Multi-Query Optimization for Subgraph Isomorphism Search

Xuguang Ren
School of Information and Communication Technology
Griffith University, Gold Coast Campus
xuguang.ren@griffithuni.edu.au

Junhu Wang
School of Information and Communication Technology
Griffith University, Gold Coast Campus
j.wang@griffith.edu.au

ABSTRACT

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 is not always the most efficient. In this paper, we study multi-query optimization for subgraph isomorphism search. We first propose a novel method for efficiently detecting useful common subgraphs and a data structure to organize them. Then 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. Extensive experiments verified the effectiveness of our solution.

Keywords

Multi-query optimization, subgraph isomorphism

1. INTRODUCTION

Given a data graph G and a query graph (a.k.a *graph pattern*) q, subgraph isomorphism search is to find all subgraphs of G that are isomorphic to q. Subgraph isomorphism is a fundamental requirement for graph databases, and is widely used as a basis for many other algorithms. The problem is known to be NP-complete, and many heuristic algorithms have been proposed to speed-up subgraph isomorphism search. These existing algorithms focus on single-query settings, where queries are isolated and evaluated independently.

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 [13]. In detecting criminal networks in social graphs [15] or finding special structures in biological networks [16], 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 [7], keys [5] and association rules [6] 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.

Contributions The main contribution of this paper is an effective solution to the multi-query optimization problem

Proceedings of the VLDB Endowment, Vol. 10, No. 3

for subgraph isomorphism search. Specifically,

(1) We introduce the concept of *tri-vertex label sequence* 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.

(5) We conduct comprehensive experiments to evaluate our techniques.

Organization We discuss related work in Section 2 and introduce the preliminaries in Section 3. In Section 4, we provide an overview of our approach. The subsequent sections present the details. Section 5 presents our methods for common subgraph computation. Section 6 presents the method to compute a good query execution order. Section 7 presents the data structure for caching intermediate results. Section 8 presents the strategies for computing the query answers utilizing the cached intermediate results. Section 9 reports our experiments. Section 10 concludes the paper.

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 [22], VF2 [3], GraphQL [10], TurboIso [9], QuickSI [20] and many others [21, 23]. Most of these algorithms follow the framework of Ullmann, with improved pruning rules and matching orders. An experimental comparison was given in [14]. A graph compression-based approach was introduced recently in [17]. 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 [19, 18]. 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.[8]) 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 [10]. `MQO` has also been studied for semi-structured data [2, 11]. For similar reasons, these methods are difficult to be adapted for our problem.

More recently, [13] studies `MQO` for SPARQL queries over RDF graphs, which is the most closely-related work to ours. The approach of [13] 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 [13] 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) [13] 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 ([13] 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 [13], 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) [13] 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 [17] 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 7. A more recent work

[1] 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 7. 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. QUBLE [12] 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 [12], we store the actual embeddings in main memory, and our data structure is very different from that used in QUBLE.

3. PRELIMINARIES

Both the dataset and the query set studied in this paper are *undirected labeled graphs.* The dataset contains a large graph G while the query set is a set of small graphs $Q = \{q_1, \ldots, q_n\}$. We use *query* to refer to a query graph.

Undirected Labeled Graph An *undirected labeled graph* is a data structure $G = (V, E, \Sigma, L)$, where *(1)* V is the set of vertices; *(2)* E is a set of undirected edges; *(3)* Σ is a set of vertex labels; *(4)* L is a function that associates each vertex v in V with a label $L(v) \in \Sigma$.

Subgraph Isomorphism Given two graphs $G_1 = (V_1, E_1, \Sigma_1, L_1)$ and $G_2 = (V_2, E_2, \Sigma_2, L_2)$, an *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 embedding f can be represented as a set of vertex pairs (u, v) where $u \in V_1$ is mapped to $v \in V_2$. If there is an embedding of G_1 in G_2, we say G_1 is *subgraph isomorphic* to G_2 and denote it by $G_1 \preceq G_2$. If $G_1 \preceq G_2$ and $G_2 \preceq G_1$, 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 $G_1 \preceq G_2$. 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 $\texttt{VCover}(f) \equiv \{f(v) | v \in V_1\}$, and call the vertices in $\texttt{VCover}(f)$ the *covered vertices* of G_2 by f.

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.*

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) $G' \preceq G_1$ and $G' \preceq G_2$.

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

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

4. 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 1.

Algorithm 1: MQO_{subiso}

```
Input: Data graph G and query set Q = {q_1, ..., q_n}
Output: All embeddings of q_i in G, ∀q_i ∈ Q
1  PCM = DetectCommonSubgraph(Q)
2  EOrder = QueryExecutionOrder(PCM)
3  Cache = ∅
4  for each q_i ∈ EOrder do
5      if q_i has PCM parents then
6          R = SubIsoSearch_Mqo(PCM, Cache, q_i, G)
7          for each parent q_j of q_i do
8              if all q_j's children are processed then
9                  Clear Cache(q_j)
10     else
11         R = SubIsoSearch(q_i, G)
12     if q_i has unprocessed children then
13         Add R to Cache(q_i)
```

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 no-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 $\sim$ 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.

5. 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 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

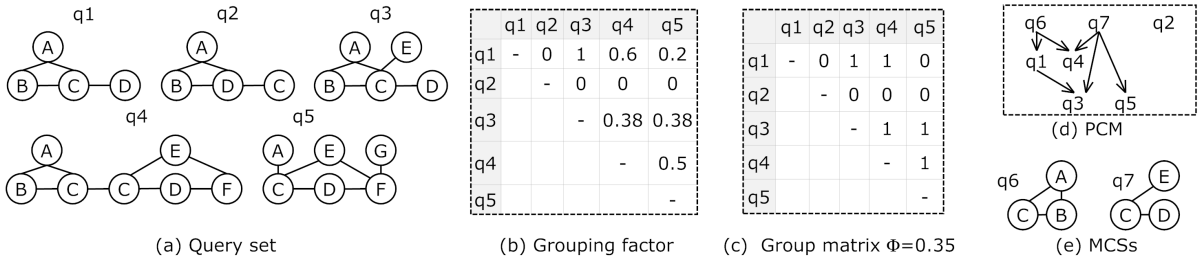


Figure 1: Building Pattern Containment Map

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.

5.1 Grouping Factor

We first define *tri-vertex label sequence*.

Definition 1 (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 $\texttt{TLS}(q)$ to denote the set of all TLSs in q. It is easy to verify $|\texttt{TLS}(q)| \leq \frac{1}{2}(|V_q| \times d_q(d_q - 1))$, 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 $\texttt{TLS}(q)$, there may be multiple instance subgraphs corresponding to it. For example, consider the subset $\{(A\text{-}B\text{-}C), (A\text{-}C\text{-}B), (B\text{-}A\text{-}C), (D\text{-}C\text{-}E)\}$ of TLSs of q_4 in Figure 1. The instances of the first three TLSs form an instance subgraph, and the instance of $(D\text{-}C\text{-}E)$ 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 $t.freq(q)$. The sum of the frequencies of all TLSs in q is denoted $\texttt{TLS}(q).size$. Given two graphs q_i and q_j, we use $\texttt{TLS}(q_i, q_j)$ to denote the set of all TLSs shared by q_i and q_j. That is, $\texttt{TLS}(q_i, q_j) = \texttt{TLS}(q_i) \cap \texttt{TLS}(q_j)$. Consider the graphs q_3 and q_4 in Figure 1. $\texttt{TLS}(q_3, q_4) = \{(A\text{-}B\text{-}C), (A\text{-}C\text{-}B), (B\text{-}A\text{-}C), (D\text{-}C\text{-}E)\}$. We use $\mathcal{LI}(q_i, \texttt{TLS}(q_i, q_j))$ to denote the number of instances in the largest instance subgraph of q_i corresponding to the TLSs in $\texttt{TLS}(q_i, q_j)$. For example, for q_3 and q_4 in Figure 1, $\mathcal{LI}(q_4, \texttt{TLS}(q_3, q_4)) = 3$ and $\mathcal{LI}(q_3, \texttt{TLS}(q_3, q_4)) = 4$.

Definition 2 (Grouping factor). *The grouping factor between two query graphs q_i and q_j, denoted $\mathcal{GF}(q_i, q_j)$, is defined as*

$$\mathcal{GF}(q_i, q_j) = \frac{min(\mathcal{LI}(q_i, \texttt{TLS}(q_i, q_j)), \mathcal{LI}(q_j, \texttt{TLS}(q_i, q_j)))}{min(\texttt{TLS}(q_i).size, \texttt{TLS}(q_j).size)} \quad (1)$$

The grouping factor has the following properties:

(1) $0 \leq \mathcal{GF}(q_i, q_j) = \mathcal{GF}(q_j, q_i) \leq 1$.

(2) If $q_i \preceq q_j$, $\mathcal{GF}(q_i, q_j) = 1$.

(3) If q_i and q_j do not have an MCS, $\mathcal{GF}(q_i, q_j) = 0$.

We will use the $\mathcal{GF}$ 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.

5.2 Pattern Containment Map

We give a formal definition of pattern containment map first.

Definition 3 (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 $q' \cong q$;

(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 $q' \not\cong q''$, $q \preceq q''$, and $q'' \preceq q'$.

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 1(d) shows an example PCM for the queries in Figure 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 $\mathcal{M}$ is given in Algorithm 2. 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 (1). If the grouping factor is larger than the given threshold Φ, we mark the element corresponding to q_i and q_j as 1 in the matrix.

Algorithm 2: TLSGroupMatrix

```
   Input: A query set Q={q_1, ..., q_n}
   Output: TLSGroupMatrix M of Q, threshold Φ
1  initialize M and set all cells to 0
2  for each q ∈ Q do
3  |  H(q) ← ComputeTLSSet(q)
4  for each pair q_i, q_j ∈ Q do
5  |  Compute GF(q_i, q_j)
6  |  if GF(q_i, q_j) > Φ then
7  |  |  M[q_i][q_j] ← 1
8  return M
```

With the query set and the matrix as input, Algorithm 3 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 $NextLevelGroup$ if there are odd number of queries). Lines 6 to 14 compute the MCSs for each pair, put the MCSs into $NextLevelGroup$, and add corresponding PCM edges (represented as a children list). The same process is repeated for queries in $NextLevelGroup$ 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 1. *For the queries in Figure 1(a), the grouping factors are shown in Figure 1(b). After applying the threshold Φ=0.35, the group matrix is shown in Figure 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 1(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_5 into $NextLevelGroup$, and compute the MCS of q_5 and q_7. The final* PCM *is shown in Figure 1(d).*

Complexity The TLS set building for each query is $O(nd^2)$ 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.

Algorithm 3: Build PCM

```
   Input: Query set Q, TLS group matrix M
   Output: Patten Containment Map PCM
1  Gr ← cliqueDetection(M)
2  for each group g ∈ Gr do
3  |  NextLevelGroup ← ∅
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 ← computeMCSs(q, q')
8  |  |  for each q_mcs ∈ Q_mcs do
9  |  |  |  if q_mcs ≅ q (assuming |q| ≤ |q'|) then
10 |  |  |  |  add q into NextLevelGroup
11 |  |  |  |  add q' into q.children
12 |  |  |  else
13 |  |  |  |  add q_mcs into NextLevelGroup
14 |  |  |  |  add both q, q' into q_mcs.children
15 |  |  if |NextLevelGroup| > 1 then
16 |  |  |  g ← NextLevelGroup
17 |  |  |  Repeat from line 3
18 mergeIsomorphicNodes(PCM)
19 transitiveReduction(PCM)
20 return PCM
```

As the query graphs are small, the MCS computation can be fast in practice.

6. 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.

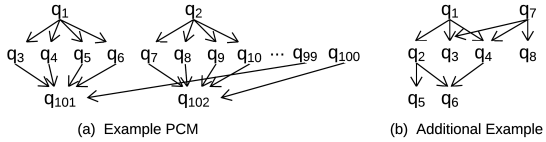


Figure 2: PCMs

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 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 ($\underline{q_1, q_3} \sim q_6, q_{99}, q_{101}, \underline{q_2, q_7} \sim q_{10}, q_{100}, q_{102}, \ldots$). 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 4, 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 4: QUERYEXECUTIONORDER

```
   Input: PCM of a query set Q, weights are initialized 0
   Output: A query execution order EOrder
1  Roots ← q ∈ PCM and q has no parents
2  q ← nextQueryGraph(Roots)
3  while q is not null do
4  |  Topo(q)
5  |  q ← nextQueryGraph(Roots)
   Subroutine Topo(query q)
1  |  if q has parents not added to EOrder then
2  |  |  changeParentsWeight(q)
   |  else
4  |  |  add q to EOrder, mark q as added
5  |  |  q' ← nextQueryGraph(q.children)
6  |  |  while q' is not null do
7  |  |  |  Topo(q')
8  |  |  |  q' ← 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 ζ)
1  |  if S ≡ {q ∈ ζ|q.added =
   |  false, q has no unadded parent} ≠ ∅} then
2  |  |  Choose q from S with the highest weight
3  |  |  return q
   |  else
5  |  |  return null
```

Algorithm 4 starts from grouping query graphs having no PCM parents into a *Roots* list (Line 1). **Subroutine** *nextQueryGraph* takes a *queryGraphList* ζ as parameter and returns a query q from ζ 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 4 iterates over all the queries in the *Roots* and calls a *Topo* subroutine for each of them (Line 3-5). In **Subroutine** *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 *Topo* (Line 6-8).

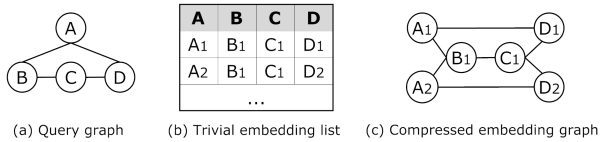
Example 2. *Consider the* PCM *in Figure 2(b). The Roots list is initialized as* $\{q_1, q_7\}$. *Starting from* q_1, *it proceeds to* q_2 *after* q_1 *is added.* q_2 *is also added because all of its parents have been added. Then it comes to* q_5 *and* q_6. *However only* q_5 *is added since* q_6 *has 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.*

Complexity Algorithm 4 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.

7. 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.

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(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.



(a) Query graph (b) Trivial embedding list (c) Compressed embedding graph

Figure 3: Trivial Structures for Caching Results

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(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 4 (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 4(a). The graphs in Figure 4(b) and (c) are partitions of G with partition widths of 2 and 3 respectively.

[1]For clarity, we use *node* to refer to the vertex of G', and *vertex* to refer to the vertex of G.

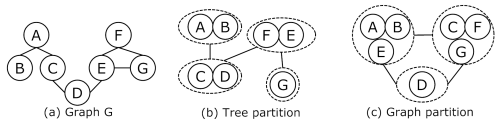


Figure 4: Graph Partitions

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 instance, for the partition shown in Figure 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 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 5 (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 4(a) and its tree partition Figure 4(b) and graph partition Figure 4(c). The embeddings of G are given in Figure 5(a), the CLLs based on these partitions are as shown in Figure 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.

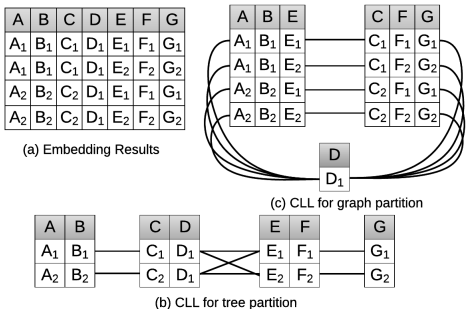


Figure 5: Cached Results

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 define $\chi(G') \equiv |E'| - |V'| + 1$ and call it *number of non-spanning tree edges* of G'. 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 non-spanning tree 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 1. *The BTLP problem is NP-complete.*

Proof sketch. 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 [4], 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 [4] (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 6 (a), the resulting MTP is shown in Figure 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 $\left\lceil \frac{|N|}{K} \right\rceil$ 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 $|N_1| = K$ randomly. This process is repeated until N is split into $\left\lceil \frac{|N|}{K} \right\rceil$ groups. For example, for the MTP in Figure 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 $\{u_5, u_6\}$.

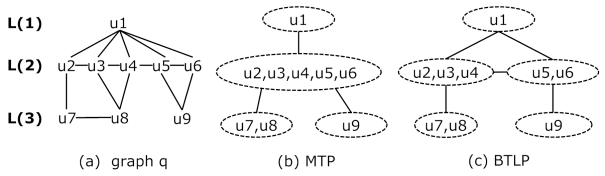


Figure 6: BTLP Steps

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.

8. 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 [14], 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) InitializeCandidates is to prepare promising candidates for each query vertex. (2) 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) 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, Σ_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 $n =$ 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*).

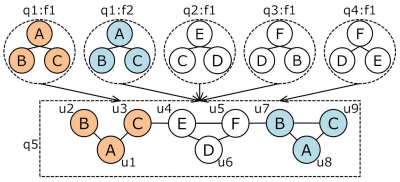


Figure 7: Query Covers

Consider the queries in Figure 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 $n_1 =$ VCover$(q_1 : f_1)$, $n_2 =$ VCover$(q_1 : f_2)$, $n_3 =$ VCover$(q_2 : f_1)$ and $n_4 =$ VCover$(q_3 : f_1)$. We have $E_P = \{(n_1, n_3), (n_3, n_4), (n_2, n_4)\}$.

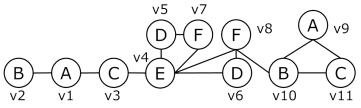


Figure 8: A Data Graph G

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 5,

Algorithm 5: BUILDINGJOINTGRAPH

Input: query q, $Parent(q)$- 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 $\quad f \leftarrow ChooseEmbedding(P)$
6 $\quad$ add f to P'
7 $\quad VCovered \leftarrow VCovered \cup$ VCover(f)
8 **return** P'

which is revised from the well-known *greedy algorithm* for set cover, to find a *good* subset P' of P.

Algorithm 5 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' untill 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 $ChooseEmbedding(P)$ (Line 5) uses these rules to choose the next embedding.

Example 3. *Consider the query q_5 and its parents in Figure 7. Suppose the data graph is G shown in Figure 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 5, 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 5, 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_{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 7 and the data graph G in Figure 8. q_1 is a parent of q_5, and both (v_1, v_2, v_3) and (v_9, 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_{parent} to q. Then any candidate of n (i.e., embedding of q_{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 7. Since v_2 (degree is 1) in Figure 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 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 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.

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.

9. EXPERIMENTS

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 [13]; (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 [9][14][17]. Wordnet was used in [21][17]. 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 1.

Table 1: Profiles of datasets

Dataset(G)	$\|V\|$	$\|E\|$	$\|\Sigma\|$	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[2] 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 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[9] and TurboIsoBoosted [17]. We also implemented a revised version for these two algorithms according to Section 8 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.

9.1 Grouping Factor Effectiveness

Table 2: Grouping Factor Effectiveness

Dataset	Yeast		Human		Wordnet	
	Tls	Jac	Tls	Jac	Tls	Jac
10%	2077	2288	4192	4218	8867	9134
30%	6337	8436	12577	13085	26625	27497
50%	12002	16600	20963	22109	45068	48920
70%	18018	26662	29348	30956	64847	73674
90%	24854	41170	37789	40464	89497	102027

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 2.

[2]We use "family" instead of "group" to distinguish it from the *groups* discussed in Section 5.

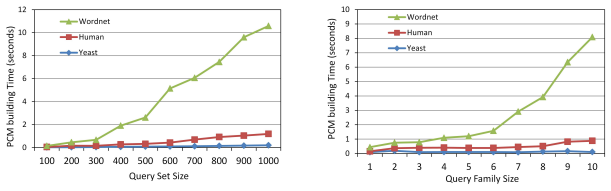
As we can see, for all the percentages, our grouping factor requires significantly fewer trials (note that computing MCS is an expensive process). The difference between the two methods is even larger with larger query sets, we omit the experimental results for larger query sets due to space limit. The time difference for computing the grouping factor and the Jaccard similarity is negligible due to the small size of query graphs.

9.2 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 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 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 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 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 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.



(a) Effect of query set size (b) Effect of family size

Figure 9: PCM Building Time

In Figure 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.

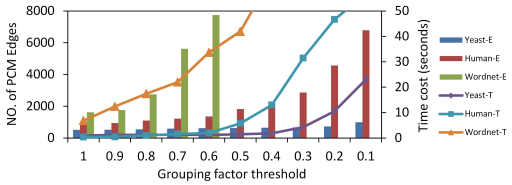


Figure 10: Effects of Grouping Factor Threshold

9.3 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. 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.

Table 3: Number of Cached Queries

Dataset	Yeast		Human		Wordnet	
	H	R	H	R	H	R
Peak	5	6	23	56	120	161
Average	1	1	6	8	23	89

9.4 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 4: Cache Memory Use in KB

Data\Width	1	2	3	4	5	Raw
Yeast	9	11	17	19	21	255×10^3
Human	24	137	358	672	1483	583×10^3
Wordnet	12	13	13	13	13	399×10^3

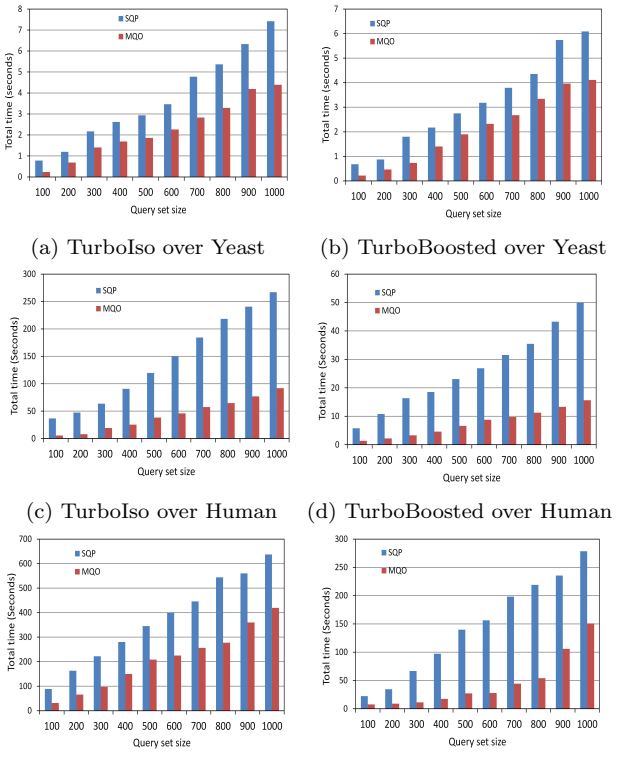
The memory cost of different types of cache results is given in Table 4. *Width* represents the partition width of the graph partition of the queries. When width is set to 1, it only allows one vertex in each partition node. This is equal to the trivial structure of compressed embedding graph. The last 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 5: Embedding Retrieval Time(ms)

Data\Width	1	2	3	4	5
Yeast	226	202	159	126	137
Human	3247	1628	952	858	770
Wordnet	3930	2314	1412	1400	1306

The embedding retrieval time is shown in Table 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.

9.5 Query Processing Time



(a) TurboIso over Yeast (b) TurboBoosted over Yeast

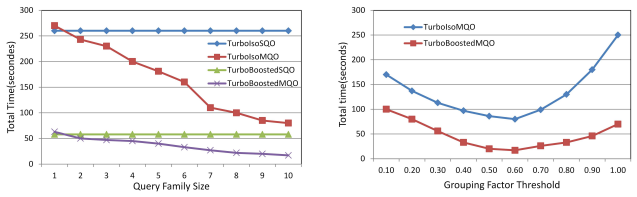
(c) TurboIso over Human (d) TurboBoosted over Human

(e) TurboIso over Wordnet (f) TurboBoosted over Wordnet

Figure 11: Performance Comparison and Scalability Test

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 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 12.

As shown in Figure 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).



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

Figure 12: Effects of query similarity and GF threshold

As shown in Figure 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 (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 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.

10. CONCLUSION

We presented a solution for MQO for subgraph isomorphism search in this paper. 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.

Acknowledgements. This work is supported by the Australian Research Council Discovery Grant DP130103051.

11. REFERENCES

[1] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. In *SIGMOD*, pages 1199–1214, 2016.

[2] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava. Navigation-vs. index-based XML multi-query processing. In *ICDE*, pages 139–150, 2003.

[3] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *Pattern Anal. Mach. Intell., IEEE Trans*, 26(10):1367–1372, 2004.

[4] A. Edenbrandt. Quotient tree partitioning of undirected graphs. *BIT Numerical Mathematics*, 26(2):148–155, 1986.

[5] W. Fan, Z. Fan, C. Tian, and X. L. Dong. Keys for graphs. *PVLDB*, 8(12):1590–1601, 2015.

[6] W. Fan, X. Wang, Y. Wu, and J. Xu. Association rules with graph patterns. *PVLDB*, 8(12):1502–1513, 2015.

[7] W. Fan, Y. Wu, and J. Xu. Functional dependencies for graphs. In *SIGMOD*, pages 1843–1857, 2016.

[8] S. J. Finkelstein. Common subexpression analysis in database applications. In *SIGMOD*, pages 235–245, 1982.

[9] W.-S. Han, J. Lee, and J.-H. Lee. TurboIso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*, pages 337–348, 2013.

[10] H. He and A. K. Singh. Query language and access methods for graph databases. In *Manag. and Min. Graph Data*, pages 125–160. 2010.

[11] M. Hong, A. J. Demers, J. E. Gehrke, C. Koch, M. Riedewald, and W. M. White. Massively multi-query join processing in publish/subscribe systems. In *SIGMOD*, pages 761–772, 2007.

[12] H. H. Hung, S. S. Bhowmick, B. Q. Truong, B. Choi, and S. Zhou. Quble: towards blending interactive visual subgraph search queries on large networks. *VLDB*, pages 401–426, 2014.

[13] W. Le, A. Kementsietsidis, S. Duan, and F. Li. Scalable multi-query optimization for SPARQL. In *ICDE*, pages 666–677, 2012.

[14] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *VLDB*, pages 133–144, 2012.

[15] M. Natarajan. Undertsanding the structure of a drug trafficing organization: a converational analysis. *Crime Preventon Studies*, 11:273–298, 2000.

[16] Y. Q and S. SH. Path matching and graph matching in biological networks. *J Comput Biol*, 14(1):56–67, 2007.

[17] X. Ren and J. Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *PVLDB*, 8(5):617–628, 2015.

[18] T. Sellis and S. Ghosh. On the multiple-query optimization problem. *TKDE*, (2):262–266, 1990.

[19] T. K. Sellis. Multiple-query optimization. *TODS*, 13(1):23–52, 1988.

[20] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 1(1):364–375, 2008.

[21] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5:788–799, 2012.

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

[23] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 3:340–351, 2010.