTree (i.e., the index nodes needed by IFTree) decreased and the candidate size increased (which is directly related to \( \mathcal{V}/\mathcal{O} \) size). We set the default values of maxSize and minSup to 4 and 500 to strike a balance between pruning and IFTree size.

I/O cost and query performance.

Average number of graph IDs. Fig. 3.10(b) shows the average number of graph IDs fetched at query time by varying the query sizes. Since the numbers for FGIndex were over 70K, we could not show them here. In Figure 3.10(b), we can see that IFTree had significantly fewer graph IDs than gIndex, especially when the query size was large. The reason was because the size of \( P_{q}^{opt} \) was small as each \( p \in P_{q}^{opt} \) was chosen by our heuristic discussed in Sec. 3.4.2. Moreover, the size of each \( D_{p} (p \in P_{q}^{opt}) \) was small.
Average number of non-answer graphs ($C_q^{R_q}$) in the candidate set. Fig. 3.10(c) shows that the average size of $C_q^{R_q}$ by varying the query sizes. IFTree produced smaller $C_q^{R_q}$ when compared to gIndex and FGIndex in most cases. For example, at Q4, the $C_q^{R_q}$ of IFTree contained 27.2% fewer graphs than that of gIndex. At Q24, IFTree resulted in 13.2% fewer graphs. As FGIndex was verification-free, Q4 queries were small graphs. Most of them were features already and in such cases, there was no non-answer graph in the candidate set. However, when queries were larger than 4, FGIndex produced larger $C_q^{R_q}$.

Average query time. Fig. 3.10(d) reports the average query time at the service provider. At Q4, the average query times on gIndex and IFTree were large since the size of $C_q^{R_q}$ was large for small queries. The subIso test on those graphs dominated the query time. FGIndex was verification-free and Q4 queries in most cases did not require to verify.

When the query size increased after Q12, the query time on IFTree became slightly

---

4Given a query graph $q$, if $q$ is a feature, i.e., $q = f$, it implies that there is no need to verify the subIso between $q$ and $g \in C_q$ as $D_f = C_q = R_q$. Such strategy is called “verification-free” [22].
larger. The reason was that the size of $P_q$ became large and finding the optimal decomposition from $P_q$ incurred relatively large overhead, while their $c_q^{\bar{R}}$s of $g\text{Index}$ and IFTree were being similar. However, the benefits of using $P_p^{opt}$ become clear in the experiments on authentication.

2 Performance of basic authentication

Query composition. Prior to a detailed performance analysis, we show the composition of queries of AIDS, presented in Fig. 3.11(a). TQ1 are queries that contain exactly one P0F in their $P_q^{opt}$. In this case, MIFTree does not perform intersections at query time. TQ2 are queries decomposed into multiple P0Fs and all proposed algorithms in MIFTree affect the performances. From Fig. 3.11(a), we note that TQ2 dominated the query sets as the query size increased.

Average number of intersections. Fig. 3.11(b) shows us the average number of intersections needed versus the query size. MIFTree required significantly fewer intersections at query time compared to MgIndex. For instance, at Q4 and Q24, MIFTree required 45.2%
and 50.8% fewer intersections, respectively, than MgIndex.

**Total VO size.** The small number of intersections performed by MIFTree is reflected in the size of VO. Fig. 3.11(c) shows the VO sizes of MIFTree and MgIndex with varying query sizes. Looking at MgIndex, when the query size increased, the size of the feature set $|F_q|$ rapidly became larger and the number of intersections performed at query time also increased accordingly. For each addition of feature, $f$, all the graph IDs of $D_f$ were added to VO (see Fig. 3.10(b)). Therefore, VO enlarged rapidly with query size. For MIFTree, VO increased with the query size, although at a slower rate. However, since $|P_q^{opt}|$ was often clearly smaller than $|F_q|$ (see Fig. 3.11(b)) and for each $p$ in $P_q^{opt}$, $|D_p|$ was relatively small (see Fig. 3.10(b)), MIFTree clearly outperformed MgIndex. Moreover, the VO of MIFTree did not increase as rapidly as that of MgIndex. We highlight that the VO size at Q4 was large since the size of non-answers in the candidate set ($C_q^{\hat{R}_q}$) was clearly larger than others, which required some VO to authenticate them.

**Average authentication time.** Fig. 3.11(d) reports the average authentication time at client side. We observed that the authentication time of MIFTree was often 4 times faster than that of MgIndex. The number of intersections, i.e., $|P_q^{opt}|$ was smaller. Thus, fewer MHTs of ID($D_p$) were reconstructed, which is a performance bottleneck during authentication. Further, the sizes of $D_p$s of $P_q^{opt}$ were smaller (refer to Fig. 3.10(b)). These factors made MIFTree clearly more efficient than MgIndex.

3 Performance of enhanced authentication.

While the basic authentication already outperformed MgIndex, in this part, we verify the enhanced method further optimizes authentication performances.

**Performance on clustered graphs.** We study the VO size due to the MHT of ID($D_{p_{min}}$) in Fig. 3.12(a) and Fig. 3.12(b). The queries used were TQ2. Fig. 3.12(a) first shows the average number of the intervals on $M_{p_{min}}$ for each queries. Recall SubSec. 3.6.2, the fewer intervals on $M_{p_{min}}$, the smaller VO size due to the MHT of ID($D_{p_{min}}$). Therefore, Fig. 3.12(b) reports such VO size, whose trends were similar to Fig. 3.12(a). We note that the average size of VO at Q4 and Q8 increased. The reason was that most of the query features were frequent, then $C_q$ in $D_{p_{min}}$ was relatively large. Therefore, the VO
became larger. At Q12 - Q24, their features contained more infrequent features. Then $C_q$ was relatively small. Hence, the $\mathcal{V}\mathcal{O}$ for computing $H_{D_p}$ decreased with the query size. However, in all queries, the graph permutations of the graph IDs of $D_{p_{min}}$ clearly led to smaller $\mathcal{V}\mathcal{O}$ size.

**Total $\mathcal{V}\mathcal{O}$ size.** Fig. 3.12(c) shows the comparison of $\mathcal{V}\mathcal{O}$ sizes between basic method and enhanced method for TQ2. For TQ1, the $\mathcal{V}\mathcal{O}$ of enhanced method was almost the same to that of basic method as there was no intersection for TQ1. The figure shows that the enhanced method reduced $\mathcal{V}\mathcal{O}$ sizes significantly. For basic method, $\mathcal{V}\mathcal{O}$ contained all the graph IDs in $\text{ID}(D_p)$ ($p \in P^{\text{opt}}_q$) that were needed to be authenticated (see Fig. 3.10(b)). Instead, $\mathcal{V}\mathcal{O}$ for enhanced method contained the $\mathcal{V}\mathcal{O}$ of MHT of $\text{ID}(D_{p_{min}})$ to authenticate. For instance, at Q24, $\mathcal{V}\mathcal{O}$ by enhanced method was about 20KB whereas that of the basic method was around 120KB.

**Average authentication time.** Fig. 3.12(d) shows the comparison of authentication time
of the basic and enhanced methods. At Q4, since the candidate set contained a large number of non-answer graphs (shown in Fig. 3.10(c)), the subiso test dominated the authentication time. When the query size went beyond Q4, more queries required the basic method to re-build the root digest of the MHT of each ID(D_p), p ∈ P_q opt and the graph IDs were intersected to determine the candidate set. Thus, the authentication time increased rapidly as the query size increased. In comparison, while the authentication time for the enhanced method increased with the query size, it increased in a much slower rate. The reason was that only M_p min and ID(D_p min) were needed to authenticate.

Overall response time. The overall response time consists of the time for query processing, data transmission and authentication. Although the query times of different methods (Fig. 3.10(d)) were close, the improvements of our methods of VΩ size (Figs. 3.11(c) and 3.12(c)) and authentication times (Figs. 3.11(d) and 3.12(d)) were often an order of magnitude more than those of the baseline, which led to better response times.

3.7.3 Experiments on Synthetic Dataset

Finally, we tested our techniques on SYN. We varied (maxSize, minSup) and observed the same trends as those from AIDS. We chose (5, 300) as default. Since the results are similar to those from AIDS, we present some major results in this subsection.

Average query time and authentication time. Fig. 3.13(a) and Fig. 3.13(b) show the query time and authentication time, respectively. In Fig. 3.13(a), we note that the query time of MIFTree was slightly longer than that of the MgIndex. Importantly, Fig. 3.13(b) shows that the authentication time of MIFTree of basic and enhanced method were at least 3 and 4 times faster than the MgIndex, respectively. The speedup of the enhanced method was up to 8 times. These results were due to smaller VΩs.

Performance on clustered graphs. Fig. 3.13(c) shows the clustering of graphs of TQ2 queries reduced at least 50% of the VΩ size due to the MHT of ID(D_p min). The reasons for the trends were the same to the AIDS. The permutations on ID(D_p min) of SYN performed even better than that of AIDS.

Total VΩ size. We compared VΩ size for TQ2 queries between basic method and en-
hanced method, shown in Fig. 3.13(d). The figure shows that the enhanced method consistently generated smaller VO s when the query sizes were larger than 4.

### 3.8 Authenticated Subgraph Queries on Lightweight Devices

This experiment verifies that MIFTree is a practical approach that enables clients to access authenticated subgraph query services via lightweight devices. We chose an extreme hardware setting where the client uses a commodity smartphone. We report both the time for query processing and energy consumption of the subgraph queries on the smartphone.

**Hardware setting.** The smartphone used in this experiment has an 1GHz processor, 1GB internal memory and 3.7 Volt, 1500 mAh battery running the Android 2.2 system.

**Software setting.** We implemented the seminal subgraph isomorphism algorithm, namely...
the Ullman’s algorithm, by Java on Android. The dataset we used is the benchmark dataset AIDS [79], which was used in Sec. 5.7. The queries we tested are Q4, i.e. the query graphs of the size 4.

**Results and discussions.** In our experiment, we consider three cases as follows. (1) Prior to our work, there is no indexing technique that supports authenticated subgraph queries. In the absence of indexing techniques, the SP or DO is required to send the whole database and the DO’s signature to the client, for each query. The client can verify the integrity of the database with DO’s signature. Next, the client scans the graphs to compute the answers. For each Q4 on the dataset AIDS (containing 10K), the smartphone took around 32.6 minutes to determine the answers and 11.7% of the battery was consumed. It is not surprising that the bottleneck is the subgraph isomorphism computation. Moreover, large queries of the AIDS dataset exhibited similar or worse performances. Hence, it is imperative to propose an efficient authentication mechanism on top of indexing techniques. (2) Suppose the client can access to an authenticated subgraph query service using the baseline method. We simulated the evaluation of Q4 again. The authentication on the smartphone required to invoke subgraph isomorphism on 960 graphs. The main reason is that the baseline method also minimizes the number of candidate graphs in the VO. It took 3.3 minutes and drained around 1.1% of the battery for one Q4 query. (3) We tested our MIFTree approach. Subgraph isomorphism was then invoked on 700 graphs only. This further reduced the battery consumption of one Q4 query to 0.81%. The answers were authenticated in around 2.4 minutes. In this case, the battery saved by using the MIFTree approach is about 27% of the battery consumption of the baseline approach.

### 3.9 VO size vs. Authentication Time

In this part, we report a supplementary experiment of the basic authentication method on AIDS dataset in order to show the relationship between VO size and authentication time. The reason for not using enhanced authentication method is that its VO size and authentication time are affected by several non-trivial optimizations, e.g., the matrix rep-
representation of graph IDs and the clustering of intersect-able graphs. Therefore, we opt to use the basic method for this supplementary experiment.

![Graphs](image)

**Figure 3.14:** \(\mathcal{V}\mathcal{O}\) size vs. authentication time of various query sizes of the basic authentication method on the AIDS dataset.

The experimental results of Q4-Q24 are reported in Figs. 3.14(a)-(f), respectively. Each figure is obtained from an experiment of a specific query set on the AIDS dataset. All the query sets used (i.e., Q4-Q24) are the same to those in Sec. 5.7. Each dot in the figure represents a query; the x-axis of the figure represents the \(\mathcal{V}\mathcal{O}\) size due to the query; and the y-axis stands for its authentication time. From the figures, we can easily observe that there are (roughly) linear correlations between \(\mathcal{V}\mathcal{O}\) size and authentication time. Therefore, a major portion of this work discusses \(\mathcal{V}\mathcal{O}\) minimization techniques for
IFTree to address efficient authenticated subgraph query processing.
Chapter 4

Structure-Preserving Subgraph Query Services

In previous chapter, authenticated subgraph query service has been studied such that the data integrity is preserved. In this part, we aim to structure-preserving subgraph query in outsourced graph database such that the confidentiality of the data is warranted.

4.1 Problem Formulation

This section presents a formulation of the problem studied in this work. More specifically, we present the system model, privacy target, attack model, and problem statement.

System model. We follow the system model that has been well received in the literature of database outsourcing (shown in Figs. 5.1 and 5.2), and known to be suitable for many applications. It consists of three parties:

1) Data owner: The owner owns and encrypts the graph data $G$. He/she then outsources the encrypted graph to the service provider and delivers the secret keys to clients for encryption of the query graphs and decryption of the encrypted result;

2) Service provider ($SP$): The $SP$ may be equipped with powerful computing utilities such as a cloud. The $SP$ evaluates a client’s query over the encrypted data, on behalf of the data owner, and returns the encrypted result to the client; and
(3) **Client**: A client encrypts the query graph $Q$ using the secret keys, submits it to the $SP$, and decrypts the returned encrypted result to obtain the final answer.

**Attack model.** We assume the dominating semi-honest adversary model [14, 15, 47, 58] from literature, where the attackers are honest-but-curious and the $SP$ may also be the attacker. For presentation simplicity, we often term the attackers as the $SP$. We assume that the attackers are the eavesdroppers and adopt the chosen plaintext attack [58]. We assume that the $SP$ and clients are not allowed to collude.

**Privacy target.** To facilitate a technical discussion, we assume that the privacy target is to protect the *structures* of a query graph $Q$ and a graph data $G$ from the $SP$ under the attack model defined above. The *structural information* of $Q$ and $G$ considered is the adjacency matrices of $Q$ and $G$, respectively. More specifically, the probability that the $SP$ correctly determines the values of the adjacency matrix of the graph is guaranteed to be lower than a threshold with reference to that of random guess.

The problem statement of this work can be stated as follows: *Given the above system and attack model, we seek an efficient approach to facilitate the subgraph isomorphism query services with preserving the above defined privacy target.*
4.2 Backgrounds and Preliminaries

In this section, we first discuss the background for the data model and revise the classical Ullmann’s algorithm.

4.2.1 Data Model

This part assumes a graph database is a large collection of graphs of modest sizes. We consider undirected labeled connected graphs. A graph is denoted as $G = (V, E, \Sigma, L)$, where $V(G)$, $E(G)$, $\Sigma(G)$ and $L$ are the set of vertices, edges, vertex labels and the function that maps a vertex to its label, respectively. We use $\text{Deg}(v_i, G)$ to denote the degree of the vertex $v_i$ in graph $G$. In this part, we focus on the graph with only vertex labels. Our proposed techniques can be extended to support the graph with edge labels with minor modifications.

4.2.2 Revised Ullmann’s Algorithm

Subgraph query has been a classical query and many algorithms, e.g., [39, 94, 99, 100], have been proposed in the literature. As motivated in Sec. 1, the Ullmann’s algorithm [99] is simple for privacy preservation. In this subsection, we revise the Ullmann’s algorithm into three interleaving steps, namely enumeration, matching and refinement. These form a foundation of our discussions, as we propose our structure preservation techniques for them.

Prior to the algorithmic details, we present some notations used in this work. We use sublso to refer to as the Ullmann’s algorithm. We denote a query as $Q = (V, M_Q, \Sigma, L)$ and graph as $G = (V, M_G, \Sigma, L)$, $m = |V(Q)|$ and $n = |V(G)|$, $M_Q$ and $M_G$ are the adjacency matrices of $Q$ and $G$, respectively. $M_Q(j, k)$ is a binary value, where $M_Q(j, k) = 1$ if $(v_j, v_k) \in E(Q)$, and otherwise 0. The values of the entries of $M_G$ are defined, similarly. Both adjacency matrices $M_Q$ and $M_G$ carry the most fundamental structural information, i.e., the edge information. We use a $m \times n$ binary matrix $M$ to represent the vertex label mapping between $Q$ and $G$. Specifically, $\forall j, k, M(j, k) = 1$ if
\[ L(v_j) = L(v_k), \text{ where } v_j \in V(Q) \text{ and } v_k \in V(G); \text{ and otherwise 0.} \]

The revised Ullmann’s algorithm (subIso) is detailed in Algo. 5. subIso takes \( Q \) and \( G \) as input and returns true if \( Q \) is the subgraph of \( G \). Initially, it determines the vertex label mapping \( M \) (Lines 1-2). Then, subIso checks from \( M \) if there is a subgraph isomorphism mapping from \( Q \) to \( G \) by using three steps: (1) Enum; (2) Match; and (3) Refine. Next, we highlight some details of each step.

**Enumeration** (Lines 8-17). Enum enumerates all possible subgraph isomorphism mappings from \( Q \) to \( G \) by \( M \). Each possible mapping is denoted as \( M_i \). Each column of \( M_i \) contains at most one 1 and each row of \( M_i \) has only one 1 (Lines 12-13). \( M_i \) is enumerated from \( M \) row by row (Line 14). When an \( M_i \) is obtained (Line 8), Match checks if \( M_i \) is a subgraph isomorphism mapping (Line 9). It is easy to see that the number of possible \( M_i \)s enumerated is \( O(n^m) \).

**Matching** (Lines 18-21). For each \( M_i \) enumerated from \( M \), if there exists a matrix \( C_i \), \( C_i = M_i M_G M_i^T \), such that \( \exists j, k, \]
\[ M_Q(j, k) = 1 \land C_i(j, k) = 0 \] (4.2.1)
then such an \( M_i \) cannot be an subgraph isomorphism mapping from \( Q \) to \( G \). Note that \( C_i \) intuitively represents the adjacency matrix of a subgraph of \( G \), that \( Q \) may be isomorphic to through \( M_i \). Formula 4.2.1 states that there is an edge between vertices \( j \) and \( k \) in \( Q \) but no corresponding edge in the subgraph of \( G \), represented by \( C_i \). Such an \( M_i \) is definitely not a mapping. We term the case in Formula 4.2.1 as a violation of subgraph isomorphism (or simply violation). \( M_i \) without violation is called a valid mapping. That is, \( Q \) is a subgraph of \( G \) through \( M_i \).

**Refinement** (Lines 22-28). The number of 1’s in \( M \) significantly increases the number of \( M_i \) to be enumerated in worst case. In the Ullmann’s algorithm, there are two optimizations, called refinements, to reduce the number of 1’s in \( M \). Intuitively, the first refinement exploits the degree constraint, whereas the second refinement relies on the neighborhood constraint: \( \forall j, k, M(j, k) = 1 \Rightarrow \]
\[ (1) \text{ Deg}(v_j, Q) \leq \text{Deg}(v_k, G); \text{ and} \]
\[ (2) \forall x, M_Q(j, x) = 1 \Rightarrow \exists y, M(x, y)M_G(k, y) = 1. \]
Algorithm 5 Revised Ullmann’s algorithm subIso \((Q, G)\)

Require: The query graph \(Q\) and the data graph \(G\).

Ensure: True if \(Q\) is a subgraph of \(G\), False otherwise.

1: Initialize \(M_i := 0\)
2: Generate \(M\) from \((V, \Sigma, L)\) of \(Q\) and \(G\)
3: if !Refine\((M, Q, G)\) /* Refinement */
   4: return False
5: if !Enum\((0, M_i, M, Q, G)\) /* Enumeration */
   6: return False
7: return True

Procedure 5.1 Enum \((d, M_i, M, Q, G)\)
8: if \(d = m\)
   9: return Match\((M_i, Q, G)\) /* Matching */
10: if !Refine\((M, Q, G)\) /* Refinement */
11: return False
12: for each \(c\), where \(c < n\), \(M(d, c) = 1\), and \(\forall d' < d\) \(M_i(d', c) = 0\)
13: \(M_i(d, c) := 1\)
14: if Enum\((d + 1, M_i, M, Q, G)\)
15: return True
16: \(M_i(d, c) := 0\)
17: return False

Procedure 5.2 Match\((M_i, Q, G)\)
18: \(C_i = M_i M_G M^T_i\) /* violation */
19: if \(\exists j, k\), \(M_Q(j, k) = 1\) \& \(C_i(j, k) = 0\)
20: return False
21: return True

Procedure 5.3 Refine\((M, Q, G)\)
22: do \(\forall j, k\), \(M(j, k) = 1\)
23: if degree constraint or neighborhood constraint fails
24: \(M(j, k) := 0\)
25: while \(M\) is not changed
26: if \(\exists j, s.t., \forall k\), \(M(j, k) = 0\)
27: return False
28: return True

Refinement is performed when (1) \(M\) is determined (Line 3) and (2) \(M_i\)'s are enumerated (Line 10). For any pair of \(j\) and \(k\), \(M(j, k) = 1\), if either one of the constraints is not satisfied, the algorithm then flips \(M(j, k)\), i.e., sets \(M(j, k) = 0\) (Lines 22-24). If any row of \(M\) contains only 0s, it reports there is no valid mapping (Lines 26-27).
Example 4.2.1. Fig. 4.2 shows an example for Algo. 5. The LHS shows the query graph $Q$ and the data graph $G$ and their adjacency matrices (below the graphs). The RHS shows the enumeration of $M_i$s. $C_1$ is computed by $M_1$, which is a valid mapping from $Q$ to $G$. Suppose we do not perform Refine, $M_2$ will be enumerated. Match determines that $M_2$ contains violations, as shown. However, when Refine is performed, $M(1,4)$ is flipped to 0 as $v_4$ of $G$ does not connect to $v_2$ and $\text{Deg}(v_1, Q) > \text{Deg}(v_4, G)$. $M_2$ is not enumerated at all.

4.3 subIso with Matrix Operations

From subIso in Algo. 5, it can be noted that the violation defined by Formula 4.2.1 in Match (Line 19) is determined by processing of the entries between $M_Q$ and $C_i$, and the neighborhood constraint (Line 23) precisely exploits edge information. Hence, as motivated in Sec. 1, we cast subIso into an algorithm that uses a series of mathematical computations, denoted as $T_{\text{subIso}}$. This enables us to derive private versions of such operations in later sections.

Foremost, we extend the definition of the query and data graph ($Q$ and $G$), defined in Def. 4.3.1. Def. 4.3.1 only differs from the one presented in Sec. 5.2 that the entries in the adjacency matrix $M_G$ are flipped, i.e., 0s (resp. 1s) are set to 1s (resp. 0s), for the trans-
formed subIso (to be detailed soon). Moreover, \( Q \) and \( G \) are extended with precomputed indexes, called static indexes (to be detailed in SubSec. 4.4.3), to enhance performances. Since our subsequent discussions always assume the extended queries/graphs, we omit the term “extended” for brevity.

**Definition 4.3.1.** The extended data graph of \( G \) is denoted as \( \bar{G} = (V, M_G, \Sigma, L, Sl_G) \) and the query graph is extended as \( Q = (V, M_Q, \Sigma, L, Sl_Q) \), where \( M_G \) are flipped, i.e., \( \forall j, k, \)

\[
M_G(j, k) = \neg M_G(j, k),
\]

and \( Sl_G \) and \( Sl_Q \) (called static indexes) are sets of bit vectors, for optimization purposes.

Based on Def. 4.3.1, we rewrite subIso into transformed subIso called TsubIso in Algo. 6. The inputs are the query graph \( Q \) and data graph \( \bar{G} \). It returns 0 if \( Q \) is a subgraph of \( G \), and non zero otherwise. The corresponding three main steps of Algo. 5 in Algo. 6 are highlighted below.

**Transformed enumeration.** The main difference in TEnum is that Refine (Lines 10-11 of Algo. 5) is removed. The reason is that Refine exploits structural information, which is required to keep private. Another difference is that TEnum is invoked with an input message \( R \) that aggregates the subgraph isomorphism information from \( Q \) to \( G \) during the enumeration of \( M_i \)s.

**Transformed matching.** In Match, the violation of Formula 4.2.1 (Line 19 of Algo 5) is checked by a condition defined on each entry of \( M_Q \) and \( C_i \), which leaks structural information. In comparison, with Def. 4.3.1, the presence of a violation is detected from the product of the matrices \( M_Q \) and \( \bar{C}_i \) (Lines 14-15) in TMatch. Further, the violation due to \( M_i \) is preserved under aggregations, i.e., the result of \( M_i \) (denoted as \( R_i \)) is aggregated into one message \( R \) (Lines 16-17).

The detection of a violation in TMatch is illustrated with Fig. 4.3. Similar to Match, TMatch computes the “subgraph” \( \bar{C}_i \) that \( Q \) may be isomorphic to. With the data graph, \( \bar{C}_i \) is computed in Line 14. There are four possible cases of the entries of \( M_Q \) and \( \bar{C}_i \) and Fig. 4.3 a) highlights the case of the violation of Formula 4.2.1. That is, \( \exists j, k, \)
Algorithm 6 TsubIso \((Q, \bar{G})\)

Require: The query graph \(Q\) and the transformed data graph \(G\).
Ensure: \(R = 0\) if \(Q\) is a subgraph of \(G\), \(R = 1\) otherwise.

1: Initialize \(R := 1\), \(M_i := 0\)
2: Generate \(M\) from \((V, \Sigma, L)\) of \(Q\) and \(\bar{G}\)
3: if \(\text{TRefine} (M, Q, \bar{G}) \) /* TRefinement */
   return \(R\)
4: \(\text{TEnum}(0, M_i, M, Q, \bar{G}, R)\) /* TEnumeration */
5: return \(R\)

Procedure 6.1 \(\text{TEnum}(d, M_i, M, Q, \bar{G}, R)\)

7: \(\text{if } d = m\)
8: \(\text{TMatch}(M_i, Q, \bar{G}, R)\) /* TMatching */
9: for each \(c\), where \(c < n\), \(M(d, c) = 1\) and \(\forall d' < d M_i(d', c) = 0\)
10: \(M_i(d, c) := 1\)
11: \(\text{TEnum}(d + 1, M_i, M, Q, \bar{G}, R)\)
12: \(M_i(d, c) := 0\)

Procedure 6.2 \(\text{TMatch}(M_i, Q, \bar{G}, R)\)

13: Initialize \(R_i := 0\), \(MC_i := 0\)
14: \(\overline{C_i} := M_i \overline{M_i} \overline{C_i}\)
15: \(\forall j, k, MC_i(j, k) := MQ(j, k) \times \overline{C_i}(j, k)\) /* Multiplication */
16: \(R_i := \sum_{\forall j, k} MC_i(j, k)\) /* Addition */
17: \(R = R_i\) /* Multiplication */

Procedure 6.3 \(\text{TRefine}(M, Q, G)\)

18: for each \(j, k, M(j, k) = 1\)
19: \(\text{if } \text{Sl}_{Q}[v_j] \cdot \text{Sl}_{Q}[v_j] \neq \text{Sl}_{Q}[v_j] \cdot \text{Sl}_{Q}[v_k]\)
20: \(M(j, k) = 0\)
21: \(\text{if } \exists j, s.t., \forall k, M(j, k) = 0\)
22: return False
23: return True

\(MQ(j, k) = 1\) and \(C_i(j, k) = 0\) (thus, \(\overline{C_i}(j, k) = 1\)), then

\[MQ(j, k) \overline{C_i}(j, k) = 1, \]  \hspace{1cm} (4.3.2)

For the other three cases, the product is 0. Therefore, by Formula 4.3.2, \(\text{TMatch}\) detects the violation and aggregates the results as follows:

1. **Multiplication** (Line 15). For each pair of \((j, k)\), \(\text{TMatch}\) computes \(MC_i(j, k) = MQ(j, k) \times \overline{C_i}(j, k)\);

2. **Addition** (Line 16). \(\text{TMatch}\) sums up the entries of the product \(MC_i\), i.e., \(R_i = \sum_{\forall j, k} MC_i(j, k)\). Note that \(R_i\) intuitively represents the validity of the mapping.
\[ M \bar{G} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \]

\[ \bar{C}_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \]

\[ \bar{C}_2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \]

\[ R_1 = 0 \]

\[ R_2 = 2 \]

Figure 4.3: (a) The truth table of \( M_Q \bar{C}_i \); (b) Illustration of \( M_Q \) and \( M_G \); and (c) TMatch.

\( M_i \), i.e., if \( M_i \) is valid, no violation is found and the value of \( R_i \) is 0, by Formula 4.3.2; and

3. Multiplication (Line 17). TMatch then aggregates \( R_i \) into \( R \) by a multiplication, i.e., \( R = R \times R_i \). If there is at least a valid \( M_i \), the value of \( R \) equals 0, and non-zero otherwise.

It is worth highlighting that if there exists a subgraph isomorphism mapping \( M_i \) from \( Q \) to \( G \), then \( M_i \) contains no violation, \( R_i = 0 \) and \( R = 0 \). Thus, \( R = 0 \) implies that \( Q \) is a subgraph of \( G \). Otherwise, \( R \) is non-zero, which implies all \( R_i \)'s are not zero and there must be some 1's in the entries of \( MC_i \), for all \( i \). By Formula 4.3.2, there is a violation in each \( M_i \) and thus, \( Q \) is not a subgraph of \( G \).

Example 4.3.1. We illustrate TMatch with the example shown in Figs. 4.3 b) and c). The query and graph are those shown in Fig. 4.2. Fig. 4.3 b) presents \( M_Q \) and \( M_G \). Fig. 4.3 c) reports the intermediate results of TMatch of two possible mappings \( M_1 \) and \( M_2 \) (Fig. 4.2). \( M_1 \) is a valid mapping as \( R_1 \) computed using \( M_Q \) and \( \bar{C}_1 \) (in Lines 15-16) is 0. In comparison, \( R_2 \) computed using \( M_Q \) and \( \bar{C}_2 \) is 2. Hence, \( M_2 \) is an invalid mapping. \( R = R_1 \times R_2 = 0 \) indicates that there is a valid mapping and thus \( Q \) is a subgraph of \( G \).

Transformed refinement. As the neighborhood constraint of Refine precisely exploits the edge information, it cannot be directly adopted. We transform Refine as TRefine that
inner products (Line 19) between our proposed static index (SI, in the form of bit vector) are used for refinements. The index is called static as the indexes of the data graphs are precomputed and those of query graphs are computed by the client prior to TsubIso.

It is worth noting that TsubIso is mainly a series of mathematical operations, i.e., additions, multiplications and inner products. This enables us to establish a connection to private query processing.

### 4.4 Structure-Preserving subIso

In this section, we propose structure-preserving sublso, denoted as SPsublso, as in Fig. 5.2. SPsublso contains three steps: (1) structure-preserving Match (SPMatch) in SubSec. 5.5.1; (2) structure-preserving Enum (SPEnum) in SubSec. 4.4.2; and (3) structure-preserving Refine (SPRefine) in SubSec. 4.4.3.

Before presenting the details, we first give the definition of the encrypted query graph $Q_k$ and the transformed graph $\bar{G}_k$, which are shared by SPMatch, SPEnum and SPRefine.

**Definition 4.4.1.** The encrypted $Q$ and $\bar{G}$ are denoted as $Q_k$ and $\bar{G}_k$, respectively, where $Q_k = (V, M_{Q_k}, \Sigma, L, SL_{Q_k})$ and $\bar{G}_k = (V, M_{G_k}, \Sigma, L, SL_{G_k})$. $M_{Q_k}$ ($M_{G_k}$) and $SL_{Q_k}$ ($SL_{G_k}$) are the encrypted $M_Q$ ($M_G$) and $SL_Q$ ($SL_G$), respectively.

It is worth remarking that we only protect $M_Q$ (resp. $M_G$) and $SL_Q$ (resp. $SL_G$) in $Q$ (resp. $\bar{G}$), by using encryption, since $(V, \Sigma, L)$ does not expose the structural information.

#### 4.4.1 Structure-Preserving Matching

In this subsection, we adopt cyclic group and propose a novel private-key encryption scheme to encrypt $M_Q$ and $M_G$. We then propose SPMatch to compute the operations of TMatch in encrypted domain, where the mapping ($M_i$) has been enumerated by SPEnum (to be discussed in SubSec. 4.4.2).

**Cyclic Group Based Encryption.** Recall that TMatch involves both additions and multiplications. Hence, the state-of-the-art partially homomorphic encryption schemes (e.g., Paillier and ElGamal) [58] cannot be adopted to our problem. On the other hand, due to
the known performance concerns of fully homomorphic encryption scheme (FHE) [35], we may not directly adopt FHE either.

Therefore, we propose a private-key encryption scheme, namely cyclic graph based encryption scheme (CGBE). CGBE not only supports both partial additions and multiplications, but also allows efficient encryption and decryption. Importantly, it is secure against CPA. However, the trade-off of using CGBE in SPMatch is that (1) it introduces negligible false positives; and (2) it requires multiple encrypted messages for aggregating a query result, which are sent to the client.

Before the detailed discussion, we first present the preliminary about cyclic group [58]. Let $G$ be a group. $p = |G|$ is denoted as the order of $G$. In particular, $\forall g \in G$, the order of $G$ is the smallest positive integer $p$ such that $g^p = 1$. Let $\langle g \rangle = \{g^i : i \in \mathbb{Z}_p, g^i \in \mathbb{Z}_n\} = \{g^0, g^1, \ldots, g^{p-1}\}$ denote the set of group elements generated by $g$. The group $G$ is called cyclic if there exists an element $g \in G$ such that $\langle g \rangle = G$. In this case, the order of $G$ is $p = |G|$ and $g$ is called a generator of $G$. Next we propose the cyclic group based encryption scheme as follows.

**Definition 4.4.2.** The cyclic group based encryption scheme is a private-key encryption scheme, denoted as CGBE = (Gen, Enc, Dec), where

- Gen is a key generation function, which generates a secret key $x \in [0, p - 1]$ uniformly at random, a cyclic group $\langle g \rangle = \{g^i : i \in \mathbb{Z}_p, g^i \in \mathbb{Z}_n\}$. It outputs the
private keys as \((x, g)\) and the value \(p\) which is known to the public.

- **Enc** is an encryption function, which takes as input a message \(m\) and the secret key \((x, g)\). It chooses a random value \(r\), and outputs the ciphertext

\[
c = mrg^x \pmod{p}
\]

- **Dec** is a decryption function, which takes as input a ciphertext \(c\), and the secret key \((x, g)\). It outputs

\[
mr = cg^{-x} \pmod{p}
\]

Note that the Dec function of CGBE only decrypts the ciphertext \(c\) as the product of the message \(m\) and random value \(r\). This is because SPMatch does not require the exact value of \(m\).

**Encryption of \(M_Q\) and \(M_G\).** To encrypt \(M_Q\) and \(M_G\), we first present an encoding for each entry of \(M_Q\) and \(M_G\).

**Definition 4.4.3.** The encoding of the entries of \(M_Q\) and \(M_G\) are: \(\forall j, k\),

\[
\begin{align*}
    & \text{if } M_Q(j, k) = 0, \text{ set } M_Q(j, k) \text{ as } q; \text{ and} \\
    & \text{if } M_G(j, k) = 0, \text{ set } M_G(j, k) \text{ as } q,
\end{align*}
\]

where \(q\) is a large prime number.

In relation to Def. 4.4.3, we have the following Formula 4.4.3 that similar to Formula 4.3.2 to detect the violation. We note that only in case of \(M_Q(j, k) = 1\) and \(\bar{C}_i(j, k) = 1\),

\[
M_Q(j, k) \times \bar{C}_i(j, k) = 1 \pmod{q}, \tag{4.4.3}
\]

where \(\bar{C}_i = M_i M_G M_i^T\), the product will be 0 otherwise. Fig. 5.5 a) shows the encoding of four possible combinations between entries, we can see that only if \(M_Q(j, k) = 1\) and \(\bar{C}_i(j, k) = 1\), the product becomes 1. Otherwise it is 0.

Under the encryption scheme CGBE in Def. 5.3.2 and the encoding in Def. 4.4.3, we are ready to define the encryption of the encoding of \(M_Q\) and \(M_G\) (in short, the encryption of \(M_Q\) and \(M_G\)) as follows.
**Definition 4.4.4.** The encryption of \( M_Q \) and \( M_{\bar{G}} \) are denoted as \( M_{Q_k} \) and \( M_{\bar{G}_k} \), respectively, where \( \forall j, k \),

\[
M_{Q_k}(j, k) = \text{Enc}(M_Q(j, k), x, g)
\]

\[
M_{\bar{G}_k}(j, k) = \text{Enc}(M_{\bar{G}}(j, k), x, g)
\]

(4.4.4)

**Example 4.4.1.** We use Fig. 5.5 b) to illustrate an example of the encryption of \( M_Q \) by CGBE. \( \forall j, k \), if \( M_Q(j, k) = 1 \),

\[
M_Q_k(j, k) = \text{Enc}(1, x, g) = r g^x \pmod{p};
\]

and if \( M_Q(j, k) = q \),

\[
M_Q_k(j, k) = \text{Enc}(q, x, g) = q r g^x \pmod{p}.
\]

Finally, we remark that the large prime number \( q \) for the encoding (Def. 4.4.3) must be kept secret. Since CGBE is a symmetric encryption scheme, both the DO and the client hold the same keys \( (x, g, p) \), whereas \( SP \) keeps \( p \) only.

**SPMatching.** Based on Def. 5.3.3, we propose a cyclic group based matching (in short, SPMatch) derived from TMatch (in Algo. 6), shown in Algo. 7. In particular, the input value \( R_k \) is the encrypted message that aggregates the violation. SPMatch first generates \( \bar{C}_i \) (Line 1), which is computed from \( M_i \) and \( M_{\bar{G}_k} \). Then the following three steps are invoked.

**Algorithm 7 SPMatch** \((M_i, Q_k, \bar{G}_k, R_k)\)

1: \( \bar{C}_i := M_i M_{\bar{G}_k} M_i^? \)

/* Multiplication */

2: \( \forall j, k, MC_i(j, k) := M_{Q_k}(j, k) \times \bar{C}_i(j, k) \pmod{p} \)

/* Addition */

3: \( R_i := \sum_{\forall j, k} MC_i(j, k) \pmod{p} \)

4: if \( i \neq 0, i \mod \omega \neq 0 \)

/* Multiplication */

5: \( R_k \times R_i \pmod{p} \)

6: else Send \( R_k \) to client, \( R_k := R_i \)

1. **Multiplication** (Line 2). For each pair of \((j, k)\), SPMatch computes \( MC_i(j, k) = M_{Q_k}(j, k) \times \bar{C}_i(j, k) \pmod{p} \);

2. **Addition** (Line 3). SPMatch sums up the entries in the product, \( i.e., R_i := \sum_{\forall j, k} MC_i(j, k) \pmod{p} \). If \( M_i \) is valid, \( i.e., \) no violation is found, the decryption of the sum is exactly 0, by Formula 4.4.3; and
3. **Multiplication** (Lines 4-6). SPMatch then aggregates $R_i$ into $R_k$ by multiplication (Line 5). If there is at least one valid mapping from $Q$ to $G$, the decryption of a $R_k$ equals 0. Otherwise, the decryption value is non zero. We remark that CGBE leads to errors if the number of $R_i$s in $R_k$ is larger than a predetermined value $\omega$. We thereby propose a *decomposition scheme* (discussed later) that sends to the client a sequence of $R_k$s, where each $R_k$ aggregates $\omega R_i$ (Line 4).

**Example 4.4.2.** Fig. 5.5 b) shows an example to illustrate the multiplication of the four possible cases of combinations between $M_{Q_k}$ and $C_i$. We observe that only under the violation (shown in grey shadow), the product of $M_{Q_k}$ and $C_i$ does not contain $q$. Fig. 5.5 c) illustrates an example of SPMatch following Fig. 4.3 c). $R_1$ and $R_2$ are computed by the summations of $MC_1$ and $MC_2$, respectively. Note that $R_2$ contains violation as $M_2$ is not a valid mapping. $R_k$ is produced.

**Decryption at the client.** After receiving all the encrypted results $R_k$, the client performs the decryption, which mainly contains two steps as belows.

1. For each message $R_k$ aggregated with $\omega R_i$s, the client computes the message encoded in $R_k$ as $R'_k = Dec(R_k, x, g)^{2\omega}$; and

2. For each encoded message $R'_k$, the client computes the final result by $R = R'_k \mod q$.

If any of $R$ equals to 0, there is at least one valid isomorphic mapping $M_i$ that contributes a 0 (Line 3) to the product $R_k$ (Lines 4-5). Thus $subIso(Q, G) = true$.

**Example 4.4.3.** We show the decryption at client following Fig. 5.5 c). The encrypted message $R_k$ client receives aggregates two $R_i$s. The client first generates $(g^{-x})^{2 \times 2}$, computes $R'_k = R_k \times g^{-4x} \mod p$, and finally computes $R = R'_k \mod q$. The result is 0 that indicates $Q$ is a subgraph of $G$.

**Decomposition scheme.** Once the number of $R_i$ aggregated by $R_k$ exceeds a predetermined value, SPMatch will result in incorrect answer. The reason leading to this problem is the multiplications when aggregating $R_i$ into $R_k$ in Line 5 of Algo. 7. Recall that in
the decryption, the client needs to compute the encoded message $R_i'$ after receiving $R_k$, once $R_k'$ exceeds $p$, the client can never recover the final result $R$ by modular $q$ correctly. We can overcome this limitation by determining the maximum number of $R_i$s that can be aggregated in $R_k$, denoted as $\omega$. We have the following formula:

$$\operatorname{Len}(R'_i) = 2 \times (\operatorname{Len}(q) + \operatorname{Len}(r)) + \log(m^2)$$

$$\operatorname{Len}(p) \geq \omega \times \operatorname{Len}(R'_i)$$

\[ (4.4.5) \]

We have:

$$\operatorname{Len}(p) \leq \frac{\operatorname{Len}(p)}{\operatorname{Len}(R'_i)}$$

where $m = |V(Q)|$, $\operatorname{Len}(x)$ is the size of the value $x$, and $R_i'$ is the message encoded in $R_i$, $i.e.$, $R_i' = \text{Dec}(R_i, x, g)^2$. In particular, with reference to Algo. 7, $(\operatorname{Len}(q) + \operatorname{Len}(r))$ is the largest size of the message encoded in each entry of $M_{Q_k}$ and $C_i$. The size of their product (Line 2) is $2(\operatorname{Len}(q) + \operatorname{Len}(r))$. There are $m^2$ additions of such products (Line 3), hence, Algo. 7 requires at most $\log(m^2)$ carry bits. This gives us the largest size of an $R_i'$. Then, the size of $\omega R_i'$ values must be smaller than that of $p$, and we obtain the inequality in Formula 5.5.2. Having computed $\omega$, the $SP$ decomposes $R_k$ into a number of aggregated messages, each of which is a product of at most $\omega$ $R_i$s.

**False positive.** When performing SPMatch, we find that two operations introduce false positive: (1) additions with computing $R_i$ (Line 3); and (2) multiplications with computing $R_k$ in each decomposition (Line 5). We prove that the probabilities of the above two false positive are negligible. Next, we first analyze the probability of false positive from the additions with computing $R_i$.

**Proposition 4.4.5.** The probability of false positive in $R_i$ is negligible.

**Proof.** The probability of false positive in $R_i$ is

$$\operatorname{Pr}(\text{false positive in } R_i) = \operatorname{Pr}(r_1 + \cdots + r_{m^2} = 0 \pmod q)$$

$$= \frac{1}{q},$$

where $m = V(Q)$, and $q$ is a large prime number, $e.g.$, 32bits. Thus, the probability is negligible in practice. \qed

Based on Prop. 5.5.1, we are able to analyze the probability of false positive with computing the $R_k$ in each decomposition.
Proposition 4.4.6. The probability of false positive in $R_k$ is negligible in each decomposition.

Proof. The probability of false positive in each $R_k$ is

$$\Pr(\text{false positive in } R_k) = \Pr(\text{false positive in all its } R_i)$$

$$= 1 - (1 - \frac{1}{q})^\omega$$

$$\approx 1 - e^{-\frac{\omega}{q}},$$

where $\omega$ is the size of the decomposition. Since $\omega \ll q$, the probability is negligible in practice. \hfill \square

4.4.2 Structure-Preserving Enumeration

The mappings $(M, s)$ processed by SPMatch are enumerated by SPEnum. Since the worst case number of all possible mappings $M, s$ from $M$ (Lines 7-12, Algo. 6) is $O(n^m)$, it has been a crucial task of SPsubIso to prune the search of useless $M, s$. For instance, we show a scenario of useless enumerations by using the LHS of Fig. 4.5. There are four subgraphs of $G$ in grey, which are disconnected from each other. In the example, only 4 mappings out of $4^6$ are possible and the remaining enumerated mappings are useless. However, since both $G$ and $Q$ are encrypted, the SP can only blindly enumerates those mappings even they may appear “certainly” invalid.

Therefore, in this subsection, we propose SPEnum that consists of a protocol between the SP and the client to prune some useless partial mappings. However, due to the pruned enumerations, a little non-trivial structural information may be leaked. Such information leakage can be well controlled by determining how often the client informs the pruning (to be analyzed in Sec. 4.5.2).

Mapping Enumeration as a Search Tree

To facilitate the discussions on pruning, we view the search of possible subgraph isomorphic mappings from $Q$ to $G$ (in the LHS of Fig. 4.5) as a search tree, as in the literature of optimizations. A sketch is shown in the RHS of Fig. 4.5. Each internal node in the
$d$-th level represents a partial mapping $M_i$, denoted as $M'_i$, whose enumeration is only up to the first $d$ rows of $M$. We denoted $Q'$ as the induced subgraph of $Q$ from the first $d$ vertices of $Q$ and $G'$ as the subgraph that $Q'$ maps to, under $M'_i$. In the example, the query size is 6, thus the height of the search tree is 6. The fanout of each internal node in $d$-th level equals to the number of 1s in the $(d+1)$-th row of $M$. Each leaf node of the search tree represents a complete $M_i$. Without pruning, TEnum enumerates all $4^6$ leaf nodes.

**SPEnumeration**

We next present SPEnum. It adopts TEnum and SPMatch, and determines if a partial mapping $M'_i$ between $Q'_k$ and $G'_k$ is a valid to be expanded or not. The inputs of SPMatch are (1) $M'_i$, which is the current partial mapping in $d$-th level; (2) $Q'_k$ and $G'_k$, which are the induced subgraph of $Q_k$ and $G_k$ with the first $d$ vertices, respectively; and (3) $R_k$, which is the same as before. Then, a naive protocol involving the client is described as follows:

1. $S\mathcal{P}$ enumerates an $M'_i$ and performs SPMatch. Its result $R_k$ is transmitted to the client for decryption immediately.

2. The client decrypts $R_k$. If it is 0, $Q'_k$ is subgraph isomorphic to $G'_k$ and $M'_i$ is valid.
The client then notifies the $SP$ to repeat Step 1 to continue the enumeration from $M'_i$. Otherwise, the search subtree at $M'_i$ can be pruned.

3. $SP$ terminates SPEnum when a valid $M_i$ is found or no more $M_i$s can be enumerated.

**Protocol with aggregated messages.** The naive protocol introduces additional network communications between the client and the $SP$. To control such communication overhead, we adopt a *breath first search* (BFS) to TEnum and exploit the aggregation supported by CGBE. Specifically, suppose SPEnum is at the $d$-th level of the search tree, it conducts two rounds of communications.

1. In the first round, for each internal nodes $n$ with a valid partial mapping at the $d$-th level (e.g., $n_1$ and $n_4$ at the 3rd level in Fig. 4.5), the $SP$ processes its child nodes of $n$ in *batches* of the size $\omega$. For each batch, the $SP$ performs SPMatch at each node and aggregates $\omega \ R_i$s into $R_k$. Recall that $\omega$ is the number of messages that CGBE correctly aggregates (see SubSec. 5.5.1). The $SP$ sends all the $R_k$s to the client.

   The client decrypts all $R_k$s. (i) If it is 0, there exists at least one child node $n'$ of $n$ such that its partial mapping is valid (e.g., $n_1$). (ii) Otherwise, there is no valid partial mapping of the $n$’s child nodes. The search subtree of $n$ can be safely pruned (e.g., $n_4$).

2. In the second round, the $SP$ then sends all $R_i$s of the child nodes of each batch that contains valid partial mappings to the client, and determines at which node(s) the search proceeds (e.g., $n_2$). Step 1 is then repeated at the $(d + 1)$-th level (e.g., at the 4th level).

3. $SP$ terminates SPEnum as in the naive protocol.

**Remarks.** It is worth noting that CGBE limits $\omega \ R_i$ to be aggregated into $R_k$. If SPMatch utilizes other encryption scheme, such a limitation may not be necessary. For example,
FHE does not limit the number of aggregated messages, but its computation cost is significant. Moreover, the number of communication rounds between the $SP$ and the client is $O(2|V(Q)|)$. In practice, most of the partial mappings of internal nodes are invalid, which result in the size of messages for each round small, e.g., 16KB in worst case for our experiments with default setting.

**Vertex ordering.** Finally, we remark that vertex ordering (the row ordering of $M$) significantly affects subgraph query performance, consistent to the literature (e.g., [39]). However, SPEnum cannot exploit structural information to choose an ordering by heuristics. We thereby order the rows of $M$ by the number of 1s in ascending order. Suppose that partial mappings are mostly useless, such an ordering prune useless enumerations with fewer communications in practice.

### 4.4.3 Structure-Preserving Refinement

In order to further optimize the search, i.e., to reduce the number of possible mappings, in this subsection, we propose a *static index* $SI_Q$ ($SI_G$) of a $Q$ ($G$) that indexes $h$-hop information. We modify Refine into inner products between $SI_G$ and $SI_Q$ as $TRefine$. $SPRefine$ achieves privacies by using *asymmetric scalar product preserving encryption* ASPE [103].

#### Static Index $SI$

The classical refinement reduces the number of 1s in $M$ (*a.k.a* flips the 1s to 0s) to minimize the large search space of subgraph isomorphism mappings. Further to reducing the number of 1s in $M$, $SPRefine$ cannot expose structural information. To address these, we propose to index vertices with their $h$-hop information. The design requirements of such index are that they can be computed before a query is run and hence, no knowledge is gained by executing the query algorithm. Moreover, they should be efficiently computed, as the client may not have powerful machines. Since the index flips the 1s in $M$ by information of the vertices that are $h$ hop away. The $SP$ cannot precisely determine the cause of the flips. Next, we define the $h$-hop vertices of a vertex $v$ as follows.
**Definition 4.4.7.** The $h$-hop vertices of $v$ ($h$-Hop$(v)$) is the set of vertices that are reachable from $v$ by exactly $h$ hops. The $h$-hop $\ell$-labeled vertices of $v$ ($h$-Hop$\ell(v)$) is

$$\{v' \mid L(v') = \ell \text{ and } v' \in h\text{-Hop}(v)\}.$$  

Fig. 4.6 a) illustrates the notations of $h$-hop vertices and $h$-hop $\ell$-labeled vertices of $v$. We assume a canonical ordering of possible labels and hence, present labels as integers.

Given a graph $G$, suppose $h = 2$, $2$-Hop$(v) = \{v_2, v_3\}$ and, $2$-Hop$0(v) = \{v_3\}$.

For illustration purposes, we choose four structural information for indexing used in SPRefine. We can determine if two vertices can be mapped or not from them. It is worth remarking that one can further extend other information for indexing. We define such information as an attribute of the set $h$-Hop$\ell(v)$ as follows:

- $h$-Hop$\ell(v).\text{MaxDeg}$ is the maximum degree of $v'$, $v' \in h$-Hop$\ell(v)$;
- $h$-Hop$\ell(v).\text{Occur}$ is $|h$-Hop$\ell(v)|$;
- $h$-Hop$\ell(v).\text{PreLabel}$ is a set of labels of the parents of occurred $h$-Hop$\ell(v)$; and
- $h$-Hop$\ell(v).\text{Sup}$ is the number of different paths that can reach from $v$ to $v'$, where $v' \in h$-Hop$\ell(v)$.

**Complexity.** One can easily compute MaxDeg, Occur, and PreLabel for all vertices and labels by simple BFS on $G$, which is in $O(|V(G)|^2)$ time. For Sup, on the other hand, it is an $\#P$ problem to count $h$-Hop$\ell(v).\text{Sup}$ for each $v$. As $G$ is always of modest size, it is still practical to compute all those Sup.

**Example 4.4.4.** We continue to discuss the example in Fig. 4.6 a). Suppose $h = 2$. Recall that $2$-Hop$0(v) = \{v_3\}$. We list some 2-hop information as follows: (1) $2$-Hop$0(v).\text{MaxDeg} = 2$, since $\text{Deg}(v_3, G) = 2$; (2) $2$-Hop$0(v).\text{Occur} = 1$, since only one label with 0 in $2$-Hop$0(v)$; (3) $2$-Hop$0(v).\text{PreLabel} = \{0\}$ as 0 is the only label of the parents of $\{v_3\}$; and (4) $2$-Hop$0(v).\text{Sup} = 1$ because there is only one path that can reach from $v$ to $v_3$.

**Encoding h-hop information in static index.** The static index of $G$ is denoted as $\text{SI}_G$. For all $v$, $h$, and $\ell$, $h \leq \text{maxH}$, maxH is a user-specified maximum hop size, $\text{SI}_G[v][h][\ell]$
is a bit vector. In the four \( h \)-hop information defined above, we identify two types. They are encoded in \( S\ell_G \) as follows.

1. **Label set (e.g., PreLabel):** for each \( \ell' \in h\text{-Hop}_\ell(v) \).PreLabel \( \Rightarrow S\ell_G[v][h][\ell].\text{PreLabel}[\ell'] = 1 \), otherwise 0; and

2. **Numerical data (e.g. MaxDeg, Occur and Sup):** We present the encoding of MaxDeg for illustration. Those of Occur and Sup are similar. We denote the maximum value for MaxDeg as MaxDeg\(_{\text{max}}\). For each \( i \leq \text{MaxDeg}_{\text{max}} \) and \( i \leq h\text{-Hop}_\ell(v) \).MaxDeg \( \Rightarrow S\ell_G[v][h][\ell].\text{MaxDeg}[i] = 1 \), otherwise 0.

The bit vector \( S\ell_G[v][h][\ell] \) is then simply a concatenation of \( S\ell_G[v][h][\ell].\text{MaxDeg}, S\ell_G[v][h][\ell].\text{Occur}, S\ell_G[v][h][\ell].\text{PreLabel} \) and \( S\ell_G[v][h][\ell].\text{Sup} \). The bit vector \( S\ell_G[v] \) is accordingly a concatenation of all \( S\ell_G[v][h][\ell] \)s for all \( v, h \leq \text{maxH} \) and \( \ell \).

**Example 4.4.5.** Fig. 4.6 a) shows a simple example of the partial \( S\ell_G[v][h][\ell] \) for \( v \) in \( G \), where \( h = 2 \), \( \ell = 0 \). We preset the default maximum value for MaxDeg, Occur and Sup to 3. We assume that the possible labels are 0 and 1. (1) For PreLabel, since 2-Hop\(_0(v) \).PreLabel = \{0\}, then \( S\ell_G[v][2][0].\text{PreLabel}[0] = 1 \), and \( S\ell_G[v][2][0].\text{PreLabel}[1] = 0 \); and (2) For MaxDeg, as 2-Hop\(_0(v) \).MaxDeg = 2, thereby \( S\ell_G[v][2][0].\text{MaxDeg}[1] = S\ell_G[v][2][0].\text{MaxDeg}[2] = 1 \).

The \( h \)-hop information abovementioned can be generated by a simple depth first traversal starting at each vertex on the data graph offline and on the query by the client on the fly. Due to space restrictions, we omit the verbose algorithm.

**Inner Products of Static Indexes**

With the static index \( S\ell \), we establish the refinement of possible subgraph isomorphism mappings by the following proposition:

**Proposition 4.4.8.** Given a user-specified \( \text{maxH} \), \( \forall v_j \in V(Q) \) and \( v_k \in V(G) \), \( M(j, k) = 1 \), iff the following of the \( h \)-hop information of \( v_j \) and \( v_k \) hold: \( \forall \ell \in \Sigma(G), h \leq \text{maxH}, \)

- \( h\text{-Hop}_\ell(v_j) \).MaxDeg \( \leq h\text{-Hop}_\ell(v_k) \).MaxDeg;
• $h\text{-Hop}_\ell(v_j).\text{Occur} \leq h\text{-Hop}_\ell(v_k).\text{Occur}$;

• $h\text{-Hop}_\ell(v_j).\text{PreLabel} \subseteq h\text{-Hop}_\ell(v_k).\text{PreLabel}$; and

• $h\text{-Hop}_\ell(v_j).\text{Sup} \leq h\text{-Hop}_\ell(v_k).\text{Sup}$.

Prop. 4.4.8 can be obtained from a proof by contradiction.

**Example 4.4.6.** We use Fig. 4.6 b) to illustrate the underlying idea of Prop. 4.4.8. For simplicity, we only show the effect of MaxDeg. Before the refinement, $M(j, k) = 1$ since $L(v_j) = L(v_k)$. Since $2\text{-Hop}_0(v).\text{MaxDeg}$ of $Q$ and $G$ are 3 and 2, respectively. Hence, $2\text{-Hop}_0(v_j).\text{MaxDeg} \leq 2\text{-Hop}_0(v_k).\text{MaxDeg}$. By Prop. 4.4.8, $v_j$ cannot be mapped to $v_k$ and $M(j, k)$ is flipped to 0.

Therefore, TRefine further transforms Prop. 4.4.8 into the inner product as follows.

**Proposition 4.4.9.** Given a user-specified $\text{maxH}$, $M(j, k) = 1$, $v_j \in V(Q)$ and $v_k \in V(G)$, iff the following of SI of $v_j$ and $v_k$ hold: $\forall \ell \in \Sigma(G)$, $h \leq \text{maxH}$,

$$\text{SI}_Q[v_j][h][\ell] \cdot \text{SI}_Q[v_j][h][\ell] = \text{SI}_Q[v_j][h][\ell] \cdot \text{SI}_G[v_k][h][\ell].$$

**Example 4.4.7.** We illustrate the Prop. 4.4.9 with the Example 4.4.6 in Fig. 4.6 b), the partial SI of both $Q$ and $G$ are shown. Since $\text{SI}_Q[v_j][2][0] \cdot \text{SI}_Q[v_j][2][0] = \text{SI}_Q[v_j][2][0] \cdot \text{SI}_G[v_k][2][0]$, then $M(j, k)$ is flipped to 0.

Note that we can further simplify the inner product in Prop. 4.4.9 to $\text{SI}_Q[v_j] \cdot \text{SI}_Q[v_j] = \text{SI}_Q[v_j] \cdot \text{SI}_G[v_k]$, where $\text{SI}_Q[v_j]$ is the concatenation for all $\text{SI}_Q[v_j][h][\ell]$s. Therefore, Line 19 of TRefine is mainly one inner product between $\text{SI}_Q[v_j]$ and $\text{SI}_G[v_k]$, using Prop. 4.4.9 for pruning the 1s in $M$.

For SPRefine, we encrypt SIs as: $\forall v_j \in V(Q)$ and $\forall v_k \in V(G)$, $\text{SI}_{Q_k}[v_j] = \text{ASPE}(\text{SI}_Q[v_j])$ and $\text{SI}_{G_k}[v_k] = \text{ASPE}(\text{SI}_G[v_k])$. The secret keys held by $SP$ and the client are the same to that of [103]. Finally, SPRefine is TRefine after replacing Line 19 with a private inner product between encrypted bit vectors ($\text{SI}_{Q_k}$ and $\text{SI}_G$), supported by ASPE.

We close this section with a remark that SPEnum and SPRefine may expose little non-trivial information in the sense that the probability of guessing the structure of a graph is not that of a random guess anymore. We shall analyze their probabilities in Sec. 4.5.2.
4.5 Privacy Analysis

In this section, we prove the privacy of the encryption method and then the query algorithm SPsubIso. The attack model is defined in Sec. 4.1 that we assume the attackers or SPs are the eavesdroppers and can adopt the chosen plaintext attack (CPA) [58].

4.5.1 Privacy of the Encryption Method

Two encryption methods are used in this work. (1) CGBE scheme is proposed to encrypt \( M_Q \) and \( M_G \), and (2) ASPE [103] is adopted to encrypt \( SI_Q \) and \( SI_G \). We first state that both the CGBE and ASPE schemes are secure against CPA and then establish that the structures of the query and the graph are protected against our attack model. Denote \( \hat{g} \) to be an arbitrary chosen from \( G \).

**Lemma 4.5.1**. [58] Let \( G \) be a finite group, and let \( m \in G \) be arbitrary. Then, choosing random \( g \in G \) and setting \( g' = m \cdot g \) gives the same distribution for \( g' \) as choosing random \( g' \in G \). I.e., for any \( \hat{g} \in G \)

\[
\Pr[m \cdot g = \hat{g}] = 1/|G|,
\]

where the probability is taken over random choice of \( g \).
Lemma 4.5.2. Let $G$ be a finite group, and let $g \in G$ be arbitrary. Then choosing random $r \in [0, |G|]$ and setting $g' = g^r$ gives the same distribution for $g'$ as choosing $g'$ from $G$. I.e., for any $\hat{g} \in G$

$$\Pr[g^r = \hat{g}] = 1/|G|,$$

where the probability is taken over random choice of $r$.

Proof. We prove the lemma in a similar style of the proof [58] of Lemma 4.5.1. Let $\hat{g} \in G$ be arbitrary. Then

$$\Pr[g^r = \hat{g}] = \Pr[r = \log_g \hat{g}]$$

Since $r$ is chosen uniformly at random, the probability that $r$ is equal to the fixed element $\log_g \hat{g}$ is exactly $1/|G|$.

Lemma 4.5.3. CGBE is secure against CPA.

Proof. We prove that the proposed CGBE scheme has indistinguishable encryptions in the presence of the eavesdroppers, which is implied by the definition of CPA secure [58].

Specifically, choosing a random value $r$, and letting $r' \in G$ such that $g^{r'} = r$, we have $\text{Enc}(m, g, x) = mrg^x = mg^{x+r'}$. First, by Lemma 4.5.2, $\Pr[g^{x+r'} = \hat{g}] = 1/|G|$, where $\hat{g}$ is arbitrary chosen from $G$. Then, by Lemma 4.5.1, $\Pr[mg^{x+r'} = \hat{g}] = 1/|G|$. Therefore, the ciphertext in the CGBE scheme is a uniformly distributed group element and, in particular, is independent of the message $m$ being encrypted, i.e., $\Pr[mrg^x = \hat{g}] = 1/|G|$. That means the entire ciphertext contains no information about $m$. Given the above, CGBE is secure against chosen plaintext attack.

Since CGBE is a secure encryption scheme against CPA, $SP$ can never attack the $M_Qk$ and $M_Gk$ without possessing the secret key against our attack model.

Lemma 4.5.4. $M_Qk$ and $M_Gk$ are preserved from $SP$ against the attack model under CGBE.

Proof. The proof is a direct application of Lemma. 4.5.3. Since CGBE is secure against CPA, $M_Qk$ and $M_Gk$ are secure against the attack model under CGBE.
Next, we state that $SI_{Q_k}$ and $SI_{G_k}$ are preserved from $SP$.

**Lemma 4.5.5.** $SI_{Q_k}$ and $SI_{G_k}$ are preserved from $SP$ against the attack model under ASPE.

$SI_{Q[v_j]}$ and $SI_{G[v_k]}$ are encrypted by ASPE, where $v_j \in V(Q)$ and $v_k \in V(G)$. Since ASPE is secure against CPA [103], it is immediate that Lemma 4.5.5 is true.

**Theorem 4.5.6.** The structure of both $Q$ and $G$ are preserved from $SP$ against our attack model under CGBE and ASPE.

**Proof.** The proof can be deduced from Lemmas 4.5.4 and 4.5.5. Recall that $Q_k = (V, M_{Q_k}, \Sigma, L, SI_{Q_k})$ and $\bar{G}_k = (V, M_{\bar{G}_k}, \Sigma, L, SI_{\bar{G}_k})$. By Lemmas 4.5.4 and 4.5.5, the $SP$ cannot break $Q_k$ and $\bar{G}_k$ since the structures of $Q_k$ and $\bar{G}_k$ (i.e., $M_{Q_k}$, $SI_{Q_k}$, $M_{\bar{G}_k}$ and $SI_{\bar{G}_k}$) are secure against CPA.

### 4.5.2 Privacy of SPsubIso

As presented in Sec 4.4, SPsubIso contains three main steps. We analyze the privacy of each of these steps in this subsection. Before we present the analysis, we clarify some notations. Given $Q$ and $G$, $m = |V(Q)|$ and $n = |V(G)|$. The function $P(n)$ returns the number of all possible graphs generated by $n$ vertices, i.e., $P(n) = 2^{n^2}$. The function $A(G)$ returns 1 if $SP$ can determine the exact structure of $G$, and 0 otherwise. The probability that the $SP$ can determine the structure of the graph $G$ is denoted as $Pr[A(G) = 1]$.

Given a graph $G$ with $n$ vertices, the probability to determine the graph structure by a random guess is $Pr[A(G) = 1] = \frac{1}{P(n)}$.

**Proposition 4.5.7.** Under SPMatch, $Pr[A(Q) = 1] = \frac{1}{P(m)}$, and $Pr[A(G) = 1] = \frac{1}{P(n)}$, which are equivalent to random guess.

**Proof.** (1) First we prove that the $SP$ can never determine any structural information from the computations in each step of SPMatch. Recall that each SPMatch comprises a constant number of mathematical operations in the encrypted domain in Algo. 7:

- Line 2 invokes a constant number $m^2$ of multiplications of $M_{Q_k}$ and $C_i$;
• Line 3 requires a constant number \( m^2 \) of additions in \( MC_i \); and

• Line 4 conducts one multiplication \( R_i \) and \( R_k \).

Further, by Lemma 4.5.3, all the intermediate computation results are securely protected against the attack model. Thus, \( SP \) cannot learn any structural information from these steps.

(2) Next, given any two SPMatches, the \( SP \) only knows that each SPMatch aggregates its \( R_i \) into \( R_k \) by one multiplication. Similarly, by Lemma 4.5.3, no other information can be learned from the \( R_i \) or \( R_k \) by the \( SP \).

Putting the above together, the \( SP \) does not learn the structures of \( Q \) or \( G \) by invoking SPMatches and the probability of determining a structure is equivalent to that of random guess. \( \square \)

**Proposition 4.5.8.** Under SPEnum, the following holds:

- If \( Q_d \) is subgraph isomorphic to \( G_d \), there is no information leakage, i.e.,
  \[
  \Pr[A(Q_d) = 1] = \Pr[A(G_d) = 1] = \frac{1}{P(d)};
  \]
  and

- Otherwise,
  \[
  \Pr[A(Q_d) = 1] = \Pr[A(G_d) = 1] = \frac{1}{(P(d) - P(d-1))},
  \]

where \( Q_d \) (resp., \( G_d \)) is the induced subgraph of \( Q \) (resp., \( G \)) that contains the mapped \( d \) vertices specified by the partial mapping \( M_i \) enumerated up to the level \( d \).

**Proof.** Recall that \( M_{Q_k} \) and \( M_{G_k} \) are preserved, by Lemma 4.5.4 and Prop. 5.6.3. Hence, we only consider the information that the \( SP \) can gain from the protocol in SPEnum. Only \( Q_d \) and \( G_d \) are analyzed as the remaining subgraphs (\( Q - Q_d \) and \( G - G_d \)) are not yet processed by the \( SP \). By the protocol of SPEnum, the client informs the \( SP \) at the \( d \)-th level of the search tree, the \( SP \) knows that the nodes at the \( d \)-th level, say \( v_j \) and \( v_k \) in \( Q \) and \( G \), cause a violation is detected or not. We thereby consider these two exhaustive cases as follows:
Case 1: If \( Q_d \) is subgraph isomorphic to \( G_d \), there is no violation between \( Q_d \) and \( G_d \). Recall Formula 4.2.1, a violation occurs when \( v_j \) is connected to some vertices (under \( M_Q \)) but \( v_k \) does not have corresponding edges (under \( C_i \)). When there is no violation, \( v_j \) may or may not be connected to other vertices in \( Q_d \). The SP cannot distinguish this because the edges of \( v_j \) (in \( M_{Q_k} \)) is preserved. Similarly, the SP does not learn any information about the edges of \( v_k \) of \( G_d \) neither. Hence, there is no information leakage; and

Case 2: If \( Q_d \) is not subgraph isomorphic to \( G_d \), there is a violation between \( Q_d \) and \( G_d \). Hence, the SP knows \( Q_d \) and \( G_d \) do not falsify Formula 4.2.1. However, if \( v_j \) is isolated in \( Q_d \), the first predicate of Formula 4.2.1 is always false; and if \( v_k \) is connected to all other vertices in \( G_d \), the second predicate of Formula 4.2.1 is always false. Contrarily, other than the above two scenarios, the SP cannot be certain the cause of the violation, as both \( M_{Q_k} \) and \( M_{\bar{G}_k} \) are protected. The above scenarios affect the probabilities as follows.

- \( v_j \) is isolated in \( Q_d \), i.e., \( \forall v_j' \in V(Q_d), v_j' \neq v_j, (v_j, v_j') \notin E(Q_d) \). Then, the possible number of \( Q_d \) with isolated \( v_j \) is \( P(d - 1) \). Thus, the probability that the SP determines \( Q_d \) is \( \Pr[A(Q_d) = 1] = \frac{1}{P(d) - P(d - 1)} \); and

- \( v_k \) is connected to all other vertices in \( G_d \), i.e., \( \forall v_k' \in V(G_d), v_k' \neq v_k, (v_k, v_k') \in E(G_d) \). Then, the possible number of \( G_d \) with \( v_k \) connecting to all other vertices is \( P(d - 1) \). Therefore, the probability that SP determines \( G_d \) is \( \Pr[A(G_d) = 1] = \frac{1}{P(d) - P(d - 1)} \).

Consider multiple SPEnum calls. Case 1 does not leak information, whereas the enumerations beyond Case 2 are pruned. In either case, an SPEnum call will not affect another.

\[ \square \]

**Proposition 4.5.9.** Under SPRefine, the following holds:

- If \( M(j, k) \) is not flipped, there is no information leakage; and
We obtain the probabilities as follows (similar to the derivations of Prop. 4.5.8’s proof):

\[
\begin{align*}
\Pr[A(Q) = 1] &= \frac{P(a+1)}{P(m)(P(a+1) - 1)}, \text{ and} \\
\Pr[A(G) = 1] &= \frac{P(b+1)}{P(n)(P(b+1) - 1)},
\end{align*}
\]

where \(a = |\text{MaxDeg}(Q)|^{\text{maxH}}, b = |\text{MaxDeg}(G)|^{\text{maxH}}, \text{ and MaxDeg}(G) \) is the maximum degree of the vertices of \( G \).

**Proof.** Recall that for any \( v_j \in V(Q), v_k \in V(G) \), \( \text{Sl}_{Q_k}[v_j] \) or \( \text{Sl}_{G_k}[v_k] \) themselves do not leak any structural information against CPA by Lemma. 4.5.5. Therefore, we only consider the private inner product between \( \text{Sl}_{Q_k}[v_j] \) and \( \text{Sl}_{G_k}[v_k] \). For each \( M(j, k) = 1 \), we divide it into two exhaustive cases as follows:

**Case 1:** If \( M(j, k) \) is not flipped, \( \text{Sl}_Q[v_j] \cdot \text{Sl}_G[v_k] = \text{Sl}_Q[v_j] \cdot \text{Sl}_Q[v_j] \) by Prop. 4.4.9. By Lemma 4.5.5, \( \mathcal{SP} \) cannot learn any structural information from \( \text{Sl}_{Q_k}[v_j] \) and \( \text{Sl}_{G_k}[v_k] \). The only information the \( \mathcal{SP} \) can deduce is that the (four) conditions listed in Prop. 4.4.8 hold. Since all the values of \( \text{MaxDeg}, \text{Occur}, \text{PreLabel} \) and \( \text{Sup} \) are encrypted, the \( \mathcal{SP} \) does not learn any structural information (i.e., \( Q \) and \( G \)) of \( v_j \) and \( v_k \). Hence, there is no information leakage; and

**Case 2:** If \( M(j, k) \) is flipped, \( \text{Sl}_Q[v_j] \cdot \text{Sl}_G[v_k] \neq \text{Sl}_Q[v_j] \cdot \text{Sl}_Q[v_j] \). Similar to **Case 1**, the \( \mathcal{SP} \) cannot deduce structural information from this, due to the encrypted operations. However, the flip of \( M(j, k) \) implies that there is a violation caused by \( v_j \) and \( v_k \) between the subgraphs \( Q_a \) and \( G_b \), where \( Q_a \) (resp., \( G_b \)) is the induced subgraph of \( Q \) (resp., \( G \)), containing at most \( a \) (resp., \( b \)) vertices that are reachable from \( v_j \) (resp., \( v_k \)) within \( \text{maxH} \) hops. This affects the probabilities similar to that in the proof of Prop. 4.5.8 as follows:

- Vertices in \( V(Q_a) \) are all isolated. The number of the possible \( Q \) containing such a \( Q_a \) is \( 2^{n^2-(a+1)^2} = P(m)/P(a+1) \); and

- Vertices in \( V(G_b) \) are connected to all other vertices. The number of the possible \( G \) containing such \( G_b \) is \( 2^{n^2-(b+1)^2} = P(n)/P(b+1) \).

We obtain the probabilities as follows (similar to the derivations of Prop. 4.5.8’s proof):

\[
\begin{align*}
\Pr[A(Q) = 1] &= \frac{1}{P(m)-P(m)/P(a+1)} = \frac{P(a+1)}{P(m)(P(a+1) - 1)}, \text{ and} \\
\Pr[A(G) = 1] &= \frac{1}{P(n)-P(n)/P(b+1)}
\end{align*}
\]
\[
\frac{P(b+1)}{P(n)(P(b+1)-1)},
\]
respectively.

Finally, each flip is independent because the subgraph of \(Q_a\) and \(G_b\) of each SPRefine can be arbitrarily different.

Finally, we remark that Props. 4.5.8 and 4.5.9 state that the client may tune the privacy offered by SPsubIso by varying the variables \(\text{maxH}\) and \(d\) of SPEnum and SPRefine. Further, the values of MaxDeg and maxH (and therefore \(a\) and \(b\)) are not known to the \(SP\). We use these values in Prop. 4.5.8 to simply quantify the privacy. In our experiment, we confirmed that SPEnum and SPRefine are effective optimizations and we may set these variables to balance privacy and performances.

### 4.6 Experimental Evaluation

In this section, we present a detailed experimental evaluation to investigate the performance of our techniques on both real world and synthetic datasets.

#### 4.6.1 Experimental Setup

**The platform.** We set up the \(SP\) as a server at Amazon EC2, equipped with a 2.8GHz CPU and 16GB memory running Ubuntu 14.04 OS. The client is a local machine with a 3.4GHz CPU and 16GB memory running Win 7 OS. For ease of exposition, we assume the \(DO\) has a machine with the same setting, to encrypt data graphs. The client is connected to an Ethernet. All techniques were implemented on the GMP library (C++). By default, our CGBE uses 2048 bits; the sizes of the prime number \(q\) and the random number \(r\) are both set to 32bits. The decomposition size \(\omega\) is 15. Our ASPE implementation is set accordingly to [103]. We have implemented a FHE-based solution. Its performance is always at least one order of magnitude slower than CGBE’s. Thus, we do not report their numbers here.

**Datasets.** We used two real-world benchmark datasets namely AIDS (A) and PubChem (P) from [40], which are widely used in [22, 39, 42, 65, 94, 106, 116, 122]. As our discussions focused on vertex labels, without loss of generality, we remove the edge labels. AIDS
consists of 10,000 graphs, which are drawn from a real antiviral dataset [79]. On average, each graph in Aids has 25.42 vertices and 27.40 edges. The number of distinct vertex labels is 51. PubChem consists of 1 million graphs, which are drawn from a real chemical database [77]. Each graph in PubChem has 23.98 vertices and 25.76 edges, on average. The number of distinct vertex labels is 81.

Query sets. For each of the aforementioned datasets, we used its existing query sets $Q_4$, $Q_8$, $Q_{12}$, $Q_{16}$, $Q_{20}$ and $Q_{24}$, which can be downloaded from [40]. Each $Q_n$ contains 1,000 query graphs, where $n$ is the number of edges for each query.

Test runs. The queries were generated from random sampling of the above datasets and their associated query sets. For each dataset and query set $Q_n$, we randomly sampled 1,000 graphs and 10 query graphs, i.e., for each $Q_n$, we performed 10,000 subgraph isomorphism testings. In addition, the average densities of the sample graphs and queries are the same as those of the original data and query sets, respectively. We report the average of the test runs by default. We use the abbreviation AQT for average query time.

Default values of parameters. The parameters used in SPRefine and SPEnum are set as follows. We set the default maxH, and maximum values for MaxDeg, Occur, and Sup to 6. We set the starting pruning depth $d$ of the protocol of SPEnum to 3.

### 4.6.2 Experiments on Real Datasets

Performance by Varying Query Sizes. We first show the performance of various query sizes in Fig. 4.7.

Encryption time by varying query sizes. We report the average encryption times in Fig. 4.7(a). The encryption time of a query $Q$ involves (1) the time for generating $SI_Q$; (2) the time of encryption of $M_Q$ by CGBE; and (3) the time of encryption of $SI_Q$ by ASPE. We observe that the average encryption times are around 100ms and 150ms for Aids and PubChem, respectively. The encryption of $M_Q$ by our proposed CGBE is efficient, which only costs several milliseconds on a commodity machine. Further, the query is encrypted only once.

Performance at the $SP$. There are two types of queries in the processing of SPsublso.
Figure 4.7: Performance on varying query sizes.
The first type of the queries are those *pruned* by SPRefine. Fig. 4.7(b) reports the percentage of such queries. In particular, we note that the PubChem queries $Q_{16}-24$ are completely pruned. Fig. 4.7(c) shows the average query time of those pruned queries, which is largely occupied by the private inner product. It is unsurprising that the time increases with the query size. They are smaller than 65ms and 140ms on Aids and PubChem, respectively.

The second type is the *non-pruned* queries that pass SPRefine. For these queries, we report the percentage of pruned possible mappings in Fig. 4.7(d), which can be calculated by the number of flipped 1s by SPRefine. The average query times are shown in Fig. 4.7(e). For most queries, the query times are smaller than 1s. The query time of $Q_8$ is the longest but it is still smaller than 1.4s.

**Performance at the client side.** We report the performance at the client side in Fig. 4.7(f). The times required are tiny, for instance, about 9ms from $Q_8$ of PubChem and clearly smaller than 2ms for other queries. The average number of rounds between $SP$ and client is usually small (Fig. 4.7(g)). Since many invalid partial mappings are pruned, the total message size sent to the client (Fig. 4.7(h)) is small (around 150KB in worst case). In each round, at most 16KB of messages are sent.

**Comparison with the naïve method.** Assume that the whole database was transferred to the client. We run one of the most popular non-indexing subgraph isomorphism algorithms VF2 [100]. The total AQT for all query sets on Aids and PubChem at the client side are up to 20ms and 30ms, respectively. In comparison, after the encryption for each query, the computation of our techniques at the client side requires only a few milliseconds on average (Fig. 4.7(f)). That is, we save most of the computations at the client.

**Effectiveness of SPEnum.** In Fig. 4.8, we verify the effectiveness of SPEnum by varying the starting pruning depth $d$ to (3, 4, 5). The query set is $Q_8$.

**Performance at the $SP$.** Fig. 4.8(a) shows the query time at $SP$. It is obvious that as the value $d$ increases, the search space increases, the query time increases.

**Performance at the client side.** Fig. 4.8(b) shows the decryption time at the client side increases with $d$ and its trend closely follows that of the query times. The average number
of rounds between $SP$ and client (Fig. 4.8(c)) decreases as the value $d$ increases because the protocol in SPEnum is a BFS. The message size increases according to $d$, as shown in Fig. 4.8(d). However, importantly, by Prop. 4.5.8, the probabilities that $SP$ can determine the structures decrease with $d$ increases.

**Effectiveness of SPRefine.** We verify the effectiveness of SPRefine by varying SI. We ranged maxH, and the maximum values for MaxDeg, Occur and Sup from 4 to 8. In this experiment, the query set is $Q_8$, and the starting pruning depth $d$ of SPEnum is 3.

**Encryption time.** Figs. 4.9 (a) and (b) show the encryption times of $G$ and $Q$, respectively. As the maximum values increase, the encryption times of both $G$ and $Q$ increase.

**Effectiveness of SPRefine.** Fig. 4.9(c) shows the average percentage of queries that are pruned by SPRefine with different maximum values in SI. We note that the pruning effectiveness on different maximum values are similar to each other, which are almost 96% for queries on both Aids and PubChem. That means for each $v_j \in V(Q)$, $v_k \in V(G)$, h-Hop$_{\ell}(v)$ may differ with each other within 4 hops with very high probabilities if $M(j, k)$ is flipped to 0. However, the $SP$ has no precise knowledge about the encrypted SIs. Fur-
Figure 4.9: Effectiveness of $S_I$.

ther, by Prop. 4.5.9, the probability that the $SP$ can determine the structures decreases as $\max H$ increases.
Performance at the $SP$. Fig. 4.9(d) shows the average query time of queries pruned by SPRefine, which mainly involves the time for private inner products. As expected, the times are small. Since the pruning of SI is very similar under different maximum values (by Fig. 4.9(c)), the query times for those non-pruned queries (the queries pass SPRefine) are similar, shown in Fig. 4.9(e). The times are around 400ms and 1.4s for Aids and PubChem, respectively.

Performance at the client side. Since the query times are similar to different maximum values on SI, the decryption times at the client side shown in Fig. 4.9(f) are also very similar. The average number of rounds between the $SP$ and the client are shown in Fig. 4.9(g), which are around 8 and 11 for Aids and PubChem respectively. The size of the received messages at client is shown in Fig. 4.9(h), which are around 17KB and 145KB, respectively.

4.6.3 Experiments on Synthetic Datasets

For the synthetic datasets, we selected three synthetic datasets (denoted as SYN-1, SYN-2 and SYN-3) from [40]. We note that the number of distinct vertex labels significantly affects the performance. We varied the number of distinct vertex labels of these three datasets to 20, 50, and 80, respectively. The average size and density of each graph are 30 and 0.5 respectively.

We note that their experimental results are similar to those of real datasets (presented in Sec 4.6). Hence, we only report two major performance results, shown in Fig. 4.10. Fig. 4.10 (a) reports the average query times on non-pruned queries. All of them are no larger than 1s. Fig. 4.10 (b) shows the decryption time at the client side. The decryption time on $Q_{16}$ under SYN-2 is the largest, which is only 5ms. In general, the average time spent at the client side is very small.
Figure 4.10: Performance on synthetic dataset.

(a) AQT on non-pruned QnS

(b) Avg. decryption time
Chapter 5

Asymmetric Structure-Preserving

Subgraph Queries for Large Graphs

In Chapter 4, we have studied the structure-preserving subgraph query in the scenario of outsourced graph database. In this chapter, we revise the system setting for another application scenario. In particular, we assume that the SP is the owner of the data graph, i.e., data graph is public. In this case, we propose subgraph query service over graph database with large graphs to protect the structural information of the query graph only.

5.1 Problem Formulation

This section formulates the technical problem. More specifically, it presents the system model, attack model, privacy target, and problem statement.

System model. The system model resembles the classical server-client model, which contains two parties (illustrated in Fig. 5.1): (1) A Service Provider (SP) and (2) the query client (or simply client). The SP is equipped with powerful computing utilities such as a cloud and hosts a subgraph query service for publicly known graph data $G$. The client encrypts his/her query $Q$ using a secret key (generated by himself/herself) as $Q_k$ and submits $Q_k$ to the SP. The SP then processes the client’s encrypted query $Q_k$ over the data $G$, and returns an encrypted result to the client. The client decrypts the result to
obtain the query answer.

**Attack model.** We assume the semi-honest (adversary) model which is widely used in the database literature [14, 15, 47, 58], where the SP is honest-but-curious. That is, the attacker may be the SP or another adversary hacking the SP. The SP performs computations according to the system model but it may be interested in inferring secrets. For presentation simplicity, we often call the attacker the SP. We assume that the attacker can be both eavesdropping and adopting the chosen plaintext attack (CPA) [58].

**Privacy target.** To facilitate technical discussions, we assume that the privacy target is to protect the structures of the query graph $Q$ from the SP under the attack model defined above. The structural information of $Q$ that we consider is the adjacency matrices of $Q$ (i.e., the edge information of $Q$). It is obvious that the complete structure of a query can be derived from the edge information.

To sum up, the problem statement of this work can be stated as follows: *Given the above system and attack model, we seek an efficient approach to complete the subgraph query service while preserving the privacy target.*

![Figure 5.1: Overview of the system model.](image)

![Figure 5.2: Overview of our approach.](image)
5.2 Preliminaries and Overview

In this section, we first provide preliminary concepts related to subgraph queries. Then, we present an overview of our proposed solution.

5.2.1 Data Model

The graph $G = (V, E, \Sigma, L)$ considered in this work is an undirected labeled connected graph, where $V(G)$, $E(G)$, $\Sigma(G)$ and $L$ are the set of vertices, edges, vertex labels and the function that maps a vertex to its label, respectively. We use $\text{NB}(v, G)$ to denote the set of neighbors of $v$ in $G$. We use $\text{occ}(\ell, G)$ to represent the number of occurrences of the label $\ell$ in $V(G)$. We use $M_G$ to represent the adjacency matrix of $G$. $M_G(v_i, v_j)$ is a binary value, where $M_G(v_i, v_j) = 1$ if $(v_i, v_j) \in E(G)$, and otherwise 0. The adjacency matrix $M_G$ represents the edge information. For the clarity of technical details, we present our technique with graphs having vertex labels only. The proposed techniques can be extended to support graphs with edge labels with minor modifications.

5.2.2 Overview of Our Approach

An overview of our solution is sketched in Fig. 5.2. Our solution essentially consists of the algorithms at the client side and those at the $SP$ side.

**Client-side algorithms.** For the algorithms at the client side, we propose performing lightweight optimization and encryption on the query graph $Q$. (1) We first analyze the query to determine the starting label $\ell_s$ and the minimum height $h$ of $Q$, which are use-
ful for minimizing the number and the sizes of candidate subgraphs of $G$. A *candidate subgraph* is a subgraph in $G$ that may contain a candidate matching, whereas a *candidate matching* is a subgraph of the candidate subgraph that may generate a *candidate mapping* between $Q$ and $G$. (2) We then propose a robust encoding scheme for $Q$ (of any size). (3) We adopt the *private-key encryption scheme* CGBE to encrypt the encoded $Q$ to encrypted query $Q_k$, which is issued to the $SP$ for query processing. (4) The client decrypts the encrypted answer returned by the $SP$.

**Server-side algorithms.** The main ideas of the algorithms at the $SP$ side are to localize and minimize the enumeration of candidate mappings between $Q$ and $G$ in candidate subgraphs. (1) The $SP$ first efficiently determines the candidate subgraphs $CS_s$ (subgraphs) starting from each starting vertex $s$ of the label $\ell_s$ with the traversal depth $h$. We propose *neighborhood containment* (NC) to minimize each $CS_s$ in the absence of the structure of $Q$. Subsequently, it minimizes the number of candidate matchings to be enumerated by the $SP$. (2) In each $CS_s$, the $SP$ enumerates all candidate matchings ($CM_s$) and candidate mappings. We propose a *canonical labeling-based subgraph cache* and apply *neighborhood equivalent class* (NEC) to further avoid redundant $CM_s$ and candidate mappings, respectively. (3) We propose *structure-preserving verification* $SPVerify$, where multiple encrypted messages $R_k$ (with negligible false positives) are returned to the client for decryption of the result.

$SPVerify$ is derived from the seminal subgraph isomorphism algorithm: the Ullmann’s algorithm [99]. The major benefit of Ullmann’s algorithm is that its computation flow is simple; hence, we can cast the algorithm into a series of matrix operations (additions and multiplications). Since the encryption of $SPVerify$ supports such matrix operations, privacy is preserved.

We also note that $SPVerify$ may send multiple messages to the client for decryption, which may result in high decryption and network communication costs. Thus we propose $SPVerify^*$. The major difference between $SPVerify^*$ and $SPVerify$ is that $SPVerify^*$ uses different query encodings according to different query sizes and significantly fewer encrypted messages are returned for decryption, and the query size is smaller than a system-
related constant.

Furthermore, we claim that our propose techniques can easily handle incremental updates for evolving graphs as they are online algorithms.

5.3 Query Preprocessing at the Client

In this section, we introduce a preprocessing method of the query graph. It comprises three steps: (1) retrieving optimization parameters; (2) encoding the query; and (3) encrypting the encoded query. The encrypted query is sent to the $SP$.

5.3.1 Retrieving Parameters for Optimization

In order to minimize (1) the size of each candidate subgraph $CS_s$ and (2) the total number of $CS_s$s, the $SP$ requires the minimum height $h$ of $Q$ and, in the meantime, the starting label $\ell_s$ of $CS_s$s that is infrequent in $G$. These parameters ($h$ and $\ell_s$) can be efficiently retrieved by the client in $O(|V(Q)|^2)$.

Given a starting label $\ell_s$, the $SP$ generates $CS_s$s by a breadth first search bounded by the depth $h$ starting at each vertex of $G$ having the label $\ell_s$ (to be detailed in Sec. 5.4.1). On the one hand, to minimize the size of each $CS_s$, we simply find the spanning tree of $Q$ with a minimum height $h$ rooted from a vertex $u$, where $u \in V(Q)$ and $\ell_s = L(u)$. Intuitively, the smaller the value $h$, the smaller the size of each $CS_s$. Note that we cannot choose the vertex $u$ with $h = 1$ since it trivially leaks the structure of $Q$ (to be analyzed in Sec. 5.6). When there is a tie ($i.e.$, when vertices $u$ and $v$ of $Q$ have the same $h$) the client selects the vertex of the label that is less frequent in $G$, simply because the number of $CS_s$s is bounded by the occurrence of the label in $G$.

Example 5.3.1. Fig. 5.3 (a) shows an example of the selection of the starting label of query $Q$. The heights of the spanning trees rooted from $u_1$, $u_3$, and $u_4$ are 2. $u_1$ is finally chosen as the starting label as $\text{occ}(0, G) < \text{occ}(2, G)$, where $L(u_1) = 0$, and $L(u_3) = L(u_4) = 2$. $u_2$ is not considered because the height of its spanning tree is 1.
5.3.2 Query Encoding

For presentation brevity, we present an encoding scheme for the query $Q$ (in Definition 5.3.1) to facilitate the discussion of the subsequent encryption scheme. This encoding is extended for further optimization (to be proposed in SubSec. 5.5.2).

**Definition 5.3.1.** The encoding of the entries of $M_Q$ are:

\[
\forall u_i, u_j \in V(Q), \begin{cases} 
M_Q(u_i, u_j) = q & \text{if } M_Q(u_i, u_j) = 0; \\
M_Q(u_i, u_j) = 1 & \text{otherwise,}
\end{cases}
\]

where $q$ is a large prime number.

**Example 5.3.2.** Fig. 5.3 (a) also shows an example of the encoding of $Q$ by Def. 5.3.1. The entries in $M_Q$ with values 0 are replaced by the large prime $q$.

5.3.3 Query Encryption

Based on the encoding of $Q$, we adopt our private-key encryption scheme (cyclic graph based encryption scheme CGBE) to encrypt the encoding of $Q$ ($M_Q$). CGBE not only allows for efficient encryption and decryption but also supports both partial additions and multiplications, which is the core of efficient structure-preserving computation.

**Background on cyclic group.** Prior to the presentation of the definition of CGBE, we first recall the preliminaries of cyclic group [58]. Let $G$ be a group. $p = |G|$ is denoted as the order of $G$. In particular, $\forall g \in G$, the order of $G$ is the smallest positive integer $p$ s.t., $g^p = 1$. Let $\langle g \rangle = \{ g^i : i \in \mathbb{Z}_p, g^i \in \mathbb{Z}_n \} = \{ g^0, g^1, \cdots, g^{p-1} \}$ denote the set of elements generated by $g$. The group $G$ is called cyclic if there exists an element $g \in G$ such that $\langle g \rangle = G$. $g$ is called a generator of $G$.

**CGBE scheme.** The cyclic group based encryption scheme is defined as follows.

**Definition 5.3.2.** The cyclic group based encryption scheme is a private-key encryption scheme, denoted as CGBE $= (\text{Gen, Enc, Dec})$, where
• **Gen** is a key generation function, which generates a secret key \( x \) uniformly at random, a cyclic group \( \langle g \rangle = \{ g^i : i \in \mathbb{Z}_p, g^i \in \mathbb{Z}_n \} \). It outputs the private keys as \((x, g)\) and the value \( p \) which is known to the public.

• **Enc** is an encryption function, which takes as input a message \( m \) and the secret key \((x, g)\). It chooses a random value \( r \), and outputs the ciphertext
  \[
e = mr g^{x} \pmod{p} \]

• **Dec** is a decryption function, which takes as input a ciphertext \( e \), and the secret key \((x, g)\). It outputs
  \[
  mr = eg^{-x} \pmod{p}
  \]

Note that the decryption function Dec in CGBE only decrypts the ciphertext \( e \) as a product of the message \( m \) and the random value \( r \).

**Query encryption.** With CGBE, we define the encryption of \( M_Q \) as follows.

**Definition 5.3.3.** The encryption of \( Q \) is denoted as \( Q_k \), \( Q_k = \{ V, M_{Q_k}, \Sigma, L \} \), where \( \forall u_i, u_j \in V(Q) \),
  \[
  M_{Q_k}(u_i, u_j) = \text{Enc}(M_Q(u_i, u_j), x, g)
  \]

**Example 5.3.3.** For example, \( \forall u_i, u_j \), if \( M_Q(u_i, u_j) = 1 \), then \( M_{Q_k}(u_i, u_j) = \text{Enc}(1) = rg^{x} \pmod{p} \); and if \( M_Q(u_i, u_j) = q \), then \( M_{Q_k}(u_i, u_j) = \text{Enc}(q) = rgq^{x} \pmod{p} \).

**Discussion.** We remark that the client holds the secret keys \((x, g)\) for decryption and moreover, determines the constant \( c \) and an encrypted value \( I \) for encrypting verification results (to be discussed in Sec. 5.5). At last, \( \ell, h, Q_k, c, I \) and \( p \) are sent to the \( SP \) for structure-preserving query processing.

### 5.4 Minimized SP Mapping Generation

The query preprocessing at the client side (in Sec. 5.3) generates \((\ell, h, Q_k, c, I, p)\) for the \( SP \). Upon receiving these, the \( SP \) performs structure-preserving sublso (termed SPsublso), presented in Algo. 8.
As outlined in Sec. 1, the $\mathcal{S}P$ first minimizes the number of candidate mappings to-be-verified. For brevity, we focus on the most crucial procedures: candidate subgraph generation (Sec. 5.4.1), candidate matching generation (Sec. 5.4.2), and candidate mapping enumeration (Sec. 5.4.3).

### 5.4.1 Candidate Subgraph Generation

To avoid enumerating mappings on a possibly large graph, the $\mathcal{S}P$ first generates candidate subgraphs (Fig. 5.3(b)), where possible mappings can only be embedded in those subgraphs. A candidate subgraph is formally described in Def. 5.4.1.

**Definition 5.4.1.** A candidate subgraph started from $s \in V(G)$, denoted as $CS_s$, is an induced subgraph of $G$, s.t.

1. $L(s) = \ell_s$;

2. $\forall v \in V(CS_s), v$ is reachable from $s$ within $h$ hops;

3. $\forall \ell, \ell \in \Sigma(CS_s) \iff \ell \in \Sigma(Q)$; and

4. $\forall \ell \in \Sigma(CS_s), \text{occ}(\ell, CS_s) \geq \text{occ}(\ell, Q)$.

**Example 5.4.1.** Suppose $L(s) = \ell_s = 0$ and $h = 2$. Fig. 5.3(b) sketches an example of a candidate subgraph $CS_s$ (the grey-colored shadow) rooted from $s$ of $G$. For each vertex $v$ in $CS_s$, $v$ is reachable from $s$ within 2 hops. The set of labels of $Q$ is the same as that of $CS_s$ (i.e., $\Sigma(CS_s) = \Sigma(Q)$). For each label $\ell$ in $CS_s$, $\text{occ}(\ell, CS_s) \geq \text{occ}(\ell, Q)$.

**Initial generation.** GenCandSubGraph (Procedure 8.1, Lines 8-17) shows the generation of candidate subgraphs. Algo. 8 first initializes the $CS_s$ as $\emptyset$ (Line 1). Then, for each vertex $s \in V(G)$, where $L(s) = \ell_s$, it invokes GenCandSubGraph (Line 1). GenCandSubGraph simply generates $CS_s$ by a simple breadth first search method started from $s$ on $G$ within $h$ hops (Lines 10-15). $V_{CS_s}$ is to record the vertices of $CS_s$ determined so far. For each vertex $v \in V_{CS_s}$, $v$ must be reachable from $s$ within $h$ hops (Lines 13-15), and $L(v) \in \Sigma(Q)$ (Line 13). If $\forall \ell \in \Sigma(CS_s), \text{occ}(\ell, CS_s) \geq \text{occ}(\ell, Q)$ (Line 16), $CS_s$ is set to the induced subgraph of $V_{CS_s}$ in $G$ (Line 17).
Minimization by neighborhood information. Since the sizes of candidate subgraphs have a significant impact on performance, we propose MinCandSubGraph (Procedure 8.2) to minimize the size of each \( CS_s \). MinCandSubGraph is derived based on our notion of neighborhood containment class (NC) of \( CS_s \), defined as follows.

**Definition 5.4.2.** Given \( N = \{v_1, v_2, \cdots, v_n\} \) of \( V(CS_s) \), \( N \) is a neighborhood containment class (NC), denoted as \( v_1 \sqsubseteq v_2 \sqsubseteq \cdots \sqsubseteq v_n \), iff \( \forall v_i, v_j \in N, i < j \),

1. \( L(v_i) = L(v_j) \);

2. (a) \( NB(v_i, CS_s) \subseteq NB(v_j, CS_s) \), if \( N \) is an independent set in \( CS_s \); or 
   
   (b) \( NB(v_i, CS_s) \cup \{v_i\} \subseteq NB(v_j, CS_s) \cup \{v_j\} \), if \( N \) is a clique of \( CS_s \).

Based on Def. 5.4.2, the vertices of a candidate subgraph \( CS_s \) exhibit a total ordering with respect to the \( \sqsubseteq \) relationships. We have the following lemma for minimizing the size of a candidate subgraph by keeping the “top” vertices in the subgraph. The intuition is that the reduced \( CS_s \) preserves all the structures of the original \( CS_s \). The proof is established via a simple contradiction.

**Lemma 5.4.3.** Denote an NC \( N \) as \( \{v_1, v_2, \cdots, v_n\} \), where \( N \subseteq V(CS_s) \) of a graph \( G \). Denote the reduced \( V(CS_s) \) (denoted as \( CS_s' \)) is the induced subgraph of \( V(CS_s) \setminus (\mathbb{N} \setminus \mathbb{N}_k) \) of \( CS_s \), i.e. \( \mathbb{N}_k = \{v_{n-k+1}, v_{n-k+2} \cdots, v_n\} \) contains top-\( k \) vertices of \( N \) that are kept, where \( k = \text{occ}(L(v_1), Q) \). Then, the answer of \( Q \) on \( CS_s \) is the same as that on \( CS_s' \).

**Proof.** (Sketch) We prove that the removed structures in \( CS_s \) are preserved in \( CS_s' \). We only consider \( N \) is an independent set for simplicity as the argument when \( N \) is a clique is similar.

We denote \( \mathbb{N}_k = N \setminus \mathbb{N}_k \) which represents the removed vertices in \( N \). Any possible structure in \( CS_s \) must be formed by some vertices from \( \mathbb{N}_k \) and \( \mathbb{N}_k \). The former can be formulated as \( \mathbb{N}_k^x \subseteq \mathbb{N}_k \), for some \( x (x \geq 1) \), whereas the latter can be \( \mathbb{N}_k^y \subseteq \mathbb{N}_k \), where \( y (y = k - x) \). Denote \( \mathbb{N}_k^x = \mathbb{N}_k^x \cup \mathbb{N}_k^y \). Denote \( CS_s^k \) as the induced subgraph of \( V(CS_s) \setminus (\mathbb{N} \setminus \mathbb{N}_k) \) and \( CS_s^{k'} \) and that of \( V(CS_s) \setminus (\mathbb{N} \setminus \mathbb{N}_k) \), respectively. For any substructures in the induced subgraph of \( CS_s^{k'} \) and for each vertex \( v_x \in \mathbb{N}_k^x \), we can always find a
Figure 5.4: (a) Construction of candidate matchings; and (b) Enumeration of possible mappings.

distinguished vertex $v_y \in N_k \setminus N^y_k$ such that $\text{NB}(v_x, CS_s) \subseteq \text{NB}(v_y, CS_s)$, by Def. 5.4.2. This implies that $CS^{k'}_s \subseteq CS^k_s$. Therefore, $CS^r_s$ preserves all the structures in $CS_s$ by only keep $N_k$.

Example 5.4.2. Reconsider Example 5.4.1. $\{v_1, v_2\}$ is an NC as $L(v_1) = L(v_2)$, $\text{NB}(v_1, CS_s) \subseteq \text{NB}(v_2, CS_s)$ and $\{v_1, v_2\}$ forms an independent set of $CS_s$ in Fig. 5.3(c). Since $\text{occ}(1, Q) = 1$, by Lemma 5.4.3, we keep the top-1 vertex. It can be seen that the answer of $Q$ remains the same after removing either $v_1$ or $v_2$ from $CS_s$. For another example, let’s consider the NC $\{v_4, v_5, v_6\}$ in Fig. 5.3(c), as the neighborhood of $v_4$ is contained by that of $v_5$. Hence, $v_4 \sqsubseteq v_5$. Similarly, $v_5 \sqsubseteq v_6$. $\{v_4, v_5, v_6\}$ forms an independent set. Again, by Lemma 5.4.3, we keep only the top-1 vertex, i.e., $v_6$. The answer of $Q$ remains the same after removing $v_4$ and $v_5$. All in all, Fig. 5.4(a) shows $CS_s$, the candidate subgraph after the minimization.

The minimization procedure MinCandSubGraph. Procedure 8.2 shows the minimization of $CS_s$ by NC. For each $\ell \in \Sigma(CS_s)$, a set $N$ of NC is first initialized as $\{\}$ (Line 18). For each vertex $v$ of $CS_s$ with the label $\ell$, sorted in ascending order of $|\text{NB}(v, CS_s)|$ (Line 19) for efficiency, MinCandSubGraph checks if there is an $N$ in $\mathcal{N}$, such that $N \cup \{v\}$ forms an NC by Def. 5.4.2 (Line 20). If so, $v$ is then inserted into $N$ (Line 21). Otherwise, the algorithm creates a new $N = \{v\}$ and unions $N$ to $\mathcal{N}$ (Line 22). After the generation of NC of $CS_s$ for the label $\ell$, $CS_s$ can be minimized by Lemma 5.4.3 via keeping the top-$k$ vertices in each $N$, $N \in \mathcal{N}$, $k = \text{occ}(\ell, Q_k)$ (Lines 23-24).

Complexity. The complexity of the generation of NC in Procedure 8.2 is $O(d_{\text{max}}|V(CS_s)|^2)$, where $d_{\text{max}}$ is the maximum degree of the vertices in $CS_s$. In practice, $|V(CS_s)|$ is often in the order of hundreds, which is small.
Algorithm 8 SPsubIso \((Q_k, G, \ell_s, h)\)

**Require:** The encrypted query graph \(Q_k\), data graph \(G\), starting label \(\ell_s\) and hop \(h\)

**Ensure:** The encrypted result \(R_h\)

1. Initialize \(CS_s = CM_s = \emptyset\), \(Cache = \emptyset\), and \(R_h = 1\)
2. **for each** vertex \(s \in V(G)\) with the starting label \(\ell_s\)
3. GenCandSubGraph\((Q_k, G, s, h, CS_s)\) /* By Def. 5.4.1 */
4. MinCandSubGraph\((Q_k, CS_s)\) /* Minimize \(CS_s\) */
5. Initialize set \(V_{CM_s} = \{s\}\)
6. GenCandMatch\((V_{CM_s}, Q_k, CS_s, R_h, Cache)\) /* By Def. 5.4.4 */
7. Return \(R_h\)

**Procedure 8.1 GenCandSubGraph \((Q_k, G, s, h, CS_s)\)**

8. Initialize a queue \(Visit\) and a set \(V_{CS_s}\) as empty
9. \(Visit.push(s)\), \(V_{CS_s}.insert(s)\), \(s.hop() = 0\)
10. **while** \(Visit\) is not empty /* BFS method */
11. \(v = Visit.pop()\)
12. **if** \((v.hop()) = h\) **continue** /* By 2. in Def. 5.4.1 */
13. **for each** \(v' = NB(v, G), v' \notin V_{CS_s} \land L(v') \in \Sigma(Q_k)\)
14. \(Visit.push(v')\), \(V_{CS_s}.insert(v')\)
15. \(v'.hop() = v.hop() + 1\) /* By 4. in Def. 5.4.1 */
16. **while** \(\exists v \in \Sigma(V_{CS_s}), s.t. \text{occ}(\ell, V_{CS_s}) < \text{occ}(\ell, Q_k)\)

    remove all \(v\) from \(\Sigma(V_{CS_s})\), where \(v \in \Sigma(V_{CS_s})\) and \(\Sigma(v) = \ell\)
17. \(CS_s = \text{GenInducedSub}(G, V_{CS_s})\)

**Procedure 8.2 MinCandSubGraph \((Q_k, CS_s)\)**

18. **for each** \(\ell \in \Sigma(CS_s), N' = \{\} \) /* \(N'\) is a set of NC */
19. /* Ascending ordered by \(|NB(v, CS_s)|\) */
20. **for each** \(v \in V(CS_s), L(v) = \ell,\)
21. **if** \(\exists N \in N', s.t.\)
22. /* By Def. 5.4.2 */
23. \(1\) \(\{v\} \cup N\) forms an independent set (or a clique); and
24. \(2\) \(NB(v, CS_s)\) (or \(NB(v, CS_s) \cup \{v\}\)) contains those of vertices in \(N\).
25. \(N.insert(v)\) /* Ordered by \(\subseteq\) */
26. **else** create a new \(N, N = \{v\}, N' = N' \cup \{N\}\)
27. **for each** \(N \in N', N_k = \{v_{n-k+1}, \ldots, v_n\}, k = \text{occ}(\ell, Q_k)\)
28. Remove \(N \setminus N_k\) from \(CS_s\) /* By Lemma. 5.4.3 */

### 5.4.2 Candidate Matching Generation

A unique challenge in structure-preserving query processing is that, in the absence of query structure, the \(SP\) matches \(Q_k\) to *multiple* possible subgraph structures in \(CS_s\). We call such subgraph structures *candidate matchings*. In contrast, if the query structures were not kept secret, the candidate matching was known to be \(Q\). Fig. 5.4(a) shows
Algorithm 9 GenCandMatch \((V_{CMs}, Q_k, CS_s, R_k, \text{Cache})\)

1: if \(V_{CMs}\) are enumerated before /* \(V_{CMs}\) is checked */
2: return
3: if \(|V_{CMs}| = |V(Q)|\)
4: \(CM_s = \text{GenInducedSub}(CS_s, V_{CMs})\)
5: if \(\text{Cache}\).isHit(can\((CM_s)\)) /* \(CM_s\) is checked before */
6: return
7: \(\text{Cache}.\text{insert}(\text{can}(CM_s))\) /* Insert \(\text{can}(CM_s)\) into Cache */
8: if \(V(Q) \leq \delta\) /* Insert subgraphs of \(CM_s\) into Cache */
9: \(\text{Cache}.\text{insert}(\text{can}(CM'))\),
   where \(CM' \subseteq CM_s, |V(CM')| = |V(CM_s)|\)
10: GenAllMap\((Q_k, CM_s, R_k)\) /* Generate candidate mappings */
11: for each \(v \in V_{CMs}\)
12: for each \(v' \in \text{NB}(v, CS_s)\) in descending order, \(v' \notin V_{CMs}\)
13: if \(\text{occ}(\ell, V_{CMs}) < \text{occ}(\ell, Q_k), \ell = L(v')\) /* By Def. 5.4.4 */
14: \(V_{CMs}.\text{insert}(v')\)
15: GenCandMatch \((V_{CMs}, Q_k, CS_s, R_k)\)
16: \(V_{CMs}.\text{remove}(v')\)

four candidate matchings, \(CM_{s1}, CM_{s2}, CM_{s3},\) and \(CM_{s4}\). For each matching, candidate mappings are enumerated. It is evident that a naive enumeration of all candidate matchings can be inefficient. In this subsection, we propose GenCandMatch to efficiently generate candidate matchings. The main idea is to avoid generating redundant matchings from \(CS_s\).

**Definition 5.4.4.** A candidate matching, denoted as \(CM_s\), is a connected induced subgraph of \(CS_s\), s.t.

1. \(|V(CM_s)| = |V(Q)|\); and
2. \(\forall \ell \in \Sigma(CS_s), \text{occ}(\ell, CM_s) = \text{occ}(\ell, Q)\).

**Example 5.4.3.** Fig. 5.4 (a) lists all the \(CM_s\)s enumerated from \(CS_s\). \(\forall CM_{si}, i \in \{1, ..., 4\}, |V(CM_{si})| = |V(Q)|, \) and \(\forall \ell \in \Sigma(CS_s), \text{occ}(\ell, CM_{si}) = \text{occ}(\ell, Q)\).

**Elimination of redundant \(CM_s\).** We make two observations from Example 5.4.3 and Fig. 5.4. (1) \(CM_{s2}\) is graph-isomorphic to \(CM_{s3}\). If candidate mappings are generated from \(CM_{s2}\), it is obvious that generating mappings from \(CM_{s3}\) is redundant. (2) \(CM_{s1}\) is a supergraph of \(CM_{s2}\). One can simply generate mappings from \(CM_{s1}\), and skip \(CM_{s2}\) and \(CM_{s3}\).
To remove the redundancies mentioned above, it is exactly to solve the following problem: “given a graph \( G \) and a graph database \( \mathcal{G} : \{G_1, \ldots\} \), how to efficiently determine if \( G \) is a subgraph of \( G' \), \( G' \in \mathcal{G} \)?” Such a problem has been extensively studied before (e.g., [94, 106, 122]). Existing solutions involve an index computed offline. In our context, candidate matchings are enumerated \textit{online}. Hence, the existing solutions cannot be directly applied.

**Canonical labeling-based subgraph cache.** Let’s recall a crucial property of canonical labeling. In the context of graph query processing, the canonical labeling of a graph \( G \) is denoted as \( \text{can}(G) \), and \( \text{can}(G) = \text{can}(G') \) if and only if \( G \) is isomorphic to \( G' \). While the cost for computing the canonical labeling of a graph is not yet known (P or NP), the cost for comparing whether two graphs are isomorphic using the canonical labeling is \( O(1) \), once computed. This work adopts the state-of-the-art labeling called \textit{minimum dfs code} [106] from the literature.

For each query, we propose \textit{Cache} to store \( \text{can}(CM_s) \), where each \( CM_s \) is the checked candidate matching. Once a new \( CM'_s \) is generated, we first check if \( \text{can}(CM'_s) \) is already in \textit{Cache}. If so, \( CM'_s \) is discarded. Otherwise, we insert \( \text{can}(CM'_s) \) into \textit{Cache}. Further, we continue to enumerate subgraphs \( CM' \)’s from \( CM'_s \), where for each \( CM' \), \( |V(CM')| = |V(CM'_s)| \), \( CM' \subseteq CM'_s \), and \( \text{can}(CM') \) is stored in \textit{Cache}. Putting subgraphs of \( CM'_s \) increases the chance of pruning by \textit{Cache}. However, the \textit{trade-off} is that as the query size increases, the computational cost for enumerating all subgraphs of a \( CM'_s \) increases exponentially. Thereby, for practical purposes, we enumerate all of the subgraphs \( CM' \)’s of \( CM'_s \) only if \( |V(Q)| \leq \delta \), where \( \delta \) is a user-defined threshold.

**Example 5.4.4.** The top of Fig. 5.4 (a) shows the idea of the canonical labeling-based cache. We assume that \( \delta \) is 3, and the sequence of the generation of \( CM_s \) is from \( CM_{s1} \) to \( CM_{s4} \). \( CM_{s3} \) is eliminated as \( \text{can}(CM_{s2}) \) is in \textit{Cache}. If we set \( \delta \) to 5, then \( CM_{s2} \) and \( CM_{s3} \) are both eliminated, because \( CM_{s2} \) is a subgraph of \( CM_{s1} \), and when \( CM_{s1} \) is processed, \( \text{can}(CM_{s2}) \) is inserted into \textit{Cache}.

**The ordering in \( CM_s \) generation.** From Example 5.4.4, it can be observed that the ordering in \( CM_s \) generation affects the performance of the cache, when \( |V(Q)| \leq \delta \). Suppose
$\delta = 5$. Assume $CM_{s2}$ is generated before $CM_{s1}$. Then, $CM_{s2}$ is not eliminated. In general, the earlier the larger $CM_{s}$s are generated, the better the performance is. Therefore, we find a simple ordering for $CM_{s}$ generation, by greedily adding vertices to the $CM_{s}$ by the degree of each vertex.

$CM_{s}$ generation. In Algo. 8, $Cache$ is initialized to $\emptyset$ (Line 1), and the vertex set of each generated $CM_{s}$, denoted as $V_{CM_{s}}$, is initialized to $\{s\}$ (Line 5). In Line 6, the algorithm for the generation of $CM_{s}$, denoted as $GenCandMatch$, presented in Algo. 9.

In $GenCandMatch$ (Algo. 9), $CM_{s}$ is generated from $CS_{s}$ (Lines 11-16) until the size of the matching is the same as the query (Lines 1-10). For each vertex $v \in V_{CM_{s}}$ (Line 12), it attempts to add $v$’s neighboring vertex $v' \in NB(v, CS_{s})$, where $v' \notin V_{CM_{s}}$, and it adds the $v'$ with a large degree first (Deg$(v', CS_{s})$). If occ$(\ell, V_{CM_{s}}) < occ(\ell, Q_{k})$ (Line 13), where $\ell = L(v')$, $v'$ is then added to $V_{CM_{s}}$ (Line 14). $GenCandMatch$ is called recursively (Line 15) until $|V_{CM_{s}}| = |V(Q_{k})|$ (Lines 1-10). $CM_{s}$ is an induced subgraph of $V_{CM_{s}}$ (Line 4).

For each generated $CM_{s}$, can($CM_{s}$) is checked if can($CM_{s}$) is already in $Cache$ (Line 5). If yes, there exists a $CM'$ in $Cache$ such that $CM'$ is isomorphic to $CM_{s}$. By the property of canonical labeling, $CM_{s}$ can be eliminated (Lines 5-6). Otherwise, we add can($CM_{s}$) into $Cache$ (Line 7). If $|V(Q)| \leq \delta$, we enumerate the subgraphs $CM'$ of $CM_{s}$, where $|V(CM')| = |V(CM_{s})|$, and insert can($CM'$) into $Cache$ (Lines 8-9).

At last, GenAllMap (see Algo. 10) is invoked (Line 10) to generate all possible mappings (see SubSec. 5.4.3) between $Q_{k}$ and $CM_{s}$.

**5.4.3 Candidate Mapping Generation**

When a new candidate matching $CM_{s}$ is generated, Algo. 10 invokes GenAllMap to enumerate all possible mappings between $Q_{k}$ and $CM_{s}$.

**Elimination of redundant mappings by NEC.** Recall that the number of mappings is exponential to the size of $CM_{s}$. However, in practice, many mappings are redundant. Hence, before generating the mappings, we utilize neighborhood equivalent classes NECs of $CM_{s}$ (Def. 5.4.5) to eliminate those redundant mappings. We remark that NEC is a
special case of NC. While a similar NEC has been proposed in [39] for query and data graphs, our NEC is applied to data graphs only.

**Definition 5.4.5.** Given an NC \( N = \{v_1, v_2, \ldots, v_n\} \) of \( CS_s \), where \( N \) is either an independent set or a clique of \( CS_s \), \( N \) is a neighborhood equivalent class (NEC), denoted as \( v_1 \simeq v_2 \simeq \cdots \simeq v_n \), iff \( \forall v_i, v_j \in N, v_i \subseteq v_j \text{ and } v_j \subseteq v_i \).

**Example 5.4.5.** Let’s consider the example of \( CM_{s4} \) in Fig. 5.4 (a), \( \{v_5, v_6\} \) is an NEC as \( L(v_5) = L(v_6) \) and \( NB(v_6, CM_{s4}) = NB(v_5, CM_{s4}) = \{v_3\} \).

Suppose that \( u_3 \) and \( u_4 \) (in Fig. 5.4 (a)) have been mapped to \( v_5 \) and \( v_6 \), respectively. It is not necessary to map \( u_3 \) and \( u_4 \) onto \( v_6 \) and \( v_5 \), respectively. This can be formalized as the following lemma. Foremost, we often use \( (u_i \mapsto v_i) \) to denote \( \text{map}[u_i] = v_i \) for ease of exposition.

**Lemma 5.4.6.** Suppose the following are true:

1. \( u_i, u_j \in V(Q), v_i', v_j' \in V(CM_s), L(u_i) = L(u_j) = L(v_i') = L(v_j'); \)
2. \( v_i' \simeq v_j'; \)
3. \( (u_i \mapsto v_i') \) and \( (u_j \mapsto v_j'). \)
Let \( \text{map}' \) be the mapping map except that \( (u_i \mapsto v_{j'}) \) and \( (u_j \mapsto v_{j'}) \). Then, \( \text{map} \) is a candidate mapping between \( Q \) and \( CM_s \) if and only if \( \text{map}' \) is also a candidate mapping.

The proof is omitted since it can be established by a simple proof by contradiction.

**Data structures.** (i) A vertex label mapping \( M \) is a \( m \times n \) binary matrix, \( m = |V(Q_k)| \) and \( n = |V(CM_s)| \). Specifically, \( \forall u, v, M(u, v) = 1 \) if \( L(u) = L(v) \), where \( u \in V(Q_k) \) and \( v \in V(CM_s) \); and otherwise 0. (ii) A vector map of the size \( |V(Q_k)| \) is to record a mapping from \( Q_k \) to \( CM_s \), \( \text{map}[u] = v \) (i.e., \( u \mapsto v \)) represents that vertex \( u \) in \( Q_k \) is mapped to vertex \( v \) in \( CM_s \). \( \text{map}[u] = 0 \) if \( u \) is not yet mapped. (iii) A vector used of the size \( |V(CM_s)| \) is to denote whether the vertex \( v \) in \( CM_s \) has been mapped to a vertex of \( Q_k \) and recorded in \( \text{map} \). \( \text{used}[v] = 0 \) if \( v \) is not yet mapped. In other words, \( \text{used}[v] = 1 \) if and only if \( \text{map}[u] = v \) for some \( u \in Q_k \).

**Algorithm for mapping generation.** The detailed algorithm GenAllMap is shown in Algo. 10. It first initializes the data structures, including \( M \), \( \text{used} \) and \( \text{map} \) in Lines 1-3. Line 4 constructs NEC of \( CM_s \), which is similar to that of NC in Procedure 8.2. EnumMap (Lines 6-14) is then invoked to enumerate all possible mappings. A mapping map is constructed vertex by vertex iteratively. Line 9 checks if \( v_j \) is a possible map of \( u_i \) by \( M \) and \( \text{used} \). We then exploit the equivalence class to further check if \( v_j \) can be possibly mapped to \( u_i \) (Lines 10-12). The vertices in a NEC are checked in a predefined order (e.g., lexicographical order). If \( \exists v_{j'} \) s.t. \( v_{j'} \simeq v_j \), \( j' < j \) and \( v_{j'} \) is not used before, then \( v_j \) is skipped (Line 10). If \( v_j \) passes the check, EnumMap is called recursively (Line 13) until a full mapping is constructed (Line 6).
Example 5.4.6. Fig. 5.4(b) illustrates the possible candidate mapping generation for those $CM$s of Example 5.4.3. Since $v_5 \simeq v_6$ in $CM_{s4}$, by Lemma 5.4.6, we only enumerate map$_5$, where $u_3 \mapsto v_5$ and $u_4 \mapsto v_6$, but the one with $u_3 \mapsto v_6$ and $u_4 \mapsto v_5$ is eliminated.

5.5 SP Mapping Verification

Section 5.4 presented a series of optimizations that reduce the number of mappings to be generated. Then, for each mapping map, the $SP$ verifies (in the encrypted domain) if there is no violation in map. The encrypted verification results are aggregated before they are transmitted to the client. In this section, we propose a basic verification (SPVerify) for our problem setting. Next, we propose an enhanced one (SPVerify$^*$) that aggregates many more messages but requires the query size to be smaller than a user-determined constant.

5.5.1 SPVerify

Given a mapping map between $Q_k$ and $CM_s$, we determine if $CM_s$ is a valid mapping or not. Specifically, we define the violation w.r.t. the encoding of $Q$ as follows: $\exists u_i, u_j \in V(Q)$,

$$M_Q(u_i, u_j) = 1 \land (v_i, v_j) \notin E(CM_s)$$  \hspace{1cm} (5.5.1)

where $v_i, v_j \in V(CM_s)$, $u_i \mapsto v_i$ and $u_j \mapsto v_j$. It states that there exists an edge between vertices $u_i$ and $u_j$ in $Q$, but there is no corresponding edge between the mapped vertices $v_i$ and $v_j$ in $CM_s$. We term the case in Formula 5.5.1 as a violation of subgraph isomorphism (or simply violation). A mapping without violation(s) is called a valid mapping.

Example 5.5.1. Let's take the two mappings map$1$ and map$2$ of $CM_{s1}$ in Fig. 5.4 (b) as an example. First, no violation is found in map$1$. Second, for map$2$, we find that $M_Q(u_1, u_3) = 1$ and $(s, v_4) \notin E(CM_{s1})$, where map$2[u_1] = s$ and map$2[i_3] = v_4$. Therefore, map$2$ is invalid.
Algorithm for SPVerify. The intuitive idea of SPVerify is to transform the verification steps into mathematical operations on $M_{Q_k}$ and $CM_s$, where (1) the violation (Formula 5.5.1) can be detected; (2) only matrix additions and multiplications are involved; and (3) the result can be aggregated with one message or multiple messages.

Algo. 11 shows the detailed algorithm. The inputs are a candidate mapping map, an encrypted query graph $Q_k$, a candidate matching $CM_s$ and an encrypted result $R_k$. We remark that $R_k$ is to record the aggregated result for $CM_s$s, where $R_k$ is initialized to 1 in Line 1 Algo. 8.

We initialize an intermediate result $R_i$ with a value 0 (Line 1). For each pair of vertices $(u_i, u_j)$ in $V(Q)$ and the mapped vertex pair $(v_i', v_j')$ in $CM_s$ (Lines 2-3), the following two steps are performed:

1. Additions (Lines 4-7): if $(v_i', v_j') \notin E(CM_s)$, $R_i$ is set to $(M_{Q_k}(u_i, u_j) + R_i)$ (mod $p$). This indicates that if $(u_i, u_j)$ is an edge in $Q$, $R_i$ must not contain a factor of $q$, and the decryption value of $R_i$ is non-zero (i.e., the current mapping map contains a violation (by Formula 5.5.1), which is not a valid mapping). Otherwise, no violation is caused by $(u_i, u_j)$. This sets $R_i$ to the value $I + R_i$ (mod $p$), where $I$ is an encrypted value with a factor $q$ issued by the client, $I = Enc(q)$; and

2. Multiplications (Line 8): it aggregates $R_i$ into $R_k$, by $R_k = R_k \times R_i$ (mod $p$). If there is at least one valid mapping from $Q$ to $G$, i.e., at least one $R_i$ whose decryption value is zero. The decryption value of $R_k$ must also be zero. Otherwise, it is non-zero. We remark that CGBE leads to errors if the number of $R_i$s aggregated in $R_k$ is larger than a predetermined value $M$.

The complexity of the above two steps is thereby $O(|V(Q)|^2)$.

Example 5.5.2. Fig. 5.5(a) depicts an example of SPVerify between $Q_k$ and $CM_{s1}$. There are two mappings from $Q_k$ to $CM_{s1}$ in Fig. 5.4(b). In map1, all the factors in $R_1$ contain $q$ since map1 is a valid mapping. However, in map2, since there exists a violation between $(u_1, u_3)$ and $(s, v_4)$, there is a factor in $R_2$ that has no prime $q$. $R_k = R_1 \times R_2$ (mod $p$).
Decryption at the client. After receiving all the encrypted messages $R_k$, the client performs two main steps:

- For each $R_k$, the client computes the plaintext of $R_k$ by $R'_k = \text{Dec}(R_k, x, g)^M$; and
- The client computes the final result by $R = R'_k \pmod{q}$.

$R$ equals zero if and only if there is at least one valid mapping from $Q$ to $G$ and thus, $\text{subIso}(Q, G) = \text{true}$.

Example 5.5.3. We show the decryption at the client by using the example in Fig. 5.5 (a). Assume $M = 2$. The encrypted message $R_k$ only aggregates two $R_i$s. The client generates the $g^{-2x}$, computes $R'_k = R_k \times g^{-2x} \pmod{p}$, and finally computes $R = R'_k \pmod{q}$. The result is zero, which indicates $Q$ is a subgraph of $G$.

Decomposition scheme. We recall that the decryption (Dec in Def. 5.3.2) uses the arithmetic modulo $p$. The message $m \ast r$ must not exceed $p$. When there are too many $R_i$s multiplied into $R_k$, the product (in the plaintext domain) may exceed $p$. Subsequently, the client will not obtain the correct plaintext under the arithmetic system. Therefore, we decompose the product into smaller numbers and the client decrypts those numbers instead. Through Formula 5.5.2 below, we can determine the maximum number of $R_i$s to be aggregated in $R_k$ ($M$):

$$\text{Len}(p) \geq M(\text{Len}(g) + \text{Len}(r))$$

$$\iff M \leq \frac{\text{Len}(p)}{(\text{Len}(q) + \text{Len}(r))},$$

where $\text{Len}(p)$ is the size of $p$.

Let’s say we set $M = 10$. From experiments, the number of mappings (after our minimizations) for our queries is around 500 on average. Each message is 2048 bits in size. Thus, the communication cost is around 12.8KB, which is very small.

False positives. Due to CGBE, the two matrix operations in SPVerify introduce negligible false positives: (1) additions with computing $R_i$ (Lines 4-7); and (2) multiplications with computing $R_k$ in each decomposed number (Line 8). However, the probabilities of the above two false positives are negligible.
Algorithm 11 SPVerify(map, Q_k, CM_s, R_k)

1: Initialize $R_i = 0$
2: for each $u_i, u_j \in V(Q), i < j$
3:     $v_i' = \text{map}[u_i], v_j' = \text{map}[u_j]$
   /* Additions */
4: if $(v_i', v_j') \notin E(CM_s)$
5:     $R_i += M_Q_k(u_i, u_j) \pmod{p}$ /* Aggregate violation */
6: else
7:     $R_i += I \pmod{p}$ /* No violation, $I = \text{Enc}(q)$ */
   /* Multiplications */
8: $R_k \times R_i \pmod{p}$ /* Decompose $R_k$ after aggregating $M R_i$ */

The probability of false positives from the aggregation (additions) while computing $R_i$ and the multiplication of $R_k$s in each decomposed number are respectively stated in Props 5.5.1 and 5.5.2, which can be established by simple arithmetics.

**Proposition 5.5.1.** The probability of false positives in $R_i$ is $\frac{1}{q}$, which is negligible.

**Proof.** The case of false positives is that each individual values in the addition are not divisible by $q$ but the sum equals a multiple of $q$. Such a probability is

$$
\Pr(\text{false positives in } R_i) = \Pr(r_1 + \cdots + r_{m(m-1)/2} = 0 \pmod{q})
= \frac{1}{q}
$$

(5.5.3)

where $m = V(Q)$, and $q$ is a large prime number, e.g., 32bits. Thus, the probability is negligible, in practice. \qed

**Proposition 5.5.2.** The probability of false positives in $R_k$ is $1 - e^{-\frac{M}{q}}$, which is negligible, in each decomposed number.

**Proof.** The probability of false positives in each $R_k$ is

$$
\Pr(\text{false positives in } R_k) = 1 - \Pr(\text{true positive in all } R_i)
= 1 - (1 - \frac{1}{q})^M
\approx 1 - e^{-\frac{M}{q}}
$$

(5.5.4)

where $M$ is the size of the decomposed number. Since $M \ll q$, the probability is negligible in practice. \qed
Algorithm 12 SPVerify\(^*(\text{map, } Q_k, CM_s, R_k)\)

1: Initialize \(R_i = 1\)
2: for each \(u_i, u_j \in V(Q), i < j\)
3: \(v_i' = \text{map}[u_i], v_j' = \text{map}[u_j]\) /* Multiplications */
4: if \((v_i', v_j') \not\in E(CM_s)\)
5: \(R_i \times = M_{Q_k}(u_i, u_j) \pmod{p}\) /* Aggregate violation */
6: else
7: \(R_i \times = I \pmod{p}\) /* No violation, \(I = \text{Enc}(1)\)*/
/* Additions */
8: \(R_k += R_i \pmod{p}\)

5.5.2 Optimized SPVerify for Queries of Bounded Sizes

Each encrypted message \(R_k\) sent by SPVerify aggregates at most \(M\) messages \(R_i\)s. In this subsection, we propose SPVerify\(^*\), which significantly reduces the number of messages returned, which in turn reduces both the communication and computational costs at the client. The main idea behind SPVerify\(^*\) is to use multiplications to detect violations since queries are often small and use additions to aggregate \(R_i\)s. Hence, the value of \(R_k\) may not exceed \(p\) even after many aggregations. However, a tradeoff of SPVerify\(^*\) is that the query size must be bounded by a pre-determined constant \(c\).

Similar to SPVerify, SPVerify\(^*\) also detects the violation by multiplications and additions. In order to achieve that, we first define a complement encoding of the query (see Def. 5.5.3).

**Definition 5.5.3.** The encoding of the entries of \(M_Q\) are:
\[\forall u_i, u_j \in V(Q),\]
\[
\begin{cases} 
    M_Q(u_i, u_j) = 1 & \text{if } M_Q(u_i, u_j) = 0 \\
    M_Q(u_i, u_j) = q & \text{otherwise}
\end{cases}
\]
where \(q\) is a large prime number.

In relation to Def. 5.5.3, we adopt Formula 5.5.1 to state the violation: \(\forall u_i, u_j \in V(Q),\)
\[
M_Q(u_i, u_j) = q \land (v_i', v_j') \not\in E(G)
\] (5.5.5)
where \(v_i', v_j' \in V(G)\), \(u_i \mapsto v_i'\) and \(u_j \mapsto v_j'\).
Algorithm for SPVerify*. For ease of comparison, we present the pseudo-code of SPVerify* (shown in Algo. 12) in the style of SPVerify. The inputs and the initialized data structures are the same as SPVerify, except that $R_k$ must be initialized to 0. We first initialize an intermediate result $R_i$ with a value 1 (Line 1). For each pair of vertex $(u_i, u_j)$ in $V(Q)$ and the mapped vertex pair $(v'_i, v'_j)$ in $CM_s$ (Lines 2-3), the following two steps are performed:

1. Multiplications (Lines 4-7): according to the violation (by Formula. 5.5.5), if $(v'_i, v'_j) \notin E(CM_s)$, set $R_i$ as the value $M_{Q_k}(u_i, u_j) \times R_i \pmod{p}$. This indicates that as soon as $(u_i, u_j)$ is an edge in $Q$, $R_i$ must contain the factor $q$, and the decryption value is zero (i.e., the current mapping map contains a violation). Otherwise, $R_i$ is set to a value $I \times R_i \pmod{p}$, where $I$ is an encrypted value without factor $q$ issued by the client, $I = \text{Enc}(1)$; and

2. Additions (Line 8): it aggregates $R_i$ to $R_k$, where $R_k = R_k + R_i \pmod{p}$. If there is at least one valid mapping from $Q$ to $G$ (i.e., at least one $R_i$ whose plain text is non-zero). The decrypted value of $R_k$ must also be non zero. Otherwise, it is zero.

The complexity is the same to that of SPVerify, i.e. $O(|V(Q)|^2)$.

Example 5.5.4. Fig. 5.5 (b) illustrates an example of SPVerify*. Similarly, since there is no violation in map1, all the factors in $R_1$ do not contain $q$. Regarding map2, since there is a violation, $R_2$ contains a factor $q$. $R_k = R_1 + R_2 \pmod{p}$.

Decryption at the client. The decryption is modified as:

- The client computes the message encoded in $R_k$ as $R'_k = \text{Dec}(R_k, x, g)^{m(m-1)/2}$, where $m = |V(Q)|$; and

- The client computes the final result by $R = R'_k \pmod{q}$.

$R$ equals non-zero if and only if there is at least one valid mapping from $Q$ to $G$. Thus $\text{subIso}(Q, G) = \text{true}$. 
**Example 5.5.5.** We show the decryption in Fig. 5.5 (b). For simplicity, we assume that \( R \) only aggregates \( R_1 \) and \( R_2 \). The client generates \( g^{-6x} \), computes \( R'_k = R_k \times g^{-6x} \pmod{p} \), and finally computes \( R = R'_k \pmod{q} \). The result is non-zero which indicates that \( Q \) is a subgraph of \( G \).

**Determining the constant** \( c \) **to decide when to use** \( \text{SPVerify} \) **or** \( \text{SPVerify}^* \). In \( \text{SPVerify}^* \), multiplications are used to aggregate violations by edges in \( CM_s \) (Line 4 in Algo. 12), instead of aggregating numerous mapping results (\( R_i \) in Line 8 of Algo. 11). Similarly, when \( R_i \) (Lines 4-7) in Algo. 12 exceeds \( p \), the client cannot recover the plaintext. The number of multiplications for each \( R_i \) is directly related to the size of the query (\(|V(Q)|\)). We can determine the maximum size of the query, denoted as \( c \), using the following inequality.

\[
\text{Len}(p) \geq \frac{c(c-1)}{2}(\text{Len}(q) + \text{Len}(r)) \\
\iff 0 \geq c^2 - c - \frac{2\text{Len}(p)}{\text{Len}(q) + \text{Len}(r)} \tag{5.5.6}
\]

Putting these together, in Lines 7-8 of Algo. 10, once \(|V(Q)| \leq c\), the \( SP \) uses \( \text{SPVerify}^* \). Otherwise, it uses \( \text{SPVerify} \).

**False positives.** Since both \( \text{SPVerify} \) and \( \text{SPVerify}^* \) use \( \text{CGBE} \), we can obtain that the probabilities of false positives of \( \text{SPVerify}^* \) are also negligible. Their proofs are almost identical to those of Props. 5.5.1 and 5.5.2, and hence, omitted.

### 5.6 Privacy Analysis

In this section, we prove the privacy of the encryption method and \( \text{SPsubIso} \). The attack model is the one defined in Sec. 5.1. The attackers or \( \text{SPs} \) are eavesdroppers and can adopt chosen plaintext attack (CPA) [58].

**Privacy of the encryption method.** \( \text{CGBE} \) is adopted to encrypt the query graph in this work, thus we have the following proposition.

**Proposition 5.6.1.** The structure of the query is preserved from the \( SP \) against the attack model under \( \text{CGBE} \).
Proof. (Sketch) The proof can be derived from the security of CGBE. After receiving $Q_k$, the $SP$ cannot break the $M_{Q_k}$ since they are secure against CPA. $V$, $\Sigma$ and $L$ do not contain structural information. Thus, the structure of query is preserved from the $SP$ against the attack model. \hfill$\Box$

Privacy of $SP_{subIso}$. $SP_{subIso}$ mainly consists of five steps: (1) GenCandSubGraph; (2) MinCandSubGraph; (3) GenCandMatch; (4) GenAllMap; and (5) SPVerify (or SPVerify$^*$). We now analyze the privacy of each step as follows. However, first, the analysis requires some notations. We denote a function $P(m, h, \Sigma)$ that returns all possible graphs of $m$ vertices with a minimum height $h$ and the labels $\Sigma$. $|P(m, h, \Sigma)|$ is exponential to the value $m$ and the size of $\Sigma$. Let $A(Q)$ is a function that returns 1 if $SP$ is able to determine the exact structure of $Q$, and 0 otherwise. The probability that the $SP$ can determine the structure of the query $Q$ is denoted as $Pr[A(Q) = 1]$. Given a query $Q$ and $(m, h, \Sigma)$, the probability of determining its structure is $Pr[A(Q) = 1] = 1/|P(m, h, \Sigma)|$.

Proposition 5.6.2. Under GenCandSubGraph, MinCandSubGraph, GenCandMatch and GenAllMap, $Pr[A(Q) = 1] = 1/|P(m, h, \Sigma)|$.

Proof. (Sketch) The proof is established by one main fact: $SP$ does not utilize any structural information of the query, except the value $h$ in the algorithm.

- GenCandSubGraph utilizes $\ell_s$, $h$, $Q_k$ and $G$ to generate all the $CS_s$s;
- MinCandSubGraph minimizes the size of each $CS_s$ by using only the structure of $CS_s$ itself;
- GenCandMatch utilizes $Q_k$ and $CS_s$ to generate $CM_s$s;
- GenAllMap enumerates all the possible mappings maps between $Q_k$ and $CM_s$.

The $SP$ cannot learn the structure of $Q$ by invoking them, and thus the probability of determining a structure remains $Pr[A(Q) = 1] = 1/|P(m, h, \Sigma)|$. \hfill$\Box$

$^1$We remark that if $h = 1$, the $SP$ is able to infer that the vertex with $\ell_s$ must connect to other vertices in $Q$. To avoid this special case, as mentioned in Sec. 5.3, we choose the starting vertex where $h$ equals or larger than 2.
In SPVerify and SPVerify*, SP sends messages to the clients. The clients may terminate the algorithm when a mapping is found, which may leak information to the SP. Such a leak can be quantified in the following proposition.

**Proposition 5.6.3.** Under SPVerify or SPVerify*, the following hold for:

- If \( Q \) is a subgraph of \( G \), \( \Pr[A(Q) = 1] = 1/|S| \), where \( S = \{ G | G \in P(m, h, \Sigma), G \subseteq CM_s, where \ CM_s \in Cache \} \); and

- If \( Q \) is not a subgraph of \( G \), \( \Pr[A(Q) = 1] = 1/|P(m, h, \Sigma)| \).

**Proof.** (Sketch) Since the algorithm SPVerify* is similar to that of SPVerify, due to the space constraint, we prove it with SPVerify only. The proof involves two aspects:

1. **SP can never determine any structural information from the mathematical computations in each steps of SPVerify:**

   Recall that SPVerify comprises a fixed number of mathematical operations in the encrypted domain in Algo. 11.

   - Lines 4-7 invoke a constant number of additions of \( M_{Q_k} \) and \( R_i \), and only structure of \( CM_s \) is considered. More specifically, \( \forall i, j, m^2 \) additions are invoked for \( M_{Q_k}(i, j) \) and \( R_i \); and

   - Line 8 requires one multiplication on each \( R_i \) and \( R_k \).

   Based on Prop. 5.6.1, all the intermediate computations results are securely protected against the attack model. Moreover, each step of SPVerify has a constant number of operations in the encrypted domain. SP cannot learn any information from them.

2. **SP may only infer some structural information from the message communications:**

   Recall that once \( M \ R_i \)s are aggregated into \( R_k \), \( R_k \) is returned to the client, the client may decide to terminate SPVerify after receiving \( R_k \)s. There are two cases:

   - Suppose there is at least one valid \( R_k \) such that \( Q \) is a subgraph of \( G \). In this case, \( Q \) must be graph (or subgraph) isomorphic to one of \( CM_s \) in Cache. Therefore, \( \Pr[A(Q) = 1] = 1/|S| \), where \( S = \{ G | G \in P(m, h, \Sigma), G \subseteq CM_s, CM_s \in Cache \} \); and
• If the client does not terminate the algorithm, $\mathcal{SP}$ does not know if there is a valid $R_k$ or not. Thus, the probability of determining the structure of $Q$ is still $Pr[A(Q) = 1] = 1/|P(m, h, \Sigma)|$.

Based on Prop. 5.6.3, we note that the client can make a tradeoff between privacy and response times by terminating the algorithm as late as acceptable.

5.7 Experimental Evaluation

In this section, we present a detailed experimental evaluation of our proposed techniques with popular real world datasets. The results show that our techniques are efficient and our optimizations are effective.

5.7.1 Experimental Setup

The platform. We conducted all our experiments on a machine with an Intel Core i7 3.4GHz CPU and 16GB memory running Windows 7 OS. All techniques were implemented on C++, and CGBE was implemented on the GMP library. We simulate the bandwidth as 10Mbits/s.

| Graph $G$     | $|V(G)|$   | $|E(G)|$   | Avg. Degree | $|\Sigma(G)|$ |
|---------------|------------|------------|-------------|--------------|
| DBLP          | 317,080    | 1,049,866  | 6.62        | 199          |
| Amazon        | 334,863    | 925,872    | 5.52        | 153          |
| Youtube       | 1,134,890  | 2,987,624  | 5.26        | 978          |
| LiveJournal   | 3,997,962  | 34,681,189 | 17.34       | 1355         |

Data and query sets. We benchmarked real-world datasets: DBLP, Amazon, Youtube, and LiveJournal [9]. Since the vertices do not have labels, we adopt the approach that

\(^2\)As discussed in Sec. 1, previous studies are not applicable to our problem, since they heavily exploit query structures, which are secret in this work.
uses the degree of the vertex as its label [51]. Some statistics of the datasets are shown in Table 5.1.

For each dataset, we generated two types of queries [96]: (1) BFS queries (BFS) and (2) DFS queries (DFS) by random BFS and DFS methods, respectively. Both BFS and DFS contain query sets $Q_2$-$Q_8$, wherein each $Q_n$ contains 1,000 query graphs, and $n$ is the number of vertices of each query of the query set. $h$ of the query sets are around 3-4 on average.

**Default values of the parameters.** In CGBE, the prime $p$ and $q$ are 2048 bits and 32 bits, respectively. The random number $r$ is 32 bits. The largest value $c$ is 12 by Formula 5.5.6. However, to study the performance of both SPVerify* and SPVerify, we first set $c$ to 6, by default. That is, if $|V(Q)| \leq 6$, we used SPVerify*. Otherwise, we used SPVerify. We finally investigated the effectiveness of SPVerify* with $c = 11$. For SPVerify*, we set $M = 100$ by default (*i.e.*, we aggregated 100 $R_i$s into each $R_k$). For SPVerify, we set $M = 10$ only. Unless specified otherwise, $\delta = 5$. Under these settings, no false positives was detected from the entire experiments.

![Figure 5.6: Average preprocessing time at the client.](image)

Figure 5.6: Average preprocessing time at the client.
5.7.2 Performance at the Client Side

Preprocessing time at the client. We report the average preprocessing time of the query \( Q \) at the client side on all datasets in Fig. 5.6. Specifically, the preprocessing of \( Q \) includes (1) the computation for \( \ell_s \) and \( h \); and (2) the encryption of \( Q \) by CGBE. We observe that the average times for each query on all datasets are around 4ms, which shows that the preprocessing is in cognitively negligible.

![Figure 5.7: Average received encrypted message size at the client.](image)

The sizes of messages received by the client. We report the sizes of the encrypted messages \( R_k \)'s that the client received in Fig. 5.7. Due to the optimizations by SPsubIso, the largest sizes of \( R_k \)'s (at \( Q6 \)) are around 13KB on LiveJournal, which can be efficiently transmitted via today’s networks. For \( Q7-Q8 \), as we set \( c \) to 6 (by default), SPsubIso uses SPVerify. The number of \( R_i \)'s aggregated in each \( R_k \) is 10. Thus, the message sizes for \( Q7-Q8 \) are larger. Since the maximum value of \( c \) is 11 in the current configuration, SPVerify* can be used to produce much smaller messages (to be discussed with Fig. 5.14).

The decryption time at the client. After receiving the encrypted messages \( R_k \)'s, the client decrypts \( R_k \)'s. The decryption time is shown in Fig. 5.8. Since the sizes of \( R_k \)'s are
small and the decryption method is simple, the average decryption times at the client are correspondingly fast at most 16ms.

5.7.3 Performance at the SP Side

The total runtime at the SP. Fig. 5.9 shows the average total runtime at the SP on all datasets, which is exactly the runtime of SPsubIso. For the simplicity of performance analysis, we terminated SPsubIso once the client found at least one valid mapping. (The client may postpone the termination to achieve higher privacy, although that introduces small but non-trivial overhead to SPsubIso.) It is not surprising that the runtimes increase exponentially with the query sizes. For Q8, the largest runtime is around 12s on LiveJournal. However, the running times for small queries (Q2-Q6) are well below 600ms for all datasets.

We further report the breakdowns of the total runtimes of SPsubIso: (1) GenCandSubGraph and MinCandSubGraph; and (2) GenCandMatch and SPVerify. For the DBLP and Amazon datasets, the breakdown percentages of both query sets are similar: 30% and 70%. For Youtube, they are 81% and 19%, and for LiveJournal, they are 53% and 47%.
The effectiveness of minimization of $CS_s$. In Fig. 5.10, we show the average percentage of the reduced vertices of $CS_s$ by NC in MinCandSubGraph. We observe that MinCandSubGraph reduces around 40% of the vertices of $CS_s$ on DBLP and Amazon, and at least 60% on Youtube. However, for LiveJournal, the percentage (on average) is
around 20%.

In our experiment, we note that a small fraction of queries have \( CS_s \)s that contain numerous candidate mappings. The reason is that \( SP_{subIso} \) cannot exploit query structures for optimizations. In this case, for each \( CS_s \), we compute an upper bound of the number of candidate mappings of a query by simple calculations on \( CS_s \). For those candidate subgraphs that may exceed 100,000 mappings, we transmit the candidate subgraphs to the client to do subIso (e.g., using [39] or [100]). The percentage of such queries is very small, at most 1% for \( Q2-Q7 \) on all datasets. For \( Q8 \), the percentage is only 10%. In other words, most subgraph queries are successfully outsourced to the \( SP \).

![Graphs showing pruned redundant CMs](image)

**Figure 5.11:** Average % of the pruned redundant \( CM_s \) by \( Cache \).

The effectiveness of the elimination of redundant \( CM_s \). Fig. 5.11 shows the average percentage of redundant \( CM_s \)s pruned by \( Cache \) in GenCandMatch. We note that as the query size increases, the effectiveness of \( Cache \) increases. For \( Q2-Q4 \) of all datasets, the percentage of the elimination of redundant \( CM_s \) increases from 0% in \( Q2 \) to 80% in \( Q4 \). For \( Q5-Q8 \) on DBLP, Amazon and Youtube, the percentages are stable at around 80%. We note the graph structures of LiveJournal are diverse and there are many distinct \( CM_s \)s. The effectiveness of \( Cache \) then decreases from 80% to 50% for \( Q5-Q8 \). This is
also reflected by the fact that the sizes of the encrypted messages $R_k$s are the largest for LiveJournal (see Fig. 5.7).

![Graphs](a) DBLP  
(b) Amazon  
(c) Youtube  
(d) LiveJournal

**Figure 5.12:** Average Cache size at $SP$.

**The memory consumption of Cache.** We report the memory consumption of Cache in Fig. 5.12. As we only store the hash code of the canonical labeling of each distinct $CM_s$, the memory consumption is very small (at most 25MB).

**The effectiveness of pruning redundant mappings by NEC.** We report the pruning of redundant mappings by using NEC in Fig. 5.13. We observe that, for most of the queries, we pruned approximately 20% of redundant mappings on average. This further saves on computations in SPVerify and SPVerify$^*$. 

**The number of aggregated messages by SPVerify$^*$.** In Fig. 5.7, since $c$ was set to 6 by default, we used SPVerify for $Q7$-$Q8$, where each $R_k$ is an aggregate of $M$ messages and $M = 10$. As discussed, the messages are small. To study SPVerify$^*$, we then set $c = 11$. We used $Q8$ with DFS and varied the values of $M$ from 10 to 100. Fig. 5.14 shows the detailed performance of all datasets. We report that for $M = 10$, the message size is the same as those values of $Q8$ DFS in Fig. 5.7. Importantly, as $M$ increases, the message size decreases accordingly.
Figure 5.13: Average % of the pruned redundant mappings by NEC.

Figure 5.14: Average size of messages $R_k$s when $c = 11$. 
Chapter 6

Conclusions and Future Work

6.1 Summary and Contributions

In this thesis, we have proposed the secure subgraph query services over two different types of graph databases. I.e. we have proposed the authenticated and structure-preserving subgraph query services.

In Chapter 3 of the thesis, we investigate the authentication of subgraph query services of outsourced graph databases. We propose an index IFTree that minimizes the I/O cost of the popular filtering-and-verification framework for subgraph query processing. We then propose MIFTree by extending IFTree to authenticate subgraph query. To optimize the \( \mathcal{VO} \) derived from MIFTree, we propose a compact \( \mathcal{VO} \) representation and a clustering of graphs having similar subset of features. We conduct a detailed experiment to evaluate the performance of our proposed techniques and the effectiveness of the enhancements.

In Chapter 4 of the thesis, we present the first work on query services for structure-preserving subgraph isomorphism (SPsubIso). SPsubIso comprises three major steps: (1) Structure-preserving matching (SPMatch) involves a novel cyclic group based encryption (CGBE) scheme to compute whether a mapping between \( Q \) and \( G \) is valid, in an encrypted domain. (2) Structure-preserving enumeration (SPEnum) comprises a protocol that involves the client for further pruning. (3) Structure-preserving refinement (SPRefine) exploits a static index for pruning the search space of possible mappings. Our analysis
shows that the structural information is preserved under SPMatch and presents the privacy preservation due to optimizations. Our experiments on both real and synthetic datasets confirm that SPsubIso is efficient.

In Chapter 5 of the thesis, we study the asymmetric structure-preserving subgraph query processing. Our techniques include deriving minimized candidate subgraphs to significantly reduce the number of candidate mappings, generating candidate matchings and then candidate mappings without redundancies and verifying candidate mappings without leaking query structures. Our extensive experiments confirm that our techniques are efficient and effective.

6.2 Future work

There are many interesting and open problems beyond the issues addressed in this thesis.

First, since graph size increases rapidly nowadays, we provide parallel algorithms in distributed environment, such that the performance can be further optimized. Numbers of researches have already studied this problem in the literature [36, 59, 88, 96], however, none of them analyze whether their proposed techniques are parallel scalable or not. We define that an algorithm is parallel scalable if the more processors used, the less time the algorithm takes. The first idea is to propose algorithm on MapReduce model [27]. Given a subgraph query, one can construct all those candidate subgraphs using a driver of the MapReduce model. Each of them are then sent to a Mapper. Each Mapper performs the subgraph isomorphism algorithm, authentication or privacy can be reconsidered on top of this. However, we note that the performance of such algorithm may be hampered by both the I/O bound property and the synchronization policy of the MapReduce model. In order to reduce the costs, we can provide another algorithm on vertex-centric asynchronous model [70, 71], where vertex program is executed on each vertex in parallel, and interacts with the neighbors of the vertex via asynchronous message passing.

Second, in the experiments of our previous works, we find out that the variation of the running time for subgraph isomorphism algorithm is often very large, even the queries are very similar. Some subgraph queries may finish in several seconds, however, some
may last for minutes or hours. The case may be even worse on large graphs. This significantly affects the usability of the query services. To improve the services, we plan to propose an estimator to predict the selectivity or the running time of the queries, and show the progress indicator [67, 115] to the users who issue the queries. Some researches have studied this problem on SQL [17], MapReduce Pipeline [74], Twig queries [85] or SPARQL queries [41, 50], but not subgraph queries. Our basic idea is to adopt machine learning techniques. In particular, we first cluster the training queries and perform feature learning on all those clusters. Based on those features, we then adopt the regression models or neural network to fit the queries. Finally, we use the well-trained model to predict the progress of subgraph query.

Third, the property graph model (e.g., RDF graph [87] or Knowledge graph [3]) is used as a de-facto standard for modeling knowledge nowadays. Therefore, it is worth investigating to extend our techniques in Chapter 5 to support property graph model. The major technical challenge for this extension is as follows. There are various state-of-the-art query language for property graph querying, e.g., SPARQL [87]. Unlike subgraph query (Chapter 5) with only label equality and topological constraint (i.e. subgraph isomorphism), their semantics are more powerful and complex (e.g., label inequality or similarity). Thus, it is challenging to protect the semantics of those queries by using our existing methods, and we need to propose novel encryption schemes and algorithms to address them.

Forth, it is of practical interests to extend our techniques in Chapter 3 for large graphs. We can adopt the graph feature called graphlet in [52] for large graphs. In particular, we can derive similar partially overlapping features POFs from those graphlets on large graphs, and build an indexing structure by adopting MHT techniques for authenticated subgraph query processing. Some specific optimization strategies may be proposed to enhance the performance for large graphs.
Bibliography


Curriculum Vitae

Academic qualifications of the thesis author, Mr. FAN Zhe:

- Received the degree of Bachelor of Engineering from South China University of Technology, July 2011.

August 2015