Exploiting Vertex Relationships in Speeding up Subgraph Isomorphism over Large Graphs

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

Subgraph Isomorphism is a fundamental problem in graph data processing. Most existing subgraph isomorphism algorithms are based on a backtracking framework which computes the solutions by incrementally matching all query vertices to candidate data vertices. However, we observe that extensive duplicate computation exists in these algorithms, and such duplicate computation can be avoided by exploiting relationships between data vertices. Motivated by this, we propose a novel approach, *BoostIso*, to reduce duplicate computation. Our extensive experiments with real datasets show that, after integrating our approach, most existing subgraph isomorphism algorithms can be speeded up significantly, especially for some graphs with intensive vertex relationships, where the improvement can be up to several orders of magnitude.

Keywords

Subgraph Isomorphism, Graph Adaptation, BoostIso

1. INTRODUCTION

The importance of graph data has long been recognized by industry as well as the research community. A fundamental processing requirement for graph data applications is subgraph isomorphism search. That is, in a given data graph, retrieve all subgraphs which are isomorphic to the query graph.

As well known, subgraph isomorphism is a NP-Complete problem [5], and extensive work has been done in trying to solve it in reasonable time for real datasets. Most 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 [9]. 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 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.
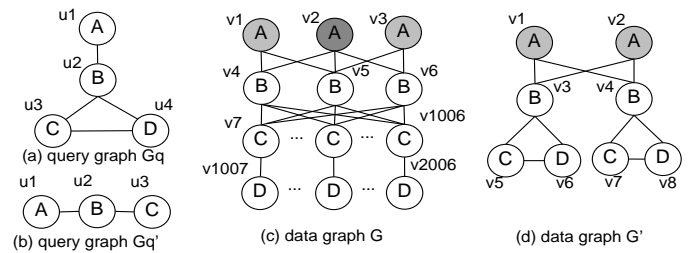

Figure 1: Example Query Graphs and Data Graphs

Example 1. *Consider the query graph G_q and the data graph G in Figure 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 2. *Consider the query graph G_q in Figure 1 (a) and the data graph G' in Figure 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 2 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 3. *Consider the query graph $G_{q'}$ in Figure 1(b) and the data graph G in Figure 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 3 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.

Contributions. We make the following contributions:

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 *adapted 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. We conduct extensive experiments to show the vertex relationships in realistic scenarios. Also 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.

Paper Organization. Section 2 discusses related work. Section 3 gives the preliminaries. Section 4 defines the four types of relationships between data vertices. Section 5 proposes the algorithm to transform the data graph into an *adapted graph* G_{sh}. Our new approach *BoostIso* is presented in Section 6. Section 7 presents the experiments. Section 8 concludes the pager.

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 [15], VF2 [3], QuickSI [12], GraphQL [7], SPath [17], STW [13] and TurboIso [6]. Most of them follow a backtracking framework. The techniques used to accelerate the matching process are *matching order optimization*, *efficient pruning rules* and *pattern-at-a-time strategies*, as briefly surveyed below.

Matching Order Optimization. The Ullmann algorithm [15] does not define the matching order of the query vertices. VF2 [3] 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 [12] proposes a matching order which accesses query vertices having infrequent vertex labels as early as possible. In contrast to QuickSI's global matching order selection, TurboIso [6] divides the candidates into separate candidate regions and computes the matching order locally and separately for each candidate region. Both STW [13] and TurboIso [6] give higher priority to query vertices with higher degree and infrequent labels.

Efficient Pruning Rules. The Ullmann algorithm [15] only prunes out the candidate vertices having a smaller degree than the query vertex. While VF2 [3] proposes a set of feasibility rules to prune out unpromising candidates, namely, 1-look-ahead and 2-look-ahead rules. SPath [17] uses a *neighbourhood signature* to index the neighbourhood information of each data vertex, and then prunes out false candidates whose candidate signature does not contain that of the corresponding query vertex. GraphQL [7] uses a pseudo subgraph isomorphism test. TurboIso [6] 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 [17] proposes an approach which matches a graph pattern at a time. The graph pattern used in SPath is path. TurboIso [6] 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 [10, 8, 2], graph summarization [11, 14], and query-preserving graph compression [4]. 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 [11] 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 [16], degree distribution, or user-specified node attributes [14]. More closely related to our work is [4], 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 adapted 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 adapted graph can be directly queried for subgraph isomorphism search. However, our adapted 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 adapted 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.

3. PRELIMINARIES

In this section, we review some fundamental concepts and the backtracking framework widely used to compute subgraph isomorphism.

Data Graph and Query Graph. A *data graph* is an undirected, vertex-labeled graph denoted as $G = (V, E, \Sigma, L)$, 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 $L(v) \in \Sigma$.

A *query graph* is an undirected, vertex-labeled graph denoted as $G_q = (V_q, E_q, \Sigma_q, L_q)$, where V_q, E_q, Σ_q, L_q have the same meaning as V, E, Σ, L of data graph G. In most cases, the query graph is much smaller than the data graph.

We assume the query graph and data graph are both connected, and will use *data vertices* (resp. *query vertices*) to refer to the vertices in the data graph (resp. query graph). In this paper, we only study undirected graph with vertex labels, but our approach can be applied to directed graphs as well.

Subgraph Isomorphism. Given a query graph $G_q = (V_q, E_q, \Sigma_q, L_q)$ and a data graph $G = (V, E, \Sigma, L)$, a *subgraph isomorphism* is an injective function f: $V_q \rightarrow V$ such that:

(1) $L_q(u) = L(f(u))$ for any vertex $u \in V_q$;

(2) For each edge $(u_i, u_j) \in E_q$, there exists an edge $(f(u_i), f(u_j)) \in E$.

f is also called an *embedding*. Note that f can be represented as a set of vertex pairs (u, v) in which $u \in V_q$ is mapped to $v \in V$ (We also say v *is matched to* u).

The Generic Framework Most subgraph isomorphism algorithms are based on a backtracking strategy which incrementally finds partial solutions by adding join-able candidate vertices. A recent survey [9] presents a generic framework for subgraph isomorphism search, which is shown in Algorithm 1.

Algorithm 1: GENERICFRAMEWORK

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     $C(u) \leftarrow initializeCandidates(G_q, G, u)$
     if $C(u) = \emptyset$ **then**
      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     $u \leftarrow nextQueryVertex()$
5     $refineCandidates(f, u, C(u))$
6   **for** *each* $v \in C(u)$ *and* v *is not matched* **do**
7   **if** *isJoinable(f, v, G, G_q)* **then**
8       $updateState(f, u, v, G, G_q)$
9       $subgraphSearch(G_q, G, f)$
10      $restoreState(f, u, v, G, G_q)$

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.

4. 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.

4.1 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 $Adj(v_j) - \{v_i\} \subseteq Adj(v_i) - \{v_j\}$, where $Adj(v_i)$ is the neighbour set of v_i and $Adj(v_j)$ is the neighbor 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 4. *In the data graph G in Figure 1(c), $L(v_1) = L(v_2) = L(v_3)$. Also $Adj(v_1) = \{v_4, v_5\}$, $Adj(v_2) = \{v_4, v_5, v_6\}$ and $Adj(v_3) = \{v_5, v_6\}$. Because $Adj(v_1) - \{v_2\} \subseteq Adj(v_2) - \{v_1\}$ and $Adj(v_3) - \{v_2\} \subseteq Adj(v_2) - \{v_3\}$, 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 and 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 $Adj(v_j) - \{v_i\} \subseteq Adj(v_i) - \{v_j\}$ and $Adj(v_k) - \{v_j\} \subseteq Adj(v_j) - \{v_k\}$. Combining these two formulas we get

$$Adj(v_k) - \{v_i\} - \{v_j\} \subseteq Adj(v_i) - \{v_k\} - \{v_j\} \quad (1)$$

There are three cases: (a) $v_j \notin Adj(v_k)$, (b) $v_j \in Adj(v_k)$ and $v_j \in Adj(v_i)$, (c) $v_j \in Adj(v_k)$ and $v_j \notin Adj(v_i)$. In the first two cases, we can easily infer $Adj(v_k) - \{v_i\} \subseteq Adj(v_i) - \{v_k\}$ from formula (1). That is, $v_i \succeq v_k$. Next we show the third case is not possible. This is because in this case $v_i \notin Adj(v_j)$, and $v_k \in Adj(v_j)$. Thus if $v_i \in Adj(v_k)$, then $Adj(v_k) - \{v_j\} \nsubseteq Adj(v_j) - \{v_k\}$, contradicting $v_j \succeq v_k$; and if $v_i \notin Adj(v_k)$, then $v_k \notin Adj(v_i)$, hence $Adj(v_j) - \{v_i\} \nsubseteq Adj(v_i) - \{v_j\}$, contradicting $v_i \succeq v_j$. $\square$

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. $\square$

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 5. *Consider the data graph G in Figure 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.*

4.2 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 $Adj(v_j) - \{v_i\} = Adj(v_i) - \{v_j\}$.*

Example 6. *Consider data graph G' in Figure 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.

4.3 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 $\mathrm{QDN}(G_q, u, v)$, is the set of data vertices $\{v_i | v_i \in Adj(v), L(v_i) \in \{L_q(u_i) | u_i \in Adj(u)\}\}$.*

Intuitively, $\mathrm{QDN}(G_q, u, v)$ is a subset of a 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 7. *Consider the query graph G_q in Figure 2(a) and the data graph G in Figure 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(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 $\mathrm{QDN}(G_q, u, v_j) - \{v_i\} \subseteq \mathrm{QDN}(G_q, u, v_i) - \{v_j\}$.*

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 8. *Consider the vertices v_3 and v_4 of data graph G in Figure 2(c) and vertex u_1 of query graph $G_{q'}$ in Figure 2(b). We have $\mathrm{QDN}(G_{q'}, u_1, v_3) = \{v_9, v_{10}, v_{13}, v_{14}\}$ and $\mathrm{QDN}(G_{q'}, u_1, v_4) = \{v_9, h_{13}, h_{14}\}$. Hence $\mathrm{QDN}(G_{q'}, u_1, v_3) \subset \mathrm{QDN}(G_{q'}, u_1, v_4)$. Therefore, we have $v_3 \succeq_{(G_q, u_1)} v_4$.*
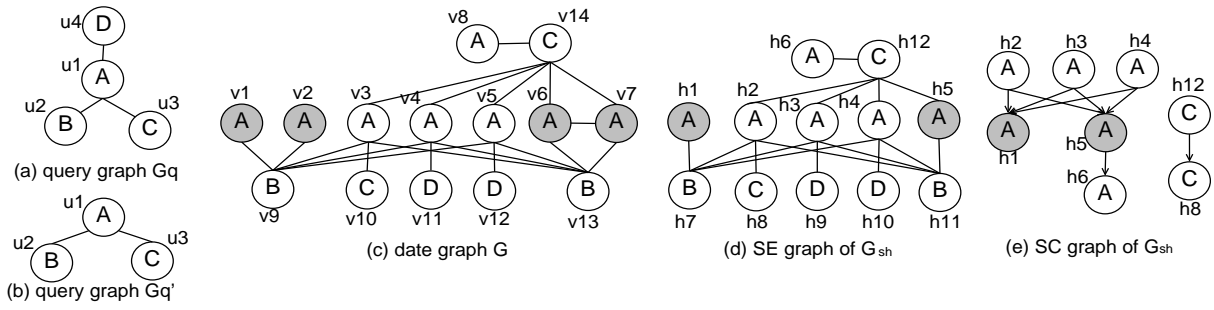
Figure 2: A Running Example

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.

4.4 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 $v_i \simeq_{(G_q,u)} v_j$, if $L(v_i) = L(v_j)$ and $\mathrm{QDN}(G_q, u, v_j) - \{v_i\} = \mathrm{QDN}(G_q, u, v_i) - \{v_j\}$.*

Clearly, query-dependent equivalence is two-way query-dependent containment. Using Proposition 4, we can infer that if $v_i \simeq_{(G_q,u)} v_j$, then for any embedding $f : G_q \to G$ 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 $v_i \simeq_{(G_q,u)} v_j$, 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.

5. GRAPH ADAPTATION

In this section, we present an algorithm to transform the data graph into an *adapted hypergraph* (or simply *adapted graph*) which is able to answer subgraph isomorphism more efficiently. We call this process *graph adaptation*.

5.1 Adapted 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 $SEC(v)$, 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, $Adj(v_i) - \{v_k\} = Adj(v_k) - \{v_i\}$. Therefore, if v_i and v_j are adjacent, that is, v_j is in $Adj(v_i)$, then v_j is also in $Adj(v_k)$, hence v_j and v_k are also adjacent. □

Proposition 5 implies that the data vertices in the same $SEC(v)$ 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 (**Adapted hypergraph**). *Given a data graph $G = (V, E, \Sigma, L)$, the* adapted hypergraph *of G is a graph $G_{sh} = (V_{sh}, E_{se}, E_{sc}, \Sigma_{sh}, L_{sh})$, such that*

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

(b) E_{se} *is a set of undirected edges such that, an edge between h and h' exists iff $(v_i, v_j) \in E$, where $h = SEC(v_i)$ and $h' = SEC(v_j)$.*

(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 $h = SEC(v)$.*

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 and only if $v_1 \succeq v_2$ for every pair of vertices where $v_1 \in h_1$ and $v_2 \in h_2$. 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 9. *Consider the data graph in Figure 2(c). We show the adapted hypergraph G_{sh} in two parts: SE graph in Figure 2(d) and SC graph in Figure 2(e). In Figure 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 adapted 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 6. *Suppose G_{sh} is the adapted 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.

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 6.

Algorithm 2: COMPUTE ADAPTED GRAPH

Input: Data graph $G = (V, E, \Sigma, L)$
Output: Adapted 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* $v' \in 1\text{-}step(v)$ *and* $L(v') = L(v)$ **do**
8   **if** $v \simeq v'$ **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* $v' \in 2\text{-}step(v)$ *and* $L(v') = L(v)$ **do**
13   **if** $v \simeq v'$ **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 $R(h) \leftarrow \{h' | h' \in adj(h) \cup 2\text{-}step(h),$
20 $L_{sh}(h) = L_{sh}(h')\}$
21   **for** *each* $h' \in R(v)$ **do**
22   **if** $h \succeq h'$ **then**
23 $E_{sc} \leftarrow E_{sc} \cup (h, h')$
24 $transitiveReduction(V_{sh}, E_{sc})$
25 return $G_{sh} = (V_{sh}, E_{se}, E_{sc}, \Sigma_{sh}, L_{sh})$

5.2 Building Adapted Graph

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

Algorithm 2 first assigns Σ to Σ_{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 *isCliques* 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 $2\text{-}step(v)$. Otherwise the algorithm will try to find S-equivalent vertices in $2\text{-}step$ 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 15~17). 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 $h \succeq h'$, 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 [1].

Example 10. *Consider the data graph G in Figure 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 $v_2 \simeq v_1$, 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(d)), SC graph will be built. We have $Adj(h_1) - h_2 \subseteq Adj(h_2) - h_1$, $h_2 \succeq h_1$, thus (h_2, h_1) is added to E_{sc}. Because $h_2 \succeq h_5 \succeq h_6$, the SC edge between h_2 and h_6 is removed by the transitive reduction. The final SC graph is shown in Figure 2(e).

Complexity. For a vertex $v \in V$, we use $2\text{-}step\text{-}SL(v)$ 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 $|2\text{-}step\text{-}SL(v)|$ 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 [1]. 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 $O(|V| \times N \times d + |E| + \Sigma_{l \in \Sigma} N_l{}^3)$.

6. 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 adapted 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.

6.1 Finding Hyperembeddings in G_{sh}

Our approach for finding the hyperembeddings follows the same framework as described in Algorithm 1 with the following modifications. (1) The *isJoinable* function is revised to allow multiple query vertices to be mapped to the same

hypernode in G_{sh}. (2) To make use of the SC relationships captured by the directed edges, we use a *dynamic candidate loading* strategy, that is, in *initializeCandidates*, we initialize $C(u)$ with the hypernodes labeled with $L_q(u)$ and having no SC-Parents (SC-Parents of hypernode h refers to the hypernodes that have a directed edge to h). Then we upload a candidate $h' \in C(u)$ for testing only when its SC-parents (namely those nodes that have a directed edge to h') have all been found to be able to match to u. (3) A boolean return value is added to the subroutine *subgraphSearch* to facilitate the implementation of the dynamic candidate loading strategy.

The revised *isJoinable* function and the process for dynamic candidate loading are presented in detail below. Note that the algorithms in this section do not consider the QDC and QDE relations. The revised *subgraphSearch* will be presented in Section 6.3.

6.1.1 The Revised isJoinable Function

Algorithm 3: Revised ISJOINABLE

Input: G_{sh}, G_q, f, h and u
Output: true if (u, h) can be added to f, false otherwise

1 **for** *each* $u_i \in V_q$ **do**
2   **if** u_i *is mapped by* f **then**
3   **if** $f(u_i) \neq h$ **then**
4   **if** $(u_i, u) \in E_q$ *and* $(f(u_i), h) \notin E_{sh}$ **then**
5   **return false**
6   **else**
7   **if** $(u_i, u) \in E_q$ *and* $h.isClique$ *is* **false** **then**
8   **return false**
9   **if** $usedTimes(h) \geq |h|$ **then**
10   **return false**
11 **return** true

The revised *isJoinable* function is shown in Algorithm 3. It takes G_q, G_{sh}, a partial hyperembedding f, a hypernode h and a query vertex u as input, and checks whether (u, h) can be added to f. Lines 1 to 5 are the usual checking which ensures that the edge between u and each matched adjacent vertex u_i of u has a corresponding edge $(f(u_i), h) \in E_{sh}$. If (u_i, u) is an edge in G and u_i is already mapped to h, then h must be a clique (Lines 6∼8); and if any u_i is mapped to h already, then the number of times h is used in the partial hyperembedding f (denoted $usedTimes(h)$) must be less than the number of data vertices in h (Lines 9,10). The correctness of the revised function follows directly from the definition of a hyperembedding.

6.1.2 Dynamic Candidate Loading

Algorithm 4 outlines the *dynamic candidate loading* process. It uses a candidate h which is known to match u in at least one hyperembedding and the candidate list $C(u)$ as input. $ump(h_i)$ is used to track the number of unmatched SC-Parents of h_i. For each SC-Child h_i of h, the algorithm first initializes $ump(h_i)$ as the number of its SC-Parents if $ump(h_i)$ is not defined yet (Lines 1∼4). Then, it decreases $ump(h_i)$ by 1 (Line 5). If $ump(h_i)$ is decreased to 0, h_i will be added to the candidate list (Lines 6,7).

Algorithm 4: DYNAMICCL

Input: $C(u)$ and hypernode h matchable to u
Output: updated $C(u)$
1 $Ch(h) \leftarrow$ SC-Children(h)
2 **for** *each* $h_i \in Ch(h)$ **do**
3   **if** $ump(h_i)$ *does not exist* **then**
4 | $ump(h_i) \leftarrow$ SCGraphIndegree(h_i)
5 $ump(h_i) \leftarrow ump(h_i) - 1$
6   **if** $ump(h_i)$ *is 0* **then**
7 | add h_i to $C(u)$

Remark. Recall that in Algorithm 1, after $C(u)$ is initialized and refined, previous algorithms verify all the candidates in $C(u)$ one by one in sequential order. In contrast, the *dynamic candidate loading* in *BoostIso* allows us to dynamically load the candidate list based on the SC relationship, so that the candidates which are S-contained by other hypernodes will always be tested later than its SC-Parents. If any one of its SC-Parents fails to match u, then they will not be loaded and will not be tested.

Example 11. *Consider the query graph G_q in Figure 2(a) and the hypergraph G_{sh} in Figure 2(d),(e). Assume the matching order is u_1-u_2-u_3-u_4. $C(u_1)$ is initialized with $\{h_2, h_3, h_4\}$ in initializeCandidates. As h_2 cannot be matched to u_1 in any embedding, h_2's SC-Children h_1, h_5 and h_6 will not be loaded and will not be tested. For query graph $G_{q'}$ in Figure 2(b), all h_2, h_3, h_4 can match u_1, h_1 and h_5 will be dynamically added into $C(u_1)$ and will be verified.*

6.2 Utilizing QDC and QDE Relationships

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.

6.2.1 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 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, $\{h | h_i \simeq_{(G_q, u)} h\}$.

Table 1: DRT for $(G_{q'}, u_1)$ and G_{sh} in Figure 2

$Node$	QDC-Children	NumOfQDC-Parent	QDE-List
h_2	$\{h_3, h_4\}$	0	$\emptyset$
h_3	$\emptyset$	1	$\{h_4\}$

As the time consumed by building *DRT*s 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) [6] 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.

Example 12. *Consider the query graph $G_{q'}$ and the hypergraph G_{sh} in Figure 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 $h_3 \simeq_{(G_{q'}, u_1)} h_4$, 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 1.*

Algorithm 5: BUILDDRT

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** $h \simeq_{(G_q, u)} h_i$ **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** $h \succeq_{(G_q, u)} h_i$ **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'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).

6.2.2 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

NumOf`QDC`-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 9 to $usedTimes(h) \geq |h| + \Sigma_{h_i \in \texttt{QDE}-List(h)}|h_i|$, 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 *DynamicCL*, we need to put both the `SC`-Children and `QDC`-Children into $Ch(h)$. Also, h may have `QDE` hypernodes which contain `SC`-Children. That is, we need to modify Line 1 to $Ch(h) \leftarrow$ `SC`-$Children(h) \cup \bigcup_{h' \in \texttt{QDE-List}(h)}$ `SC`-$Children(h') \cup$ `QDC`-$Children(h)$. Besides, the initial value of $ump(h_i)$ should be changed to $SCGraphInDegree(h) +$ NumOf`QDC`-Parents(h) in Line 4.

6.3 Putting It All Together

Algorithm 6: Revised SUBGRAPHSEARCH

Input: Adapted graph G_{sh} of G and query graph G_q where $V_q = (u_1 \cdots u_n)$
Output: All embeddings of G_q in G
$subgraphSearch(G_q, G_{sh}, f)$

1    $flag \leftarrow$ false
2   **if** $|f| = |V_q|$ **then**
3      $finalEG(G_q, G_{sh}, DRT, f)$
4      $dynamicCL(C(u_n), f(u_n))$
5   **return** true
   else
7   u $\leftarrow nextQueryVertex()$
8      $refineCandidates(f, u, C(u))$
9   **for** *each* $h \in C(u)$ **do**
10   **if** $isJoinable(f, h, G_{sh}, G_q)$ **then**
11          $updateState(f, u, h, G_{sh}, G_q)$
12   **if** $subgraphSearch(G_q, G_{sh}, f)=$true **then**
13            $flag \leftarrow$ true
14          $restoreState(f, u, h, G_{sh}, G_q)$
15   **if** $flag =$ true **then**
16   u$' \leftarrow previousQueryVertex(u)$
17      $dynamicCL(C(u'), f(u'))$
18    $resetCL(C(u))$
19   **return** $flag$

Subroutine $finalEG(G_q, G_{sh}, DRT, f)$
1   **for** *each* $u \in V_q$ **do**
2      $R_u \leftarrow \{v|v \in f(u)\} \cup$ $\{v|v \in h, h \in$ `QDE`-$List(f(u))\}$
3    $enumberateEmbeddings(R_{u_1} \cdots R_{u_n})$

We are now ready to present the revised framework for subgraph isomorphism search. Algorithm 6 outlines the *revised* subgraphSearch subroutine with full integration of *BoostIso*. In Algorithm 6, a local variable *flag* is used to indicate whether the current partial embedding can lead to any successful embedding. Different from Algorithm 1 which simply reports the embedding once it finds one, Algorithm 6 generates all the final embeddings of G_q from f once a hyperembedding f is found, as shown in *finalEG* (Lines 2,3). The subroutine for generating the final embeddings is shown

at the bottom of the Algorithm and is self-descriptive (Note that if `QDE` and `QDC` relations are not considered, then Line 2 of $finalEG$ should be changed to $R_u \leftarrow \{v|v \in f(u)\}$). Each time a hyperembedding f is found, we also update the $C(u_n)$ for the last mapped query vertex u_n and return a true value to notify a hyperembedding is found (Lines 4,5). Once the recursive call for u ends, we update the candidate list of u's previous query vertex u' according to the query vertex matching order (Lines 15~17). After all the combinations containing the current partial embedding are verified, $C(u)$ will be reset to its initial state as in *initializeCandidates* and $flag$ will be returned (Lines 18,19).

Example 13. *Consider the query graph $G_{q'}$ in Figure 2(b) and the adapted graph G_{sh} in Figure 2(d),(e). The DRT for vertex u_1 is given in Table 1. The revised matching process is shown in Figure 3 where each straight arrow line represents* dynamic candidate loading *and each curved arrow line represents a returned value, being false if there is a X mark in the middle. $C(u_1)$ is initialized with $\{h_2\}$, while $C(u_2)$ and $C(u_3)$ are initialized with $\{h_7, h_{11}\}$ and $\{h_{12}\}$ respectively. After (u_3, h_{12}) is added into the partial embedding $\{(u_1, h_2), (u_2, h_7)\}$, h_8 will be loaded into $C(u_3)$ and a true value will be returned. Once all the candidates of $C(u_2)$ are verified, the process backtracks to h_2 with a true value returned. Because h_2 is successfully matched to u_1 in some hyperembeddings, h_3 will be loaded into $C(u_1)$ and will be tested. This process goes on until all the final embeddings are found.*
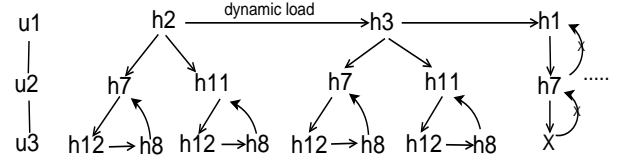

Figure 3: Example of revised subgraph search

7. 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.

7.1 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 [15], VF2 [3], QuickSI [12], TurboIso [6] and GQL (r=2) [7]. Ullmann is the first and canonical backtracking algorithm, QuickSI and GQL had the best overall performance among those compared in [9], and TurboIso is the state-of-the-art which outperforms all others, as reported in [6]. 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 [6][9], Wordnet was used in [13]. We obtained the Youtube, Email and DBLP datasets from Stanford Large Network Dataset 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 2.

Table 2: Profiles of datasets

Dataset(G)	$|V|$	$|E|$	$|\Sigma|$	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.

7.2 G_{sh} Statistics

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 hypenodes, 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 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 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

---

[1] http://snap.stanford.edu/data/

---

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 3: Statistics of G_{sh}

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

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 4, 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.
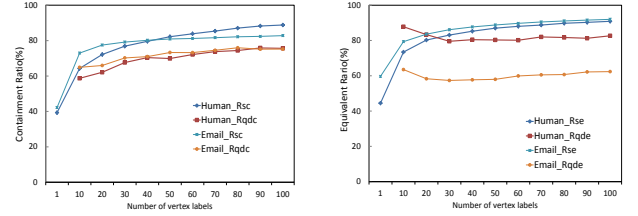
(a) Containment relationships   (b) Equivalent relationships

Figure 4: Relationship ratios with varying number of labels

7.3 Building Time and Scalability

The first column of Table 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 labels is 100. Table 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.

7.4 Efficiency of Query Processing

Measurements. As in [6][9], we measured the performance of an algorithm in two aspects: time elapsed and number of

Table 4: The Time for Building G_{sh}

Vertex number	0.1M	0.5M	1M	1.5M	2M
$T(min)$	0.5	3	7.3	11.7	19.28

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.
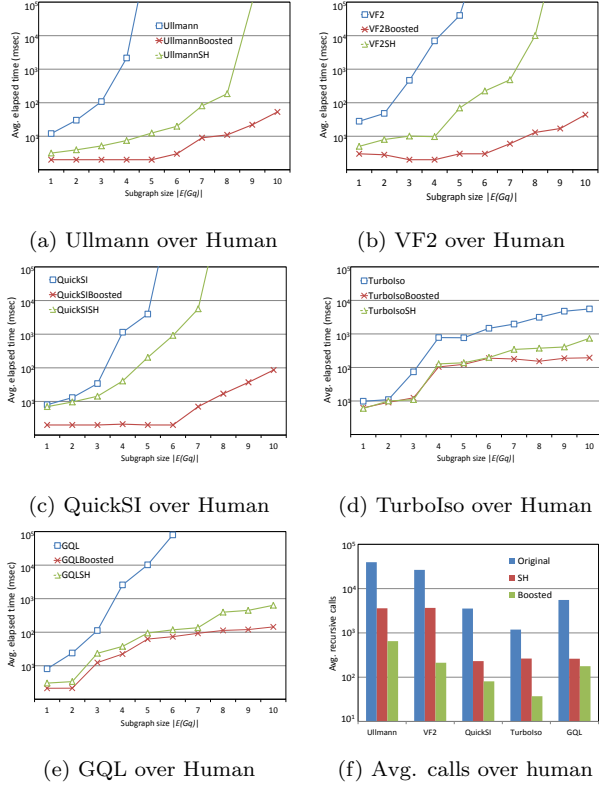

(a) Ullmann over Human

(b) VF2 over Human

(c) QuickSI over Human

(d) TurboIso over Human

(e) GQL over Human

(f) Avg. calls over human

Figure 5: Experiment results over Human

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 5(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, QuickSIBoosted 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-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 TurboIsoBoosted 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 5(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

(a) Ullmann over Yeast

(b) VF2 over Yeast

(c) QuickSI over Yeast

(d) TurboIso over Yeast
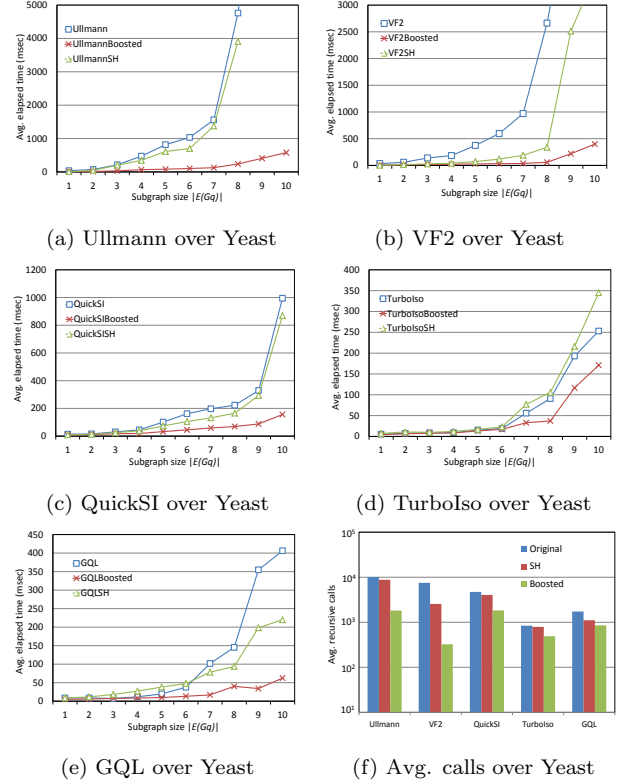
(e) GQL over Yeast

(f) Avg. calls over Yeast

Figure 6: Experiment results over Yeast

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 6(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 2 and Table 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 6(f), for Ullmann and VF2, the avg. recursive calls of the Boostd-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

(a) Ullmann over Wordnet

(b) VF2 over Wordnet

(c) QuickSI over Wordnet

(d) TurboIso over Wordnet

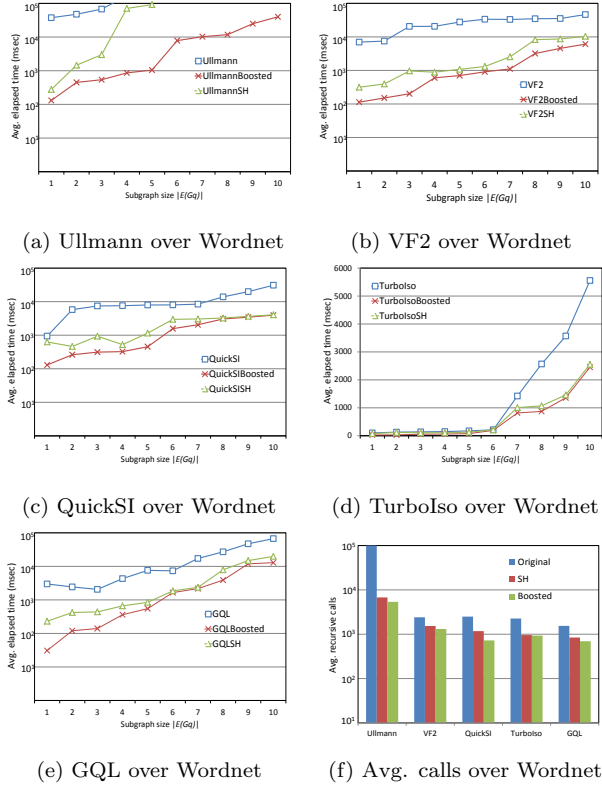(e) GQL over Wordnet

(f) Avg. calls over Wordnet

Figure 7: Experiment results over Wordnet

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 6(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 hypernodes. 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 7(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.

8. CONCLUSION

In this paper, 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 data sets 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 an extended version of this paper.

9. REFERENCES

[1] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM*, 1:131–137, 1972.

[2] Q. Chen, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *SIGMOD*, pages 134–144, 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] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *SIGMOD*, pages 157–168, 2012.

[5] M. R. Garey and D. S. Johnson. Computer and intractability. *A Guide to the NP-Completeness*, 1979.

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

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

[8] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*, pages 129–140, 2002.

[9] 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.

[10] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, pages 277–295, 1999.

[11] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph summarization with bounded error. In *SIGMOD*, pages 419–432, 2008.

[12] 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.

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

[14] Y. Tian, R. A. Hankins, and J. M. Patel. Efficient aggregation for graph summarization. In *SIGMOD*, pages 567–580, 2008.

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

[16] N. Zhang, Y. Tian, and J. M. Patel. Discovery-driven graph summarization. In *ICDE*, 2010.

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