Speeding up Subgraph Isomorphism Search in Large Graphs

By

Xuguang Ren
B.Eng., Jinan University, Guangzhou, China

School of Information and Communication Technology (ICT)
Griffith Sciences
Griffith University, Australia

Submitted in fulfilment of the requirements of the degree of

Doctor of Philosophy

April, 2018
Abstract

Graph is a widely used model to represent complicated data in many domains. Finding subgraph isomorphism is a fundamental function for many graph databases and data mining applications handling graph data. This thesis studies this classic problem by considering a set of novel techniques from three different aspects.

This thesis first considers speeding up subgraph isomorphism search by exploiting relationships among data vertices. Most of the subgraph isomorphism algorithms of the In-Memory model (IM) are based on a backtracking method which computes the solutions by incrementally enumerating all candidate combinations. We observed that all current algorithms blindly verify each individual mapping separately, often leading to extensive duplicate calculations. We propose two novel concepts, Syntactic Equivalence and Query Dependent Equivalence, by using which we group specific candidate data vertices into a hypervertex. The data vertices belonging to the same hypervertex can be mapped to the same query vertex. Thus, all the vertices falling into the same hypervertex can be determined whether to contribute to a solution simultaneously instead of calculating them separately. Our extensive experimental study on real datasets shows that existing subgraph isomorphism algorithms can be significantly boosted by our approach.

Secondly, this thesis considers multi-query optimization where multiple queries are processed together so as to reduce the overall processing time. We propose a novel method for efficiently detecting useful common subgraphs and a data structure to organize them. We propose a heuristic algorithm based on the data structure to compute a query execution order so that cached intermediate results can be effectively utilized. To balance memory usage and the time for cached results retrieval, we present a novel structure for caching the intermediate results. We provide strategies to revise existing single-query subgraph isomorphism algorithms to seamlessly utilize the cached results, which leads to significant performance improvement. Experiments over real datasets proved the effectiveness and efficiency of our multi-query optimization approach.

In the third part, this thesis considers the subgraph isomorphism search under distributed environments. We observed that current state-of-the-art distributed solutions
either rely on crippling joins or cumbersome indices, which leads those solutions hard to be practically used. Moreover, most of them follow the synchronous model whose performance is often bottlenecked by the machine with the worst performance in the cluster. Motivated by this, in this thesis, we utilize a dramatically different approach and propose PADS, a Practical Asynchronous Distributed Subgraph enumeration system. We conducted extensive experiments to evaluate the performance of Pads. Compared with existing join-oriented solution, our system not only shows significant superiority in terms of query processing efficiency but also has outstanding practicality. Even compared with heavy indexed solution, our approach also has better performance in many cases.
Declaration

This work has not previously been submitted for a degree or diploma in any university. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made in the thesis itself.

Signed: _______________________
Xuguang Ren
April, 2018
Acknowledgements

This thesis is not only a summary of the work during the period of chasing my Ph.D., but also a milestone of the journey in experiencing the meaning of my life.

First and foremost, I express my sincere gratitude to my supervisor Assoc. Prof. Junhu Wang who dug me up from a universe of chaos and led me the way to higher knowledge. I appreciate his patience in listening to my ideas. Without his excellent advice, I cannot complete all my publications and this thesis.

My immeasurably thankfulness goes to my partner Lei Lei who solved a harder problem of mine compared with subgraph isomorphism, being single. Together we proved the theoretical prerequisites of love: patience, understanding, support ...

From the bottom of my heart, I thank Stephen Feng and Nigel Franciscus for so much fun time we had together. We designed many methods to balance study and entertainment.

My special thanks go to Professor Wook-Shin Han and the team members in the database lab of POSTECH, from whom I learned a lot.

Last but not the least, my deepest gratitude goes to my dear family: my parents and my brother.
List of Publications


Contents

1 Introduction
   1.1 Graph Model ........................................... 1
   1.2 Preliminaries ........................................ 3
      1.2.1 Data Structures .................................. 3
      1.2.2 Pattern Matching ................................. 4
   1.3 Proposed Approaches ................................... 5
      1.3.1 Exploiting Vertex Relationships ................. 6
      1.3.2 Multi-Query Optimization ....................... 7
      1.3.3 Distributed Processing .......................... 7
   1.4 Thesis Outline ....................................... 9

2 Exploiting Vertex Relationships ........................................... 11
   2.1 Motivation Problems ................................ 11
   2.2 Related Work ....................................... 13
   2.3 Relationships Between Data Vertices ............... 16
      2.3.1 Syntactic Containment ......................... 16
      2.3.2 Syntactic Equivalence ......................... 17
      2.3.3 Query-Dependent Containment ............... 18
      2.3.4 Query-Dependent Equivalence ............... 19
   2.4 Graph Compression .................................. 19
      2.4.1 Compressed Graph for Subgraph Isomorphism .... 20
      2.4.2 Building Compressed Graph ................... 22
2.5 BoostIso ................................................................. 24
  2.5.1 Utilizing SC and SE ........................................ 25
  2.5.2 The algorithm for hyperembedding search ........... 31
  2.5.3 Utilizing QDC and QDE ................................. 34
    2.5.3.1 Building DRT ...................................... 34
    2.5.3.2 Integrating DRT into Hyperembedding Search ... 35
  2.6 Experiments ....................................................... 36
    2.6.1 Experimental Setup .................................... 36
    2.6.2 $G_{sh}$ Statistics on Real Datasets .................. 37
    2.6.3 Building Time and Scalability ....................... 38
    2.6.4 Efficiency of Query Processing ...................... 39

2.7 Conclusion ......................................................... 44

3 Multi-Query Optimization ........................................... 45
  3.1 Motivation and Challenges ................................... 45
  3.2 Related work .................................................... 46
  3.3 Overview of Our Approach .................................... 48
  3.4 Detecting Common Subgraphs .................................. 49
    3.4.1 Grouping Factor ....................................... 50
    3.4.2 Pattern Containment Map ............................... 51
  3.5 Query Execution Order ......................................... 54
  3.6 Caching Results ................................................ 55
  3.7 Subgraph Isomorphism Search ............................... 61
  3.8 Experiments ....................................................... 65
    3.8.1 Grouping Factor Effectiveness ....................... 66
    3.8.2 PCM Building Time .................................... 67
    3.8.3 Query Execution Order ................................. 68
    3.8.4 Intermediate Result Caching ......................... 69
    3.8.5 Query Processing Time ................................. 70
  3.9 Conclusion ....................................................... 72

4 Distributed Subgraph Enumeration: A Practical Asynchronous System 73
  4.1 Motivations and Problems .................................... 73
  4.2 Related Work .................................................... 75
    4.2.1 Multi-round Join-oriented .............................. 75
    4.2.2 Exchanging Data Vertices .............................. 76
Contents

4.2.3 Compression ................................................................. 77
4.3 PADS ................................................................. 77
    4.3.1 Architecture Overview ................................................. 77
    4.3.2 R-Meef .......................................................... 79
    4.3.3 Algorithm of SubEnum ............................................... 83
4.4 Computing Execution Plan ................................................ 86
    4.4.1 Minimising Number of Rounds ...................................... 86
    4.4.2 Moving Forward Verification Edges ............................... 87
4.5 Inverted Trie ............................................................. 89
4.6 Memory Control Strategies ................................................. 94
    4.6.1 Estimating Cached Data ............................................... 95
    4.6.2 Grouping Regions .................................................. 96
4.7 Experiment ............................................................... 98
    4.7.1 Intermediate Result Storage Latency ............................ 98
    4.7.2 Performance Comparisons ......................................... 99
4.8 Conclusion ............................................................... 102

5 Conclusion and Future Work ................................................. 103
    5.1 Conclusion .......................................................... 103
    5.2 Future Work ........................................................ 104

References ......................................................... 105
List of Figures

1.1 Subgraph Isomorphism Example ........................................ 2
2.1 Illustration of duplicate computation ................................. 12
2.2 A Running Example of BoostIso approach .......................... 18
2.3 Example SC-Graph ...................................................... 30
2.4 (a) and (b) Example graphs where nodes $A$ and $A_i$ are all labeled $A$, and so on; (c) Embeddings stored as a trie ............................. 32
2.5 Relationship ratios with varying number of labels .................. 39
2.6 BoostIso Experiment results over Human ............................ 40
2.7 BoostIso Experiment results over Yeast ............................. 42
2.8 Experiment results over Wordnet ...................................... 43
3.1 Building Pattern Containment Map .................................... 50
3.2 Illustration of multiple-query execution order ....................... 54
3.3 Trivial Structures for Caching Results ............................... 57
3.4 Illustration of Graph Partitions ....................................... 58
3.5 Illustration of Cached Results ....................................... 59
3.6 Illustration of BTLP Building Steps .................................. 61
3.7 Illustration of Query Covers .......................................... 62
3.8 An Illustration Graph $G$ for revised subgraph isomorphism search .... 62
3.9 PCM Building Time on Real Datasets ................................. 68
3.10 Effects of Grouping Factor Threshold ................................ 68
3.11 Performance Comparison and Scalability Test ...................... 71
3.12 Effects of query similarity and GF threshold ....................... 71
4.1 PADS Architecture .................................................... 78
4.2 Running Example ..................................................... 82
4.3 R-Meef workflow ..................................................... 82
4.4 Example of Inverted Trie ........................................ 91
4.5 Query Sets .................................................... 99
4.6 Query Performance over Roadnet dataset ................. 100
4.7 Query Performance over Human Dataset ................. 101
4.8 Query Performance over DBLP dataset ..................... 101
4.9 Query Performance over LiveJournal dataset .......... 101
List of Tables

2.1 DRT for \((G_{q'}, u_1)\) and \(G_{sh}\) in Figure 2.2 . . . . . . . . . . . . . . . . . . 34
2.2 Profiles of datasets used for BoostIso experiments . . . . . . . . . . . . . . . . . . 37
2.3 Statistics of \(G_{sh}\) on real datasets . . . . . . . . . . . . . . . . . . . . . . . . . 38
2.4 The Time for Building \(G_{sh}\) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

3.1 Profiles of datasets for Multi-query optimization Experiment . . . . . . . . . . . 65
3.2 Grouping Factor Effectiveness . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
3.3 Number of Cached Queries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
3.4 Cache Memory Use in KB . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
3.5 Embedding Retrieval Time (ms) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70

4.1 Illustration of the Index Size . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
4.2 Profiles of datasets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
4.3 Triangle-Listing Time Cost (ms) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
Chapter 1

Introduction

1.1 Graph Model

There is no doubt that we are living in an increasingly connected and blended world. The Big Data movement has resulted in more data being collected at higher rates, and the data is related, dynamic, and constantly evolving. Traditionally, the data has been stored in, accessed from and processed by relational databases where fairly fixed schemas have to be designed. However relational database is not suitable for today’s unstructured and complex data. Moreover, the data in the relational database is not easily traversed as the relationship among data is not "first-class citizens" within the database itself. The relational database bears significant performance issues when processing queries involving extensive traversals. The need to efficiently store and process highly connected data has resulted in huge interest in the graph databases whose data model is graph. In graph data model, relationships (edges) have as much value as the data (vertices) itself. Applying graph theory to those organized graph data allows us to efficiently discover patterns that are not easy to see. Graph data model supports schema-less storage which enables graph databases great flexibility in handling unstructured and complex data.

However, due to the fact that graph-based algorithms always suffer a high computational cost, graph has been practically left unused for a long period of time despite its novel properties. With the improvement of the computational power of new generation computers, graph is being widely adopted practically to model various application data. Nowadays, graph data has already been used in numerous domains such as social network, bioinformatics, biochemical and image analysis. Efficient and effective graph data processing becomes an urgent requirement for graph data applications. A fundamental processing requirement for graph applications is graph pattern matching. For example,
in the biochemical area, it is common to search all the similar compounds in the chemical database by a given chemical structure [9]; in social networks, friends searching based on graph pattern matching is highly desired since it presents the users with a natural and easy way of searching [16]. Because of considerable practical requirements of graph pattern matching techniques, graph pattern matching has been a hot topic of academia for many years [7]. It has been well studied typically by several definitions based on various matching requirements as follows:

1. **Subgraph Isomorphism**: Given two graphs $G$ and $P$, subgraph isomorphism is to determine whether $G$ contains a subgraph that is isomorphic to $P$. In subgraph isomorphism, the vertices and edges are exact one-to-one mapping subject to syntactic and semantic restrictions. Subgraph isomorphism is proved to be a NP-Complete problem [52].

   ![Fig. 1.1 Subgraph Isomorphism Example](image)

   **Example 1.** Consider the query graph and data graph in Figure 1.1, there are two subgraph isomorphisms of the query graph in the data graph (marked with red).

2. **Graph Simulation**: Graph simulation [11][17][14] is a binary relation $S$ between the vertices of given graph $G$ and $P$, each pair of vertices in $S$ possess the same neighbourhood structures. Graph simulation relaxes the restrictions by permitting one-to-many and many-to-one mappings, which can be computed in quadratic time [14].

3. **Graph Homomorphism**: Graph homomorphism [49] is a mapping relation between two graphs $G$ and $P$. Graph homomorphism is an injective function and differs from subgraph isomorphism which is a bijective function. Its mapping conditions are different, in which an edge may be mapped to a path. Graph homomorphism is also a NP-Complete problem.

   Finding subgraph isomorphism is a fundamental function for many graph databases and data mining applications handling graph data. For example, it is used for network
motif computing\cite{3,37} to support the design of large network from neurobiology, ecology, and bioinformatics. It can also be used to compute the graphlet kernels for large graph comparison \cite{44,55} and property generalization for biological networks \cite{36}. In chemistry \cite{47}, it is considered as a key operation for the synthesis of target structures. The list goes on, subgraph isomorphism search can also be utilized to illustrate the evolution of social networks \cite{28} and to discover the information trend in recommendation networks \cite{35}. Besides those practical applications, the subgraph isomorphism is also a basis for some other algorithms in the literature. For instance, as a special case of subgraph isomorphism search, triangle listing is a mandatory step in cluster coefficient calculation \cite{60} and community detection \cite{59}.

Although subgraph isomorphism search is a well-known NP-complete problem, extensive and continuous efforts have been committed in speeding up the the searching process for more than 40 years. However, there are still many open issues that can be explored to further speed up subgraph isomorphism search.

In this thesis, we study this classic problem by considering a set of novel techniques. Our goal is to speed up the subgraph isomorphism search in large graphs.

1.2 Preliminaries

1.2.1 Data Structures

Although the definitions of the following graph concepts can be found in many textbooks, for the self-containment of this thesis, we explain some key graph concepts for readers to easily understand the content of this thesis hereafter.

**Graph** A graph is denoted as \( G = (V, E) \) which consists a set of vertices \( V \) and a set of edges between the vertices \( E \subseteq V \times V \). An edge is represented by \( e = (v_1, v_2) \), where \( v_1 \) and \( v_2 \) are known as nodes of \( e \).

**Vertex Labelled Graph** A vertex labelled data graph is an undirected, labelled graph denoted as \( G = (V, E, \Sigma, L) \), where

(1) \( V \) is the set of vertices.

(2) \( E \subseteq V \times V \) is a set of undirected edges.

(3) \( \Sigma \) is a set of vertex labels.

(4) \( L \) is a function that associates each vertex \( v \) in \( V \) with a label \( L(v) \in \Sigma \).
The work presented in Chapter 2 and 3 uses vertex labelled graph while the work in Chapter 4 uses unlabelled graph.

**Directed and Undirected** An undirected graph is one in which edges have no orientation such that the edge \(e = (v_1, v_2)\) is identical to the edge \(e' = (v_2, v_1)\). While directed graph is one where edges have orientation, \((v_1, v_2)\) and \((v_2, v_1)\) are two distinguished edges and may co-exist.

All the work presented in this thesis assumes undirected graph.

**Data Graph and Query Graph** The data graph is a large graph and is the pre-saved data to be queried on. Given a particular context, we sometimes use data to refer to the data graph for short. In contrast, the query graph is much smaller than the data graph. We also use query pattern or simply query, pattern the same as query graph.

**Adjacency List** An adjacency list of a graph is a data structure representing the graph with a collection of unordered lists, one for each vertex in the graph. Each list shows the set of neighbours of the corresponding vertex. The set of neighbours of a vertex \(v\), also known as adjacent vertices of the vertex, is denoted as \(Adj(v)\).

**Vertex Degree** For an undirected graph, the degree of a vertex \(v\) is the number of edges incident on it, which is \(|Adj(v)|\).

**Complete Graph** A complete graph is an undirected graph in which every pair of distinct vertices is connected by a unique edge. A complete graph with \(n\) vertices has \(n(n - 1)/2\) edges.

### 1.2.2 Pattern Matching

**Subgraph Isomorphism** Given two vertex labelled graphs \(G_1 = (V_1, E_1, \Sigma_1, L_1)\) and \(G_2 = (V_2, E_2, \Sigma_2, L_2)\), a subgraph isomorphism (a.k.a embedding) of \(G_1\) in \(G_2\) (or, from \(G_1\) to \(G_2\)) is an injective function \(f: V_1 \rightarrow V_2\) such that:

1. \(L_1(v) = L_2(f(v))\) for any vertex \(v \in V_1\);
2. For any edge \((v_1, v_2) \in E_1\), there exists an edge \((f(v_1), f(v_2)) \in E_2\).

The above definition applies to the unlabelled graph as well by ignoring the first condition.

The embedding \(f\) can be represented as a set of vertex pairs \((u, v)\) where \(u \in V_1\) is mapped to \(v \in V_2\) and we have \(f^{-1}(v) = u\). If there is an embedding of \(G_1\) in \(G_2\), we say
1.3 Proposed Approaches

$G_1$ is subgraph isomorphic to $G_2$ and denote it by $G_1 \leq G_2$. If $G_1 \leq G_2$ and $G_2 \leq G_1$, we say $G_1$ is isomorphic to $G_2$, denoted $G_1 \cong G_2$.

There may be multiple embeddings of $G_1$ in $G_2$ if $G_1 \leq G_2$. We use $F(G_1, G_2)$ to denote the set of all such embeddings. For each $f \in F(G_1, G_2)$, we define $\text{VCover}(f) = \{ f(v) | v \in V_1 \}$, and call the vertices in $\text{VCover}(f)$ the covered vertices of $G_2$ by $f$.

**Symmetry Breaking** A symmetry breaking technique based on automorphism is conventionally used to reduce duplicate embeddings [32][40]. As a result the data vertices in the final embeddings should follow some preserved orders of query vertices. We apply this technique in this thesis by default and we will specify the preserved order when necessary.

**Degree Filter** A degree filter is based on that a data vertex $v$ cannot map to query vertex $u$ if the degree $v$ is smaller than that of $u$. We apply this filter by default.

**Partial Embedding** A partial embedding of graph $q$ in graph $G$ is an embedding in $G$ of a vertex-induced subgraph of $q$. The following lemma is obvious.

**Lemma 1.** Let $f$ be an embedding from $q$ to $G$. Let $V'_q$ be a subset of the vertices in $q$. Restricting $f$ to $V'_q$ will always produce a partial embedding from $q$ to $G$.

**Subgraph Listing & Enumeration** In this thesis, we regard the problem of subgraph isomorphism search as to find all the embeddings of the given query graph from the data graph. This search process is also refereed to as subgraph listing and subgraph enumeration. We use them interchangeably.

**Maximal Common Subgraph** Given two graphs $G_1$ and $G_2$, a maximal common subgraph (MCS) of $G_1$ and $G_2$ is a connected graph $G'$ such that

1. $G' \leq G_1$ and $G' \leq G_2$.
2. there is no connected graph $G''$ such that $G'' \leq G_1$, $G'' \leq G_2$, and $G' \leq G''$, but $G' \not\cong G''$.

Note that the MCS is required to be connected. Clearly, there can be multiple MCSs between two graphs.

1.3 Proposed Approaches

In this thesis, we propose three efficient approaches to speed up subgraph isomorphism search in large graphs. Here we present a brief overview of those proposed approaches.
1.3.1 Exploiting Vertex Relationships

Most of the subgraph isomorphism algorithms of the In-Memory model (IM) are based on a backtracking method which computes the solutions by incrementally enumerating all candidate combinations. This backtracking method was proposed by Ullmann’s 1976 paper. After that various pruning rules, query vertex matching orders and auxiliary indexing techniques are proposed to improve the overall performance. However, we observed that all current algorithms blindly verify each individual mapping separately, often leading to extensive duplicate calculations. Typically, duplicate calculations caused by candidate vertices with the same or similar structure dramatically slow down the matching progress, which has not yet drawn enough attention.

In this thesis, we first speed up the subgraph isomorphism search by exploiting vertex relationships. To be specific,

(1) We define four types of relationships between vertices in the data graph, namely **syntactic containment**, **syntactic equivalence**, **query-dependent containment** and **query-dependent equivalence**. We show some interesting properties of such relationships.

(2) We show how the original data graph can be transformed into an **compressed hypergraph** $G_{sh}$ based on the first two types of relationships identified above, and how $G_{sh}$ can be used to speed-up subgraph isomorphism search. $G_{sh}$ can be built off-line, and used for any query graph.

(3) To further reduce duplicate computation using the last two types of relationships, we propose **BoostIso**, an approach that uses on-line **Dynamic Relationship Tables** with respect to each specific query graph, as well as $G_{sh}$. **BoostIso** can be integrated into the generic subgraph isomorphism framework and used by all backtracking algorithms.

(4) By implementing five subgraph isomorphism algorithms with the integration of our approach, we show that most existing subgraph isomorphism algorithms can be significantly speeded-up, especially for some datasets with intensive vertex relationships, where the improvement can be up to several orders of magnitude.
1.3 Proposed Approaches

1.3.2 Multi-Query Optimization

Existing work on subgraph isomorphism search mainly focuses on *a-query-at-a-time* approaches: optimizing and answering each query separately. When multiple queries arrive at the same time, sequential processing (SQO) is not always the most efficient.

The second approach we propose is to optimize the processing of multiple-queries (MQO) together so as to reduce the overall processing time. To be specific,

1. We introduce the concept of *tri-vertex label sequence* (TLS) and propose a novel grouping factor between two query graphs, which can be used to efficiently filter out graphs that do not share helpful common subgraphs.

2. We propose a heuristic algorithm to compute a good query execution order, which guarantees the cached results can be shared effectively, and enables efficient cache memory usage.

3. We propose a new type of graph partition, based on which we design a novel structure to store the query results of common subgraphs in main memory. This structure can effectively balance the cache memory size and efficiency of utilizing the cached results. We prove the new graph partition problem is NP-complete, and provide a heuristic algorithm that can produce a good partition for our purpose.

4. We present strategies to revise the current state-of-the-art sequential query optimizers for subgraph isomorphism search so that they can seamlessly utilize the cached intermediate results.

We conducted extensive experiments to show that, using our techniques, the overall query processing time when multiple queries are processed together can be significantly shorter than if the queries are processed separately when the queries have many overlaps. Furthermore, the larger the data set or the more time it takes for an average query, the more savings we can achieve. When the queries have no or little overlap, our filtering technique can detect it quickly, resulting in only a slight overhead compared with SQO.

1.3.3 Distributed Processing

The subgraph isomorphism solutions of IM model are facing critical I/O problems when the data graph cannot fit in memory. Many efforts have been committed in order to
tackle this problem under distributed and parallel environments, while the state-of-the-art solutions either rely on crippling joins or cumbersome indices, which makes those solutions hard to be practically used. Moreover, most of them follow a synchronous model whose performance is often bottlenecked by the machine with the worst performance in the cluster.

Motivated by this, in the third part of this thesis, we switch our focus to distributed settings and propose PADS, a Practical Asynchronous Distributed Subgraph enumeration system. To be specific, we have the following contributions in this chapter:

1. We propose a new subgraph enumeration approach called R-Meef (region-grouped multi-round expand-verify-filter), which borrows the incremental verification strategy from backtracking algorithms to filter out false intermediate results as early as possible and there are no more joins.

2. We propose an algorithm to compute the efficient execution plan which firstly minimizes the number of rounds based on the concept of minimum connected dominating set, and secondly brings forward the verification edges so as to filter out unpromising candidates as early as possible.

3. We introduce an inverted trie structure to save the embedding candidates in a compact format which significantly reduces memory cost. Moreover, each embedding candidate has a unique ID in the inverted trie, which facilitates the lightweight data-exchange of R-Meef. And the inverted trie enables us to efficiently retrieve, remove and incrementally expand the embeddings compressed inside.

4. We propose a grouping strategy to group the data vertex candidates into separated Region Groups considering the capability of each machine. Our system processes each region group separately so that the peak number of intermediate results in memory can be controlled. The candidates within the same group are relatively close to each other so that the searching process starting from them can be effectively shared.

5. Another technique we proposed for effective memory usage is the strategy of maximum-data caching. Based on the capacity of local memory or a given memory threshold, we cache the fetched data as much as we can and dump the caches only when it reaches the limit. This technique significantly reduces the number of data that are exchanged through network.
We experimentally demonstrate the problem of intermediate latency of the join-oriented approach compared with backtracking method. We also conducted extensive experiments to evaluate the performance of our approach.

1.4 Thesis Outline

The rest of the thesis is organized as follows. In Chapter 2, we present our approach to speed up subgraph isomorphism search by exploiting vertex relationships. In Chapter 3, we give our multiple-query optimization solution. In Chapter 4, we present our system for subgraph enumeration under distributed settings. Finally, we conclude the thesis with a conclusion and future work in Chapter 5.
Chapter 2

Exploiting Vertex Relationships

In this chapter, we present our approach of exploiting vertex relationships. We first give our motivation problems in Section 2.1. Then we present the related work in Section 2.2. After that Section 2.3 defines four vertex relationships and Section 2.4 proposes the algorithm to transform the data graph into a compressed graph $G_{sh}$. Our approach $BoostIso$ is presented in Section 2.5. Section 2.6 presents the experiments. Section 3.9 concludes this chapter.

2.1 Motivation Problems

Most in-memory subgraph isomorphism algorithms are based on a backtracking method which computes the solutions by incrementally enumerating and verifying candidates for all vertices in a query graph [34]. A variety of techniques has been proposed to accelerate the matching process, such as matching order selection, efficient pruning rules and pattern-at-a-time matching strategies (see Section 2.2 for a brief survey of these techniques). However, we observe that all existing algorithms suffer from extensive duplicate computation that could have been avoided by exploiting the relationships between vertices in the data graph, as shown in the following examples.

Example 2. Consider the query graph $G_q$ and the data graph $G$ in Figure 2.1. Assume the matching order is $u_1$-$u_2$-$u_3$-$u_4$. Each query vertex $u$ in $G_q$ has a candidate list $C(u)$ which contains the data vertices having the same label as $u$. Then we have $C(u_1) = \{v_1, v_2, v_3\}$. In the backtracking process, $v_1, v_2, v_3$ will be checked one by one to see whether they can match $u_1$. For each of them, there are $|C(u_2)| \times |C(u_3)| \times |C(u_4)|$ combinations to be verified. However, observe that the set of neighbors of both $v_1$ and $v_3$ are subsets of that of $v_2$. 
Therefore, if \( v_2 \) is first computed and fails to match \( u_1 \), then \( v_1 \) and \( v_3 \) can be known not to be able match \( u_1 \) immediately, without further computation.

The above example shows that, if the candidate vertices in \( C(u) \) are checked in an appropriate order, then some duplicate computation may be avoided. Note that some previous algorithms considered the ordering of query vertices, but to the best of our knowledge, they did not consider the ordering of candidate vertices in the data graph.

**Example 3.** Consider the query graph \( G_q \) in Figure 2.1 (a) and the data graph \( G' \) in Figure 2.1 (d). In \( G' \), \( v_1 \) and \( v_2 \) share exactly the same set of neighbors. If there is any embedding \( f \) involving \( v_1 \), we may get another embedding simply by replacing \( v_1 \) with \( v_2 \) in \( f \), and vice versa. For instance, from the embedding \( \{(u_1, v_1), (u_2, v_3), (u_3, v_5), (u_4, v_6)\} \), we can obtain another embedding \( \{(u_1, v_2), (u_2, v_3), (u_3, v_5), (u_4, v_6)\} \) by replacing \( v_1 \) with \( v_2 \).

Example 3 shows that, if data vertices have the same neighbourhood structure, then they can be regarded as “equivalent” in that if one can be matched to a query vertex, so can the others. Thus we only need to verify one of them, instead of all of them.

**Example 4.** Consider the query graph \( G_q' \) in Figure 2.1 (b) and the data graph \( G \) in Figure 2.1 (c). Although data vertices \( v_7 \) and \( v_{1006} \) do not have identical neighbour set, their \( B \)-labeled neighbours are identical. Notice that the query vertex \( u_3 \) has only a \( B \)-labeled neighbour. Therefore, if \( v_7 \) can be matched to \( u_3 \), then \( v_{1006} \) can also be matched to \( u_3 \), and vice versa.

Example 4 shows that, even if two vertices in the data graph do not share the same set of neighbours, they may still be regarded as “equivalent” with respect to a specific query vertex when searching for isomorphic subgraphs.

The above examples motivate us to identify useful relationships between data vertices and develop techniques to exploit such relationships in speeding up subgraph isomorphism search. We find that the vertex relationships are abundant in many real graphs,
such as protein networks, collaboration networks and social networks. For instance, in Human (a protein interaction network), more than 53% of data vertices hold equivalent relationships and among those that are not equivalent, 56.8% hold containment relationships. In Youtube (a social network), more than 37% of data vertices can be reduced by equivalent relationships and a further 42% of data vertices hold containment relationships.

### 2.2 Related Work

**Existing Subgraph Isomorphism Algorithms.** Subgraph isomorphism has been investigated for many years. Existing algorithms can be divided into two classes: (1) Given a graph database consisting of many small data graphs, retrieve all the data graphs containing a given query graph. (2) Given a query graph, find all embeddings in a single large graph. Our work belongs to the second class. Existing algorithms falling into this class include Ullmann [58], VF2 [8], QuickSI [53], GraphQL [25], SPath [62], STW [56] and Turboiso [24]. Most of them are based on a backtracking strategy which incrementally finds partial solutions by adding join-able candidate vertices. A recent survey [34] presents a generic framework for subgraph isomorphism search, which is shown in Algorithm 1.

In Algorithm 1, the inputs are a query graph and a data graph, the outputs are all the embeddings. Each embedding is represented by a list $f$ which comprises pairs of a query vertex and a corresponding data vertex. $initializeCandidates$ is to find a set of candidate vertices $C(u)$ for each query vertex $u$. If any $C(u)$ is empty, the algorithms terminates immediately. In each recursive call of $subgraphSearch$, once the size of $f$ equals to the number of query vertices, a solution is found and reported. $nextQueryVertex$ returns the next query vertex to match according to the query vertex matching order. Pruning rules are implemented in $refineCandidates$ to filter unpromising candidates. $isJoinable$ is the final verification to determine whether the candidate vertex can be added to the partial solution. $updateState$ adds the newly matched pair $(u, v)$ into $f$ while $restoreState$ restores the partial embedding state by removing $(u, v)$ from $f$.

Matching Order Optimization. The Ullmann algorithm [58] does not define the matching order of the query vertices. VF2 [8] starts with a random vertex and selects the next vertex which is connected with the already matched query vertices. By utilizing global statistics of vertex label frequencies, QuickSI [53] proposes a matching order which accesses query vertices having infrequent vertex labels as early as possible. In contrast to
Algorithm 1: GENERIC FRAMEWORK

Input: Data graph \( G \) and query graph \( G_q \)
Output: All embeddings of \( G_q \) in \( G \)

1. \( f \leftarrow \emptyset \)
2. for each \( u \in V_q \) do
3. \( C(u) \leftarrow \text{initializeCandidates}(G_q, G, u) \)
4. if \( C(u) = \emptyset \) then
5. return
6. subgraphSearch\((G_q, G, f)\)

Subroutine subgraphSearch\((G_q, G, f)\)

1. if \( |f| = |V_q| \) then
2. report \( f \)
3. else
4. \( u \leftarrow \text{nextQueryVertex}() \)
5. \( \text{refineCandidates}(f, u, C(u)) \)
6. for each \( v \in C(u) \) and \( v \) is not matched do
7. if \( \text{isJoinable}(f, v, G, G_q) \) then
8. \( \text{updateState}(f, u, v, G, G_q) \)
9. subgraphSearch\((G_q, G, f)\)
10. \( \text{restoreState}(f, u, v, G, G_q) \)

QuickSI’s global matching order selection, TurboIso [24] divides the candidates into separate candidate regions and computes the matching order locally and separately for each candidate region. Both STW [56] and TurboIso [24] give higher priority to query vertices with higher degree and infrequent labels.

Efficient Pruning Rules. The Ullmann algorithm [58] only prunes out the candidate vertices having a smaller degree than the query vertex. While VF2 [8] proposes a set of feasibility rules to prune out unpromising candidates, namely, 1-look-ahead and 2-look-ahead rules. SPath [62] uses a neighbourhood signature to index the neighbourhood information of each data vertex, and then prunes out false candidates whose neighbourhood signature does not contain that of the corresponding query vertex. GraphQL [25] uses a pseudo subgraph isomorphism test. TurboIso [24] exploits a neighborhood label filter to prune out unpromising data vertices.

Pattern-At-A-Time Strategies. Instead of the traditional vertex-at-a-time fashion, SPath [62] proposes an approach which matches a graph pattern at a time. The graph pattern used in SPath is path. TurboIso [24] rewrites the query graph into a NEC tree, which matches the query vertices having the same neighbourhood structure at the same time.

Different from these previous techniques, our method focuses on (1) reducing the
search space by grouping “equivalent” vertices together, and (2) optimizing the candidate vertex matching order to avoid duplicate computation. Our approach is not a single algorithm, it is an approach that can be integrated into all existing backtracking algorithms.

**Graph Summary and Graph Compression.** The grouping of data vertices into hypernodes in our approach bears some similarity to structural summaries [6, 30, 39], graph summarization [42, 57], and query-preserving graph compression [15]. Structural summaries are designed for path expressions, hence they group vertices sharing the same set of incoming label paths into a hypernode. The graph summarization proposed in [42] is in effect a compression technique that aims at saving storage space. It consists of two parts: a graph summary and a set of edge corrections. The summary part groups nodes with similar neighbors into a hypernode, while the edge corrections are used to ensure accuracy during decompression. A second type of graph summarization aims at reducing the size of a large graph to help users understand the characteristics of the graph. These techniques group vertices into hypernodes based on a variety of statistics, such as node attributes values [61], degree distribution, or user-specified node attributes [57].

More closely related to our work is [15], which proposes a framework for query-preserving graph compression as well as two compression methods that preserve reachability queries and pattern matching queries (based on bounded simulation) respectively. Both methods are based on equivalence relations defined over the vertices of the original graph $G$, and compress $G$ by merging vertices in the same equivalent class into a single node. Part of our compressed graph is based on a similar idea, that is, we combine vertices that are “equivalent” for subgraph isomorphism queries into a hypernode, and like the compressed graphs for reachability and for bounded simulation, our compressed graph can be directly queried for subgraph isomorphism search. However, our compressed graph goes beyond grouping nodes into hypernodes. It also includes edges that represent “containment” relationships for subgraph isomorphism, which can be utilized to effectively optimize the candidate vertex matching order. Moreover, besides the compressed graph constructed offline, we provide a method to further speed-up query processing on-the-fly by utilizing query-dependent equivalence and query-dependent containment relationships among data vertices, which proves to be highly effective in our experiments. These, to the best of our knowledge, have not been studied in previous work.
2.3 Relationships Between Data Vertices

In this section, we identify four types of relationships between the vertices of a data graph and show some useful properties of these relationships.

2.3.1 Syntactic Containment

Definition 1. Given a data graph \( G \) and a pair of vertices \( v_i, v_j \) in \( G \), we say \( v_i \) syntactically contains (or simply S-contains) \( v_j \), denoted \( v_i \succeq v_j \), if \( L(v_i) = L(v_j) \) and \( \text{Ad j}(v_j) - \{v_i\} \subseteq \text{Ad j}(v_i) - \{v_j\} \), where \( \text{Ad j}(v_i) \) is the neighbour set of \( v_i \) and \( \text{Ad j}(v_j) \) is the neighbour set of \( v_j \).

The above definition defines a binary relation among the vertices of \( G \). If \( v_i \succeq v_j \), then \( v_i \) and \( v_j \) have the same label, and the neighbour set (excluding \( v_i \)) of \( v_j \) is a subset of the neighbour set (excluding \( v_j \)) of \( v_i \). Hereafter, we refer to syntactic containment relation as SC relation for short.

Example 5. In the data graph \( G \) in Figure 2.1(c), \( L(v_1) = L(v_2) = L(v_3) \). Also \( \text{Ad j}(v_1) = \{v_4, v_5\} \), \( \text{Ad j}(v_2) = \{v_4, v_5, v_6\} \) and \( \text{Ad j}(v_3) = \{v_5, v_6\} \). Because \( \text{Ad j}(v_1) - \{v_2\} \subseteq \text{Ad j}(v_2) - \{v_1\} \) and \( \text{Ad j}(v_3) - \{v_2\} \subseteq \text{Ad j}(v_2) - \{v_3\} \), we have \( v_2 \succeq v_1 \) and \( v_2 \succeq v_3 \).

The SC relation is transitive, as shown in the proposition below.

Proposition 1. For any three nodes \( v_i, v_j, v_k \) in \( G \), if \( v_i \succeq v_j \) and \( v_j \succeq v_k \), then \( v_i \succeq v_k \).

Proof. By definition, if \( v_i \succeq v_j \) and \( v_j \succeq v_k \), we have \( \text{Ad j}(v_j) - \{v_i\} \subseteq \text{Ad j}(v_i) - \{v_j\} \) and \( \text{Ad j}(v_k) - \{v_j\} \subseteq \text{Ad j}(v_j) - \{v_k\} \). Combining these two formulas we get

\[
\text{Ad j}(v_k) - \{v_i\} - \{v_j\} \subseteq \text{Ad j}(v_i) - \{v_k\} - \{v_j\}
\]  

(2.1)

There are three cases: (a) \( v_j \notin \text{Ad j}(v_k) \), (b) \( v_j \in \text{Ad j}(v_k) \) and \( v_j \notin \text{Ad j}(v_i) \), (c) \( v_j \in \text{Ad j}(v_k) \) and \( v_j \notin \text{Ad j}(v_i) \). In the first two cases, we can easily infer \( \text{Ad j}(v_k) - \{v_i\} \subseteq \text{Ad j}(v_i) - \{v_k\} \) from formula (2.1). That is, \( v_i \succeq v_k \). Next we show the third case is not possible. This is because in this case \( v_i \notin \text{Ad j}(v_j) \), and \( v_k \notin \text{Ad j}(v_j) \). Thus if \( v_i \in \text{Ad j}(v_k) \), then \( \text{Ad j}(v_k) - \{v_j\} \nsupseteq \text{Ad j}(v_j) - \{v_k\} \), contradicting \( v_j \succeq v_k \); and if \( v_i \notin \text{Ad j}(v_k) \), then \( v_k \notin \text{Ad j}(v_i) \), hence \( \text{Ad j}(v_i) - \{v_i\} \nsupseteq \text{Ad j}(v_i) - \{v_j\} \), contradicting \( v_i \succeq v_j \).

Since we assume the data graph is connected, for any two vertices \( v_i, v_j \) in \( V \), if \( v_i \succeq v_j \), then either \( v_i \) is a neighbour of \( v_j \), or \( v_i \) and \( v_j \) share at least one common neighbour. Therefore, we have
Proposition 2. Any two data vertices satisfying the SC relation is 1-step reachable or 2-step reachable from each other. That is, there is a 1-edge or 2-edge path between them.

The next proposition indicates how the SC relation can be used in subgraph isomorphism search. Intuitively, if $v_i \succeq v_j$, then replacing $v_j$ with $v_i$ (assuming $v_i$ is unused) in any embedding will result a new embedding.

Proposition 3. Given a pair of vertices $v_i, v_j$ in data graph $G$, if $v_i \succeq v_j$, then for any embedding $f$ of any query graph $G_q$ in $G$, where $f$ maps query vertex $u$ to $v_j$, and maps no query vertex to $v_i$, $f' = f - \{(u, v_j)\} + \{(u, v_i)\}$ is also an embedding of $G_q$ in $G$.

Proof. We only need to show that $f'$ maps every edge incident on $u$ in the query graph to an edge in the data graph $G$. Suppose $(u, u')$ is an edge in the query graph. Since $f$ is an embedding, $(f(u), f(u'))$ is an edge in $G$, that is, $(v_j, f(u'))$ is an edge in $G$. Since $v_i \succeq v_j$, we know there is an edge $(v_i, f(u'))$ in $G$ (note that $f(u') \neq v_i$ because we assume $v_i$ is not used in $f$). Since $f'(u) = v_i$, and $f'(u') = f(u')$, we know $(f'(u), f'(u'))$ is an edge in $G$. □

From the above proposition, it is also clear that if $v_i \succeq v_j$ and $v_i$ is pruned in the matching process, then $v_j$ can also be safely pruned. This is because if $v_i$ cannot be matched to a query vertex by some embedding, then $v_j$ cannot either.

Example 6. Consider the data graph $G$ in Figure 2.1(c), we have $v_2 \succeq v_1$ and $v_2 \succeq v_3$. For query graph $G_q$, $v_2$ fails to match to query vertex $u_1$, thus we know immediately that $v_1$ and $v_3$ cannot be matched to $u_1$.

2.3.2 Syntactic Equivalence

Definition 2. Given a data graph $G$ and any pair of vertices $v_i, v_j$ in $G$, we say $v_i$ is syntactically equivalent (or simply S-equivalent) to $v_j$, denoted $v_i \simeq v_j$, if $L(v_i) = L(v_j)$ and $\text{Adj}(v_j) - \{v_i\} = \text{Adj}(v_i) - \{v_j\}$.

Example 7. Consider data graph $G'$ in Figure 2.1(d). $v_1$ and $v_2$ share the same label and the same set of neighbors. Thus we have $v_1 \simeq v_2$.

Clearly, syntactic equivalence is two-way syntactic containment. It defines a relation among the vertices of $G$ which is reflexive, symmetric and transitive (The transitivity is evident from Proposition 1). Thus the syntactic equivalence relation is a class. Hereafter, we refer to syntactic equivalence relation as SE relation for short.
From Proposition 3, we know that if two data vertices \(v_i, v_j\) satisfy the SE relation, then if there is an embedding \(f\) that maps a query vertex to \(v_i\), there is also an embedding that maps the query vertex to \(v_j\) (if \(v_j\) is not used in \(f\)), while the two embeddings are identical on other query vertices. If an embedding \(f\) maps \(u_1\) to \(v_i\), and \(u_2\) to \(v_j\), then swapping the images of \(u_1\) and \(u_2\) will result in another embedding.

### 2.3.3 Query-Dependent Containment

Before we give the definition of query-dependent containment, let us first define query-dependent neighbors.

**Definition 3.** Given a query graph \(G_q\), vertex \(u \in V_q\), a data graph \(G\), and vertex \(v \in V\), where \(L(v) = L_q(u)\), the set of query-dependent neighbors of \(v\) w.r.t \(u\), denoted \(QDN(G_q, u, v)\), is the set of data vertices \(\{v_i | v_i \in \text{Adj}(v), L(v_i) \in \{L_q(u_i) | u_i \in \text{Adj}(u)\}\}\).

Intuitively, \(QDN(G_q, u, v)\) is a subset of \(v\)’s neighbors with the requirement that the labels of these neighbours must appear as labels of \(u\)’s neighbours in the query graph.

**Example 8.** Consider the query graph \(G_q\) in Figure 2.2(a) and the data graph \(G\) in Figure 2.2(c). \(QDN(G_q, u_1, v_4) = \{v_9, v_{11}, v_{13}, v_{14}\}\). But for query graph \(G_{q'}\) in Figure 2.2(b), \(u_1\) has no neighbor with label \(D\), any data vertices with label \(D\) will be ignored. Thus we have \(QDN(G_{q'}, u_1, v_4) = \{v_9, v_{13}, v_{14}\}\).

We can now define query-dependent containment.

**Definition 4.** Given a query vertex \(u\) in \(G_q\), and two data vertices \(v_i, v_j\) in \(G\), we say \(v_i\) query-dependently contains (or simply QD-contains) \(v_j\) with respect to \(u\) and \(G_q\), denoted \(v_i \succeq_{(G_q,u)} v_j\), if \(L(v_i) = L(v_j)\) and \(QDN(G_q, u, v_j) - \{v_j\} \subseteq QDN(G_q, u, v_i) - \{v_i\}\).
Hereafter, we refer to query-dependent containment relation as QDC relation for short. The essential difference between QDC and SC is that latter is not related to any query graph, but the former is defined with respect to a specific query vertex of a query graph. It is easy to verify that, if \( v_i \succeq v_j \) holds, then \( v_i \succeq_{(G_q,u)} v_j \) holds for any query vertex \( u \) of any query graph \( G_q \).

**Example 9.** Consider the vertices \( v_3 \) and \( v_4 \) of data graph \( G \) in Figure 2.2(c) and vertex \( u_1 \) of query graph \( G_q' \) in Figure 2.2(b). We have \( \text{QDN}(G_q', u_1, v_3) = \{v_9, v_{10}, v_{13}, v_{14}\} \) and \( \text{QDN}(G_q', u_1, v_4) = \{v_9, h_{13}, h_{14}\} \). Hence \( \text{QDN}(G_q', u_1, v_3) \subseteq \text{QDN}(G_q', u_1, v_4) \). Therefore, we have \( v_3 \succeq_{(G_q,u_1)} v_4 \).

Similar to SC, QDC is transitive, and it can be utilized in searching for isomorphic subgraphs.

**Proposition 4.** Given data vertices \( v_i, v_j \) in \( G \) and a query vertex \( u \) in \( G_q \), if \( v_i \succeq_{(G_q,u)} v_j \), then for each embedding \( f \) of \( G_q \) in \( G \) that maps \( u \) to \( v_j \) but no query vertex to \( v_i \), \( f' = f - \{(u, v_j)\} + \{(u, v_i)\} \) is also an embedding of \( G_q \) in \( G \).

The proof of Proposition 4 is similar to that of Proposition 3. Hence it is omitted.

### 2.3.4 Query-Dependent Equivalence

**Definition 5.** Given a query vertex \( u \) in \( G_q \) and two data vertices \( v_i, v_j \) in \( G \), we say \( v_i \) is query-dependently equivalent (or simply QD-equivalent) to \( v_j \) with respect to \( u \) and \( G_q \), denoted \( v_i \equiv_{(G_q,u)} v_j \), if \( L(v_i) = L(v_j) \) and \( \text{QDN}(G_q, u, v_j) - \{v_i\} = \text{QDN}(G_q, u, v_i) - \{v_j\} \).

Clearly, query-dependent equivalence is two-way query-dependent containment. Using Proposition 4, we can infer that if \( v_i \equiv_{(G_q,u)} v_j \), then for any embedding \( f : G_q \rightarrow G \) that maps \( u \) to \( v_i \) but no query vertex to \( v_j \), \( f' = f - \{(u, v_j)\} + \{(u, v_i)\} \) is also an embedding, and vice versa.

Hereafter, if \( v_i \succeq_{(G_q,u)} v_j \) but not \( v_i \equiv_{(G_q,u)} v_j \), then we say \( v_i \) strictly QD-contains \( v_j \) w.r.t \( u \) and \( G_q \), and will denote it by \( v_i \succ_{(G_q,u)} v_j \). We refer to query-dependent equivalence relation as QDE relation for short.

### 2.4 Graph Compression

In this section, we present an algorithm to transform the data graph into an compressed hypergraph (or simply compressed graph) which is able to answer subgraph isomorphism more efficiently. We call this process graph compression.
2.4.1 Compressed Graph for Subgraph Isomorphism

We need to define syntactic equivalence class first.

**Definition 6.** Given a data graph $G$, the syntactic equivalence class of a data vertex $v$ in $G$, denoted $\text{SEC}(v)$, is a set of data vertices which are $S$-equivalent to $v$.

As mentioned earlier, the syntactic equivalence relation is a class. Therefore, any pair of vertices in the same syntactic equivalence class are $S$-equivalent.

The next proposition is important.

**Proposition 5.** Data vertices in the same syntactic equivalence class either form a clique (i.e., they are pairwise adjacent), or are pairwise non-adjacent.

**Proof.** It suffices to prove that, for any three distinct data vertices $v_i$, $v_j$, and $v_k$ in the same syntactic equivalent class, if $v_i, v_j$ are adjacent, then $v_j, v_k$ are also adjacent. Since $v_i \simeq v_k$, by definition, $\text{Adj}(v_i) - \{v_k\} = \text{Adj}(v_k) - \{v_i\}$. Therefore, if $v_i$ and $v_j$ are adjacent, that is, $v_j$ is in $\text{Adj}(v_i)$, then $v_j$ is also in $\text{Adj}(v_k)$, hence $v_j$ and $v_k$ are also adjacent. ∎

Proposition 5 implies that the data vertices in the same $\text{SEC}(v)$ are either all 1-step reachable from each other (when they form a clique) or all 2-step reachable from each other (when they are not adjacent to each other but share the same set of neighbours).

**Definition 7 (Compressed hypergraph).** Given a data graph $G = (V, E, \Sigma, L)$, the compressed hypergraph of $G$ is a graph $G_{sh} = (V_{sh}, E_{se}, E_{sc}, \Sigma_{sh}, L_{sh})$, such that

(a) $V_{sh} = \{h | h = \text{SEC}(v), v \in V\}$ is the set of hypernodes.

(b) $E_{se}$ is a set of undirected edges such that, an edge from $h$ to $h'$ exists iff $(v_i, v_j) \in E$, where $h = \text{SEC}(v_i)$, $h' = \text{SEC}(v_j)$ and $h$ must be a clique when $h'$ is a clique.

(c) $E_{sc}$ is the smallest set of directed edges such that a path from $h$ to $h'$ exists iff $h \succeq h'$.

(d) $\Sigma_{sh} = \Sigma$.

(e) For each $h \in V_{sh}$, $L_{sh}(h) = L(v)$ where $h = \text{SEC}(v)$.

**Remark** The hypergraph $G_{sh}$ captures the structure of the original data graph as well as the SE and SC relationships between the data vertices.

1. Each hypernode groups all the $S$-equivalent data vertices together, thus two data vertices are $S$-equivalent if and only if they are in the same hypernode.
2. $E_{se}$ is a set of undirected edges that capture the structure of the original graph. Observe that if there is an edge between $v_1 \in h_1$ and $v_2 \in h_2$, then there is an edge between every pair of vertices $v_i, v_j$ where $v_i \in h_1$ and $v_j \in h_2$.

3. $E_{sc}$ is a set of directed edges that capture SC relations among the hypernodes. Observe that for any two hypernodes $h_1$ and $h_2$, $h_1 \succeq h_2$ if $v_1 \succeq v_2$ for every pair of vertices where $v_1 \in h_1$, $v_2 \in h_2$ and $h' = SEC(v_j)$ and $h$ must be a clique when $h'$ is a clique.

It is worth noting that $E_{sc}$ is a minimal set of directed edges such that if $h_i \succeq h_j$, there is a path from $h_i$ to $h_j$. This requirement is to reduce the size of $G_{sh}$.

4. The hypergraph can be divided into two parts: the SE graph and the SC-graph. The SE-graph consists of the hypernodes and the undirected edges, while the SC-graph consists of the hypernodes and the directed edges. Note that these two parts share the same set of hypernodes.

**Example 10.** Consider the data graph in Figure 2.2(c). We show the compressed hypergraph $G_{sh}$ in two parts: SE-graph in Figure 2.2(d) and SC-graph in Figure 2.2(e). In Figure 2.2(e) we omit the hypernodes that are not incident on the directed edges.

**Definition 8 (Hyperembedding).** Given a query graph $G_q$ and the compressed hypergraph $G_{sh}$, a hyperembedding of $G_q$ in $G_{sh}$ is a mapping $f_h: V_q \rightarrow V_{sh}$, such that

1. $L_{sh}(f_h(u)) = L_q(u)$ for all $u \in V_q$.
2. For each edge $(u_i, u_j) \in E_q$, if $f_h(u_i) \neq f_h(u_j)$, there exists an edge $(f_h(u_i), f_h(u_j)) \in E_{se}$.
3. For each edge $(u_i, u_j) \in E_q$, if $f_h(u_i) = f_h(u_j)$, all the data vertices in $f_h(u_i)$ form a clique.
4. For each $h \in V_{sh}$, $h$ can be matched to up to $|SEC(v)|$ vertices of $V_q$, where $h = SEC(v)$ and $|SEC(v)|$ is the number of data vertices in $SEC(v)$.

The following theorem shows the relationship between hyperembeddings and subgraph isomorphism.

**Theorem 1.** Suppose $G_{sh}$ is the compressed hypergraph of data graph $G$, and $G_q$ is any query graph.
(1) Let $f_h$ be a hyperembedding of $G_q$ in $G_{sh}$. Let $f : V_q \rightarrow V$ map every node $u \in V_q$ to a data vertex $v \in f_h(u)$ such that $v$ has not been matched to other query vertices by $f$. Then $f$ is an embedding of $G_q$ in $G$.

(2) Every embedding of $G_q$ in $G$ can be obtained from a hyperembedding of $G_q$ in $G_{sh}$, in the way described above.

\textbf{Proof.} (1) First, we note that $f$ is a valid injective function: it maps different nodes in $V_q$ to different nodes in $V$, and since $f_h$ maps no more than $|h|$ query vertices to $h$, we have enough data vertices in $h$ to be matched to query vertices which are mapped to $h$ by $f_h$. Second, for every $u \in V_q$, we have $L(f(u)) = L_h(f_h(u)) = L_q(u)$. Third, for every edge $(u, u') \in E_q$, either there is an edge $(f_h(u), f_h(u'))$ in $G_{sh}$ or $f_h(u) = f_h(u')$ and $f_h(u)$ is a clique. In both cases, there is an edge $(f(u), f(u'))$ in $G$. Therefore, $f$ is an embedding of $G_q$ in $G$.

(2) Let $f$ be an embedding of $G_q$ in $G$. Construct a mapping $f_h : V_q \rightarrow V_{sh}$ as follows: $\forall u \in V_q$, let $f_h$ map $u$ to the hypernode representing $SEC(f(u))$. It is easy to verify that $f_h$ is a hyperembedding of $G_q$ in $G_{sh}$, and $f$ can be obtained from $f_h$ by choosing $f(u) \in SEC(f(u))$ as the image, for any $u \in V_q$.

A backtracking algorithm slightly modified from Algorithm 1 can be used to find all hyperembeddings, as we will discuss later in Section 2.5.

\subsection*{2.4.2 Building Compressed Graph}

We give an algorithm, shown in Algorithm 2, for transforming the original graph $G$ into $G_{sh}$.

Algorithm 2 first assigns $\Sigma$ to $\Sigma_{sh}$ (Line 1) as $G_{sh}$ shares the same label set with the original graph. Then for each unvisited data vertex $v \in V$, it marks $v$ as visited and creates a new hypernode $h$ (Lines 2~4). It initializes $h$ by setting its $isClique$ as false and its label as that of $v$ (Line 5). Then it puts $v$ into $h$ (Line 6). The flag $isClique$ is used to indicate whether $h$'s data vertices form a clique or only share the same set of neighbours but not adjacent to each other. The algorithm first iterates through all the neighbours of $v$ and finds all data vertices belonging to $SEC(v)$ (Lines 7~10). If some S-equivalent vertices are found in its neighbours, then there is no need to iterate through 2-step($v$). Otherwise the algorithm will try to find S-equivalent vertices in 2-step reachability of $v$ (Lines 11~14). Once all the hypernodes are obtained, the edges between hypernodes will be added if there exists an edge between the data vertices in the corresponding hypernodes (Lines
2.4 Graph Compression

Algorithm 2: COMPUTE COMPRESSED GRAPH

**Input:** Data graph $G = (V, E, \Sigma, L)$

**Output:** Compressed graph $G_{sh} = (V_{sh}, E_{se}, E_{sc}, \Sigma_{sh}, L_{sh})$

1. $\Sigma_{sh} \leftarrow \Sigma$

2. **for each** $v \in V$ **do**
   
   | 3. if $v$ is not visited then |
   | 4. mark $v$ as visited, create hypernode $h$ |
   | 5. $h.isClique \leftarrow false$, set $L_{sh}(h) = L(v)$ |
   | 6. $V_{sh} \leftarrow V_{sh} \cup \{h\}$ |
   | 7. **for each** $v' \in 1$-step($v$) and $L(v') = L(v)$ **do** |
   | 8. if $v \approx v'$ then |
   | 9. $h.isClique \leftarrow true$ |
   | 10. add $v'$ to $h$ and mark $v'$ as visited |
   | 11. if $h.isClique$ is false then |
   | 12. **for each** $v' \in 2$-step($v$) and $L(v') = L(v)$ **do** |
   | 13. if $v \approx v'$ then |
   | 14. add $v'$ to $h$ and mark $v'$ as visited |

15. **for each** edge $(v, v') \in E$ **do**

16. if $v \in h$, $v' \in h'$ and $h \neq h'$ then

17. $E_{se} \leftarrow E_{se} \cup \{(h, h')\}$

18. **for each** $h \in V_{sh}$ **do**

19. $R(h) \leftarrow \{h'|h' \in adj(h) \cup 2$-step($h$), $L_{sh}(h')\}$

20. $L_{sh}(h) = L_{sh}(h')$

21. **for each** $h' \in R(v)$ **do**

22. if $h \geq h'$ then

23. $E_{sc} \leftarrow E_{sc} \cup (h, h')$

24. **transitiveReduction**($V_{sh}, E_{se}$)

25. **return** $G_{sh} = (V_{sh}, E_{se}, E_{sc}, \Sigma_{sh}, L_{sh})$

After the SE-graph is built, based on the SE-graph, for each hypernode $h \in V_{sh}$, the algorithm visits each node $h'$ in $h$’s neighbour set or in $h$’s neighbour’s neighbour sets that have the same label as $h$ (Lines 18~22). If $h \geq h'$, then an directed edge $(h, h')$ is added to $E_{sc}$ (Line 22). After all the SC edges are found, a transitive reduction is executed to minimize the number of the SC edges. Transitive reduction has been well studied, and we utilize an transitive reduction algorithm based on the idea given in [2].

**Example 11.** Consider the data graph $G$ in Figure 2.2(c). Algorithm 2 first finds S-equivalent vertices for each vertex of each label. $v_1$ is the first to be visited, $h_1$ is created with label $A$ and $v_1$ is put into $h_1$. As $v_1$ has no S-equivalent vertices in its neighbours, then its 2-step reachable vertices having label $A$, $v_2, v_3, v_3, v_4$, will be visited. Only $v_2 \approx v_1$, thus $v_2$ is
marked as visited and added into $h_1$. Because $v_1, v_2$ are not a clique, $h_1.isClique = false$. The same process goes on with $v_6$ and $v_7$ being grouped into $h_5$ and $h_5.isClique = true$. After all the hypernodes are created, edges between hypernodes will be added. Because $v_1 \in h_1, v_9 \in h_7$ and $(v_1, v_9) \in V$, we add $(h_1, h_7)$ to $E_{se}$. Once the SE-graph is created (Figure 2.2(d)), SC-graph will be built. We have $Adj(h_1) - h_2 \subseteq Adj(h_2) - h_1$, $h_2 \geq h_1$, thus $(h_2, h_1)$ is added to $E_{sc}$. Because $h_2 \geq h_5 \geq h_6$, the SC edge between $h_2$ and $h_6$ is removed by the transitive reduction. The final SC-graph is shown in Figure 2.2(e).

**Complexity.** For a vertex $v \in V$, we use 2-step-$SL(v)$ to denote the set of vertices that are reachable from $v$ within 1 or 2 steps and have the same label as $v$. In Algorithm 2, to find the hypernodes (Lines 2~14), for each vertex $v$, we may have to visit all of its neighbours and 2-step reachable vertices. For each pair of vertices $v_1, v_2$, it takes $d_1 + d_2$ to find their SE relationship where $d_i$ is the degree of $v_i$ (We note the neighbours are ordered by vertex ID). Therefore, computing the hypernodes takes $O(|V| \times N \times d)$ where $d$ is the maximal vertex degree in $G$ and $N$ is the maximal value of $|2$-step-$SL(v)|$ for all $v \in V$. Computing the SE edges (Lines 15~17) takes $O(|E|)$. Computing the SC edges (Lines 18~22) takes no more than $O(|V| \times N \times d)$. In addition, the complexity of transitive reduction is $O(n^3)$ for a graph of $n$ vertices [2]. Since the transitive reduction is only carried out on hypernodes with the same label, line 23 takes $O(\Sigma_{l \in \Sigma} N_l^3)$ where $N_l$ is the number of nodes with label $l$. Therefore, the overall complexity for constructing $G_{sh}$ is $O(|V| \times N \times d + |E| + \Sigma_{l \in \Sigma} N_l^3)$.

## 2.5 BoostIso

We present our approach for subgraph isomorphism search in this section. We refer to our approach as *BoostIso*.

In BoostIso, we search for hyperembeddings directly over $G_{sh}$ and then expand these hyperembeddings into embeddings. To reduce duplicate computation, we exploit QDC and QDE relations as well as the SC and SE relations. For clarity, we first present the revised algorithm for computing hyperembeddings when QDC and QDE relations are not considered. Then we discuss how to integrate the QDC and QDE relationships into the revised algorithm.

The data structures used are: (1) Two in-memory adjacency lists to store the two parts of the compressed graph. One is to store the SE graph, the other is to store the SC graph. For each hypernode $h$, we first group its neighbours by hypernode labels and then sort them in ascending order according to hypernode ID in each group. This enables us to...
compute the QDC and QDE relationships more efficiently. (2) An inverted vertex label list for the SE graph to efficiently access all hypernodes with a specific label.

### 2.5.1 Utilizing SC and SE

In this subsection, we first discuss how to utilize the SC and SE relationships in order to filter out unpromising candidates.

**Proposition 6.** Suppose $h$ and $h'$ are two hypernodes in the SE-Graph $G_{SE}$ and $h \succeq h'$. In any of the following cases (1), (2) and (3), for any hyperembedding $f$ of any query graph in $G_{SE}$, where $f$ maps vertex $u$ to $h'$, $f' = f - \{(u, h')\} \cup \{(u, h)\}$ is also a hyperembedding:

1. $f$ has not mapped any query vertex to $h$, and one of the following conditions holds:
   
   (a) $f$ has not mapped any query vertex other than $u$ to $h'$.
   
   (b) $h'$ is an independent set (this includes the case where $|h'| = 1$).
   
   (c) there is an edge between $h$ and $h'$.

2. $f$ has mapped less than $|h|$ query vertices to $h$, $h$ is a clique (this includes the case where $|h| = 1$), and one of the following conditions holds:
   
   (a) $f$ has not mapped any query vertex other than $u$ to $h'$.
   
   (b) $h'$ is an independent set (this includes the case where $|h'| = 1$).
   
   (c) there is an edge between $h$ and $h'$.

3. $f$ has mapped less than $|h|$ query vertices to $h$, there is no edge between $h$ and $h'$, and one of the following conditions holds:
   
   (a) $f$ has not mapped any query vertex other than $u$ to $h'$.
   
   (b) $h'$ is an independent set (this includes the case where $|h'| = 1$).

**Proof.** We only need to show that, for every edge $(u, u')$ in the query graph, $(f'(u), f'(u'))$ is an edge in the hypergraph $G_{SE}$, or $f'(u') = f'(u')$ and $f'(u)$ is a clique containing more than one vertex, in each of the cases (1) (2) and (3) above.

Suppose $(u, u')$ is an edge in the query graph. Since $f$ is a hyperembedding, either (i) $(f(u), f(u'))$ is an edge in $G_{SE}$, that is, $(h', f(u'))$ is an edge in $G_{SE}$, or (ii) $f(u) = f(u') = h'$ and $h'$ is a clique containing more than one vertex.
(1) Suppose condition (1) is satisfied. In case (i), since \( h \geq h' \), we know there is an edge \((h, f(u'))\) in \( G_{SE} \), that is, there is an edge \((f'(u), f'(u'))\) in \( G_{SE} \) (note that it is impossible for \( f(u') = h \) since we assume \( f \) has not mapped any query vertex to \( h \)). Therefore, \( f' \) is still a hyperembedding. In case (ii), \( f(u') = f(u) = h' \) and \( h' \) is a clique containing more than one vertex. This is impossible if condition (a) or (b) holds. If condition (c) holds, since \( f'(u) = h, f'(u') = h' \), we know there is an edge \((f'(u), f'(u'))\), hence \( f' \) is still a hyperembedding.

(2) Suppose condition (2) is satisfied. In case (i), if \( f(u') \neq h \), since \( h \geq h' \), there will be an edge \((h, f(u'))\) in \( G_{SE} \), that is, there is an edge \((f'(u), f'(u'))\), hence \( f' \) is still a hyperembedding. If \( f(u') = h \), that is, \( f'(u') = f'(u) \), since \( h \) is a clique, we also know \( f' \) is still a hyperembedding. In case (ii), \( f \) maps both \( u' \) and \( u \) to \( h' \) and \( h' \) is a clique containing more than one vertex. This is impossible if conditions (a) or (b) holds. If condition (c) holds, since \( f'(u) = h, f'(u') = h' \), we know there is an edge \((f'(u), f'(u'))\), hence \( f' \) is still a hyperembedding.

(3) Suppose condition (3) is satisfied. In case (i), if \( f(u') \neq h \), since \( h \geq h' \), there must be an edge \((h, f(u'))\) in \( G_{SE} \), that is, there is an edge \((f'(u), f'(u'))\). Note it is impossible for \( f(u') = h \), otherwise there must be an edge \((h, h')\) since \( f(u) = h' \), contradicting the assumption there is no edge between \( h \) and \( h' \). Therefore, \( f' \) is still a hyperembedding. Case (ii) is impossible because \( f \) maps no other vertex than \( u \) to \( h' \) (if condition (a) holds) or \( h' \) is an independent set (if condition (b) holds).

\[ \square \]

Intuitively, the main condition in each of (1),(2) and (3) in the proposition ensures mapping \( u \) to \( h \) will cause no conflicts, that is, either \( u \) is the only vertex mapped to \( h \), or no more than \( |h| \) vertices are mapped to \( h \) and \( h \) is a clique, or no more than \( |h| \) vertices are mapped to \( h \) and there are no edges between \( u \) and those vertices already mapped to \( h \); and each of the sub-conditions (a), (b) and (c) ensures that after mapping \( u \) to \( h \), any edge \((u, u')\) in the query graph still corresponds to an edge \((f'(u), f'(u'))\) in the compressed data graph.

Given query graph \( G_q \) and the compressed graph \( G_{SE} \), suppose \( u_1, u_2, \ldots, u_n \) are the query vertices, and the mapping order in the backtracking process is \( u_1, u_2, \ldots, u_n \). Whenever we have successfully mapped some vertices \( u_1, \ldots, u_k \) \((k \leq n)\) to some hypernodes, say \( h_1, h_2, \ldots, h_k \), we get a partial hyperembedding \( f_k = [(u_1, h_1), \ldots, (u_k, h_k)] \). Now suppose \( h \) and \( h' \) are two candidate nodes of query vertex \( u \equiv u_{k+1} \), and \( h \geq h' \). Then ac-
According to Proposition 6, if \( f_k \) can be extended to a full hyperembedding \( f \) that contains \((u, h')\), then it can be extended to a full hyperembedding that contains \((u, h)\), provided \( f, h, h' \) satisfy the conditions (1), or (2), or (3) in Proposition 6. It is important to note the difference between \( f_k \) and \( f \) here, for example, even if \( f_k \) may have not mapped any other query vertices than \( u \) to \( h' \) (or \( h \)), \( f \) may; even if \( f_k \) has mapped less than \(|h|\) query vertices to \( h \), \( f \) may have mapped \(|h|\) query vertices to \( h \); Therefore, to use the above proposition, we need to consider the cases where the extended full hyperembedding \( f \) also satisfies the conditions (1), or (2), or (3).

For convenience, in the following, we will use embedding to mean hyperembedding, and partial embedding to mean partial hyperembedding. We will also use \( \text{usedTimes}(f_k, h) \) to denote the number of times \( h \) has been used by partial embedding \( f_k \), that is, the number of query vertices mapped to \( h \) by \( f_k \), and use \( \text{unmapped}(f_k, u) \) to denote the number of query vertices that have the same label as \( u \) and have not been mapped by \( f_k \). The following corollary can be easily obtained from Proposition 6:

**Corollary 1:** Suppose \( h \) and \( h' \) are two hypernodes in the SE-Graph \( G_{SE} \) and \( h \succeq h' \). Suppose \( f_k \) is a partial embedding, and \( f_k \) has not mapped query vertex \( u \) to any node in \( G_{SE} \) yet. If \( f_k \) cannot be extended to a full embedding containing \((u, h)\), then it cannot be extended to a full embedding containing \((u, h')\), if any of the following conditions is satisfied:

1. \( \text{usedTimes}(f_k, h) = 0 \), \( \text{unmapped}(f_k, u) = 1 \), and one of the following conditions holds:
   - (a) \( \text{usedTimes}(f_k, h') = 0 \).
   - (b) \( h' \) is an independent set.
   - (c) there is an edge between \( h \) and \( h' \).

2. \( \text{usedTimes}(f_k, h) < |h| \), \( h \) is a clique, \( \text{usedTimes}(f_k, h) + \text{unmapped}(f_k, u) \leq |h| \), and one of the following conditions holds:
   - (b) \( h' \) is an independent set.
   - (c) there is an edge between \( h \) and \( h' \).

3. \( \text{usedTimes}(f_k, h) < |h| \), \( h \) is a clique, \( \text{usedTimes}(f_k, h') = 0 \), and \( \text{unmapped}(f_k, u) = 1 \).
(4) \( \text{usedTimes}(f_k, h) < |h| \), there is no edge between \( h \) and \( h' \), \( \text{usedTimes}(f_k, h') = 0 \), and \( \text{unmapped}(f_k, u) = 1 \).

(5) \( \text{usedTimes}(f_k, h) < |h| \), there is no edge between \( h \) and \( h' \), \( \text{usedTimes}(f_k, h) + \text{unmapped}(f_k, u) \leq |h| \), and \( h' \) is an independent set.

**Proof.** It is easy to verify that, under each of the above conditions, if \( f_k \) is extended to a full embedding \( f \) containing \((u, h')\), then \( f \), \( u \), \( h \) and \( h' \) will satisfy one of the conditions in Proposition 6, thus \( f_k \) would be able to be extended to a full embedding containing \((u, h)\), contradicting our assumption. For example, under condition (1), since \( \text{unmapped}(f_k, u) = 1 \) (that is, there is no other vertices than \( u \) that has the same label as \( u \) and has not been mapped by \( f_k \) ), and \( \text{usedTimes}(f_k, h) = 0 \), if \( f_k \) is extended to a full embedding \( f \) containing \((u, h')\), we will know \( f \) has not mapped any query vertex to \( h \). Moreover, if \( \text{usedTimes}(f_k, h') = 0 \), then \( f \) can not map any vertex other than \( u \) to \( h' \). Therefore \( f \), \( u \), \( h \) and \( h' \) will satisfy condition (1) of Proposition 6. Similarly, under conditions (2) or (3) (resp. conditions (4) or (5)), if \( f_k \) is extended to a full embedding \( f \) containing \((u, h')\), then \( f \), \( u \), \( h \) and \( h' \) will satisfy condition (2) (resp. condition (3)) of Proposition 6. □

Corollary 1 provides the conditions using which we can filter impossible candidates. There are two cases where \( f_k \) cannot be extended to a full embedding that contains \((u, h)\): the first case is when \( \text{isJoinable}(f_k, u, h) \) is \( \text{false} \). In this case, \( \text{isJoinable}(f_k, u, h') \) will be \( \text{false} \) as well, provided \( f_k, h, h' \) (if \( f_k \) is treated as \( f \) ) satisfy the conditions in Proposition 6 (note that the conditions in Corollary 1 implies the conditions in Proposition 6). Formally we have the proposition below:

**Proposition 7.** Suppose \( h \) and \( h' \) are two hypernodes in the SE-Graph \( G_{SE} \) and \( h \geq h' \). Suppose \( f_k = \{(u_1, h_1), \ldots, (u_k, h_k)\} \) is a partial hyperembedding. If \( \text{isJoinable}(f_k, u, h) = \text{false} \), then \( \text{isJoinable}(f_k, u, h') = \text{false} \), provided one of the following conditions holds:

1. \( \text{usedTimes}(f_k, h) = 0 \), and one of the following conditions holds:
   
   (a) \( \text{usedTimes}(f_k, h') = 0 \).
   
   (b) \( h' \) is an independent set.
   
   (c) there is an edge between \( h \) and \( h' \).

2. \( \text{usedTimes}(f_k, h) < |h| \), \( h \) is a clique, and one of the following conditions holds:
(a) \(\text{usedTimes}(f_k, h') = 0\).
(b) \(h'\) is an independent set.
(c) there is an edge between \(h\) and \(h'\).

(3) \(\text{usedTimes}(f_k, h) < |h|\), there is no edge between \(h\) and \(h'\), and one of the following conditions holds:

(a) \(\text{usedTimes}(f_k, h') = 0\).
(b) \(h'\) is an independent set.

Proof. Suppose \(\text{isJoinable}(f_k, u, h) = \text{false}\), that is, there exists \(u_i (i \in [1, k])\), such that 
\((u_i, u) \in E_q\), but either (A) \(h_i \neq h\) and there is no edge \((h_i, h)\) in \(G_{SE}\), or (B) \(h_i = h\) but \(h\) is not a clique.

Consider condition (1) in the proposition. Under this condition \(h \neq h_i\), otherwise \(\text{usedTimes}(f_k, h)\) will not be 0. Since \(h > h'\), if \(h' \neq h_i\), then there is no edge \((h_i, h')\) in \(G_{SE}\). Next we consider the possibility \(h' = h_i\).

If condition (a) \(\text{usedTimes}(f_k, h') = 0\) holds, then it is not possible to have \(h' = h_i\) otherwise \(\text{usedTimes}(f_k, h')\) will not be 0;

If condition (b) \(h'\) is an independent set holds, then if \(h' = h_i\), \(\text{isJoinable}(f_k, u, h') = \text{false}\); if condition (c) (there is an edge \((h, h')\)) holds, it will not be possible to have \(h' = h_i\), otherwise there will be an edge \((h, h_i)\), which contradicts (A).

Therefore, under condition (1), \(\text{isJoinable}(f_k, u, h') = \text{false}\).

Now consider condition (2) in the proposition. Under this condition \(\text{usedTimes}(f_k, h) < |h|\), and \(h\) is a clique. Thus it is not possible to have \(h_i = h\), otherwise there will be a contraction with (B). The remaining part of the proof under condition (2) is identical to that under condition (1).

Finally consider condition (3) in the proposition. Under this condition \(\text{usedTimes}(f_k, h) < |h|\), and there is no edge between \(h\) and \(h'\).

Since \(h > h'\), if \(h' \neq h_i\), and \(h \neq h_i\), then there is no edge \((h_i, h')\) in \(G_{SE}\) (because of (A)).

If \(h' \neq h_i\), and \(h = h_i\), then because there is no edge \((h, h')\), there is no edge \((h', h)\), hence \(\text{isJoinable}(f_k, u, h') = \text{false}\).

Under condition (3)(a) \(\text{usedTimes}(f_k, h') = 0\), it is not possible to have \(h' = h_i\), otherwise \(\text{usedTimes}(f_k, h') \neq 0\).

Under condition (3)(b) \(h'\) is an independent set, thus if \(h' = h_i\), then \(\text{isJoinable}(f_k, u, h') = \text{false}\). Therefore, under condition (3) \(\text{isJoinable}(f_k, u, h') = \text{false}\). \(\square\)
Because of Proposition 7, when \( isJoinable(f_k, u, h) \) is false, we will not check the descendants of \( h \).

The second case where \( f_k \) cannot be extended to a full embedding that contains \((u, h)\) is when \( isJoinable(f_k, u, h) \) is \text{true}, but after adding \((u, h)\) into the partial embedding, we find some remaining query vertex that cannot be mapped into any node in \( G_{SE} \). In this case, \( isJoinable(f_k, u, h') \) may or may not be true, but \( f_k \) cannot be extended to a full embedding provided \( f_k, h, \) and \( h' \) satisfy one of the conditions in Corollary 1.

To handle the second case, we use a boolean variable \( matchable(f, u, h) \) to represent whether partial embedding \( f \) can be extended into a full embedding that includes \((u, h)\). Initially this variable is set to \text{false} for all \( f, u, h \). After backtracking, if we can find an embedding that contains the pair \((u, h)\), we will set \( matchable(f, u, h) \) to \text{true}. The details are in Algorithm 4.

Given query vertex \( u \), we use \( \text{List}(u) \) to denote the list of candidate nodes sorted according to some topological order. For example, for the SC-Graph in Fig. 2.3, \((h_1, h_2, h_3, h_4, h_5)\) is such a list. For each node \( h \) in \( \text{List}(u) \), we use a boolean variable \( h\.masked(u, f) \) to denote whether the node is marked off for query vertex \( u \) and partial embedding \( f \). Note that a node that can be marked off for vertex \( u \) for one partial embedding may not be able to be marked off for another partial embedding. Initially, the variable is set to \text{false} for all candidates in \( \text{List}(u) \), meaning none of them is marked off.

**Algorithm 3: CandidateMask**

**Input:** \( u \) and \( \text{List}(u) \), \( h \), \( f \)

**Output:** updated \( \text{List}(u) \) with possibly more elements in the list marked off

1. for each \( h' \in \text{SC-Descendent}(h) \) s.t. \( h\.masked(u, f) = \text{false} \) do
2.   if any of the conditions (1),(2),(3),(4) or (5) in Corollary 1 holds then
3.     \( h\.masked(u, f) \) $\leftarrow$ \text{true}
Algorithm 3 describes the function CandidateMask() to mark off candidate nodes in $Clist(u)$. If $f$ cannot be extended to a full embedding containing $(u, h)$, the function will be called to check every descendant $h'$ of $h$ in the SC-Graph to see whether $u$ certainly should not be mapped to $h'$ (that is, $f$ cannot be extended to a full embedding containing $(u, h')$), and hence $h'$ can be marked off (by setting $h'.masked(u, f)$ to true), using the conditions in Corollary 1.

2.5.2 The algorithm for hyperembedding search

**Algorithm 4: HyperembeddingSearch**

<table>
<thead>
<tr>
<th>Input: $G_{SE}, G_{SC}$ and query vertices in the order $u_1, u_2, \ldots, u_n$</th>
<th>Output: All hyperembeddings of $G_q$ in $G_{SE}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f \leftarrow \emptyset$</td>
<td></td>
</tr>
<tr>
<td>for each $u \in V_q$ do</td>
<td></td>
</tr>
<tr>
<td>$Clist(u) \leftarrow \text{initializeCandidates}(G_q, G_{SE}, u)$</td>
<td></td>
</tr>
<tr>
<td>if $Clist(u) = \emptyset$ then</td>
<td></td>
</tr>
<tr>
<td>return</td>
<td></td>
</tr>
<tr>
<td>subgraphSearch($G_q, G_{SE}, f$)</td>
<td></td>
</tr>
</tbody>
</table>

**Subroutine** subgraphSearch($G_q, G_{SE}, f$)

| if $|f| = |V_q|$ then | report $f$ |
|---|---|
| else | |
| $u \leftarrow \text{nextQueryVertex}()$ | |
| refineCandidates($f, u, Clist(u)$) | |
| for each $h \in Clist(u)$ s.t. usedTimes($h, f$) < $|h|$ and $h$.masked($u, f$) do | |
| if isJoinable($f, u, h$) then | |
| updateState($f, u, h$) | |
| subgraphSearch($G_q, G_{SE}, f$) | |
| restoreState($f, u, h$) | |
| if matchable($f, u, h$) = false then | |
| CandidateMask($f, u, h$) | |

In this subsection, we present the detailed algorithm for finding all embeddings which makes use of the CandidateMask() function, as shown in Algorithm 4.

Algorithm 4 is similar to the generic framework except the following modifications. (A) The function isJoinable() is revised as in [46]. (B) There are two cases where the CandidateMask() function may be called, the first case is when isJoinable($f, u, h$) = false. However, as shown in Proposition 7, in this case if the conditions in Corollary 1 is satisfied, then isJoinable($f, u, h'$) will be false as well. Therefore, marking off descendent $h'$ does little help because the only benefit is saving the cost of checking isJoinable($f, u, h'$),
which is not more costly than checking the conditions in Corollary 1. Therefore in this case we will not call CandidateMask(). The second case is when isJoinable(f, u, h) = true but f cannot be extended to a full embedding that contains the pair (u, h). We use a boolean variable matchable(f, u, h) and the condition matchable(f, u, h) = false to denote the second case. The value matchable(f, u, h) can be tested as follows. Suppose there are n query vertices u_1, u_2, ..., u_n and the matching order is also u_1, u_2, ..., u_n. Each partial embedding f that has mapped u_i to h_i (for i = 1, ..., k) can be denoted by (h_1, ..., h_k). For the next query vertex u = u_{k+1} and its candidate h, the value matchable(f, u, h) will be true if and only if there is an embedding (h_1, ..., h_k, h, h_{k+2}, ..., h_n) which has been found by the time the search restored f to (h_1, ..., h_k) from (h_1, ..., h_k, h), that is, if we treat a partial embedding as a string, then h_1, ..., h_k, h is a substring of some full embedding that has already been found. In our implementation, we can store the embeddings found in a trie so that the condition matchable(f, u, h) can be searched for quickly. (C) For each candidate node h in Clist(u), we will check whether u can be mapped to h only when usedTimes(h, f) < |h| and h is not marked off for u, f (line 6).

Fig. 2.4 (a) and (b) Example graphs where nodes A and A_1 are all labeled A, and so on; (c) Embeddings stored as a trie

**Example 12.** Consider the query graph in Fig. 2.4 (a) and the SE-Graph in Fig. 2.4 (b). Note that A_1 ≥ A_2, B_1 ≥ B_3, B_2 ≥ B_3, C_1 ≥ C_2, and D_2 ≥ D_1. Suppose the matching order is A, B, C, D, and Clist(A) = [A_1, A_2], Clist(B) = [B_1, B_2, B_3], Clist(C) = [C_1, C_2], and Clist(D) = [D_2, D_1]. Initially we have f = φ. Using Algorithm 4, we first match A to A_1, B to B_1 and C to C_1, generating the partial embedding f_{1.1} = (A_1, B_1, C_1). Now we find isJoinable(f_{1.1}, D, D_1) = false for i = 1, 2, therefore we backtrack by calling restoreState(), and return to the partial embedding f_{1.1} = (A_1, B_1), and u = C and h = C_1. Since there is no full embedding found yet, we know matchable(f_{1.1}, C, C_1) = false,
hence $\text{CandidateMask}(f_1, C, C_1)$ is called, which will mark off $C_2$ for $f_1$ and $C$ because condition (1) (a) of Corollary 1 is true, that is, $C_2$.masked($f_1, C$) is set to true. Now since there are no more candidates for $C$, we backtrack again, setting $f$ to the partial embedding $f_1 = (A_1)$, $u$ to $B$ and $h$ to $B_1$. Since matchable($f_1, B, B_1$) = false, we call CandidateMask($f_1, B, B_1$), which will set $B_3$.masked($f_1, B$) to true, that is, $B_3$ is marked off for $f_1$ and $B$ (Since usedTimes($f_1, B_1$) = usedTimes($f_1, B_1$) = 0 and unmapped $(f_1, B)$ = 1, condition (1) (a) of Corollary 1 is satisfied).

Next we find isJoinable($f_1, B, B_2$) = true, thus generating partial embedding $f_{1.2} = (A_1, B_2)$. Then we find $C$ can be mapped to $C_1$, and then $D$ can be mapped to either $D_2$ or $D_1$, thus we get two full embeddings $(A_1, B_2, C_1, D_2)$ and $(A_1, B_2, C_1, D_1)$. Now we backtrack to $f_{1.2}$ and $u = C$, and we find $A_1, B_1, C_1$ is a prefix substring of some embeddings already found, therefore CandidateMask($f_{1.2}, C, C_1$) will not be called. We find $C$ can be mapped to $C_2$, then $D$ can be mapped to $D_2$ but not $D_1$. Hence we generate another embedding $(A_1, B_2, C_2, D_2)$. After that we backtrack to $f = \varnothing$, and map $A$ to $A_2$. Then we find $B$ can be mapped to $B_1$, $C$ can be mapped to $C_1$, generating partial embedding $f_{2.1, 1} = (A_2, B_1, C_1)$. Then we find $D$ cannot be mapped to any candidate, hence we backtrack to $f_{2.1}$. Now CandidateMask($f_{2.1}, C, C_3$) will be called, and $C_2$ will be marked off. After backtracking again, we return to $f_2 = (A_2)$, and will call CandidateMask($f_2, B, B_1$), which will mark off $B_3$ for $f_2$ and $B$. Then we find $B$ cannot be mapped to $B_2$, so the process will stop.

**Discussion.** Algorithm 4 differs from the general framework (Algorithm 1) only in that (1) there is additional filtering of the candidate nodes (line 6): for candidate node $h$, only when the conditions in line 6 are satisfied will $h$ be checked to see if it can be matched to the query vertex. This will generally reduce the number of candidates hence likely to make the search faster. (2) Each time the algorithm backtracks, it will test the condition matchable($f, u, h$) = false, and if the condition holds it will run the CandidateMask($f, u, h$) procedure (lines 11 and 12). This is an additional overhead. However, in practice, the savings of the additional filtering (line 6) usually significantly outweighs the overheads of lines 11 and 12.

Algorithm 4 and Algorithm 1 have the same worst-case complexity, which is exponential in the size of the data graph. Note that the conditions in line 6 of Algorithm 4 can be tested in constant time, the procedure CandidateMask($f, u, h$) can be done in $O(|E_{SC}|)$ where $E_{SC}$ is the edge set of $G_{SC}$ (testing the conditions in Corollary 1 can be done in constant time, and for a candidate node $h$, all of its descendants in $G_{SC}$ may need to be tested). The function matchable($f, u, h$) checks the trie of already-found hyperembeddings, thus its run time depends on the size of the trie.
2.5.3 Utilizing QDC and QDE

Since QDC and QDE relationships are relative to specific query vertices, we must find these relationships on-line. BoostIso builds a dynamic relationship table (DRT) to store these relationships for each query vertex. In this section, we will first present an algorithm for generating DRTs and then discuss how to integrate the information in the DRTs when searching for subgraph isomorphisms.

2.5.3.1 Building DRT

For each query vertex \( u \), we build a DRT, denoted \( DRT(u) \), which captures the QDC and QDE relations w.r.t \( u \). As shown in Table 2.1, \( DRT(u) \) is a table in which each tuple consists of four columns with the hypernode as the index. For the tuple indexed by \( h_i \), the second column, QDC-Children, contains the hypernodes strictly QD-contained by \( h_i \) and which are indexed in the table (i.e., which appear in the first column), that is, \( \{h| h_i \succ (G_{q,u}) h, h \text{ is indexed}\} \). The third column, NumOfQDC-Parents, contains the number of hypernodes indexed in the table that strictly QD-contain \( h_i \) w.r.t \( u \). The fourth column, QDE-List, contains the hypernodes QD-equivalent to \( h_i \), that is, \( \{h| h_i \equiv (G_{q,u}) h\} \).

Table 2.1 DRT for \((G_{q'}, u_1)\) and \(G_{sh}\) in Figure 2.2

<table>
<thead>
<tr>
<th>Node</th>
<th>QDC-Children</th>
<th>NumOfQDC-Parent</th>
<th>QDE-List</th>
</tr>
</thead>
<tbody>
<tr>
<td>(h_2)</td>
<td>({h_3, h_4})</td>
<td>0</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(h_3)</td>
<td>(\emptyset)</td>
<td>1</td>
<td>({h_4})</td>
</tr>
</tbody>
</table>

As the time consumed by building DRTs will be added to the total time of answering the query, we want to minimize the number of hypernodes indexed in the DRT, while still capture some important QDC and QDE relationships. To this end, we apply two filters to select the hypernodes to be indexed. For each query vertex \( u \) we have a candidate list \( C(u) \). (1) In the SE graph, we first apply a neighborhood label frequency filter (NLF filter) \(|?|\) to remove unpromising candidates from \( C(u) \). For each distinct label \( l \) of \( u \)’s neighbors, NLF filter excludes the candidate \( v \) if \( |adj(v, l)| \leq |adj(u, l)| \) where \( |adj(v, l)| \) is the number of \( v \)’s neighbors with label \( l \). (2) We remove the hypernodes whose SC graph in-degree is not 0. That is, we only consider the hypernodes not S-contained by any other hypernodes. Then, we build the DRT over the filtered candidate list. The second filter makes it possible to miss some QDC and QDE relationships. However, the trade-off is that we will spend less time building the DRTs. Note that we do not index hypernodes which are listed in the QDE-List of another indexed hypernode.
2.5 BoostIso

Example 13. Consider the query graph $G_q'$ and the hypergraph $G_{sh}$ in Figure 2.2. For query vertex $u_1$, $h_1$ will be filtered by the NLF filter, $h_5$ and $h_6$ will be filtered as they are $S$-contained by other hypernodes. Thus, only $h_2, h_3, h_4$ are left for the DRT. As $h_3 \equiv_{(G_q', u_1)} h_4$, we put $h_4$ into the QDE-List of $h_3$ and only index $h_3$. The final DRT for $u_1$ is shown in Table 2.1.

Algorithm 5: BUILD DRT

```
Input: A filtered candidate list $C(u)$
Output: DRT($u$), the DRT for $u$

1 for each $h \in C(u)$ do
2    for each $h_i \in C(u)$ and $h_i$ is after $h$ in $C(u)$ do
3        if $h \equiv_{(G_q', u)} h_i$ then
4            add $h_i$ to $h$ tuple’s QDE-List
5            remove $h_i$ from DRT and $C(u)$
6    for each $h \in C(u)$ do
7        for each $h_i \in C(u)$ and $h_i$ is after $h$ in $C(u)$ do
8            if $h \preceq_{(G_q', u)} h_i$ then
9                add $h_i$ to $h$’s QDC-Children
10               increase $h_i$’s NumOfQDC-Parent by 1
11            else if $h_i \succeq_{(G_q', u)} h$ then
12                add $h$ to $h_i$’s QDC-Children
13                increase $h_i$’s NumOfQDC-Parent by 1
14 return dynamic relationship table
```

Algorithm 5 presents the method to compute the DRT. For the candidate list $C(u)$, it compares every pair of vertices, and finds the QDE-List for each candidate $h_i$. Those nodes that are in the QDE-List of an indexed hypernode will be removed from $C(u)$ as they do not need to be indexed (Lines 1∼5). Then it scans each pair of the remaining hypernodes in $C(u)$ again and then updates the corresponding tuple of DRT according to the relationship (Lines 6∼13).

2.5.3.2 Integrating DRT into Hyperembedding Search

To exploit the QDC and QDE relationships captured in the DRTs, we need to slightly modify the search process. Specifically,

- In initializeCandidates, we first build the DRTs for each filtered candidate list $C(u)$ and then initialize $C(u)$ with those hypernodes indexed in DRT($u$) whose
NumOfQDC-Parents is 0. That is, we start with (a subset of) those hypernodes which are not strictly S-contained or QD-contained by other hypernodes.

- In isJoinable, we need to change Line 6 to usedTimes(h, f) leq |h| + \sum_{h_i \in \text{QDE-List}(h)} |h_i|, that is, we ensure that the number of times h is used in the partial hyperembedding is less than the total number of data vertices in h or in the hypernodes QD-equivalent to h.

- In CandidateMask, for each failed candidate h we not only need to test the SC children of h based on conditions (1) to (5) but also need to test the QDC children of h.

2.6 Experiments

This section presents our experiments. The purpose of the experiments is to evaluate (1) the size of the adapted graph and time to build them for real data sets, as well as the percentage of vertices having SC, SE, QDC and QDE relationships in realistic scenarios, and (2) the performance improvement of backtracking algorithms after integrating our approach.

2.6.1 Experimental Setup

Implementation and Running Environment. We implemented Algorithm 2 for computing adapted graphs. We used five backtracking algorithms to evaluate the performance of our approach: Ullmann [58], VF2 [8], QuickSI [53], Turbolso [24] and GQL (r=2) [25]. Ullmann is the first and canonical backtracking algorithm, QuickSI and GQL had the best overall performance among those compared in [34], and Turbolso is the state-of-the-art which outperforms all others, as reported in [24]. For each of these algorithms, we implemented the original version (denoted by their original name), a version that incorporates only the SC and SE relations (denoted by -SH), and a version that integrates the DRT as well (denoted by -Boosted). All of the algorithms were implemented in C++ with VC++ 2010 as our compiler. All the experiments were carried out under 64-bit Windows 7 on a machine with an Intel 3GHz CPU and 4GB memory.

Datasets. We used six real datasets in our experiments: Human, Youtube, Yeast, Email, Wordnet and DBLP. Human and Yeast were used in [24][34], Wordnet was used in [56]. We obtained the Youtube, Email and DBLP datasets from Stanford Large Network Dataset.
2.6 Experiments

Collection 1. As no label information is available for Email, we randomly assigned a label for each vertex from a label set of 130. The profiles of the datasets are given in Table 4.2.

Table 2.2 Profiles of datasets used for BoostIso experiments

| Dataset | |V| | |E| | |Σ| | Avg. degree |
|---------|----------------|---------|---------|---------|---------|---------|---------|
| Human   | 4675           | 86282   | 90      | 36.82   |
| Youtube | 1.1M           | 2.9M    | 1       | 5.26    |
| Yeast   | 3112           | 12915   | 184     | 8.05    |
| Email   | 36692          | 183831  | 130     | 10.01   |
| Wordnet | 82670          | 133445  | 5       | 3.28    |
| DBLP    | 317080         | 1.04M   | 1       | 6.62    |

Query Sets. We generated all the query graphs by randomly selecting connected subgraphs of the data graphs. This will ensure every query has at least one embedding in the data graph. The query graph size (number of edges) ranges from 1 to 10. Each query set contains ten query files and each query file contains 1000 query graphs with the same number of edges.

2.6.2 $G_{sh}$ Statistics on Real Datasets

Measurements. Given data graph $G = \{V, E, \Sigma, L\}$ and its adapted graph $G_{sh} = \{V_{sh}, E_{se}, E_{sc}, \Sigma, L_{sh}\}$, we use $R_{sh} = |G_{sh}|/|G|$ to measure the size of $G_{sh}$ over $G$ where $|G_{sh}| = |V_{sh}| + |E_{se}| + |E_{sc}|$ and $G = |V| + |E|$. We use $R_{se} = |V_{sh}|/|V|$ to measure the vertex compression ratio by the SE relationships, and $R_{sc} = |V'_{sh}|/|V_{sh}|$ to measure the percentage of hypernodes that are not S-contained by other hypernodes, where $V'_{sh}$ is the set of hypernodes whose SC indegree is 0. To roughly estimate the frequency of QDE and QDC relationships between hypernodes, we randomly select 5 labels from the label set of $G_{sh}$ when testing the relationships between a pair of hypernodes, so that neighbours with a label different from the 5 selected ones will be ignored. We define $R_{qde} = |V_h|/|V_{sh}|$ where $V_h$ is the set of hypernodes left after merging QDE hypernodes, and define $R_{qdc} = |V''_h|/|V'_h|$ where $V'_h$ is the set of nodes in $V_h$ whose SC indegree is 0, and $V''_h$ is the set of nodes in $V_h$ whose QDC indegree is 0.

Statistics for Real Datasets. As shown in Table 2.3, the adapted graphs for Human is smaller than the original data graph. For Human, the data vertices are reduced to 46.9% by SE relationships, and it can be further reduced by another 54.5% when QDE is taken

1http://snap.stanford.edu/data/
into consideration. Additionally, 56.8% of hypernodes for Human are S-contained by other hypernodes, and another 36.1% of hypernodes are QD-contained by other hypernodes whose SC indegree is 0. Because Youtube and DBLP have only one label, we did not compute their $R_{qde}$ and $R_{qdc}$. It is worth noting that Human and DBLP have a low ratio of $R_{sc}$, which could give the subgraph matching process a highly optimized candidate vertex matching order. For the other datasets, the size of the adapted graph is slightly larger than that of the original graph because of the edge set $E_{sc}$. However, the vertex compression ration of Youtube and Wordnet reached 62.6% and 68% respectively, and for Yeast and Email, after integrating the QDE relationships, the number of hypernodes can be further reduced significantly.

Table 2.3 Statistics of $G_{sh}$ on real datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>$T(s)$</th>
<th>$R_{sh}$</th>
<th>$R_{se}$</th>
<th>$R_{sc}$</th>
<th>$R_{qde}$</th>
<th>$R_{qdc}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human</td>
<td>13.2</td>
<td>0.23</td>
<td>46.9%</td>
<td>43.2%</td>
<td>45.5%</td>
<td>63.9%</td>
</tr>
<tr>
<td>Youtube</td>
<td>612</td>
<td>1.1</td>
<td>62.6%</td>
<td>58.0%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Yeast</td>
<td>3.7</td>
<td>1.07</td>
<td>95%</td>
<td>79.5%</td>
<td>59.7%</td>
<td>91.1%</td>
</tr>
<tr>
<td>Email</td>
<td>78.1</td>
<td>1.06</td>
<td>93%</td>
<td>91%</td>
<td>64.3%</td>
<td>80.8%</td>
</tr>
<tr>
<td>Wordnet</td>
<td>93</td>
<td>1.01</td>
<td>68%</td>
<td>49%</td>
<td>97%</td>
<td>70.9%</td>
</tr>
<tr>
<td>DBLP</td>
<td>227</td>
<td>1.13</td>
<td>74.3%</td>
<td>38.5%</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Effect of Label Set Size.** To evaluate the effect of label set size on the frequency of the four types of relationships, we used the Human and Email datasets to generate 22 new datasets by randomly re-assigning a label for each vertex from a label set of size 1, 10, 20, …, and 100 respectively.

As we can see in Figure 2.5, both $R_{se}$ and $R_{sc}$ show an increasing trend when we increase the size of the label set. However, the ratios of query-dependent relationships keep steady, with only a slight increment for QDC. This is because the query dependent relationships ignore all the labels not related to the query graph, thus they are little affected by the number of labels of the data graph.

### 2.6.3 Building Time and Scalability

The first column of Table 2.3 shows the time for building the adapted graph. As can be seen, all of the adapted graphs were built in a very short time for the real data sets.

To test the scalability for building time, we generated five synthetic datasets using the graph generator given by GQL. The number of vertices ranges from 0.1M to 2M with edges ranging from 1 million to 20 million. The average degree is 20. The number of
2.6 Experiments

Table 2.4 shows the building time for each synthetic dataset. All the datasets can be built in a reasonable time with less than 20 minutes for the largest dataset. As we increase the number of vertices, the building time nearly follows a linear trend (note that the average degree keeps steady, so the building time largely depend on $|V|$). This shows the good scalability in terms of time cost for building the adapted graph.

<table>
<thead>
<tr>
<th>Vertex number</th>
<th>0.1M</th>
<th>0.5M</th>
<th>1M</th>
<th>1.5M</th>
<th>2M</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(min)$</td>
<td>0.5</td>
<td>3</td>
<td>7.3</td>
<td>11.7</td>
<td>19.28</td>
</tr>
</tbody>
</table>

2.6.4 Efficiency of Query Processing

Measurements. As in [24][34], we measured the performance of an algorithm in two aspects: time elapsed and number of recursive calls. All the algorithms terminated once 1000 embeddings were found for a single query. Because of page limits, we only presented the average number of recursive calls of the ten query sets.

Experiments on Human. Human is a graph with a large average degree where each query vertex has a large number of candidate vertices. As shown in Figure 2.6(a)∼(e), Ullmann, VF2, QuickSI, and GQL show an exponential increase as the query size increases. Especially for large queries, these algorithms take more than $10^5$(msec) to compute one single query. With the integration of BoostIso, all of UllmannBoosted, VF2Boosted, QuickSI-Boosted and GQLBoosted behave much better. All of the queries can be answered within $10^2$(msec). The overall improvement is 3 orders of magnitude. TurboIso is the most state-
Fig. 2.6 BoostIso Experiment results over Human
of-the-art subgraph isomorphism algorithm, which shows steady but less drastic increment with the growth in query size. However, with the integration of our approach, it achieves a much better performance. For small queries, the improvement by TurboIso-Boosted is not significant while for large queries, the performance can be 19 times faster. Even without the consideration of QDE and QDC, all SH-algorithms have significant improvements in overall performance because Human has a large number of SE vertices.

As shown in Figure 2.6(f), the average number of recursive calls of Boosted algorithms are significantly less than the original algorithms by up to 4 orders of magnitude. Compared with TurboIso, TurboIsoBoosted has a decrement from 3914 to 112 on average. Again, both the SH-algorithms and the Boosted-algorithms perform much better than the original algorithms, and the Boosted-algorithms perform better than the SH-algorithms.

**Experiments on Yeast.** Yeast is a sparse graph with a large number of vertex labels. The time consumed and average number of recursive calls of all the algorithms are much less than that for Human. As shown in Figure 2.7(a)∼(e), as the query size increases, all the algorithms first experience a linear and then a very sharp increase in the average elapsed time, while the growth rates of all Boosted-algorithms are far lower than that of the original ones. All SH-algorithms achieved minor improvements, and this could be because Yeast has very few SE vertices (see Table 4.2 and Table 2.3). As for Ullmann and VF2, the Boosted-algorithms are 10 times faster on average. For QuickSI, the improvement is about 5 times faster. For GQL, it is about 9 times faster. While for TurboIso, there are slight improvements when the query graph has less than 6 edges while it can be 2 times faster when the query size increases to 10. The figure shows a trend that with the query graph growing larger, TurboIsoBoosted has a larger improvement over TurboIso.

In Figure 2.7(f), for Ullmann and VF2, the avg. recursive calls of the Boosted-algorithms are more than 10 times less than that of the original algorithms and the SH-algorithms. Even for QuickSI, GQL and TurboIso whose original algorithms have achieved a good performance over Yeast, the avg. number of recursive calls is reduced by 2896, and 880 and 349 respectively.

**Experiments on Wordnet.** Wordnet is a much larger and sparser graph than Human and Yeast, thus its computing time is much longer. Also, Wordnet has only 5 labels which results in a large number of candidates for a query vertex. As shown in Figure 2.7(a)∼(e), all of the SH algorithms achieved a performance which is not much worse its Boosted Algorithms, this is because there are not many QDE and QDC relationships among the hy-
Exploiting Vertex Relationships

Fig. 2.7 BoostIso Experiment results over Yeast
### 2.6 Experiments

#### Fig. 2.8 Experiment results over Wordnet

<table>
<thead>
<tr>
<th>Method</th>
<th>Original</th>
<th>SH</th>
<th>Boosted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. recursive calls</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ullmann</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VF2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QuickSI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TurboIso</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GQL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Subgraphs over Wordnet

- **(a) Ullmann over Wordnet**
- **(b) VF2 over Wordnet**
- **(c) QuickSI over Wordnet**
- **(d) TurboIso over Wordnet**
- **(e) GQL over Wordnet**
- **(f) Avg. calls over Wordnet**
pernodes. For VF2, QuickSI and GQL, the SH and Boosted algorithms are 10 times faster than the original ones on average. For Ullmann, Boosted Algorithm performs much better than the original and SH algorithms. This is because Ullmann has not defined any matching order or pruning rules which leads to big time differences even for a small difference of the search space size. For TurboIso, the SH and Boosted Algorithms are 2 times faster than the original one on average. The avg. recursive calls, shown in Figure 2.8(f), is consistent with the elapsed time with similar performances for the SH and Boosted algorithms, while both of them are less than that of the original one.

2.7 Conclusion

In this chapter, we presented an approach, BoostIso, for speeding-up subgraph isomorphism search. Our approach differs from previous algorithms in that it utilizes the relationships between data vertices, and it can be integrated into all existing backtracking algorithms. Our extensive experiments with real and synthetic datasets demonstrated that, with the integration of our approach, most existing subgraph isomorphism algorithms can be speeded up significantly.

To apply our approach in practice, efficient maintenance of the adapted graphs is important. Intuitively, the adapted graphs can be incrementally maintained efficiently because a vertex S-contains another only if the two vertices are connected by a 1-edge or 2-edge path. We will discuss this problem in detail in future research.
Chapter 3

Multi-Query Optimization

In this chapter, we present our approach of multi-query optimization of subgraph isomorphism. We provide an overview of our approach in Section 3.3. The subsequent sections present the details. Section 3.4 presents our methods for common subgraph computation. Section 3.5 presents the method to compute a good query execution order. Section 3.6 presents the data structure for caching intermediate results. Section 3.7 presents the strategies for computing the query answers utilizing the cached intermediate results. Section 3.8 reports our experiments. Section 3.9 concludes this chapter.

3.1 Motivation and Challenges

Motivation Existing work on subgraph isomorphism search mainly focuses on a-query-at-a-time approaches: optimizing and answering each query separately. However, in some scenarios multiple queries can be processed as a batch. For example, in semantic web applications, multiple SPARQL queries often need to be processed together [33]. In detecting criminal networks in social graphs [41] or finding special structures in biological networks [45], a user may be interested in finding subgraphs isomorphic to any one in a collection of query graphs, or she may not know the exact structure of the query graph, and submit a group of possible queries instead. In such cases, multiple query graphs can be evaluated at the same time. Recently, graph functional dependencies [19], keys [12] and association rules [18] are all defined based on subgraph isomorphism. We envisage that batch subgraph isomorphism search will be useful in checking the satisfaction of a set of graph functional dependencies and/or keys, as well as in verifying collections of graph association rules.
Motivated by the above, we study the problem of multiple query optimization (MQO) for subgraph isomorphism search in this paper. Given a data graph $G$ and a set of query graphs $Q = \{q_1, \ldots, q_n\}$, our aim is to efficiently find all subgraphs of $G$ that are isomorphic to one of the query graphs. Specifically, (1) when there are significant overlaps (i.e., common subgraphs) among the query graphs, we want to make maximum use of the intermediate results of these common subgraphs to speed-up the process, so that the overall processing time is significantly shorter than if we process the queries in $Q$ one by one sequentially. (2) When there are little or no overlaps among the query graphs, we want to be able to detect it quickly so that the total processing time will be about the same as sequential processing.

**Challenges** Although the basic idea of MQO is simple, there are some challenging technical issues. *First*, how to identify overlaps among query graphs that are worthwhile to extract? Since detecting and extracting common subgraphs takes time, we need to ensure the benefits of extracting and evaluating the common subgraphs outweigh the overhead. *Second*, how to compute an optimal processing order that enable us to effectively share the intermediate results? *Third*, how should we store the intermediate results, i.e., the matchings of the common subgraphs, to ensure a good trade-off between memory usage and the time to retrieve these intermediate results? *Last*, how can we integrate the intermediate results into current state-of-the-art subgraph isomorphism search algorithms to maximize the performance? We will address these issues in this paper and provide an effective solution to the multi-query processing problem in the context of subgraph isomorphism search. To the best of our knowledge, our work is the first on multi-query processing for subgraph isomorphism search over general graphs.

### 3.2 Related work

**Subgraph Isomorphism** The problem of subgraph isomorphism search has been investigated for many years. The algorithms can be divided into two categories: (1) Given a graph database consisting of many small data graphs, find all of the data graphs containing a given query graph. (2) Given a query graph, retrieve all of the isomorphic subgraphs of a single large data graph.

Algorithms falling into the second category include Ullmann [58], VF2 [8], GraphQL [25], TurboIso [24], QuickSI [53] and many others [56, 62]. Most of these algorithms follow the framework of Ullmann, with improved pruning rules and matching orders. An
3.2 Related work

experimental comparison was given in [34]. A graph compression-based approach was introduced recently in [46]. These algorithms focus on a single query, they do not consider batch processing of multiple queries. Our work belongs to the second category. However, different from previous work, we treat multiple queries as a batch and process them together. Our solution can be seamlessly integrated with the single-query algorithms.

**Multi-Query Optimization (MQO)** MQO has been well studied for relational databases [50, 51]. Most works on relational MQO assume the existence of a cost model based on statistics about the data, and search for a globally optimal access plan among all possible combinations of access plans for the individual queries. Each access plan consists of a sequence of tasks, and some tasks can be shared by multiple queries. These methods for relational MQO cannot be directly used for MQO for subgraph isomorphism search, since we do not assume the existence of statistics or indexes on the data graph, and some relational optimization strategies (e.g., pushing selection and projection) are inapplicable to subgraph isomorphism search. Methods for identifying common relational subexpressions (e.g.,[21]) are also difficult or inefficient to be adapted for common subgraph computation, just as it is inefficient to evaluate graph pattern queries by converting them into relational queries [25]. MQO has also been studied for semi-structured data [5, 26]. For similar reasons, these methods are difficult to be adapted for our problem.

More recently, [33] studies MQO for SPARQL queries over RDF graphs, which is the most closely-related work to ours. The approach of [33] works as follows. Each SPARQL query is represented as a graph pattern (GP) which is a set of triple patterns. Given a batch of graph patterns, (a) it uses common predicates (i.e., edge labels), Jaccard similarity and the $k$-means method to cluster the GPs into disjoint groups. For the GPs in each group, it uses bottom-up hierarchical clustering and a selectivity-based cost model to further divide them into finer groups, such that only queries in the same finer group are likely to benefit from batch optimization. (b) For each finer group $M_i$, it rewrites the queries $P_1, \ldots, P_m$ within $M_i$ into a single query consisting of the common GP $P$ and the optional GPs $P_1 - P, \ldots, P_m - P$. This rewritten query is then evaluated using a SPARQL engine that has the ability to answer queries with optional conditions.

Our work and [33] are different in the following aspects: (1) We do not assume the existence of statistics about the data graph and a cost model (like most single-query isomorphism search algorithms). The only heuristic for us is that the larger the MCS, the more beneficial to process the graphs together. (2) [33] uses edge labels, Jaccard similarity and the $k$-means method to cluster the GPs into disjoint groups. The problems with
this approach include: (a) The $k$-means algorithm assumes a given number of clusters, while in practice this number is difficult to guess ([33] uses $|Q|/40$ without explanation). (b) Using the number of common edge-labels as the basis to measure similarity can put two graphs that share many disconnected single edges into the same group, while putting two groups sharing a few number of connected common edges into different groups. In contrast to [33], we use a novel TLS to ensure two graphs in the same group share a common connected subgraph of at least 2 edges, and we do not need to pre-give the number of groups. (3) [33] mainly focuses on grouping the queries and extracting common sub-patterns. It leaves the task of result caching to the RDF engine, and relies on the RDF engine’s ability to answer optional queries to reuse the cached results. In contrast, we devote significant attention to techniques for result caching and efficient algorithms for reusing the cached results.

**Result Caching** In-memory result caching is used in some existing works on subgraph isomorphism search. For example, TurboIso [46] stores the embeddings of a path in-memory as a collection of lists, which is similar to the trivial table structure discussed in our Section 3.6. A more recent work [4] uses a data structure called compact path index (CPI) to store the potential embeddings of a spanning tree of the query graph. The overall structure of a CPI can be seen as a tree with the same shape as the spanning tree, consisting of nodes which contain highly-likely candidates of the corresponding query vertices. Since the cached embeddings is that of a tree, the CPI is similar to the fully compressed data structure discussed in our Section 3.6. In contrast, we store the embeddings of a subgraph which is usually not a tree, and our data structure is based on sophisticated graph partition in order to balance memory usage and cached result retrieval time. QU- BLE [27] is an interactive system where a user can dynamically add edges to the query. The system decomposes the large data graph into many small graphlets. To make query processing fast, it builds indexes for frequent fragments and small infrequent fragments. Based on the indexes, an in-memory structure called G-SPIG is used to record the IDs of the graphlets that may contain the query. Different from [27], we store the actual embeddings in main memory, and our data structure is very different from that used in QUBLE.

### 3.3 Overview of Our Approach

Given a data graph $G$ and a set of query graphs $Q = \{q_1, \ldots, q_n\}$, our problem is to efficiently find all embeddings in $G$ of the query graphs. An overview of our solution is given in Algorithm 6.
A crucial first step is to detect subgraphs shared by the queries. We organize the common subgraphs and the original queries in a structure called pattern containment map (PCM) (Line 1). Based on the PCM, we compute a query execution order (Line 2) which is an order for processing the queries in the PCM. The purpose of the execution order is to guarantee the results of the common subgraphs can be reused, and to enable efficient memory use by releasing non-longer useful cached results as early as possible. For each query graph in the PCM, we evaluate it using a framework revised from single-query isomorphism search that can utilize the cached results of its PCM parents (Line 6). For each parent $q_j$ of $q_i$, we release the cache for $q_j$ if all $q_j$’s children are processed (Lines 7 ~ 9). We cache the results of $q_i$ if it has unprocessed children in the PCM (Line 12,13). The results are cached in main memory for fast retrieval. To balance cache memory usage and result retrieval time, we design a special data structure to store the results. The key idea is to partially compress the result set by dividing the query vertices into disjoint lists based on a special type of graph partition, and if several results map a list of query vertices into the same list of data vertices, we store this list of data vertices only once.

### 3.4 Detecting Common Subgraphs

In this section, we present the process of detecting common subgraphs for the query set. We are only interested in MCSs that are likely to help in reducing the overall query processing time. We do not consider single-edge MCSs as such MCSs are considered not

---

**Algorithm 6: MQO<sub>subiso</sub>**

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PCM = DETECTCOMMONSUBGRAPH(Q)</td>
</tr>
<tr>
<td>2</td>
<td>EOrder = QUERYEXECUTIONORDER(PCM)</td>
</tr>
<tr>
<td>3</td>
<td>CACHE = ∅</td>
</tr>
<tr>
<td>4</td>
<td>for each $q_i$ ∈ EOrder do</td>
</tr>
<tr>
<td>5</td>
<td>if $q_i$ has PCM parents then</td>
</tr>
<tr>
<td>6</td>
<td>$R = SUBISOSEARCH&lt;sub&gt;Mqo&lt;/sub&gt;(PCM, Cache, $q_i$, G)</td>
</tr>
<tr>
<td>7</td>
<td>for each parent $q_j$ of $q_i$ do</td>
</tr>
<tr>
<td>8</td>
<td>if all $q_j$’s children are processed then</td>
</tr>
<tr>
<td>9</td>
<td>Clear CACHE($q_j$)</td>
</tr>
<tr>
<td>10</td>
<td>else</td>
</tr>
<tr>
<td>11</td>
<td>$R = SUBISOSEARCH(q_i, G)$</td>
</tr>
<tr>
<td>12</td>
<td>if $q_i$ has unprocessed children then</td>
</tr>
<tr>
<td>13</td>
<td>Add $R$ to CACHE($q_i$)</td>
</tr>
</tbody>
</table>
very helpful, and there can be too many of them. In the following, when we say MCS, we mean an MCS with 2 or more edges.

The naive strategy to compute the MCSs of all pairs of queries is impractical when the query set is large, and a random query pair often do not share any MCSs. To address these problems, we design a grouping factor based on the concept of tri-vertex label sequence. We use it to divide the queries into groups, where queries within the same group are likely to have an MCS. Then we will divide the queries in the same group into random pairs. For each pair we will compute their MCS. We treat the MCSs as new queries and repeat the process until we get all the MCSs for the whole group. These MCSs and the original queries will be organized into a hierarchy called a pattern containment map (PCM).

The following subsections give the details.

### 3.4.1 Grouping Factor

We first define tri-vertex label sequence.

**Definition 9** (Tri-Vertex Label Sequence). Given a pair of connected edges \((v_i, v_j)\) and \((v_j, v_k)\) of a graph \(q\), assuming \(L(v_i) \leq L(v_k)\), we call the label sequence \(L(v_i)-L(v_j)-L(v_k)\) a Tri-Vertex Label Sequence (TLS), and \((v_i, v_j, v_k)\) an instance of the TLS in \(q\).

We will use \(\text{TLS}(q)\) to denote the set of all TLSs in \(q\). It is easy to verify \(|\text{TLS}(q)| \leq \frac{1}{2}(|V_q| \times d_q(d_q - 1))\), where \(V_q\) is the vertex set of \(q\), and \(d_q\) is the maximum vertex degree in \(q\). Each TLS may have multiple instances in the graph. Two instances of the same or different TLSs are connected if they share common vertices. Multiple connected instances form a connected subgraph, referred to as an instance subgraph hereafter. Given a subset of \(\text{TLS}(q)\), there may be multiple instance subgraphs corresponding to it. For example, consider the subset \{\((A-B-C), (A-C-B), (B-A-C), (D-C-E)\)\} of TLSs of \(q_4\) in Figure 3.1. The instances of the first three TLSs form an instance subgraph, and the instance of \((D-C-E)\) forms another.
Intuitively, if two graphs share a connected common subgraph of 2 or more edges, they must share a TLS, and the more TLSs they have in common, the more overlap they have. Furthermore, if two graphs share a large connected common subgraph, they must share common TLSs whose instances in each of them form a large instance subgraph. Based on this observation, we define a grouping factor between a pair of query graphs.

Before that, we need the following notation.

Let \( t \) be a TLS of graph \( q \). We call the number of times \( t \) occurs in \( q \), i.e., the number of instances of \( t \) in \( q \), the frequency of \( t \) in \( q \), and denote it by \( t . \text{freq} (q) \). The sum of the frequencies of all TLSs in \( q \) is denoted \( \text{TLS size} (q) \).

Given two graphs \( q_i \) and \( q_j \), we use \( \text{TLS}(q_i, q_j) \) to denote the set of all TLSs shared by \( q_i \) and \( q_j \). That is, \( \text{TLS}(q_i, q_j) = \text{TLS}(q_i) \cap \text{TLS}(q_j) \). Consider the graphs \( q_3 \) and \( q_4 \) in Figure 3.1. \( \text{TLS}(q_3, q_4) = \{(A-B-C), (A-C-B), (B-A-C), (D-C-E)\} \). We use \( \text{LI}(q_i, \text{TLS}(q_i, q_j)) \) to denote the number of instances in the largest instance subgraph of \( q_i \) corresponding to the TLSs in \( \text{TLS}(q_i, q_j) \).

For example, for \( q_3 \) and \( q_4 \) in Figure 3.1, \( \text{LI}(q_4, \text{TLS}(q_3, q_4)) = 3 \) and \( \text{LI}(q_3, \text{TLS}(q_3, q_4)) = 4 \).

**Definition 10 (Grouping factor).** The grouping factor between two query graphs \( q_i \) and \( q_j \), denoted \( GF(q_i, q_j) \), is defined as

\[
GF(q_i, q_j) = \frac{\min(\text{LI}(q_i, \text{TLS}(q_i, q_j)), \text{LI}(q_j, \text{TLS}(q_i, q_j)))}{\min(\text{TLS}(q_i).\text{size}, \text{TLS}(q_j).\text{size})}
\]

The grouping factor has the following properties:

1. \( 0 \leq GF(q_i, q_j) = GF(q_j, q_i) \leq 1 \).
2. If \( q_i \preceq q_j \), \( GF(q_i, q_j) = 1 \).
3. If \( q_i \) and \( q_j \) do not have an MCS, \( GF(q_i, q_j) = 0 \).

We will use the \( GF \) to divide the query graphs into groups. Queries will be put into the same group if and only if their pairwise grouping factor are all above a specified threshold, and this threshold can be used as a parameter to control the group size so as to balance the PCM building time and the number of MCSs detected.

Once we have divided the queries into groups, we can compute multiple levels of MCSs, and organize them into a PCM, as discussed in the next subsection.

### 3.4.2 Pattern Containment Map

We give a formal definition of pattern containment map first.
**Definition 11** (Pattern Containment Map). *Given a query set \( Q = \{q_1, \ldots , q_n\} \), a pattern containment map is a directed graph \( \mathcal{P}_Q = \{V_{pcm}, E_{pcm}\} \) where each \( q_{pcm} \in V_{pcm} \) represents an undirected vertex-labelled graph, such that

1. \( \forall q_i \in Q, \) there is a \( q_{pcm} \in V_{pcm} \) such that \( q_i \cong q_{pcm}; \)

2. \( \forall q \in V_{pcm}, \) there exists \( q_i \in Q \) such that \( q \preceq q_i; \)

3. There are no two nodes \( q \) and \( q' \) in \( V_{pcm} \) such that \( q' \equiv q; \)

4. A directed edge \( (q, q') \in E_{pcm} \) exists only if \( q \preceq q' \) and there is no \( q'' \in V_{pcm} \) such that \( q' \not\equiv q'', \) \( q \preceq q'' \), and \( q'' \preceq q'. \)

---

**Algorithm 7: TLSGroupMatrix**

- **Input:** A query set \( Q = \{q_1, \ldots , q_n\} \)
- **Output:** TLSGroupMatrix \( M \) of \( Q \), threshold \( \Phi \)

1. initialize \( M \) and set all cells to 0
2. for each \( q \in Q \) do
   3. \( H(q) \leftarrow \text{ComputeTLSSet}(q) \)
4. for each pair \( q_i, q_j \in Q \) do
   5. \( \text{Compute} \ G(q_i, q_j) \)
   6. if \( G(q_i, q_j) > \Phi \) then
      7. \( M[q_i][q_j] \leftarrow 1 \)
8. return \( M \)

Intuitively, a PCM is a structure that represents the subgraph isomorphic relationships among a set of graphs. Each node in the PCM is either a query in \( Q \), or a subgraph of some other nodes in the PCM. Each query in \( Q \) either appears as a node in the PCM, or is isomorphic to a node in the PCM (if some queries in \( Q \) are isomorphic to each other, only one of them will appear in the PCM). Conditions (3) and (4) in the above definition ensure that the PCM is a DAG. Figure 3.1(d) shows an example PCM for the queries in Figure 3.1(a).

We can now present the process for computing the MCSs and the PCM for these queries.

We first build a matrix based on the grouping factor of each query pair. The process of building the group matrix \( M \) is given in Algorithm 7. For each query \( q \), we use a hashmap \( H(q) \) to contain its TLS set with each TLS as key and its corresponding instances as value (Line 2-3). Then for each query pair \( q_i, q_j \), we compute its grouping factor using Equation (3.1). If the grouping factor is larger than the given threshold \( \Phi \), we mark the element corresponding to \( q_i \) and \( q_j \) as 1 in the matrix.
Algorithm 8: BUILD PCM

**Input:** Query set $Q$, TLS group matrix $M$

**Output:** Pattern Containment Map PCM

1. $\mathcal{G}_r \leftarrow \text{cliqueDetection}(M)$
2. **for each** group $g \in \mathcal{G}_r$ **do**
   3. $\text{NextLevelGroup} \leftarrow \emptyset$
   4. create a PCM node for each query in $g$
   5. divide the queries in $g$ into disjoint pairs
   6. **for each** random pair $q, q'$ **do**
   7. $Q_{mcs} \leftarrow \text{computeMCSs}(q, q')$
   8. **for each** $q_{mcs} \in Q_{mcs}$ **do**
      9. if $q_{mcs} \cong q$ (assuming $|q| \leq |q'|$) then
         10. add $q$ into $\text{NextLevelGroup}$
         11. add $q'$ into $q.\text{children}$
      else
         12. add $q_{mcs}$ into $\text{NextLevelGroup}$
         13. add both $q, q'$ into $q_{mcs}.\text{children}$
   14. if $|\text{NextLevelGroup}| > 1$ then
      15. $g \leftarrow \text{NextLevelGroup}$
      16. Repeat from line 3
   17. $\text{mergeIsomorphicNodes}(\text{PCM})$
   18. $\text{transitiveReduction}(\text{PCM})$
19. **return** PCM

With the query set and the matrix as input, Algorithm 8 proceeds to compute the final PCM. It runs a clique detection process over the matrix where each “1” element represents an edge (Line 1). Each clique detected represents a group of queries. Within each group, we create a PCM node for each query (Line 4). We divide the nodes in each group into random disjoint pairs (we put the last query into $\text{NextLevelGroup}$ if there are odd number of queries). Lines 6 to 14 compute the MCSs for each pair, put the MCSs into $\text{NextLevelGroup}$, and add corresponding PCM edges (represented as a children list). The same process is repeated for queries in $\text{NextLevelGroup}$ until there is no more than one query left (Line 15~17). Finally we do a merge for the isomorphic PCM nodes (Line 18) and a transitive reduction (Line 19) to remove redundant edges.

**Example 14.** For the queries in Figure 3.1(a), the grouping factors are shown in Figure 3.1(b). After applying the threshold $\Phi=0.35$, the group matrix is shown in Figure 3.1(c). Two cliques can be detected from the matrix, which are $(q_1, q_3, q_4)$ and $(q_3, q_4, q_5)$. Consider the first group and take $q_1$ and $q_3$ as a pair, $q_1$ is the MCS. A PCM edge $(q_1, q_3)$ is added.
Then the algorithm puts $q_1$ and $q_4$ into NextLevelGroup and computes their MCS ($q_6$ in Figure 14(e)). For the second group, we can choose the first pair ($q_3$, $q_4$) and find two MCSs ($q_6$ and $q_7$ in the figure), put them and $q_3$ into NextLevelGroup, and compute the MCS of $q_5$ and $q_7$. The final PCM is shown in Figure 3.1(d).

**Complexity** The TLS set building for each query is $O(n d^2)$ where $n$ is the number of vertices and $d$ is the maximum degree. The complexity for building TLS matrix is $O(n^2)$ where $n$ is the number of queries. The group matrix is usually sparse, therefore the clique detection can be very quick. As the query graphs are small, the MCS computation can be fast in practice.

### 3.5 Query Execution Order

In this section, we investigate the issue of query execution order. Our target is to minimize the number of cached results in memory.

To make sure the results of PCM parents can be utilized by their children, we must process the parents first and cache their results. Therefore the query processing order needs to be a topological order. There can be multiple topological orders available for the PCM, some of them will lead to the problem of inefficient memory usage.

Consider the PCM in Figure 3.2(a). To effectively share the results of common subgraphs, the results of PCM parents of $q_{101}$ which are $q_3$ to $q_6$ and $q_{99}$ have to be cached before $q_{101}$ is processed. If we choose a query execution order as the increasing order of the PCM node ID, the results of $q_{101}$’s parents will stay in memory when we process $q_7$ to $q_{99}$. This may lead to the memory leak problem if too many cached results are kept in memory. Assuming we choose an order ($q_1$, $q_3$ $\sim$ $q_6$, $q_{99}$, $q_{101}$, $q_2$, $q_7$ $\sim$ $q_{10}$, $q_{100}$, $q_{102}$, $\ldots$). All results for $q_{101}$’s parents can be released after $q_{101}$ is processed, thus the memory can be more effectively used.
Next we present a heuristic algorithm, Algorithm 9, for finding a good query execution order, which combines both topological order and priority weight together. The basic idea is as follows. (1) Each node will not be added to the list \( EOrder \) (Execution order) until all of its parents have been added. (2) A priority weight is assigned to each query that has not been added to \( EOrder \): Initially all weights are 0. If a node cannot be added to \( EOrder \) because it has unadded parents, the weight of these parents will be increased by 1, and this increase will propagate to the ancestors of these parents as well. (3) For a given set of unadded queries that have no unadded parents, priority will be given to those that have the highest weight.

Algorithm 9 starts from grouping query graphs having no \( PCM \) parents into a \( Roots \) list (Line 1). \textbf{Subroutine} nextQueryGraph takes a \( queryGraphList \) \( \zeta \) as parameter and returns a query \( q \) from \( \zeta \) where \( q \) is not added to \( EOrder \) yet, but all of its parents have been added, and \( q \) has the highest weight among such queries. Algorithm 9 iterates over all the queries in the \( Roots \) and calls a \textit{Topo} subroutine for each of them (Line 3-5). In \textbf{Subroutine} \textit{Topo}, if \( q \) has unadded parents, we increase the parent’s weight and propagate the increment to the ancestors (Line 1-2). Otherwise we add \( q \) to the \( EOrder \) and mark it as added (Line 4-5). For each of the unadded children of \( q \) whose parents have all been added, we recursively call \textit{Topo} (Line 6-8).

\textbf{Example 15.} Consider the \( PCM \) in Figure 3.2(b). The \( Roots \) list is initialized as \( \{q_1, q_7\} \). Starting from \( q_1 \), it proceeds to \( q_2 \) after \( q_1 \) is added. \( q_2 \) is also added because all of its parents have been added. Then it comes to \( q_5 \) and \( q_6 \). However only \( q_5 \) is added since \( q_6 \) has one unadded parent \( q_4 \). Recursively, we increase the weight of \( q_4 \) and \( q_4 \)’s parent \( q_7 \). Then it comes to \( q_7 \). After \( q_7 \) is added, the algorithm adds \( q_4 \) first as \( q_4 \) has larger weight than that of \( q_3 \) and \( q_8 \). After \( q_3 \) and \( q_8 \) are added, and the algorithm terminates.

\textbf{Complexity} Algorithm 9 costs \( O(nmk) \) where \( n \) is the number of nodes in \( PCM \), \( m \) is the maximum number of edges among the ancestors of a node, and \( k \) is the maximum number of a parents of a node. In practice, both \( m \) and \( k \) are very small, and the time for computing the execution order is trivial.

\section*{3.6 Caching Results}

In this section, we study the data structure and algorithm to cache the intermediate results. The challenge here is to find a structure that balances effective memory use and fast cached result retrieval.
Algorithm 9: **QUERYEXECUTIONORDER**

**Input:** PCM of a query set $Q$, weights are initialized 0

**Output:** A query execution order $EOrder$

1. $Roots \leftarrow q \in PCM$ and $q$ has no parents
2. $q \leftarrow nextQueryGraph(Roots)$
3. **while** $q$ is not null **do**
   4. $Topo(q)$
   5. $q \leftarrow nextQueryGraph(Roots)$

**Subroutine** $Topo(query q)$

1. **if** $q$ has parents not added to $EOrder$ **then**
   2. $changeParentsWeight(q)$
   **else**
   3. add $q$ to $EOrder$, mark $q$ as added
   4. $q' \leftarrow nextQueryGraph(q.children)$
   5. **while** $q'$ is not null **do**
   6. $Topo(q')$
   7. $q' \leftarrow nextQueryGraph(q.children)$

**Subroutine** $changeParentsWeight(query q)$

1. **for each** parent $q'$ of $q$ not added to $EOrder$ **do**
   2. $q'.weight++$
   3. $changeParentsWeight(q')$

**Subroutine** $nextQueryGraph(queryGraphList \zeta)$

1. **if** $S \equiv \{q \in \zeta | q.added = false, q$ has no unadded parent$ \neq \emptyset \}$ **then**
   2. Choose $q$ from $S$ with the highest weight
   3. **return** $q$
   **else**
   5. **return** null

Assuming a fixed order of the query vertices, an embedding can be represented as a list of corresponding data vertices. A trivial structure for caching the embeddings is a table where each row stores an embedding. An example is shown in Figure 3.3(b). This structure allows very fast retrieval of the embeddings. However, the problem with this structure is that it may take too much memory. To see this, consider a query graph with 10 vertices, we use 4 bytes to represent one vertex and 40 bytes to store one embedding. It needs 40M to store 1 million embeddings of this query. It is not unusual for a single query to have millions of embeddings in a large graph. In our experiment, even for Human data set which contains only 4675 vertices, the space of embeddings for 50 graphs can easily be over 500MB. Thus, this structure is impractical when dealing with graphs with millions of vertices.
An intuitive improvement over the table structure is to group the data vertices together and add corresponding edges of the query edges to link the data vertices. The result is a *compressed embedding graph* which is a subgraph of the data graph. Figure 3.3(c) is a compressed embedding graph and we have \((A_1, B_1), (A_1, D_1), (B_1, C_1), (C_1, D_1)\) derived from embedding \([A_1, B_1, C_1, D_1]\). This structure saves cache space, but the process of retrieving the embeddings is a subgraph isomorphism search over the compressed embedding graph, hence can be too slow.

To balance the space cost and the time efficiency, we propose a data structure *Compressed Linked List* for the storage of intermediate results. Before that, we need to define *graph partition*.

**Definition 12 (Graph partition).** Given a graph \(G\), a partition of \(G\) is a graph \(G'\) where

1. each node\(^1\) in \(G'\) is a non-empty set of vertices in \(G\);
2. the vertex sets corresponding to different nodes of \(G'\) are disjoint;
3. there is an edge between two nodes \(C_i\) and \(C_j\) in \(G'\) iff there is an edge \((u, v)\) in \(G\) such that \(u \in C_i\) and \(v \in C_j\).

The size of the largest vertex set in \(G'\) is called the *partition width*. When \(G'\) is a tree, it is called a *tree partition of \(G\).*

Consider the graph \(G\) in Figure 3.4(a). The graphs in Figure 3.4(b) and (c) are partitions of \(G\) with partition widths of 2 and 3 respectively.

Let \(q\) be a query graph. Given a partition \(q'\) of \(q\) which divides the vertices of \(q\) into \(K\) groups, an embedding of \(q\) in \(G\) can be divided into \(K\) lists accordingly, with each list corresponding to a node in \(q'\) (or equivalently, a vertex list in \(q\)). Actually, each list represents an embedding of the graph induced by the corresponding vertex group. We link two lists together if and only if there is an edge between the two corresponding nodes in \(q'\). For

\(^1\)For clarity, we use *node* to refer to the vertex of \(G'\), and *vertex* to refer to the vertex of \(G\).
instance, for the partition shown in Figure 3.4(b), the embedding \((A_1, B_1, C_1, D_1, E_1, G_1)\) can be represented as a linked list \((A_1, B_1)-(C_1, D_1)-(E_1, F_1)-(G_1)\). For the partition shown in Figure 3.4(c), the same embedding can be represented as the lists \((A_1,B_1,E_1)\), \((C_1,F_1,G_1)\), \((D_1)\) pairwise linked together. With this in mind, if multiple embeddings map the vertices in a node of \(q'\) to the same list of vertices in the data graph, we only need to cache the list once. In this way, we save cache space, and meanwhile we can retrieve the embeddings easily following the links between the lists.

Formally, we define a data structure called compressed linked lists (CLL) for the storage of intermediate embeddings.

**Definition 13** (Compressed Linked Lists). Given a data graph \(G\), a query graph \(q\) and a partition \(q'\) of \(q\), the compressed linked lists (CLL) of \(q\) with respect to \(q'\) and \(G\) consists of the set of lists defined and linked as follows:

1. For every embedding \(f\) of \(q\) in \(G\), and every node \(C\) of \(q'\), there is a list which is the projection of \(f\) onto the vertices in \(C\).

2. There is a link between two lists if they can be obtained from the same embedding of \(q\) in \(G\), and there is an edge between the corresponding nodes in \(q'\).

Intuitively, the CLL of \(q\) w.r.t \(q'\) and \(G\) is a compact representation of all embeddings of \(q\) in \(G\), which stores every embedding of the graph induced by each vertex group in \(q'\) exactly once. Moreover, every individual embedding of \(q\) can be retrieved from the CLL by following the links between the lists. For example, consider the query \(G\) in Figure 3.4(a) and its tree partition Figure 3.4(b) and graph partition Figure 3.4(c). The embeddings of \(G\) are given in Figure 3.5(a), the CLLs based on these partitions are as shown in Figure 3.5(b) and (c) respectively.

It is worth noting that each query graph may have many partitions and each of them leads to a different CLL. Different CLLs have different performance in terms of space and the time for retrieval. Intuitively, the larger the partition width \(k\), the more space we will need. For instance, when \(K = |V|\), all vertices of \(G\) are put into one group and the partition
consists of a single node, but the embeddings of $G$ will be stored as a single table. Also, for a fixed $K$, the closer the partition is to a tree, the quicker it is to assemble the original embeddings as for tree edges we can just follow the links, while for non-tree edges we need to do extra check to ensure connectivity. Based on these observations, it is clear that a tree partition of small width, if it exists, will be ideal. However, the tree partition width (which is the minimum partition width at which there exists a tree partition) is often too large. Therefore, we propose to use a partition of bounded width which is closest to a tree, called bounded-width tree-like partition to generate the CLL.

Given a connected graph $G'$ with node set $V'$ and edge set $E'$, we have $\chi(G') = |E'| - |V'| + 1$. Intuitively, a spanning tree of $G'$ has $|V'| - 1$ edges, and $|E'| - |V'| + 1$ is the number of edges we must remove from $G'$ to obtain a spanning tree.

**Definition 6** (Bounded-width Tree-like Partition). Given a graph $G$ and an integer $K$, a Bounded-Width Tree-like Partition (BTLP) of $G$ is a partition $G'$ of $G$ such that

1. $G'$ has partition width at most $K$.
2. $G'$ has the least number of edges among all partitions of width $K$ or less.

The complexity of the bounded-width tree-like partition problem is NP-complete. To prove this claim, we only need to prove the following decision problem is NP-complete.

**Definition 7** (BTLP problem). Given graph $G = (V, E)$ and integers $K < |V|, M < |E|$, is there a graph partition $G'$ of $G$ with partition width $\leq K$ and $\chi(G') \leq M$?

**Theorem 2.** The BTLP problem is NP-complete.
Clearly the BTLP problem is in NP. To show it is NP-complete, we reduce the bounded-width tree partition (BTP) problem, which is known to be NP-complete [10], to an instance of the BTLP problem. The BTP problem is: Given graph $G = (V, E)$ and integer $K < |V|$, is there a tree partition of $G$ with partition width $\leq K$?

Given $G = (V, E)$ and integers $K < |V|, M < |E|$, we construct a new graph as follows: construct $M$ cliques $c_1, \ldots, c_M$ of size $3K$, and connect one vertex in each clique to a vertex in $G$. Denote this new graph by $G_1$. It can be easily verified that $G$ has a tree partition of width $\leq K$ iff $G_1$ has a graph partition of width $\leq K$ with no more than $M$ non-spanning tree edges (we omit the details here).

As discussed earlier, given query graph $q$, we would like to find a graph partition of $q$ with width no more than $K$, and with the minimum number of non-spanning tree edges. However, since the problem is NP-complete, we use a heuristic procedure to find a partition which meets the partition width requirement strictly, and the number of spanning tree edges is likely to be small.

Our heuristic procedure consists of two steps: In Step 1, we use the algorithm in [10] (referred to as the MTP algorithm hereafter) to compute a maximal tree partition (MTP), which is a tree partition where splitting any node will make the partition no longer a tree. Given graph $q$, the MTP algorithm uses BFS to divide the vertices into different levels: $L(1)$ contains a random vertex $v$, $L(i + 1)$ contains vertices which are adjacent to those vertices in $L(i)$ but not in $L(i)$ or previous levels. It then splits the vertices in each level into disjoint nodes: two vertices at $L(i)$ are put in the same node iff they are connected via vertices at the same level or vertices at $L(i + 1)$. For example, for the graph $q$ in Figure 3.6 (a), the resulting MTP is shown in Figure 3.6 (b). In Step 2, we check each node in the MTP starting from nodes at the largest level. If there is a node $N$ such that $|N| > K$, we will split it into $\lceil |N|/K \rceil$ groups of size no more than $K$ with some simple heuristic rules: if there are $K$-vertices in $N$ that are not connected to other vertices in $N$ via vertices in $N$, or via the same node at the next level, we will put these vertices into $N_1$ and the other vertices into $N_2$. Otherwise we split $N$ into $N_1$ and $N_2$ such that $|N_1| = K$ and $|N_2| = \lceil |N| - K \rceil$. This process is repeated until $N$ is split into $\lceil |N|/K \rceil$ groups. For example, for the MTP in Figure 3.6 (b) and $K = 3$, the vertices $u_2, u_3, u_4$ are not connected to the vertices $u_5, u_6$ via the same node below them, therefore, we can split the node $\{u_2, u_3, u_4, u_5, u_6\}$ into two nodes $\{u_2, u_3, u_4\}$ and $\{u_5, u_6\}$.

The MTP algorithm takes $O(m)$ where $m$ is the number of edges in $q$. Our heuristic process also takes $O(m)$. In practice the MTP algorithm is likely to find a tree partition with small width. If the width is $\leq K$, we are done. Otherwise our heuristic rules will split
the big nodes in a way that is likely to produce fewer additional edges than a random split.

### 3.7 Subgraph Isomorphism Search

In this section, we present our approach for multi-query subgraph isomorphism search which efficiently utilizes the PCM and cached results. For queries that have no PCM parents, we just relay them to the single-query subgraph isomorphism algorithm to process. For queries that do have PCM parents, we must revise the single-query subgraph isomorphism algorithm so as to utilize the cached results of the PCM parents.

Most single-query subgraph isomorphism algorithms follow the framework proposed in [34], which is a backtracking strategy looking for solutions by incrementing partial solutions or abandoning them when it determines they cannot be completed. In the framework, (1) \textsc{InitializeCandidates} is to prepare promising candidates for each query vertex. (2) \textsc{IsJoinable} is to decide whether a candidate can be matched to the query vertex by various pruning rules, given those query vertices already matched. (3) \textsc{NextQueryVertex} returns the next query vertex according to the mapping order. Before presenting our strategies to revise this framework, we need to define the concept of a joint graph.

**Definition 8** (Joint Graph). *Given a query \(q=(V_q, E_q, \Sigma_q, L_q)\) and a set of embeddings \(P\) from \(q\)'s PCM parents to \(q\), we construct a joint graph \(q_P = \{N_P, E_P\}\) as follows:

1. For any \(f \in P\), there is a node \(n \in N_P\) such that \(n = \text{VCover}(f)\).
2. For each non-covered vertex \(u \in V_q\), there is a node \(n \in N_P\) such that \(n = \{u\}\).
3. There exists an edge \((n_i, n_j) \in E_P\) iff \(n_i \cap n_j \neq \emptyset\) OR there exists \((u_i, u_j) \in E_q\) where \(u_i \in n_i\) and \(u_j \in n_j\).
The nodes (resp. edges) in a joint graph will be referred to as *joint nodes* (resp. *joint edges*).

Consider the queries in Figure 3.7. For clarity, in the figure (and in subsequent examples) we use a pair \( q_i : f \) to indicate that \( f \) is an embedding from \( q_i \). Given the set of embeddings \( P = \{q_1 : f_1, q_1 : f_2, q_2 : f_1, q_3 : f_1\} \), we have a joint graph \( q_P \) with \( N_P = \{n_1, n_2, n_3, n_4\} \) where \( n_1 = \text{VCover}(q_1 : f_1), n_2 = \text{VCover}(q_1 : f_2), n_3 = \text{VCover}(q_2 : f_1) \) and \( n_4 = \text{VCover}(q_3 : f_1) \). We have \( E_P = \{(n_1, n_3), (n_3, n_4), (n_2, n_4)\} \).

The basic idea to revise the single-query framework is to use a joint graph instead of the original query graph in the search. Intuitively, the vertices in each joint node can be mapped to the data vertices as a group, and candidates of the group are the cached embeddings of the PCM parents. Obviously, if any of the PCM parents has no embeddings, then there will be no embeddings of the original query graph. Therefore, we will build a joint graph to replace the original query graph only when every PCM parent has some embeddings.

Given a query graph \( q \) and its PCM parents, we can build different joint graphs by choosing different subsets of embeddings from the PCM parents to \( q \). Let \( P \) be the set of all embeddings from the PCM parents to \( q \), and \( V_P \) be the set of query vertices covered by these embeddings. To make good use of the cached results and minimize the number of nodes in the joint graph, we would like to find a minimum subset of \( P \) that covers \( V_P \). This
is essentially a set cover problem which is NP-complete. Therefore, we use Algorithm 10, which is revised from the well-known greedy algorithm for set cover, to find a good subset $P'$ of $P$.

**Algorithm 10: BUILDINGJOINTGRAPH**

**Input:** query $q$; $\text{Parent}(q)$ - the set of PCM parents of $q$, $P$ - the set of embeddings from $q$’s parents to $q$

**Output:** a subset $P'$ of $P$

1. $P' \leftarrow \emptyset$
2. $V_P \leftarrow \bigcup_{f \in P} VCover(f)$
3. $VCovered \leftarrow \emptyset$
4. while $|VCovered| < |V_P|$ do
5.   $f \leftarrow \text{ChooseEmbedding}(P)$
6.   add $f$ to $P'$
7.   $VCovered \leftarrow VCovered \cup VCover(f)$
8. return $P'$

Algorithm 10 is very simple. Initially $P'$ is empty. A variable $VCovered$ is used to record the covered vertices by the embeddings in $P'$. We add the embeddings one by one into $P'$ until $P'$ covers $V_P$ (Lines 4 to 7) using the following heuristic rules: embeddings that can cover the most not-yet covered vertices come first (this is from the greedy algorithm), and if several embeddings can cover the same number of non-covered vertices, then we choose one from a parent that has the least number of cached results (this is to reduce the number of candidates for the joint-node). The function $\text{ChooseEmbedding}(P)$ (Line 5) uses these rules to choose the next embedding.

**Example 16.** Consider the query $q_5$ and its parents in Figure 3.7. Suppose the data graph is $G$ shown in Figure 3.8. The graphs $q_1$, $q_2$, $q_3$ and $q_4$ (parents of $q_5$) have 2, 2, 1, and 2 embeddings respectively. Using Algorithm 10, we will start with $q_3 : f_1$ to generate the subset $\{q_3 : f_1, q_1 : f_1, q_1 : f_2, q_2 : f_1\}$ or $\{q_3 : f_1, q_1 : f_1, q_1 : f_2, q_4 : f_1\}$.

Once we have a joint graph we will use it as the input graph, and try to map a joint node (instead of a single vertex) in each iteration, as described below.

**InitializeCandidates** In the original framework of subgraph isomorphism, the candidates for each query vertex of $q$ are retrieved by utilizing label constraints (and other filtering conditions such as vertex degree constraints). In the modified framework, the input graph $q$ is replaced with the joint graph obtained using Algorithm 10, and the candidates for each joint node are the embeddings of the corresponding PCM parent in the data graph $G$. These embeddings are cached in the CLL and can be easily retrieved.
To accelerate the process, we use two conditions to filter out impossible candidates:

1. An embedding $f$ (in $G$) of the parent graph $q_{\text{parent}}$ of $q$ does not always form an embedding of the subgraph of $q$ induced by the vertices in the joint nodes. In such cases $f$ can be safely filtered out. This is because of Lemma 1. For example, consider the query graphs in Figure 3.7 and the data graph $G$ in Figure 3.8. $q_1$ is a parent of $q_5$, and both $(v_1, v_2, v_3)$ and $(v_5, v_{10}, v_{11})$ are embeddings of $q_1$ in $G$. However, the first embedding is an impossible candidate for the joint node produced by $q_1 : f_2$ since it is not an embedding of the subgraph of $q_5$ induced by the vertices in the joint node.

2. Suppose the joint node $n$ contains a query vertex $u$, and $u$ cannot be mapped to data vertex $v$ due to degree constraints or other filtering conditions used in single-query algorithms. Assume $n$ is produced by the embedding $h$ from $q_{\text{parent}}$ to $q$. Then any candidate of $n$ (i.e., embedding of $q_{\text{parent}}$) that maps $h^{-1}(u)$ to $v$ can be safely filtered out. Consider the joint node $n$ produced by $q_1 : f_2$ in Figure 3.7. Since $v_2$ (degree is 1) in Figure 3.8 cannot be matched to query vertex $u_7$ (degree is 3), any embedding of $q_1$ in $G$ that maps the $B$-node in $q_1$ to $v_2$ is not a valid candidate for $n$.

**IsJoinable** This function must be modified to test whether a candidate can be matched to the current joint node. Suppose we have matched joint nodes $n_1, \ldots, n_{k-1}$ to their candidates $c_1, \ldots, c_{k-1}$ before, and $c_k$ is a candidate of the current joint node $n_k$. We must make sure matching $n_k$ to $c_k$ (together with matching $n_1, \ldots, n_{k-1}$ to $c_1, \ldots, c_{k-1}$) will generate a partial embedding of the query graph in the data graph, i.e., an embedding of the subgraph induced by the vertices in $n_1, \ldots, n_k$. Specifically,

1. Each vertex in $n_1, \ldots, n_k$ must be mapped to a distinct data vertex.
2. If $n_k$ and $n_i$ ($i \in [1, k-1]$) have a common vertex $u$, then $n_k$ and $n_i$ must map $u$ to the same data vertex. Consider the joint node $n_2$ produced by $q_2 : f_1$ and $n_3$ by $q_3 : f_1$ in Figure 3.7. $n_2$ has a common vertex $u_6$ with $n_3$. If we have mapped $n_2$ to $(v_3, v_4, v_5)$ in Figure 3.8, then we cannot map $n_3$ to $(v_6, v_8, v_{10})$ because $u_6$ cannot be mapped to $v_5$ and $v_6$ at the same time.
3. If there are query vertices $u' \in n_k$ and $u \in n_i$ such that there is an edge $(u', u)$ in the query graph, and $n_k$ and $n_i$ map $u'$ to $v'$ and $v$ respectively, then there must be an edge $(v', v)$ in the data graph. For example, if we have mapped the joint node $n_2$ produced by $q_1 : f_2$ to $(v_9, v_{10}, v_{11})$, then we cannot map the joint node $n_3$ produced by $q_3 : f_1$ to $(v_4, v_5, v_7)$, as there is an edge between $u_5$ and $u_7$ but there is no edge between $v_7$ and $v_{10}$.

Note that the above conditions (2) and (3) need to be checked only if there is an edge between $n_k$ and $n_i$ in the joint graph.
3.8 Experiments

The correctness of the modified subgraph isomorphism search is clear from the observation that there is a 1:1 correspondence between the embeddings of \( q \) and the embeddings of the joint graph.

In this section, we report our experiments to evaluate our solution. Specifically, (1) we compare the effectiveness of our grouping factor with edge-label based Jaccard similarity which is used in [33]; (2) we evaluate the factors that affect PCM building time; (3) to evaluate the effectiveness of our query execution order, we compare the number of cached queries under different execution orders; (4) for CLL, we give the results to illustrate the effects of partition width on the memory usage and the time for retrieving embeddings from the CLL; (5) we compare the performance of our MQO with sequential query processing (SQO), and evaluate the effects of grouping factor threshold and query similarity on the performance of our solution.

Datasets. We used three benchmark datasets: Yeast, Human, and Wordnet. Human and Yeast were used in [24][34][46]. Wordnet was used in [56][46]. The three datasets have different characteristics. Yeast is a graph with small number of data vertices but many labels. Human is a clip of social network graph with much larger vertex degrees. Compared with Yeast and Human, Wordnet is a much larger data graph, however it is much sparser and has very few vertex labels. The profiles of the datasets are given in Table 4.2.

| Dataset | \( |V| \) | \( |E| \) | \( |\Sigma| \) | Avg. degree |
|---------|-------|-------|-------|------------|
| Human   | 4675  | 86282 | 90    | 36.82      |
| Yeast   | 3112  | 12915 | 184   | 8.05       |
| Wordnet | 82670 | 133445| 5     | 3.28       |

Query Graphs. We designed two generators to generate the query graphs based on the data sets. (1) Random graph generator, which takes the number of query graphs \( N \) and family size \( S \) as input, and randomly chooses \( \frac{N}{S} \) data vertices as core-vertices. For each core-vertex \( v \), it picks 5 to 10 vertices within a distance of 5 from \( v \), and generates a family of \( S \) connected queries by randomly connecting the vertices with an edge. The number of edges in each query ranges from 5 to 15. Queries within each family share the same

\[\text{family}^2\]
core-vertex and are likely to have more overlaps. Thus family size acts as a parameter to control the overlaps of the queries: generally the larger the family size, the more overlaps among the queries. The family size used in our experiments ranges from 1 to 10. If family size is 1, the generator is a pure random graph generator, which generates queries with rare overlaps. (2) Subgraph generator, which is used to generate subgraphs of the data graph for testing result caching strategies. Given the number of queries $N$, it randomly chooses $N$ data vertices. For each vertex, it generates a subgraph around this vertex by random walk. The number of edges ranges from 5 to 15. Each subgraph generated this way is guaranteed to have at least one embedding.

**Experimental Settings.** We implemented 2 recent single-query subgraph isomorphism algorithms: TurboIso[24] and TurboIsoBoosted [46]. We also implemented a revised version for these two algorithms according to Section 3.7 so as to support MQO. All the algorithms were implemented in C++. All the experiments were carried out under 64-bit Windows 7 on a machine with 4GB memory.

### 3.8.1 Grouping Factor Effectiveness

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Yeast</th>
<th>Human</th>
<th>Wordnet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tls</td>
<td>Jac</td>
<td>Tls</td>
</tr>
<tr>
<td>10%</td>
<td>2077</td>
<td>2288</td>
<td>4192</td>
</tr>
<tr>
<td>30%</td>
<td>6337</td>
<td>8436</td>
<td>12577</td>
</tr>
<tr>
<td>50%</td>
<td>12002</td>
<td>16600</td>
<td>20963</td>
</tr>
<tr>
<td>70%</td>
<td>18018</td>
<td>26662</td>
<td>29348</td>
</tr>
<tr>
<td>90%</td>
<td>24854</td>
<td>41170</td>
<td>37789</td>
</tr>
</tbody>
</table>

Intuitively a more effective grouping factor or similarity measure is the one that can obtain the same number of MCSs with less trials (a trial means trying to compute the MCS for one pair of queries). To compare the effectiveness of our TLS-based grouping factor against the edge-label based Jaccard similarity, we fix the query set and compare the number of trials of each method under their maximum threshold that can obtain a fixed percentage of all MCSs. For each dataset, we use the random graph generator to generate 500 (family size is 10) queries. The percentage of required MCSs varies from 10% to 90%. The result is given in Table 3.2. As we can see, for all the percentages, our grouping factor requires significantly fewer trials (note that computing MCS is an expensive process). The difference between the two methods is even larger with larger query sets, we omit the experimental results for larger query sets due to space limit. The time
difference for computing the grouping factor and the Jaccard similarity is negligible due to the small size of query graphs.

### 3.8.2 PCM Building Time

We conducted three sets of experiments to evaluate the PCM building time. (1) To evaluate the scalability, we tested the PCM building time under different query set sizes ranging from 100 to 1000. The family size was set to 10. The results are given in Figure 3.9(a). (2) To evaluate the PCM building time under different query similarities, we tested the building time for 10 sets of queries with family sizes from 1 to 10. Each query set has 1000 queries. The results are given in Figure 3.9(b). (3) To evaluate the effect of grouping factor threshold, we tested the PCM building time and the number of PCM edges detected under different thresholds ranging from 0.1 to 1. The results are given in Figure 3.10. All of the queries for the three tests were generated by the random graph generator. The grouping factor threshold used in tests (1) and (2) is 0.5 for Yeast, 0.6 for Human and 0.9 for Wordnet. The query set size is 1000 and family size is 10 for test (3).

As shown in Figure 3.9(a), the PCM building time for both Yeast and Human has a slight increment when the query set size is increased, while the time for Wordnet shows a sharper increment. As aforementioned, Wordnet only has 5 labels, which results in much higher possibility of two queries sharing common subgraphs. For Yeast and Human, due to the diversity of labels, queries are not easy to share common subgraphs, and the PCM building time is under 2 seconds for 1000 queries.

As shown in Figure 3.9(b). For Yeast and Human, the PCM building time only shows a slight increment with increasing family size. While the time for Wordnet shows a much larger increment. Although the PCM building time can be more than 10 seconds for Wordnet with family size 10, and it grows with larger family size due to more overlaps, this cost can be easily paid back because there will be more MCSs detected, which will lead to much larger query time savings.

In Figure 3.10, the horizontal axis represents the grouping factor threshold, the left vertical axis is the number of PCM edges, and the right vertical axis is the time for building the PCM. The lines show the PCM building time, and the bars show the number of PCM edges. As shown in the figure, both the time and number of PCM edges increase for all three datasets when the threshold is decreased. The number of PCM edges increases faster with smaller thresholds (We omit the bars of wordnet whose value is more than 8000). This is because many of the connected common subgraphs under a small threshold are
small graphs, and it is easy for two queries to have small common subgraphs while it is much harder for them to have a large one.

Fig. 3.9 PCM Building Time on Real Datasets

3.8.3 Query Execution Order

To evaluate the effectiveness of our execution order, we compare the number of cached queries of our heuristic execution order (H) with that of a random topological order (R). When the algorithm reaches the stage to compute the subgraph isomorphism for any query, we record the number of cached queries at this time point. For each dataset and for each order, we use **Peak** and **Average** to represent the highest number and the average number at all the time points respectively. The results are given in Table 3.3. The query set size is 1000 and generated by the random graph generator. As we can see, both the peak and the average number of our order is less than that of the random topological order. Especially for Wordnet and Human, our order significantly reduced the peak number of cached queries.
3.8 Experiments

Table 3.3 Number of Cached Queries

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Yeast</th>
<th>Human</th>
<th>Wordnet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>H</td>
<td>R</td>
<td>H</td>
</tr>
<tr>
<td>Peak</td>
<td>5</td>
<td>6</td>
<td>23</td>
</tr>
<tr>
<td>Average</td>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

3.8.4 Intermediate Result Caching

To evaluate the power of CLL, we tested the effects of partition width on the memory use and the time cost for retrieving embeddings from the CLL. For each of the three datasets, we used the Subgraph Generator to generate one query set containing 50 different queries. We first conducted subgraph isomorphism search under different cache settings for each of the queries and cached all the final embeddings after the search. Then we did an embedding enumeration to test the time for recovering the embeddings from the cache.

Table 3.4 Cache Memory Use in KB

<table>
<thead>
<tr>
<th>Data\Width</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Raw</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yeast</td>
<td>9</td>
<td>11</td>
<td>17</td>
<td>19</td>
<td>21</td>
<td>$255 \times 10^3$</td>
</tr>
<tr>
<td>Human</td>
<td>24</td>
<td>137</td>
<td>358</td>
<td>672</td>
<td>1483</td>
<td>$583 \times 10^3$</td>
</tr>
<tr>
<td>Wordnet</td>
<td>12</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>$399 \times 10^3$</td>
</tr>
</tbody>
</table>

The memory cost of different types of cache results is given in Table 3.4. Width represents the partition width of the graph partition of the queries. When width is set to 1, it only allows one vertex in each partition node. This is equal to the trivial structure of compressed embedding graph. The last column is marked as Raw which represents the trivial table structure without any grouping of vertices. As shown, the sizes of the cache for the table structure are much larger than that of the grouped results. For Human data set which is a relatively dense graph with small size, the trivial table structure can use more than 500MB for caching the results of only 50 queries. 500M is not a problem for modern computers with gigabytes of memory. However, it can be much worse for larger and denser data graphs. The cached result sizes show an overall increasing trend with the increment of the partition width. Not surprisingly, the sizes of cached results for Wordnet become stable when the partition width is larger than 2. This is because Wordnet is a sparse graph, the queries generated from Wordnet are sparse graphs as well. Thus the queries can be partitioned into trees with small width. The partition would not change much given a larger allowed width as our algorithm usually produces a tree partition of
the least partition width.

<table>
<thead>
<tr>
<th>Data</th>
<th>Width</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yeast</td>
<td></td>
<td>226</td>
<td>202</td>
<td>159</td>
<td>126</td>
<td>137</td>
</tr>
<tr>
<td>Human</td>
<td></td>
<td>3247</td>
<td>1628</td>
<td>952</td>
<td>858</td>
<td>770</td>
</tr>
<tr>
<td>Wordnet</td>
<td></td>
<td>3930</td>
<td>2314</td>
<td>1412</td>
<td>1400</td>
<td>1306</td>
</tr>
</tbody>
</table>

The embedding retrieval time is shown in Table 3.5. As expected, the retrieval time shows an decreasing trend with the increment of the partition width. A smaller partition width may lead to more non-spanning tree edges, checking the connection of these edges when retrieving the embeddings takes more time.

### 3.8.5 Query Processing Time

In this subsection, we present the comparisons of the performances of MQO and sequential query processing (SQO). We report the results from three perspectives. (1) The scalability of the query processing. For each query set, we generated 10 query sets, with set size ranging from 100 to 1000. The family size was set to 10. The grouping factor threshold was set to 0.5 for Yeast, 0.6 for Human and 0.9 for Wordnet. The partition width is set to 3 for all data sets. The results are given in Figure 3.11. (2) The effects of query similarity over the query processing time. We used different family sizes (from 1 to 10) when generating the queries. The query set size is 1000. (3) The effects of different grouping factor thresholds over the query processing time. We used query sets with 1000 queries and the family size was set to 10. Due to space limit, we only present the results for Human for experiments (2) and (3) in Figure 3.12.

As shown in Figure 3.11, our MQO approach achieved significant improvement over SQO for all three datasets. Compared with Yeast, both Human and Wordnet achieved larger improvement. There are two possible reasons for this: (1) Yeast is a small graph where the average query processing time is short, hence the space for time savings is not as big as for the other data sets. (2) Yeast contains many labels which makes the queries harder to share common subgraphs. The improvement of MQO over SQO for TurboIso is larger than that for TurboIsoBoosted for all datasets in terms of absolute time saved (note the different time units in the figures).

As shown in Figure 3.12(a), with the increment of the family size, the performance of MQO show larger improvement over SQO. We used two horizontal lines to represent the average SQO processing time over all queries for TurboIso and TurboBoosted respectively.
3.8 Experiments

(a) TurboIso over Yeast
(b) TurboBoosted over Yeast
(c) TurboIso over Human
(d) TurboBoosted over Human
(e) TurboIso over Wordnet
(f) TurboBoosted over Wordnet

Fig. 3.11 Performance Comparison and Scalability Test

(a) Effects of query similarity
(b) Effects of threshold

Fig. 3.12 Effects of query similarity and GF threshold
Multi-Query Optimization

(note that the family size does not affect the average time cost of SQO). As can be seen, when the family size is 1 (which means the queries have little overlap), the performance of our MQO is only slightly worse than that of SQO.

As shown in Figure 3.12(b), neither a very large nor a very small grouping factor threshold can achieve the best performance. The former leads to many useful MCSs not being detected, and the latter leads to relatively long PCM building time with many small common subgraphs being detected. The PCM building time cannot be easily paid back in such cases.

3.9 Conclusion

We presented our solution of MQO for subgraph isomorphism search in this chapter. Our experiments show that, using our techniques, the overall query processing time when multiple queries are processed together can be significantly shorter than if the queries are processed separately when the queries have many overlaps. Furthermore, the larger the data set or the more time it takes for an average query, the more savings we can achieve. When the queries have no or little overlap, our filtering technique can detect it quickly, resulting in only a slight overhead compared with SQO. To the best of our knowledge, our work is the first on MQO for general subgraph isomorphism search. It demonstrates that MQO for subgraph isomorphism search is not only feasible, but can also be highly effective.
Chapter 4

Distributed Subgraph Enumeration: A Practical Asynchronous System

In this chapter, we present a Practical Asynchronous Distributed Subgraph enumeration system (PADS). Section 4.1 presents some practical motivations of distributed subgraph enumeration and discusses some problems existing in previous research. We further discuss the techniques proposed by previous research in Section 4.2. In Section 4.3, we give the architecture of PADS and present the principles of R-Meef based on which we designed SubEnum, the core of PADS. The algorithm of building explanation plan is given in Section 4.4. In Section 4.5, we present an inverted trie data structure to compress intermediate results. After that, two memory control strategies are given in Section 4.6. Then we present our experimental study in Section 4.7. At last we conclude this chapter in Section 5.1.

4.1 Motivations and Problems

In the real world, the data graphs are often fragmented and distributed across different sites. For example, a social network graph maybe distributed across different servers and data centers for performance and security considerations [22][38]. This phenomenon highlights the importance of distributed systems for subgraph enumeration. Also, the increasing size of modern graph makes it hard to load the whole graph into memory, which further strengthens the requirement for distributed systems that could share the workload among the computing resources of a cluster.

Assuming the data graph is partitioned over multiple machines, many distributed processing approaches of subgraph enumeration have been proposed [1][13][54][31][32][13].
However they are facing the following problems:

- **Memory Crisis** A fatal issue of the single-round join-oriented solutions [1][13] is that the graph partition held within each machine easily outnumbers the available memory of a single machine. The reason is that, for any particular machine, those solutions have to copy a large part of data graphs from other machines to the memory of this machine. With the ability of avoiding maintaining a large part of the data graph in memory, the algorithms [31][32][54] have achieved some success by following a multi-round style. However, current multi-round algorithms suffer from the fact that a huge number of intermediate results are generated in each round, causing severe burden not only on memory but also on network communication.

- **Heavy Index** Aiming to solve the output crisis of finally found embeddings, [40] proposed a compression strategy based on which it also provided a distributed subgraph enumeration framework. However, [40] needs to pre-compute and index all the cliques of the data graph. Although this solution achieved some speed-up, the size of the indexed cliques can be much larger than the graph itself (see Section 4.7) in our experiments. For instance, the original data graph of liveJournal takes 1G space, while the index of [40] takes more than 15G space. Maintaining such a heavy index generates huge overhead especially when the graph needs to be updated frequently.

- **Intermediate Result Storage Latency** Most of the existing distributed approaches are based on the join-oriented method where the embeddings of two small subgraphs are first loaded in memory and then joined together to get the embeddings of the union of these two subgraphs [43][31][32]. It is known that CPU is faster than RAM access, therefore, besides the memory crisis aforementioned, the latency generated by saving a large size of intermediate results in memory can also be a serious problem for these join-oriented approaches. To our best knowledge, this issue has never been noted in early research.

- **Synchronization Delay** Most existing multi-round distributed approaches [54][31][32] are following a synchronous model where machines are synchronized within each round. To be specific, before any machine proceeds to the next round, the embeddings from every machine of the current round have to be shuffled across the network. Therefore, those machines that have completed the current round have to wait for those that have not. The time cost of each round will highly depend on the worst machine. This
synchronization delay can be a big problem when the datasets are large and when the computing power of a cluster is not evenly distributed.

4.2 Related Work

In this section, we extend the discussion of existing distributed subgraph enumeration solutions in Section 4.1 and highlight the differences between them and our work. We classify them into three categories by their main techniques: Multi-round Join-oriented, Exchanging Data Vertices, and Compression.

4.2.1 Multi-round Join-oriented

The state-of-the-art distributed subgraph enumeration solutions \cite{43} follow a multi-round join-oriented approach. They first decompose the query pattern into small join units (a subgraph of the query pattern) and then launches multiple rounds of joins to get all the embeddings, whose process can be formulated as:

\[
\mathbb{R}(P) = \mathbb{R}(p_0) \Join \mathbb{R}(p_1) \Join \cdots \Join \mathbb{R}(p_n). \tag{4.1}
\]

where \(\mathbb{R}(p_i)\) denotes the embeddings of \(p_i\) in data graph \(G\).

The most critical problem of the above join-oriented approach is that a huge number of intermediate results will be generated and they may quickly drain the memory. Secondly, those intermediates need to be exchanged in each round, leading to heavy network communication and therefore, a huge network latency.

One more issue is that this approach inherits the intermediate latency of using join to solve subgraph enumeration. Compared with join-oriented approach, the backtracking based method does not generate any intermediate results therefore has no such latency. Although the backtracking based TurboIso \cite{24} has proved much faster than a typical join-oriented approach \cite{56}, it is hard to determine whether the intermediate latency is a big problem because of other filtering and pruning techniques used in \cite{24}. We better demonstrated this issue by comparing the performance of join-oriented approach with backtracking approach through a triangles listing task which avoids the effects of any auxiliary techniques. In our experiment, we show that the backtracking method outperformed join-oriented approach by 10 times in real datasets as shown in experiment 4.7.1.

Aiming to relieve the intermediate letancy of \cite{31}, the SubEnum thread of PADS
utilizes an R-Meef approach which verifies and filters out the failed embedding candidates as early as possible. The intermediate latency is significantly reduced.

Similar to [31][32], PADS still generates some intermediate embeddings. However, we propose an inverted trie structure to effectively compress those intermediates. More importantly, we introduce a grouping strategy which divides the data candidates into region groups so that we can process each group sequentially and separately. The maximum number of intermediate results of each group is much smaller than the sum of processing all groups together. The ability to tune the peak number of intermediate results significantly increases the flexibility of our approach.

In contrast to [31][32], we do not send any intermediate results which can be quite explosive. Instead, we exchange edges that are need to be verified and adjacent-lists that are required (more details in next subsection), both of which are bounded by $|E|$.

### 4.2.2 Exchanging Data Vertices

The distributed subgraph enumeration solutions that choose to exchange data vertices instead of intermediate results include [1][13]. By saying exchanging data vertices, we mean exchanging the adjacency-list of those vertices. Both [1] and [13] follow a single-round style where, for any specific machine, it has to fetch a large set of data vertices from other machines. A fatal issue of these solutions is that the graph data held within each machine easily drains the memory of a single machine.

Similar to [1][13], PADS chooses to exchange data vertices. Differently, PADS uses a multi-round approach so that the data vertices exchanged are divided into multiple-rounds. Instead of coping data vertices blindly, we also introduce maximum-caching strategy which releases previously fetched/used data vertices when it nearly drains the memory or has reached a given threshold. Those strategies grant us the ability to control the peak number of data vertices held in memory during the process.

Moreover, for non-spanning tree query edges, to determine whether some particular data edges (which cannot be verified locally) can be matched to them, we do not fetch related data vertices to the local machine. Instead, we send those edges to be verified to other machines and get the verification results from them. Sending particular edges and verification results is more lightweight than sending data vertices. This technique significantly reduces the workload that needs network communication.
4.2.3 Compression

Aiming to solve the output-crisis of the subgraph enumeration, [40] introduced a compression strategy to compress the output embeddings. By pre-indexing all the cliques of the data graph, [40] also proposed a distributed enumeration solution. Although [40] achieved some success in speeding up the subgraph enumeration process, it has to admit that its index is quite large (as shown in Table 4.1.), which not only takes a large storage space but also increases the overhead of maintaining, especially when the data graph is frequently updated. This issue dramatically drops the practicality of [40]. In contrast, our approach does not need any auxiliary index except for the original data graph.

![Table 4.1 Illustration of the Index Size](image)

<table>
<thead>
<tr>
<th>Dataset (G)</th>
<th>Original File Size</th>
<th>Index File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>13M</td>
<td>210M</td>
</tr>
<tr>
<td>RoadNet</td>
<td>87M</td>
<td>569M</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>501M</td>
<td>6.5G</td>
</tr>
</tbody>
</table>

4.3 PADS

4.3.1 Architecture Overview

Given an unlabelled data graph $G$ and $m$ machines $\{M_1, \ldots, M_m\}$ in a distributed environment, a partition of $G$ is denoted as $\{G_1, \ldots, G_m\}$ where $E_G = \{E_{G_1} \cup \cdots \cup E_{G_m}\}$ and $G_t$ is the graph partition located in the $t^{th}$ machine $M_t$. PADS finds all subgraphs of $G$ that are isomorphic to $P$ from all the partitions. In this chapter, we assume each partition is stored as a adjacency-list. For any data vertex $v$, we assume its adjacency-list is stored in a single machine $M_t$ and we say $v$ is owned by $M_t$ (or resides in $M_t$) (denoted as $v \in V_{G_t}$). We say a data edge $e$ is owned by (or resides in) $M_t$ (denoted as $e \in E_{G_t}$) if either end vertex of $e$ resides in $G_t$. Note that an edge can reside in two machines.

The architecture of PADS is given in Figure 4.1.

Given an unlabelled query pattern $P$, PADS processes subgraph enumeration by executing the following two threads within each machine:

- **Daemon Thread** listens to requests from other machines and supports two functionalities: verifyE is to return the edge verification results for a given request consisting of vertex pairs. For example, given a request $\{(v_0, v_1), (v_2, v_3)\}$ posted to $M_1$, $M_1$ will
return \( \{true, false\} \) if \((v_0, v_1)\) is an edge in \(G_1\) while \((v_2, v_3)\) is not. \textit{fetchV} is to return the adjacency lists of requested vertices of the data graph.

- **SubEnum Thread** is the core subgraph enumeration thread which applies a new approach \textbf{R-Meef} (region-grouped multi-round expand-verify-filter). When necessary, SubEnum thread sends undetermined edges and remote vertex requests to the Daemon threads located in other machines.

We use \textbf{PADS} when referring to our distributed subgraph enumeration system as a whole and use SubEnum to refer to the core subgraph enumeration thread.

Utilizing the above framework, \textbf{PADS} elegantly solves the aforementioned issues that are preventing existing solutions from being practically applied. To be specific, \textbf{PADS} achieves

\begin{enumerate}
  \item \textbf{Asynchronism} The core subgraph enum thread of \textbf{PADS} runs separately and asynchronously where there is no waiting latency among them. This leads the overall performance of the process to be dramatically improved.
  \item \textbf{Index Free} Except for the adjacency-list of the data graph, \textbf{PADS} does not need any auxiliary index. This characteristic significantly improved the practicality of \textbf{PADS} especially when the data graphs needs frequent update.
  \item \textbf{Lightweight Communication} Unlike \cite{54}\cite{31}\cite{32}, \textbf{PADS} does not send any intermediate embeddings through the network. The requests and responses for both \textbf{verifyE} and \textbf{fetchV} are much more lightweight compared to previous approaches.
\end{enumerate}
(4) **Relieved Intermediate Result Storage Latency** Instead of join-oriented method, the core subgraph enumeration algorithm follows a novel multi-round verify-expand-filter idea which is similar to the incremental verification strategy of backtracking algorithms. It filters out unpromising embedding candidates by edge verification as early as possible. In this way, it effectively reduces the number of intermediate results that are needed to be stored in memory. Therefore the intermediate result storage latency is relieved in **PADS**.

(5) **Memory Control** The memory usage of **PADS** is controlled by a set of new techniques: a compression strategy to compress intermediate results, a grouping strategy to split the workload and a maximum-data caching mechanism to reduce communication.

### 4.3.2 R-Meef

The architecture of our distributed subgraph enumeration system **PADS** has already been presented in the above subsection. The functions within the daemon thread are very intuitive to understand, therefore we omit any further details about them. In this section, we focus on the explanation of the SubEnum thread, which is the core component of **PADS**.

Before we present the algorithmic details of SubEnum, let us first explain its underlying principles which can be abstracted as a *region-grouped multi-round expand-verify-filter* (**R-Meef**) approach where the following definitions are important.

**Definition 9.** *Given a graph partition* $G_t$ *of data graph* $G$ *located in machine* $M_t$ *and a query pattern* $P$, *the injective function* $f_{G_t}: V_P \rightarrow V_G$ *is an embedding candidate of* $P$ *in* $G$ *w.r.t* $G_t$ *such that for any edge* $(u_i, u_j) \in E_P$, *if either* $f_{G_t}(u_i) \in V_{G_t}$ *or* $f_{G_t}(u_j) \in V_{G_t}$, *there exists an edge* $(f_{G_t}(u_i), f_{G_t}(u_j)) \in E_{G_t}$.

The definition of *embedding candidate* is similar to that of partial embedding while the embedding candidate is restrictedly defined within a given graph partition $G_t$. For any query edge, it only requires that the corresponding mapped data edge $e$ must exist if either vertex of $e$ is owned by $G_t$. We use $\mathcal{R}_{G_t}(P)$ to denote a set of embedding candidates of $P$ in $G$ w.r.t $G_t$.

$f_{G_t}^{-1}(v)$ is used to denote the query vertex mapped to data vertex $v$ in $f_{G_t}$.

**Definition 10.** *Given an embedding candidate* $f_{G_t}$ *of query pattern* $P$, *for any* $e = (v, v')$ *where* $(f_{G_t}^{-1}(v), f_{G_t}^{-1}(v')) \in E_P$, *$e$ is called an undetermined edge of* $f_{G_t}$ *if neither* $v$ *nor* $v'$ *resides in* $G_t$.
Obviously if we want to determine whether $f_{G_t}$ is actually an embedding of the query pattern, we have to verify its undetermined edges in other machines. For any undetermined edge $e$, if its two vertices reside in two different machines, we can use any of them to verify whether $e \in E_{G_t}$ or not.

**Example 17.** Consider a random graph partition $G_t$ of a data graph $G$ and a triangle query pattern $P$ where $V_P = \{u_0, u_1, u_2\}$, $f_{G_t} = \{(u_0, v_0), (u_0, v_1), (u_0, v_2)\}$ is an embedding candidate of $P$ in $G$ w.r.t $G_t$ if $v_0 \in V_{G_t}$, $v_1 \in Adj(v_0)$ and $v_2 \in Adj(v_0)$ while neither $v_1$ nor $v_2$ resides in $G_t$. $(v_1, v_2)$ is an undetermined edge of $f_{G_t}$.

Next let us assume that we have an identification strategy such that each $f_{G_t}$ can be looked up by a unique ID. (We will discuss our identification strategy in Section 4.5.)

**Definition 11.** Given an embedding candidate set $\mathcal{R}_{G_t}(P)$, the edge verification index (EVI) is a key-value map where for any tuple $\{e, IDs\}$,

- the key $e$ is a vertex pair $(v, v')$,
- the value IDs represent the set of IDs of the embedding candidates of $\mathcal{R}_{G_t}(P)$.

and (1) if the ID of any $f_{G_t} \in \mathcal{R}_{G_t}(P)$ is in $IDs$, $e$ is an undetermined edge of $f_{G_t}$. (2) for any undetermined edge $e$ of $f_{G_t} \in \mathcal{R}_{G_t}(P)$, there exists a tuple in $EVI$ with $e$ as the key and the ID of $f_{G_t}$ as the value.

Intuitively, the edge verification index is a structure to organize the undetermined edges of a set of embedding candidates. It is straightforward to see:

**Lemma 2.** Given a data graph $G$, query pattern $P$ and an edge verification index $EVI$, for any $\{e, IDs\} \in EVI$, if $e \not\in E_G$, then none of the embedding candidates corresponding to $IDs$ can be an embedding of $P$ in $G$.

We have explained $f_{G_t}$, $\mathcal{R}_{G_t}(P)$ and $EVI$, which are key structures of the data flowing in the workflow of R-Meef. Now we are ready to define the execution plan which guides the workflow of R-Meef. Before that, we present the definition of our query decomposition.

**Definition 12.** A decomposition of query pattern $P = \{V_P, E_P\}$ is a set of decomposition units $\mathcal{D}E = \{d_{p_0}, \ldots, d_{p_l}\}$ where for every $d_{p_i} \in \mathcal{D}E$, $d_{p_i} = \{u_{pivot}, V_{leaf}\}$ and

1. $u_{pivot} \in V_P$, $d_{p_i}.V_{leaf} \subseteq V_P$, and for every $u' \in V_{leaf}$, $(u_{pivot}, u') \in E_P$. 


4.3 PADS

(2) $V_{d_{pi}} = \{d_{pi}.u_{pivot}\} \cup V_{leaf}$, and $E_{d_{pi}} = E_{d_{pi}}^{star} \cup E_{d_{pi}}^{sib}$ where $E_{d_{pi}}^{star} = \{d_{pi}.u_{pivot}, u'\}$, and $E_{d_{pi}}^{sib} = \{u, u' \in V_{leaf} \mid (u, u') \in E_{p}\}$.

(3) $\bigcup_{d_{pi} \in \mathcal{P}} (V_{d_{pi}}) = V_{p}$, and for any two $d_{pi}, d_{pj}$, $E_{d_{pi}} \cap E_{d_{pj}} = \emptyset$.

Similar to the star decomposition within the left-deep join of [31], our decomposition units cover all the vertices of $P$ and share no common edges with each other. Each of the decomposition units can be treated as a subgraph of $P$. While differently, our unit is not restricted to star since it allows edges among leaves. We use $E_{d_{pi}}^{star}$ represent the edges from pivot to leaves and $E_{d_{pi}}^{sib}$ represents the edges among leaves. Also it is worth noting that $\bigcup_{d_{pi} \in \mathcal{P}} (E_{d_{pi}}) \subset E_{p}$.

Based on the above decomposition, we define the execution plan as following:

**Definition 13.** Given a query pattern $P$, the execution plan is an ordered list of decomposition units, denoted as $\mathcal{P} \mathcal{L} = \{d_{p_0}, \ldots, d_{p_l}\}$ where for any $d_{pi} \in \mathcal{P} \mathcal{L}$,

(1) $P_i = \{V_{p_i}, E_{p_i}\}$ where $V_{p_i} = \bigcup_{p_{jsi} \in \mathcal{P} \mathcal{L}} (V_{p_j})$ and $E_{p_i} = \bigcup_{p_{jsi} \in \mathcal{P} \mathcal{L}} (E_{p_j})$.

(2) $d_{pi}.u_{pivot} \in V_{p_{i-1}} (i > 0)$ and $d_{pi}.V_{leaf} \cap V_{p_{i-1}} = \emptyset$.

It is easy to see that $P_i$ is the union of the first $i$ units. As per the order of execution plan, it requires that the pivot-vertex of $d_{pi}$ must appear in some units before $d_{pi}$ and any of the leaf-vertex of unit $d_{pi}$ should never appear in any unit before $d_{pi}$. Considering the condition $(3)$ of Definition 12, it is obvious that $V_p = V_{p_1}$.

For any $d_{pi}$ of a decomposition plan $\mathcal{P} \mathcal{L}$, let us define:

$$E_{d_{pi}}^{anc} = \bigcup_{u \in d_{pi}.V_{leaf}, u' \in V_{p_{i-1}}} (u, u') \in E_{p}$$

We call $E_{d_{pi}}^{star}$ expansion query edges and call the union of $E_{d_{pi}}^{sib}$ and $E_{d_{pi}}^{anc}$ verification query edges. The $E_{d_{pi}}^{anc}$ is an empty set. We have the following lemma:

**Lemma 3.** Given an execution plan $\mathcal{P} \mathcal{L} = \{d_{p_0}, \ldots, d_{p_n}\}$ of query pattern $P$, it holds that $|\bigcup_{d_{pi} \in \mathcal{P} \mathcal{L}} (E_{d_{pi}}^{star})| = |V_{p}| - 1$.

Intuitively, the union of the expansion edges of all the units forms a spanning tree of pattern $P$. The verification query edges are non-spanning tree edges of the pattern $P$.

**Example 18.** Consider the query pattern in Figure 4.2(a), we have an execution plan $\mathcal{P} \mathcal{L} = \{d_{p_0}, d_{p_1}, d_{p_2}, d_{p_3}\}$ where $d_{p_0}.u_{pivot} = u_0$, $d_{p_0}.V_{leaf} = \{u_1, u_2, u_7\}$, $d_{p_1}.u_{pivot} = u_1$, $d_{p_1}.V_{leaf} = \{u_3, u_4\}$, $d_{p_2}.u_{pivot} = u_0$, $d_{p_2}.V_{leaf} = \{u_6, u_9\}$ and $d_{p_3}.u_{pivot} = u_2$, $d_{p_3}.V_{leaf} = \{u_5, u_6\}$.
Definition 14. Given an execution plan \( \mathcal{P} \) and a list of candidate vertices \( C(d_{p0}.u_{pivot}) \) in \( M_t \) that can be mapped to \( d_{p0}.u_{pivot} \), we divide \( C(d_{p0}.u_{pivot}) \) into region groups, denoted as \( \mathcal{RG} = \{r_{g0}, \ldots, r_{gh}\} \) where each region group is a set of data vertices and

1. for any two \( r_{gi}, r_{gj} \in \mathcal{RG} \), \( r_{gi} \cap r_{gj} = \emptyset \).
2. \( C(d_{p0}.u_{pivot}) = \bigcup_{r_{gi} \in \mathcal{RG}} r_{gi} \).

With the above concepts, we are ready to present the principles of our \textbf{R-Meef} approach.

Given query pattern \( P \), data graph \( G \) and its partition \( G_t \) within a machine \( M_t \), \textbf{R-Meef} finds a set of embeddings of \( P \) in \( G_t \) based on the following:

1. \textit{Region-Grouped} From the vertices residing in \( M_t \), \textbf{R-Meef} divides those which can be mapped to the first query vertex of the execution plan into different region groups. Then it process each group sequentially and separately. The idea is to divide the workload into multiple independent processes so that the maximum intermediate results cached in the memory can be significantly decreased, therefore memory crash can be effectively avoided.

2. \textit{Multi-round} For each region group, \textbf{R-Meef} processes one unit at a round based on the execution plan \( \mathcal{P} \). In \( i^{th} \) round, the workflow can be illustrated in Figure 4.3.

\[
\mathcal{RG}_t(P_{i-1}) \xrightarrow{\text{Expand}} \widehat{\mathcal{RG}}_t(P_i) \xrightarrow{\text{Verify \\& Filter}} \mathcal{RG}_t(P_i)
\]

Fig. 4.3 \textbf{R-Meef} workflow

In Figure 4.3, the \( \mathcal{RG}_t(P_{i-1}) \) represents the embedding generated and cached from last round. If it is the first round, \( \mathcal{RG}_t(P_{i-1}) \) will be initialized as \( \bigcup \{(d_{p0}.u_{pivot}, v)\} \) where
$v$ is the candidate vertex that can be mapped to $dp_0.u_{pivot}$. By expanding $\mathbb{R}_{G_i}(P_{i-1})$, we get all the embedding candidates $\mathbb{H}_{G_i}(P_i)$ of $P_i$ w.r.t $M_t$. After verification and filtering, we get all the embedding of $P_i$ within $M_t$, which are partial embeddings of $P$ in $G$.

In each round, the **expand** and **verify & filter** work as follows:

- **Expand** According to our execution plan, given an embedding $f$ from last round, $dp_i.u_{pivot}$ has already been matched to a data vertex $v$ by $f$. By searching the neighbourhood of $v$, we expand $f$ and find all the embedding candidates of $dp_i$ containing $(dp_i.u_{pivot}, v)$ w.r.t $M_t$. It is worth noting that if $v$ does not reside in $M_t$, we have to fetch its adjacency-list from other machines. Different embeddings have different vertices to fetch in order to expand. For all the embeddings from last round, we gather all the vertices that need to be fetched and then fetch their adjacency-lists together.

- **Verify & Filter** Upon having a set of embedding candidates $\mathbb{H}_{G_i}(P_i)$, we can easily build an $\mathcal{EVI}$ from them. Then we send the keys of $\mathcal{EVI}$ to other machines to verify their existence. One important assumption here is that each machine has a record of the ownership information (say which machine a particular $v$ resides at) of all the vertices. This record can be constructed offline as a map whose size is $|V|$. When constructing the record, each machine broadcasts the IDs of their local vertices to other machines, whose time cost is very small. After we get the verification results, we filter out the failed embedding candidates from $\mathbb{H}_{G_i}(P_{i(i)})$. And the remaining ones are $\mathbb{H}_{G_i}(P_i)$. The output of the final round is the set of embeddings of query pattern $P$ in $G$ within $M_t$.

### 4.3.3 Algorithm of SubEnum

Next we look at the implementation of **R-Meef** as shown in Algorithm 11. Algorithm 11 is executed in each machine $M_t$ simultaneously and asynchronously.

**Data Structure**

- Within each machine, we first compute an execution plan $\mathcal{PL}$ based on the given query pattern $P$ (Line 2). The $\mathcal{PL}$s computed in every machine are the same.

- We group the candidate data vertices of $dp_0.u_{pivot}$ within $M_t$ into into region groups (Line 3). For each region group $rg$, a multi-round mapping process is conducted (Line 4 to 21).
Algorithm 11: **SUBEnum Framework**

**Input:** Query pattern $P$ and Data graph $G$ partitioned into $\{G_1, G_2, \ldots, G_m\}$ located at $m$ machines from $M_1$ to $M_m$

**Output:** All embeddings of $Q$ in $G$

for each machine $M_t (1 \leq t \leq m)$ simultaneously do

1. $\mathcal{PL} \leftarrow$ executionPlan($P$)
2. $\mathcal{RG} = \{r_{g_0} \ldots r_{g_k}\} \leftarrow$ regionGroups($C(dp_0.u_{pivot}, M_t)$)
3. for each region group $r_g \in \mathcal{RG}$ do
   4. init inverted trie $\mathcal{IT}$ with size $|V_P|$
   5. init edge verification index $\mathcal{EVI}$
   6. for each data $v \in r_g$ do
      7. $f \leftarrow (dp_0.u_{pivot}, v)$
      8. expandInvertedTrie($f, M_t, dp_0, \mathcal{IT}, \mathcal{EVI}$)
      9. $\mathcal{R} \leftarrow$ verifyForeignE($\mathcal{EVI}$)
      10. filterFailedEmbed($\mathcal{R}, \mathcal{IT}$)
   11. for Round $i = 1$ to $|\mathcal{PL}|$ do
      12. clear $\mathcal{EVI}$
      13. fetchForeignV($i$)
      14. for each $f \in \mathcal{IT}$ do
         15. expandInvertedTrie($f, M_t, dp_i, \mathcal{IT}, \mathcal{EVI}$)
         16. $\mathcal{R} \leftarrow$ verifyForeignE($\mathcal{EVI}$)
         17. filterFailedEmbed($\mathcal{R}, \mathcal{IT}$)
   18. return the embeddings within $\mathcal{IT}$

- In Line 5, We use an inverted trie $\mathcal{IT}$ to save the generated intermediate results. We use intermediate results for short in the rest of the paper. We will give more details of inverted trie in Section 4.5.
- The edge verification index $\mathcal{EVI}$ is initialized in Line 6, which will be reset for each round of processing (line 13).

(2) **First Round** Starting from each candidate $v$ of $r_g$, we match $v$ to $dp_0.u_{pivot}$ in the execution plan. With the pivot-vertex being matched, we find all the embedding candidates of $dp_0$ with regarding to $M_t$ and compress them into $\mathcal{IT}$. We use a function `expandInvertedTrie` to represent this process (Line 9). For each embedding candidate compressed in $\mathcal{IT}$, its undetermined edges need to be verified in order to determine whether this embedding candidate is a partial embedding of $P$. We record this information in the edge verification index $\mathcal{EVI}$, which is constructed in the `expandInvertedTrie`. After we have $\mathcal{EVI}$ in $M_t$, we send network requests and verify those undetermined edges within $\mathcal{EVI}$ in the machine who has the ability to
verify it (*verifyForeignE* in Line 10). After the edges in *EVI* are all verified, we filter out the failed embedding candidates from *IT* (Line 11).

(3) **Other Rounds** For each of the remaining rounds of the execution plan, we first clear the *EVI* from previous round (Line 13). In the *i*th round, we want to find all the embedding candidates of *dP* based on the embeddings *R* <sup>i</sup> (where the *dP*:*upivot* has been matched). While in those embeddings, not all the data *v* matched to *dP*:*upivot* are within the *Mi*. We have to fetch the adjacency-lists of those foreign vertices from other machines in order to expand from them. A sub-procedure *fetchForeignV* is used to represent this process (Line 14). After that, for each embedding *f* from last round, we find all the embedding candidates of *Pi* by expanding from *f*(*dP*:*upivot*) and compress them into *IT* (Line 16). Then *verifyForeignE* and *filterFailedEmbed* are called to make sure the expanded *IT* only contains partial embeddings of *Pi* (Line 17, 18).

Finally, after all the rounds have finished, we report the final embeddings compressed in *IT* (Line 19).

**Example 19.** Consider the query pattern *P* and the data graph *G* in Figure 4.2, the vertices marked with dash border are owned by *M*<sub>0</sub> and the other vertices reside in *M*<sub>1</sub>. We use the execution plan given in Example 18. We assume the preserved orders due to symmetry breaking: *u*<sub>1</sub> < *u*<sub>2</sub>, *u*<sub>8</sub> < *u*<sub>9</sub> and *u*<sub>3</sub> < *u*<sub>4</sub> < *u*<sub>5</sub> < *u*<sub>6</sub>.

There is one vertex {*v*<sub>0</sub>} in *M*<sub>0</sub> and two vertices {*v*<sub>2</sub>, *v*<sub>10</sub>} in *M*<sub>1</sub> with a degree larger than that of *dP*:*upivot*. Therefore in *M*<sub>1</sub>, we have *C*(*dP*:*upivot*) = {*v*<sub>0</sub>} and in *M*<sub>2</sub> we have *C*(*dP*:*upivot*) = {*v*<sub>2</sub>, *v*<sub>10</sub>}. After division, assume we have *RG* = {*rg*<sub>0</sub>} where *rg*<sub>0</sub> = {*v*<sub>0</sub>} in *M*<sub>1</sub> while we have *RG* = {*rg*<sub>0</sub>, *rg*<sub>1</sub>} where *rg*<sub>0</sub> = {*v*<sub>2</sub>} and *rg*<sub>1</sub> = {*v*<sub>10</sub>} in *M*<sub>2</sub>.

Consider the region group *rg*<sub>0</sub> in *M*<sub>1</sub>, we first match *v*<sub>0</sub> to *dP*:*upivot*. Expanding from *v*<sub>0</sub>, we may have embedding candidates *f*<sub>1</sub> = {{*u*<sub>0</sub>, *v*<sub>0</sub>}, {*u*<sub>1</sub>, *v*<sub>1</sub>}, {*u*<sub>2</sub>, *v*<sub>2</sub>}, {*u*<sub>7</sub>, *v*<sub>7</sub>}}, *f*<sub>1</sub><sup>i</sup> = {{*u*<sub>0</sub>, *v*<sub>0</sub>}, {*u*<sub>1</sub>, *v*<sub>1</sub>}, {*u*<sub>2</sub>, *v*<sub>7</sub>}, {*u*<sub>7</sub>, *v*<sub>2</sub>}} and *f*<sub>1</sub><sup>ii</sup> = {{*u*<sub>0</sub>, *v*<sub>0</sub>}, {*u*<sub>1</sub>, *v*<sub>2</sub>}, {*u*<sub>2</sub>, *v*<sub>8</sub>}, {*u*<sub>7</sub>, *v*<sub>1</sub>}}. The edges (*v*<sub>1</sub>, *v*<sub>2</sub>) and (*v*<sub>2</sub>, *v*<sub>8</sub>) cannot be determined in *M*<sub>1</sub>. Therefore, we load *f*<sub>1</sub>, *f*<sub>1</sub><sup>i</sup> into *IT* and put {{*v*<sub>1</sub>, *v*<sub>2</sub>}, < *f*<sub>1</sub>)} and {{*v*<sub>2</sub>, *v*<sub>8</sub>}, < *f*<sub>1</sub><sup>i</sup>)} into *EVI*. However, there is no undetermined edges in *f*<sub>1</sub><sup>i</sup> and {{*v*<sub>1</sub>, *v*<sub>7</sub>}} verified failed in *M*<sub>1</sub>. *f*<sub>1</sub><sup>i</sup> is simply locally filtered. After verification of *EVI*, a false returned for (*v*<sub>2</sub>, *v*<sub>8</sub>), *f*<sub>1</sub><sup>ii</sup> will be removed from *IT*.

In the next round, among the embeddings from last round, we have a *f* = {{*u*<sub>0</sub>, *v*<sub>0</sub>}, {*u*<sub>1</sub>, *v*<sub>1</sub>}, {*u*<sub>2</sub>, *v*<sub>2</sub>}, {*u*<sub>7</sub>, *v*<sub>7</sub>}}. To expand from *v*<sub>1</sub>, we need to fetch its adjacency-list first. After the adjacency-list of foreign vertices of all embeddings from last round, we expand each *f*. Following the above process, after we process the last round, all the final embeddings
starting from this region group in machine $M_1$ will be saved in $IT$.

In the following sections, we will study of several optimizing strategies to improve the performance of the SubEnum framework.

### 4.4 Computing Execution Plan

It is obvious that we may have multiple valid execution plans for a query pattern and different execution plans may have different performances, therefore the challenge is to find the most efficient one among them. In this section, we present two techniques to tackle this challenge.

#### 4.4.1 Minimising Number of Rounds

Given a query pattern $P$ and an execution plan $PL$, recall Algorithm 11, we have $|PL|$ rounds for each region group, where in each round the workload can be shared. To be specific, a single undetermined edge $e$ may be shared by multiple embedding candidates. If they are generated in the same round, the cost of network communication and verification by $e$ of be shared among them effectively. The same principle applies to the foreign vertices whose cost of fetching and memory space can be shared among multiple embedding candidates if they happen in the same round. Therefore, in order to optimize the performance, our first heuristic is to minimize the number of total rounds. Here we present a technique to compute query execution plans, which guarantees a minimum number of rounds.

Let us review the concept of connected dominating set [20].

**Definition 15.** Given a query pattern $P$, a connected dominating set $DS$ is a subset of the vertex set of a $P$ where any two vertices are reachable to each other and any vertex of the $P$ is either in $DS$ or adjacent to a vertex in $DS$.

A minimum dominating set $MDS$ is the one with smallest cardinality among all connected dominating sets. A query pattern may have multiple $MDS$s while they all have the same cardinality. We have the following theorem.

**Theorem 3.** Given a $DS$ of query pattern $P$, there exists a valid execution plan $PL$ where $|PL| = |DS|$. 
Proof. To prove Theorem 3, let us try to construct a $\mathcal{PL}$ where the $u_{pivot}$ of each unit is a vertex within $\mathcal{DS}$. Sequentially following an increasing order of vertex ID, for each vertex $u \in \mathcal{DS}$, we create a unit $d_{pi}$ and set $d_{pi}.u_{pivot} = u$. For any $u' \in (Adj(u) \cap \mathcal{DS})$, if $u'$ has neither been used as a pivot nor been added into leaf vertices of any other unit yet, we add $u'$ to $d_{pi}.V_{leaf}$. After all units are created, for any $u' \in (\mathcal{VP} - \mathcal{DS})$, we add $u'$ to $d_{pi}.V_{leaf}$ where the degree of $d_{pi}.u_{pivot}$ is the largest among $Adj(u') \cap \mathcal{DS}$. Finally arranging those units into $\mathcal{PL}$ following the ascending order of the ID of their $u_{pivot}$, it is easy to see that $\mathcal{PL}$ is a valid execution plan.

Theorem 4. Given a $\mathcal{PL}$ of a query pattern $P$, $\bigcup_{d_{pi} \in \mathcal{PL}} \{d_{pi}.u_{pivot}\}$ is a connected dominating set of $P$.

Proof. It is intuitive to prove Theorem 4, since every leaf vertex is connected to a pivot in $\mathcal{PL}$. And the pivot vertices of $\mathcal{PL}$ is a connected subset of $P$ as per its definition.

Based on Theorem 3 and Theorem 4, it is straightforward to have the Corollary 2.

Corollary 2: Given a $\mathcal{MDS}$ of a query pattern $P$, $|\mathcal{MDS}|$ is the minimum number of rounds for any valid execution plan that $P$ may have.

Given the above three theorems, it is guaranteed that we can generate at least one execution plan $\mathcal{PL}$ given a $\mathcal{MDS}$ of a query pattern and the $\mathcal{PL}$ is guaranteed to have a minimum number of rounds.

Example 20. Consider the query pattern in Figure 4.2(a), we can get a minimum dominating set $\mathcal{MDS} = \{u_0, u_1, u_2\}$, based on which, we have an minimum round execution plan $\mathcal{PL} = \{d_{p0}, d_{p1}, d_{p2}\}$ where $d_{p0}.u_{pivot} = u_0$, $d_{p0}.V_{leaf} = \{u_1, u_2, u_7, u_8, u_9\}$ and $d_{p1}.u_{pivot} = u_1$, $d_{p1}.V_{leaf} = \{u_3, u_4\}$ and $d_{p2}.u_{pivot} = u_2$, $d_{p2}.V_{leaf} = \{u_5, u_6\}$.

4.4.2 Moving Forward Verification Edges

Given a query pattern $P$, we may have multiple execution plans with the minimum number of rounds. In order to further optimize the performance of our approach, we need to find the one with the best performance from them. Then the question will be: given two execution plans with the same number of rounds, how could we determine one is more efficient than another? To answer this question, we propose the following scoring function $S(\mathcal{PL})$ for a given plan $\mathcal{PL} = \{d_{p0}, \ldots, d_{pl}\}$:

$$SC(\mathcal{PL}) = \sum_{d_{pi} \in \mathcal{PL}} \frac{\rho}{(i + 1)} \times (|E_{d_{pi}}^{sib}| + |E_{d_{pi}}^{anc}|)$$  (4.2)
Algorithm 12: ComputeExecutionPlan

**Input:** Query pattern \( P \)

**Output:** The execution plan \( \mathcal{P}L = \{ dp_0 \ldots dp_l \} \)

1. \( MDSs \leftarrow \text{minimumCDS}(P) \)
2. \( \mathcal{P}L_{\text{max}} \leftarrow \emptyset, \mathcal{P}L \leftarrow \emptyset \)
3. For each \( MDS \in MDSs \) do
4.   For each \( u \in MDS \) do
5.     Create a unit \( dp \)
6.     \( dp, upivot \leftarrow u \)
7.     \( NR' \leftarrow \text{Adj}(u) \cap MDS \)
8.     \( dp, V_{\text{leaf}} \leftarrow NR' \)
9.     Add \( dp \) to \( \mathcal{P}L \)
10.   NextRoundUnit(\( NR' \))
11. Remove \( dp \) from \( \mathcal{P}L \)

Subroutine NextRoundUnit(\( NR \))

1. If \( NR = \emptyset \) then
2.   For each \( u \in (V_p - MDS) \) do
3.     \( \mathcal{P}L_{\text{sub}} \leftarrow \bigcup_{dp \in \mathcal{P}L, dp, upivot \in (\text{Adj}(u) \cap MDS)} dp \)
4.     \( dp \leftarrow \text{earliestUnit}(\mathcal{P}L_{\text{sub}}) \)
5.     Add \( u \) to \( dp, V_{\text{leaf}} \)
6.     If \( SC(\mathcal{P}L) > SC(\mathcal{P}L_{\text{max}}) \) or \( \mathcal{P}L_{\text{max}} = \emptyset \) then
7.       \( \mathcal{P}L_{\text{max}} \leftarrow \mathcal{P}L \)
8.   For each \( u \in NR \) do
9.     \( NR' \leftarrow (NR - u) \)
10. Create a unit \( dp \)
11. \( dp, upivot \leftarrow u \)
12. For each \( u' \in \text{Adj}(u) \) and \( u' \) is a neither a pivot nor a leaf in \( \mathcal{P}L \) do
13.   If \( u' \in MDS \) then
14.     Add \( u' \) to \( dp, V_{\text{leaf}} \) and to \( NR' \)
15.     Add \( dp \) to \( \mathcal{P}L \)
16. NextRoundUnit(\( NR' \))
17. Remove \( dp \) from \( \mathcal{P}L \)
The union of $E_{dpi}^{stib}$ and $E_{pi}^{anc}$ are the query edges that need to be verified in each round. Our heuristic is to move forward those verification edges so that their filtering power could be put into effect early and the failed candidates can be filtered out early. Function $SC(PL)$ assigns a score to the unit of each round and sums the score together. The early round $dp_i$ is and the more verification $dp_i$ brings, the higher score $i^{th}$ round will get. A larger $i$ leads to a weak contribution from the verification edges of $dp_i$. Since we index the units from 0, thus the dominator is set as $i+1$. $\rho$ is a parameter within a range of $(0, 1)$ to slightly tune up the function.

Based on the above scoring function, we are ready to present the algorithm to compute the execution plan of the query pattern $P$. The algorithm is shown in Algorithm 12.

We first compute the minimum connected dominating sets for the query pattern (Line 1) where we can utilize the algorithm proposed in [20]. We use $PL_{max}$ to save the execution plan with the largest score (Line 2) and use $PL$ to storage the status of the current partial execution plan being tested. For each $MDS$, we try to use every of its vertices as the pivot-vertex of $dp_0$ (Line 4 to 6). We set the $dp_0.\text{Vleaf}$ as those neighbours of $dp_0.\text{vpivot}$ which are in the $MDS$ (Line 7). After adding the $dp$ to $PL$, we call a recursive function to enumerate every possible $MDS$ vertex as the pivot-vertex of the unit in each round respectively.

The Subroutine $nextRoundUnit$ takes a $NR$ as input which is a set of $MDS$ vertices that have already been used as leaf-vertices in $PL$. Then for every $u$ of $NR$, we first make a copy $NR'$ of the input not including $u$ (Line 9). Then we try to use $u$ as the pivot-vertex of the unit $dp$ in this round (Line 10, 11). For each neighbour $u'$ of $u$, if $u'$ has never used in $PL$ and is a $MDS$ vertex, we add $u'$ to both the $dp.\text{Vleaf}$ and $NR'$ (Line 12 to 14). Then we put $dp$ to $PL$ and call the recursive function to go deeper round (Line 15, 16). After we the recursive function returns, we remove $dp$ from $PL$ and try another $u$ in the loop (line 17). After we create all the units, for each query vertex $u$ not in $MDS$, we find the earliest added unit $dp$ from $PL$ where $u$ is connected to $dp.\text{vpivot}$ and then add $u$ to $dp.\text{Vleaf}$ (Line 2 to 5). Once the current execution plan is finished, we compute its score based on Equation 4.2. If its score is larger than that of the $PL_{max}$, we update $PL_{max}$ as $PL$ (Line 6, 7).

### 4.5 Inverted Trie

It is easy to see that the structure to cache the intermediate results (partial embeddings and embedding candidates) plays a significant role in our approach. Firstly, it requires
the structure to compress those intermediate results to save memory as much as possible. Secondly, it requires that each cached intermediate result to have a unique ID. Thirdly, it requires that an intermediate result can retrieved or removed it efficiently given a particular ID. Last, the structure should support incremental expansion where we can easily expand the intermediate results. With the above requirements in mind, we propose the *inverted trie* in this section.

First let us define a matching order based on a given execution plan \( PL \).

**Definition 16.** Given a execution plan \( PL \) of query pattern \( P \), \( MO \) is a function to assign the order of the query vertices being matched, and

1. \( MO \) starts with 0 with \( MO(dp_0.u_{pivot}) = 0 \).
2. for any \( dp_i \in PL \) and any \( u \in dp_i.V_{leaf} \), \( MO(u) = MO(dp_i.u_{pivot}) + |V_{largerD}|+1 \) where \( V_{largerD} \) is a subset of \( dp_i.V_{leaf} \) and whose vertices have a larger degree than that of \( u \).

Based on the definition of \( PL \), a query vertex is either a pivot-vertex of \( dp_0 \) or a leaf-vertex of a unit \( dp_i \) of \( PL \). We assign 0 to the matching order of \( dp_0.u_{pivot} \). While for a leaf vertex \( v \) of any \( dp_i \), we need to get \( |V_{largerD}| \) which is the number of leaf vertices in the same \( dp_i \) whose degree is larger than that of \( v \). To break the tie, if the degree a leaf \( v' \) is equal to that of \( v \), \( v' \) also belongs to \( V_{largerD} \) if the ID of \( v' \) is larger than \( v \). Then the sum of \( MO \) and \( V_{largerD} \) is the order of the leaf vertex \( v \). Consider the execution plan in Example 20, \( MO(u_0)=0, MO(u_1)=1 \) and \( MO(u_4)=3 \).

It is worth noting that the matching order \( MO \) is different from the partial order generated from symmetry breaking. The matching order is the order following which the query vertex is matched in SubEnum and the intermediate results are loaded into the inverted trie. We denote it as \( u_i \rightarrow u_j \) if \( u_i \) is the query vertex whose matching order is one less than \( u_j \).

Now we are ready to present the details of inverted trie.

**Definition 17.** Given a query pattern \( P \) and a matching order \( MO \), inverted trie is a collection of node lists where each query vertex corresponds to one of them. For any trie node \( N \) of \( u_i \)'s list, it has,

1. \( v \): the data vertex mapped to \( u_i \) in all intermediate results.
2. \( childN \): the pointer points to a trie node \( N' \) of the corresponding list of \( u_j \) where \( (u_j \rightarrow u_i) \).
4.5 Inverted Trie

(3) **parentNodeNo:** the number of trie nodes pointing to \( N \).

It is easy to see that the direction of the pointers in the trie node lists is opposite to the matching order \( \mathcal{MO} \). We set the child node as null for the nodes in the list of first query vertex in the matching order. Given a node \( N \) points to \( N' \), \( N' \) is called the a child node of \( N \) and \( N \) is called as a parent node of \( N' \). Within the inverted trie, we call the trie nodes without any parent nodes as tail nodes.

Intuitively, the inverted trie applies a hierarchical strategy to group the intermediate results. Given a set of embeddings of query pattern \( P \), in the first level, it groups the embeddings by the data vertices mapped to the first query vertex in the matching order. Then for the embeddings falling in the same group of the first level, it groups them by the data vertices mapped to the second query vertex in the matching order. After that the hierarchy goes on until the last query vertex.

![Fig. 4.4 Example of Inverted Trie](image)

To easily understand it, see the following example:

**Example 21.** Consider a query pattern \( P \) with \( V_P = \{u_0, u_1, u_2\} \) and assume we have a matching order \( (u_0 \rightarrow u_1 \rightarrow u_2) \), we have an inverted trie as shown in Figure 4.4. The inverted trie concealed 4 intermediate results which are \([\{(u_0, v_0), (u_1, v_2), (u_2, v_3)\}, \{(u_0, v_0), (u_1, v_3), (u_2, v_5)\}, \{(u_0, v_1), (u_1, v_4), (u_2, v_5)\}, \{(u_0, v_1), (u_1, v_3), (u_2, v_3)\}]\).

Although the structure of inverted trie is simple, it elegantly meet the requirements aforementioned.

- **Compression** The compression of inverted trie is inherited from the widely known trie structure. Each of our intermediate results accounts for a combination of data vertices. We group the intermediate by the common data vertices in each level so as to reduce the size of intermediate results.
Algorithm 13: REMOVE TRIE NODE

Input: A failed trie node \( N \)

1. \( N' = N \cdot \text{childN} \)
2. while \( N' \) is not Null do
3. \( N'.\text{parentNodeNo} -- \)
4. if \( N'.\text{parentNodeNo} = 0 \) then
5. \( N'' = N', N' = N'.\text{childN} \)
6. remove \( N'' \) from \( IT \)
7. remove \( N' \) from \( IT \)

- **Unique ID** For each intermediate result in inverted trie, the location address of its tail node in memory can be used as the unique ID.

- **Retrieve and Remove** Given a specific tailnode, we can easily track down its pointer childN step-by-step to retrieve the corresponding intermediate result. To remove any intermediate result, we can remove its corresponding tail trie node and decrease the parentNodeNo of its childnode by 1. If parentNodeNo of this childnode reaches 0, we remove this childnode. And this process recursively affects its descendants of the tail node. The detailed algorithm is as shown in Algorithm 13.

- **Incremental Expansion** The function expandInvertedTrie() which conceals implementation of the incremental expansion of the inverted trie \( IT \). Recall the framework of SubEnum where the embedding candidates of a unit of \( PL \) are incrementally added to \( IT \) in expandInvertedTrie() and the \( EVI \) is constructed at the same time.

The algorithm is shown as in Algorithm 14. It intakes a partial embedding \( f \) of \( P \) from previous round and tries to find all the embedding candidates containing \( f \) and covering the new vertices \( dp_i.V_{leaf} \). Based on the framework SubEnum, the \( dp_i.u_{pivot} \) is always mapped in \( f \). We get the mapping \( v_{pivot} \) of \( dp_i.u_{pivot} \) in Line 1. For each leaf-vertex \( u \) of \( dp_i \), we find its candidate \( C(u) \) by joining \( Adj(v) \) with the \( Adj \) of the data vertices (resides in \( Mt \)) mapped to query vertices that are mapped in \( f \) and has an edge with \( u \) (Line 2 to 8). Upon having a list of candidates for each leaf-vertex, we get the current tailnode corresponding to \( f \) (Line 9, 10) and call a recursive function to enumerate the combinations of the candidates of each leaf-vertex (Line 11).

In the Subroutine, it takes a childnode \( N \) and a leaf-vertex \( u \) as input. In Line 1 to 4, it refines the candidates for \( u \) by considering the recently mapped leaf-vertices of \( dp_i \) in \( f \) where those leaf-vertices has an edge with \( u' \). If the data vertices mapped to those leaf-vertices resides in \( Mt \), we shrink the candidates of \( u \) by joining it with
Algorithm 14: \textsc{ExpandInvertedTrie}

\textbf{Input:} an embedding candidate \( f \) of \( P \), local machine \( M_t \), unit \( dp_i \), edge verification index \( EVI \)

\textbf{Output:} updated \( IT \) and \( EVI \)

\begin{enumerate}
\item \( v_{\text{pivot}} \leftarrow f(dp_i, \text{u_{pivot}}) \)
\item \textbf{for each} \( u \in dp_i, V_{\text{leaf}}f \) \textbf{do}
  \begin{enumerate}
  \item \( C(u) \leftarrow Adj(v_{\text{pivot}}) \)
  \item \textbf{for each} \( (u, u') \in E_{\text{anc}}^{dp_i} \) \textbf{do}
    \begin{enumerate}
    \item if \( f(u') \) resides in \( M_t \) then
      \begin{enumerate}
      \item \( C(u) \leftarrow Adj(f(u') \cap C(u)) \)
      \end{enumerate}
    \end{enumerate}
  \end{enumerate}
\end{enumerate}
\item return \( \) \( \)
\item get \( u \) where \( MO(u) = |f| - 1 \)
\item get \( N \) corresponding to \( f(u) \)
\item \textbf{adjEnum(} \( \mathcal{MO}(|f|), N \) \textbf{)}
\end{enumerate}

\textbf{Subroutine \textsc{adjEnum}(} u, N \textbf{)}

\begin{enumerate}
\item \( C_r(u) \leftarrow C(u) \)
\item \textbf{for each} \( u' \) mapped in \( f \) and \( (u, u') \in E_{\text{sib}}^{dp_i} \) \textbf{do}
  \begin{enumerate}
  \item if \( f(u') \) resides in \( M_t \) then
    \begin{enumerate}
    \item \( C_r(u) \leftarrow Adj(f(u') \cap C_r(u)) \)
    \end{enumerate}
  \end{enumerate}
\item \textbf{for each} \( v \in C_r(u) \) \textbf{do}
  \begin{enumerate}
  \item if \( isJoinable \) \( (f, u, v) \) then
    \begin{enumerate}
    \item \( \text{flag} \leftarrow \text{false} \)
    \item create a trie node \( N' \)
    \item \( N'.dataV \leftarrow v' \)
    \item \( N'.childN \leftarrow N \)
    \item \( u' \leftarrow \mathcal{MO}(|f|) \)
    \item add \( (u, v) \) to \( f \)
    \item if \( |f| = |V_{\text{pos}}| \) then
      \begin{enumerate}
      \item \textbf{for each} unverified \( (v, v') \) mapped to \( (u, u') \in E_{\text{p}} \) \textbf{in} \( f \) \textbf{do}
        \begin{enumerate}
        \item add \( N' \) to \( EVI[(v, v')] \)
        \end{enumerate}
      \end{enumerate}
    \item else
      \begin{enumerate}
      \item \( \text{flag} \leftarrow \text{adjEnum}(\mathcal{MO}(|f|), N') \)
      \item if \( \text{flag} \) is \text{true} then
        \begin{enumerate}
        \item \( N'.parentNodeNo += \)
        \item push \( N' \) to \( IT [|f| - 1] \)
        \end{enumerate}
      \end{enumerate}
    \end{enumerate}
  \end{enumerate}
\item \begin{enumerate}
\end{enumerate}
\item remove \( (u, v) \) from \( f \)
\end{enumerate}

\( Adj(u') \). Then for each vertex \( v \) within refined candidate \( C_r(u) \), if \( v \) can be mapped to \( u \), we create a new trie node \( N' \) and points its \text{childN} to \( N \) (Line 10). We update \( f \) by adding \( (u, v) \) to it. And if \( u \) is the last leaf of \( dp_i \) according to the matching order, we update the \( EVI \) by adding the unverified edges of \( f \) to \( EVI \) (Line 14, 15) and set a
flag to true to indicate we find a embedding candidate in this recursive (Line 16). If \( u \) is not the last leaf, we further call the recursive function \( adj \ Enum \) to map the next leaf-vertex (Line 18) and set the flag to true if any embedding candidate can be found in deeper recursive. If there is any embedding candidate found extending \( f \), we increase \( parentNodeNo \) of the childnode and put the trie node created to \( IT \) (Line 19 to 21). We remove \((u, v)\) from \( f \) after trying to map \( v \) to \( u \) (Line 22).

### 4.6 Memory Control Strategies

In contrast to the performance optimization of the previous sections, this section focuses on the practicality of R-Meef. Since SubEnum still caches extra data graph and intermediate results in memory, the memory crisis is also a critical issue threatening the practicality of the SubEnum. We propose two techniques to optimize the memory usage of our approach: (1) maximum-caching which is to control the size of cached data graph and (2) region groups which contributes to controlling both the size of cached data graph and the cached intermediate results.

The maximum-caching is intuitive to understand. Before the expansion phrase of each round, we need to fetch the adjacency-lists of the foreign vertices which will be used to expand the partial embeddings found in a local machine. After the expansion of this round, we do not release those cached foreign vertices as they may be used in the next round. Only until there is no space available for new data to be cached, we release some cached foreign vertices from previous rounds or even from this round after the expansion is done. This strategy enables us to avoid unnecessary data transfer by using the available memory as much as possible.

In the rest of this section, we particularly focus on the technique of region groups. The concept of region is originally used in TurboIso[24] where the the whole searching process is divided into regions each of which starts from a given data vertex. While in SubEnum, we need to further group those regions into region groups. Recall that SubEnum is a multi-round approach where some old cached data are filtered and removed, some new ones are incrementally added. Therefore our goal is to ensure the size of cached data in each round is under the available memory \( \Phi \). \( \Phi \) can be set as a particular threshold in advance by user, otherwise SubEnum can detect and set all free memory as \( \Phi \). The key idea of SubEnum is to divide the candidates of the \( d_{p \_0 \_u \_pivot} \) into disjoint region groups and process the region groups separately. In this way, the overall cached data of the whole processing will be divided into the processing of each region group.
Assuming a naive grouping algorithm which randomly puts vertices into a group, there may be very few common foreign data vertices and edges that need to be fetched and be verified resp. when processing this group. This will lead to little workload being shared and therefore the overall performance of the subgraph enumeration will be jeopardized. Thus, a good grouping algorithm not only needs to achieve the aforementioned goal w.r.t memory usage but also requires the vertices within the same group should be close to each other so that the searching workload could be shared properly. This is a very challenging task. Before we present our grouping algorithm, let us study our method to estimate the space cost of cached data.

### 4.6.1 Estimating Cached Data

In SubEnum, we only need to take care of the intermediate results and the fetched foreign data vertices in each round, since the space cost of other data structures is trivial.

Let us consider a single candidate region of a particular data vertex $v_{reg}$. Starting by mapping $v_{reg}$ to $dp_{0..pivot}$, $\tilde{\mathbb{R}}_{G_i}(P_i)$ represents the embedding candidates of $P_i$ of round $i$ and $V_i^{for}$ represents the foreign vertices that need to be fetched of round $i$. The space cost of cached data in round $i$ is the sum of the space cost of $e_{\mathbb{R}G_i}(P_i)$ and $V_i^{for}$, which can be formulated as:

$$SP(i) = SP(\tilde{\mathbb{R}}_{G_i}(P_i)) + SP(V_i^{for})$$

**Component-1: $SP(\tilde{\mathbb{R}}_{G_i}(P_i))$** For easy implementation, we use a loose estimation by ignoring the compression of the inverted trie. We have:

$$SP(\tilde{\mathbb{R}}_{G_i}(P_i)) = |\tilde{\mathbb{R}}_{G_i}(P_i)| \times |V_{P_i}| \times \alpha$$

where $|\tilde{\mathbb{R}}_{G_i}(P_i)|$ is the number of embedding candidates in $|\tilde{\mathbb{R}}_{G_i}(P_i)|$ and $\alpha$ represents the space cost of each single inverted trie node\(^1\). $\tilde{\mathbb{R}}_{G_i}(P_i)$ is expanded from $\mathbb{R}_{G_i}(P_{i-1})$ of last round. For any $f \in \tilde{\mathbb{R}}_{G_i}(P_i)$, assume $d$ is the average degree of the mapped data vertex $v$ mapped to $dp_{i..pivot}$, we have:

$$|\tilde{\mathbb{R}}_{G_i}(P_i)| = |\mathbb{R}_{G_i}(P_{i-1})| \times \frac{d!}{(d - |dp_{i..Vleaf}|)!}$$

The above equation assumes that all the $|dp_{i..Vleaf}| - permutation$ on $Adj_{(reg)}$ can pass the local verification and be appended to inverted trie, which is also a loose estima-

\(^1\)It depends on the implementation and operating system.
tion. Recall again that \( R_{G_t}(P_{i-1}) \) is generated from \( R_{G_t}(P_{i-1}) \) of round \( i - 1 \) by filtering out failed candidates. There are \( E^{\text{ Shib}}_{dP_{i-1}} \) and \( E^{\text{ Anc}}_{dP_{i-1}} \) query edges whose mapped data edges should be verified from \( R_{G_t}(P_{i-1}) \). For any data edge that need to be verified, let us assume its success probability as \( \rho \). Then we have:

\[
|\overline{R}_{G_t}(P_{i-1})| = |\overline{R}_{G_t}(P_{i-1})| \times \rho^{(|E^{\text{ Shib}}_{dP_{i-1}}| + |E^{\text{ Anc}}_{dP_{i-1}}|)} \tag{4.6}
\]

Based on Equation 4.5 and Equation 4.6, we may defer the relationship between \( |\overline{R}_{G_t}(P_i)| \) and \( |\overline{R}_{G_t}(P_{i-1})| \). As a starting point of the region starting from \( v \), we hav:

\[
|\overline{R}_{G_t}(P_0)| = \frac{|Adj(\text{reg})|!}{(|Adj(\text{reg})| - |dp_0.\text{Vleaf}|)!} \tag{4.7}
\]

Based on the deferred relationship between \( |\overline{R}_{G_t}(P_i)| \) with \( |\overline{R}_{G_t}(P_{i-1})| \) and the starting point \( |\overline{R}_{G_t}(P_0)| \) in Equation 4.7, we can estimate the space cost of the \( SP(\overline{R}_{G_t}(P_i)) \) in \( i^{th} \) round.

**Component-2**: \( SP(V^\text{ for}_i) \), Assume \( d \) is the average degree of all data vertices, we simply compute \( SP(V^\text{ for}_i) \) by: \( SP(V^\text{ for}_i) = |V^\text{ for}_i| \times d \times \beta \) where \( \beta \) is the space cost for each vertex ID in the adjacency-list.

Assuming all data vertices mapped to \( dp_i.u_{\text{pivot}} \) in \( R_{G_t}(P_{i-1}) \) are not within the local machine, a naive loose estimation for \( |V^\text{ for}_i| \) is \( d\gamma \) where \( \gamma \) is the shortest-distance from \( dp_0.u_{\text{pivot}} \) to \( dp_i.u_{\text{pivot}} \) within \( P_{i-1} \). In order to remove those local data vertices, for each data vertex \( v' \) in \( G_t \), we pre-compute the longest shortest distance, denoted as \( BOR(v) \), between \( v \) with any border vertex \( v' \) in \( G_t \) (who has neighbour not residing in \( G_t \)). Then get \( |V^\text{ for}_i| = d\gamma - BOR(v_{\text{par}}) \). If \( \gamma \) is smaller than \( BOR \), \( |V^\text{ for}_i| \) is less than 1 which means \( v_{\text{par}} \) is further away from the border, and there is no need to fetch its neighbours which are in the local machine.

### 4.6.2 Grouping Regions

Now we are ready to present our grouping algorithm as shown in Algorithm 15. It takes the candidate vertex of \( dp_0.u_{\text{pivot}} \) as input and outputs region groups \( RG \). We use a \text{flag} to mark whether a vertex \( v \) is already added to a region group or not. We first pick an unadded vertex \( v \) and create a new group \( rg \) for it (Line 2). Then we use a map \( \theta \) to mark the closeness of the unadded vertices with the vertices already added to group \( rg \) (Line 4). If \( \theta \) is not empty, we get a largest \( v' \) within \( \theta \) and remove it from \( \theta \) (Line 5). By calling \text{underSpaceLimit}, we test whether it will exceed the space limit if adding \( v' \) to
Algorithm 15: REGIONGROUPS

Input: the pivot vertex $d_{p_0,u_{pivot}}$ and $M_t$
Output: region groups $RG = \{r_{g_0} \ldots r_{g_n}\}$
1 \[ flag[v] \leftarrow false \]
2 \[ for \ any \ v \ of \ M_t \ and \ flag[v] \ is \ false \ do \]
3 \[ create \ a \ new \ region \ group \ r_{g} \]
4 \[ init \ closeness \ map \ \theta, \ \phi[v] \leftarrow 0 \]
5 \[ while \ \theta \ is \ not \ empty \ do \]
6 \[ get \ and \ remove \ v' \ from \ \theta \ with \ largest \ value \]
7 \[ if \ underSpaceLimit(v', r_{g}) \ is \ false \ then \]
8 \[ break \ the \ while-loop \]
9 \[ add \ v' \ to \ r_{g} \]
10 \[ for \ each \ v'' \in Adj(v') \ and \ flag[v''] \ is \ false \ do \]
11 \[ if \ v'' \ not \ in \ \theta \ then \]
12 \[ \theta[v''] \leftarrow 0 \]
13 \[ else \ \theta[v''] \leftarrow \theta[v''] + 1 \]

$rg$ (Line 6). If it exceeds, we end this loop and this region group is finalized (Line 7,8). If it does not, we add $v'$ as a new member to $rg$ (Line 9). For each unadded neighbour $v''$ of $v'$, we put $v''$ to $\theta$ with initial value 0 if $v''$ is not in $\theta$. Otherwise we increase the value of $v''$ within $\theta$.

The key idea of Algorithm 15 is to ensure the data vertices falling into the same group are as close as possible to each other. The Subroutine is based on the aforementioned estimation of cached data.

Remark The task of estimating the size of cached data is a challenging task and our estimation strategy and grouping algorithm cannot guarantee the cached data never leak. A simple solution of handling such leaking cases is to break the current region group into small region groups when we encounter any leaks. To be specific, if we encounter a leak when processing region group $rg$, we first evenly break the $rg$ into two parts $rg_i$ and $rg_j$ and remove all the intermediate results originating from the vertices in $rg_j$. Then we continue the searching process only for $rg_i$. If we further encounter any leaks, we break $rg_i$ again and restrict the searching process to an even smaller region group. Once the smaller group is finished, we re-process the other small groups generated along the way. This solution may have some duplication problems, while the leaking cases are very rare since our estimation strategy is very loose. And we have a tuning parameter to control the estimation to make sure the the number of leaking cases is minimized.
4.7 Experiment

In this section, we present our experimental study. We first experimentally compare the performance of join-oriented approach with the backtracking approach in triangle listing so as to demonstrate the intermediate result latency problem of join-oriented sub-graph enumeration approaches. Then we compare the performance of our system with existing state-of-the-art solutions.

Table 4.2 Profiles of datasets

| Dataset | $|V|$ | $|E|$ | Average degree | Type             |
|---------|-----|-----|----------------|------------------|
| Human   | 4675| 86282| 36.82          | Protein Interaction |
| Yeast   | 3112| 12915| 8.05           | Protein Interaction |
| DBLP    | 0.3M| 1.0M | 6.62           | Paper Citations   |
| RoadNet | 1.9M| 5.5M | 5.26           | Road Network      |
| LiveJournal | 4.8M| 12915| 68.9           | Blogging Community |

4.7.1 Intermediate Result Storage Latency

As aforementioned, backtracking based approaches incrementally and recursively enumerate all the possible embeddings one by one [34] where the data edges are verified immediately when to decide whether it can be added to the partial embedding or not. Each data edge may be verified many times during the process. While the join-oriented approaches first cache all the embeddings of two small subgraphs and join them to get all the embeddings of the united graph of these two small subgraphs. Join-oriented approaches sacrifice the memory so as to list all the embeddings by a traditional join approach.

Backtracking based TurboIso [24] has proved much faster then [56]. However it is hard to see whether latency of caching those intermediate results is a problem slowing down the process of searching. Here, we use a triangle listing task to compare the performance of join-oriented and backtracking approach. Triangle-listing is a naive case of subgraph enumeration where the techniques of degree filtering, order selection etc. play a trivial role in the listing process.

We tested the triangle-listing over four real datasets: Yeast, Human, Roadnet and DBLP whose profiles can be found in Table 4.2. The algorithm we used for backtracking is Ullmann[58]. Regarding the join-oriented triangle-listing algorithm, we first cache the embeddings of a star with two connected edges and the embeddings of a single edge.
in memory. And then we join them together using hashjoin. All the algorithms were implemented in C++. All the experiments were carried out under 64-bit Windows 7 on a machine with Intel i-5 CPU and 4GB memory.

Table 4.3 Triangle-Listing Time Cost (ms)

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Yeast</th>
<th>Human</th>
<th>Roadnet</th>
<th>DBLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backtracking</td>
<td>6.1</td>
<td>360.6</td>
<td>182.7</td>
<td>977.8</td>
</tr>
<tr>
<td>Join-oriented</td>
<td>74.2</td>
<td>764.4</td>
<td>3095.9</td>
<td>5258.7</td>
</tr>
</tbody>
</table>

We present the results in Table 4.3. As we can see, the time cost of backtracking algorithm for all the four datasets is much less than that of backtracking. For Yeast and Roadnet, the backtracking algorithms demonstrate more than 10 times faster than the Join-oriented approaches. As for the Human which is a denser graph than the others, which will make the verification of backtracking much slower, it still shows that the backtracking is 2 times faster than the Join-oriented.

Based on the above experiments results, we may conclude that the intermediate result storage latency is a big problem of the backtracking approach. Therefore, unpromising intermediate results should be filtered as early as possible so as to minimize the latency caused by those intermediate results.

4.7.2 Performance Comparisons

Next we present the performance comparisons of subgraph enumeration between our approach with two state-of-the-art subgraph enumeration solutions: Seed [32] and
Implementations: We implemented all components of PADS using C++ with g++11 as our compiler. We utilized the MPICH2 [23] as our MPI library. We used Boost Asio [48] library to implement the two daemon threads of PADS.

Datasets: We used four real datasets in our experiments: Human, DBLP, Roadnet and LiveJournal. The profiles of the datasets are given in Table 4.2. Human is a tiny graph and has never been used in distributed subgraph enumeration algorithms. The reason we included Human is to see the performance of distributed solutions of subgraph enumeration when the data graph can be loaded in memory. For each of the other three data graphs, we partition it using the graph partition package provided by Metis [29].

Cluster Configurations: We set up our cluster by using the Amazon EC2 computing resource. Our cluster consists of fifteen nodes whose type are all m4.large(2 Intel Xeon CPU, 8GB memory). As for Seed, we use one of the fifteen nodes as the master node and use the rest as slave nodes.

Query Set: We used the query set proposed by Qiao [40]. However we moved the $q_9$ from the original query set, because it contains one standalone vertex with degree as one. This standalone vertex is particularly picked by [40], which has performance disadvantage to the others. The query set is given in Figure 4.5.

Comparison Results: The results of Human, Roadnet, DBLP and LiveJournal are given in Figure 4.7, 4.6, Figure 4.8 and Figure 4.9, respectively. For each dataset, we report the time cost (in seconds) of the three algorithms when processing each of the eight queries. In the figures of results, the performances of Seed, Qiao and Pads are marked as empty, back slash lines and slash lines, respectively.

![Fig. 4.6 Query Performance over Roadnet dataset]
It is easy to see that the performance of Qiao and Pads are much better than that of Seed \textit{w.r.t} all the datasets and query sets. Although with the help of heavy index, Qiao is outperformed by Pads \textit{w.r.t} to many tests in the experiments.

For the Roadnet dataset, the performance of Pads are much better than both Qiao and
Distributed Subgraph Enumeration: A Practical Asynchronous System

Seed. Roadnet is a sparse data graph where the number of embeddings for the queries is much smaller than that of other datasets. Therefore most partial embeddings cannot lead to any final embeddings. While the filtering strategy of Pads plays a significant role is filtering those unpromising partial embeddings at early stage. This explains the big improvement of Pads over Seed and Qiao w.r.t Roadnet. For the Human dataset which is a denser data graph, Pads performed better than Qiao when processing $q_2$ to $q_5$ while worse when processing the rest. For the DBLP dataset, Pads demonstrated a worse performance when processing $q_4$ while had a better performance for the rest. Especially for queries $q_1$ and $q_2$, the improvement of Pads over Qiao is more than 20 times faster. For the Livejournal dataset, Pads delivered an improvement over Qiao when processing $q_1$, $q_2$, $q_3$ and $q_4$, while performed slightly worse when processing the rest.

In summary, Pads achieved a much better performance than the state-of-the-art join-oriented approach Seed for all datasets and query sets. Pads also outperformed Qiao, the heavy index solution, for all query sets when processing Roadnet and some query sets of other datasets.

4.8 Conclusion

In this chapter, we have presented a practical asynchronous subgraph enumeration system PADS. We discussed some problems of current existing subgraph enumeration solutions and better demonstrated the problem of intermediate result storage latency of join-oriented approaches. We proposed a region-grouped multi-round expand-verify-filter approach based on which we designed SubEnum, the core of PADS. We conducted extensive experiments to evaluate the performance of Pads. Compared with the state-of-the-art join-oriented solution, our system shows significant superiority in terms of query processing efficiency. Even compared with heavy indexed solution, our approach also has a better performance in many cases.
Chapter 5

Conclusion and Future Work

5.1 Conclusion

With the improvement of the computational power of new generation computers, graph is being widely adopted practically to model various application data. Efficient and effective graph data processing becomes an urgent requirement for graph data applications. A fundamental processing requirement for graph applications is graph pattern matching whose core problem is subgraph isomorphism. Subgraph isomorphism has been used in many practical areas where a fast processing performance is highly urgently demanded. This thesis speeds up the subgraph isomorphism search by providing three different approaches.

Our first approach is to exploit the neighbourhood relationships among the data vertices and convert the data graph into a smaller compressed graph. The vertex relationships proposed are syntactic containment, syntactic equivalence, query-dependent containment and query-dependent equivalence. Based on those relationships, we proposed the concept of compressed graph and designed the algorithm to compress the original graph. By querying on the compressed graph, we achieved a significantly improvement of the subgraph isomorphism search.

Our second approach is multiple-query optimization of subgraph isomorphism search where we process multiple queries at the same time so as to share the workload. We not only proposed an efficient method to detect useful common subgraphs but also designed a data structure to organize those common them. We presented a heuristic algorithm to compute the query execution order so that the cached results from previous queries can be reused by later queries. To control the size of the cached results in memory, we designed a novel data structure to compress them. We revised the existing sequential
subgraph isomorphism algorithms to utilize the cached results. Through extensive experiments over real datasets, we proved that our approach significantly improved the query processing time compared with sequential subgraph isomorphism search.

The third approach is focused on the subgraph isomorphism under distributed settings. We observed that there are still several critical issues existing in the current distributed subgraph isomorphism solutions. We experimentally identified the problem of intermediate result storage latency of join-oriented approaches. We designed a practical subgraph enumeration system **PADS** which follows the principles of multi-round and asynchronous and does not need any heavy index. We presented several efficient strategies to control the memory usage and optimize the performance of **PADS**. We conducted extensive experiments to evaluate the performance of **PADS**.

### 5.2 Future Work

For the research of subgraph isomorphism, we listed the following issues that need to be further investigated.

1. **Subgraph Isomorphism On Direct Graphs** In the real world, directed graphs are widely used in many domains. However most existing subgraph isomorphism algorithms assume the data graph and query graph are undirected. It is still questionable whether those algorithms can be applied to process directed graphs. Also it is valuable to investigate whether the direction property of directed graphs can be used to further speed up the subgraph isomorphism search.

2. **Estimating The Number Of Embeddings** As we already noticed that the subgraph isomorphism search is a slow process. Especially when the data graph is large, this exact pattern matching method is hard to be accepted in the real world. While in many scenarios, an estimated number of embeddings of a query graph in the data graph is enough for a user to make some decisions. Therefore it is an important issue of estimating the number of embeddings of a query graph in a data graph. In the memory control of our distributed approach, we proposed a simple method to estimate the embeddings starting from a data vertex. While it deserves more research in order to achieve a good estimation.
References


