Efficient Subgraph Matching by Postponing Cartesian Products

Fei Bi[†], Lijun Chang[†], Xuemin Lin[†], Lu Qin[‡], Wenjie Zhang[†]
[†]University of New South Wales, Australia
[‡]University of Technology, Sydney, Australia
f.bi@student.unsw.edu.au, {ljchang,lxue,zhangw}@cse.unsw.edu.au, lu.qin@uts.edu.au

ABSTRACT

In this paper, we study the problem of subgraph matching that extracts all subgraph isomorphic embeddings of a query graph q in a large data graph G. The existing algorithms for subgraph matching follow Ullmann's backtracking approach; that is, iteratively map query vertices to data vertices by following a matching order of query vertices. It has been shown that the matching order of query vertices is a very important aspect to the efficiency of a subgraph matching algorithm. Recently, many advanced techniques, such as enforcing connectivity and merging similar vertices in query or data graphs, have been proposed to provide an effective matching order with the aim to reduce unpromising intermediate results especially the ones caused by redundant Cartesian products. In this paper, for the first time we address the issue of unpromising results by Cartesian products from "dissimilar" vertices. We propose a new framework by postponing the Cartesian products based on the structure of a query to minimize the redundant Cartesian products. Our second contribution is proposing a new path-based auxiliary data structure, with the size $O(|E(G)| \times |V(q)|)$, to generate a matching order and conduct subgraph matching, which significantly reduces the exponential size $O(|V(G)|^{|V(q)|-1})$ of the existing path-based auxiliary data structure, where $V(G)$ and $E(G)$ are the vertex and edge sets of a data graph G, respectively, and $V(q)$ is the vertex set of a query q. Extensive empirical studies on real and synthetic graphs demonstrate that our techniques outperform the state-of-the-art algorithms by up to 3 orders of magnitude.

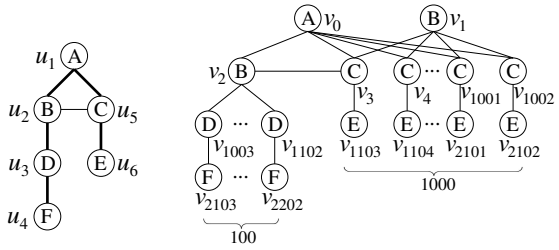
1. INTRODUCTION

In recent years, graph analysis has been playing an increasingly important role in the area of data analytics. Subgraph matching is one of the most fundamental problems in graph analysis. Given a query graph q and a large data graph G, the problem of subgraph matching is to extract all *subgraph isomorphic embeddings* of q in G. Subgraph matching has a wide range of applications including protein interaction network analysis [13], social network analysis [17], and chemical compound search [20].

Despite the NP-completeness of subgraph matching [5], recent research efforts lead to significant advances in developing computing paradigms to conduct subgraph matching [4, 8, 14, 15, 22]. A key issue is to reduce the number of unpromising intermediate results when iteratively mapping vertices one by one from a query graph to a data graph, especially to prevent generating unpromising immediate results caused by redundant Cartesian products. VF2 [4] and QuickSI [15] propose to enforce the connectivity to reduce redundant Cartesian products; that is, prevent generating an intermediate embedding, in a data graph for a query graph q, from two disjoint subgraphs q_1 and q_2 of q such that there are no edges connecting a vertex in q_1 and another vertex in q_2. To further reduce the chance of unnecessarily enumerating Cartesian products, $\text{Turbo}_{\text{ISO}}$ [8] proposes to merge together the *similar* vertices in a *query graph* q (i.e., the vertices with the same labels and the same neighborhoods), and [14] significantly extends the work of [8] to compress a *data graph* G by merging together the similar vertices in G to boost the performance of the technique in [8].

The second key issue is to generate an effective matching order for iteratively mapping vertices one by one from a query graph to a data graph with the aim to minimize the total number of intermediate results. QuickSI [15] proposes to generate a matching order based on the *infrequent-labels* first strategy. SPath [22] proposes to generate a matching order based on the *infrequent-paths* first strategy to resolve the limitations of only considering vertices and edges in [15]. The technique $\text{Turbo}_{\text{ISO}}$ in [8] proposes to exactly enumerate all paths to overcome the limitations in [22] that possibly overestimates the join cardinality by an estimation formula.

Challenges and Our Approaches. Our initial empirical study demonstrated that $\text{Turbo}_{\text{ISO}}$ [8] and its boost [14] can be very inefficient when the size of a query graph gets larger. This motivates us to develop new, more efficient and scalable techniques to conduct subgraph matching. Below are the two challenges that we will deal with in the paper.



(a) Query q (b) Data graph G

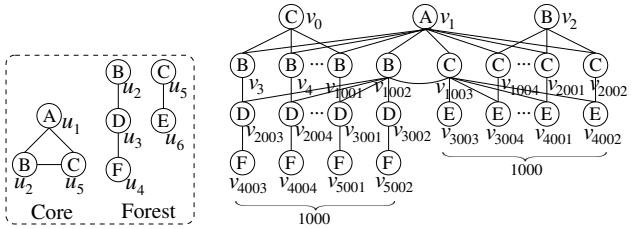
Figure 1: Challenge 1

Challenge 1: Redundant Cartesian Products by Dissimilar Vertices. Consider the query q in Figure 1(a) and the data graph G in Figure 1(b). It is immediate that q and G cannot be compressed by the techniques in [8] and [14], respectively, since *there are no sim-*

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2915236

ilar vertices in q or G (i.e., no vertices with the same labels and the same neighborhoods). Note that the recent techniques [8, 14, 15, 22] are all based on iteratively extending a mapping (i.e., embedding) from q to G along a spanning tree of q and also checking the non-tree edges adjacent to a newly extended vertex in q. Suppose that the spanning tree q_T of q is as depicted by thick lines in Figure 1(a); that is, (u_1, u_2), (u_2, u_3), (u_3, u_4), (u_1, u_5), and (u_5, u_6). The state-of-the-art edge-based ordering (i.e., QuickSI) and path-based ordering (i.e., $\mathsf{Turbo}_{\mathsf{ISO}}$) techniques will both choose the matching order in q_T as $(u_1, u_2, u_3, u_4, u_5, u_6)$. Consequently, the 100 partial mappings $(v_0, v_2, v_{1000+i}, v_{2100+i})$ $(3 \le i \le 102)$ of (u_1, u_2, u_3, u_4) have to be combined with the 1000 partial mappings (v_0, v_j) $(3 \le j \le 1002)$ of (u_1, u_5) before checking the mapping of the non-tree edge (u_2, u_5). Clearly, in this case the Cartesian product of $(100000 - 100)$ partial mappings are false positive and redundant.



(a) Decomposition of q (b) Another Data Graph

Figure 2: Our Approach

Our Approach: Postpone Cartesian Products. Regarding the above example, if we use the matching order $(u_1, u_2, u_5, u_3, u_4, u_6)$, then we can avoid this Cartesian product of 100×1000 partial mappings. In fact, the second matching order leads to only $100 + 1000$ partial mappings since an early checking of the non-tree edge (u_2, u_5) can eliminate the 999 partial mappings (v_0, v_j) $(4 \le j \le 1002)$ of (u_1, u_5) before extending the mapping to u_3 in q. This can be generalized into a new framework by decomposing a query graph into a dense subgraph (i.e., *core*) and a *forest* such that we process the core first; for example, Figure 2(a) illustrates such a decomposition of the query graph in Figure 1(a). Note that a dense query subgraph has a stronger pruning power (thus, reduces the number of unpromising partial mappings) while a forest potentially may have more mappings in G. Therefore, processing core first potentially postpones Cartesian products and thus reduces the chance to generate redundant Cartesian products.

Moreover, we can further postpone possible redundant Cartesian products by processing all leaf (i.e., degree-one) vertices (if exist) of a query in the last. For example, suppose that now the data graph is the one in Figure 2(b), we process the leaf vertices u_4 and u_6 of q in the last. In this case, the 1000 partial mappings of the subgraph of q induced by (u_1, u_2, u_5, u_3) will be generated. Nevertheless, we will leave their Cartesian product with the 1000 mappings of (u_5, u_6) to the last. The advantage of doing this is that such Cartesian product could be redundant if the query graph in Figure 1(a) has additional parts; thus it should be postponed, to be avoided.

Challenge 2: Exponential Size of The Path-based Data Structure in $\mathsf{Turbo}_{\mathsf{ISO}}$. The state-of-the-art approach, $\mathsf{Turbo}_{\mathsf{ISO}}$ [8], outperforms the other approaches due to its accurate calculation of join cardinality of a root-to-leaf query path in a spanning tree of a query q and the materialization of all embeddings in a data graph for each of such query paths so that they can be used to generate subgraph isomorphic embeddings. This immediately results in the exponential size $O(|V(G)|^{|V(q)|-1})$ of such path embeddings in the worst case, where $|V(G)|$ and $|V(q)|$ are the number of vertices in a data graph G and a query graph q, respectively (see Section 4.1 for more details). To resolve this, $\mathsf{Turbo}_{\mathsf{ISO}}$ [8] in its implementation only materializes k embeddings for each root-to-leaf query path to compute the matching order of query vertices, if we only retrieve k subgraph isomorphic embeddings for q. Note that such a materialization cannot always guarantee to generate k subgraph isomorphic embeddings for q. Thus, more path embeddings may be materialized on demand when enumerating subgraph isomorphic embeddings. Moreover, if we want to retrieve all subgraph isomorphic embeddings, then we have to materialize all path embeddings; that is, the worst-case exponential size is unavoidable in $\mathsf{Turbo}_{\mathsf{ISO}}$. As a result, $\mathsf{Turbo}_{\mathsf{ISO}}$ cannot scale to large queries or large data graphs.

Our Approach: Compact Path-based Data Structure with Polynomial Size. As shown in [8], the auxiliary path-based data structure can greatly speed up the computation of subgraph matching. To resolve the issue of the exponential size of the path-based data structure in $\mathsf{Turbo}_{\mathsf{ISO}}$, in this paper we propose not to enumerate and materialize all embeddings of query paths. Instead, we compute a data structure, called compact path-index (CPI), to store the candidates of embeddings of each query path. Since we generate subgraph isomorphic embeddings from CPI, it is immediate that the smaller the size of CPI, the more efficient conducting subgraph matching. We can show that while minimizing the size of CPI is NP-hard, the size of the CPI generated by our techniques is polynomial $O(|E(G)| \times |V(q)|)$ and our CPI construction algorithm runs in $O(|E(G)| \times |E(q)|)$ time, where $|E(G)|$ and $|E(q)|$ are the number of edges in G and q, respectively.

Moreover, in this paper we also propose to compute a matching order of query vertices based on a cost model and the CPI. Our experiments demonstrate that our ordering technique significantly outperforms the existing techniques.

Contributions. Our main contributions are summarized as follows.

- We develop a new framework with the aim to postpone Cartesian products. The new framework decomposes a query graph into a core and a forest for subgraph matching and proposes to deal with all leaf vertices of a query (if exist) in the last. We also develop an effective technique to compress the mappings of leaf vertices on the fly to avoid generating redundant Cartesian products.
- We design a compact auxiliary path-based data structure CPI with size $O(|E(G)| \times |V(q)|)$ for accurately estimating the number of embeddings of query paths and for generating subgraph isomorphic embeddings.
- While showing that minimizing CPI is NP-hard, we propose an efficient heuristic to build CPI in $O(|E(G)| \times |E(q)|)$ time.
- We develop efficient and effective algorithms for conducting subgraph matching.

Our extensive experiments on various dataset and query settings demonstrate that our techniques outperform the state-of-the-art techniques [8, 14] by up to 3 orders of magnitude, even excluding the cases when $\mathsf{Turbo}_{\mathsf{ISO}}$ [8] and its boost [14] cannot terminate.

Organization. The rest of the paper is organized as follows. A brief overview of related work follows immediately. Section 2 defines the problem of subgraph matching. Section 3 presents our new framework to postpone Cartesian products. We propose our CPI-based techniques in Section 4, while CPI construction techniques are presented in Section 5. Experimental results are reported in Section 6. We give a conclusion in Section 7. *Proofs are omitted due to space limits and can be found in Section A.1 in the Appendix.*

Related Works. Related works are categorized as follows.
1) Subgraph Matching over a Single Large Data Graph. The problem of subgraph matching over a single large graph has been studied for decades. The first result is Ullmann's algorithm [19] pro-

posed in 1976, which iteratively maps vertices one by one from a query graph q to a data graph G by following the input order of query vertices. To enhance the performance, *connected matching order* is used in VF2 [4] and QuickSI [15] that propose to generate the matching order by selecting a vertex connected to one of the already selected vertices rather than a random selection; this enables to prune false-positive candidates at an early stage, especially those caused by redundant Cartesian products. QuickSI [15] proposes to further remove false-positive candidates by first processing vertices and edges of q that are infrequent in G. GraphQL [9] and SPath [22] focus on reducing the candidates of query vertices by exploiting neighborhood-based filtering. As mentioned earlier, Turbo$_{ISO}$ [8] and the boost technique in [14] propose to merge similar vertices (i.e., vertices with the same labels and the same neighborhoods) in q and G, respectively. In this paper, we propose new efficient and effective techniques to scalably conduct subgraph matching. Firstly, we for the first time address the issue of unpromising results by Cartesian products from "dissimilar" vertices, in addition to that from similar vertices [8, 14]. Secondly, we propose to compute a connected matching order of query vertices based on a cost model and a compact path-based data structure rather than simple edge-frequencies in [15]. Thirdly, we propose a new light-weight candidate filtering technique to reduce the candidates of query vertices, in addition to the existing ones in [9, 22].

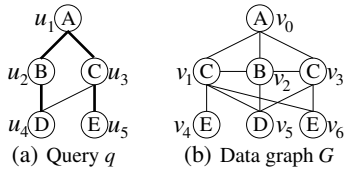
While the above techniques are based on the *depth-first paradigm* that is shown as the most efficient strategy to conduct subgraph matching on a single computer, the problem of subgraph matching has also been investigated in a distributed environment [11, 16, 18], where the *join paradigm* has been demonstrated as the most popular strategy. The goal of this paper is to develop novel techniques on a single computer to resolve the scalability issues in the existing techniques. Recently, the problem of similarity subgraph matching, which is to retrieve all subgraphs from a large data graph that are similar to a query graph, is also studied in the database community (e.g., [21, 24]). Our contributions in this paper may also be useful to study the problem of similarity subgraph matching.

2) Subgraph Containment Search over a Graph Database. The problem of subgraph containment search over a graph database is to identify the data graphs (from a graph database) that contain a query graph. This involves performing the subgraph isomorphism search over a graph database. To efficiently conduct this, many graph-feature based approaches have been proposed, following the filtering-and-verification framework, and can be classified into two categories: *frequent subgraph mining based approaches* (e.g., gIndex [20], Tree+Δ [23], and FG-Index [3]) and *exhaustive enumeration based approaches* (e.g., gCode [25], CT-Index [10] GraphGrepSX [2], and Grapes [7]). Although both subgraph containment search and subgraph matching involve subgraph isomorphism search, they are inherently different; the problem of subgraph matching, the problem we study in the paper, is harder since it requires enumerating all embeddings.

2. PRELIMINARIES

In this paper, we focus on a *vertex-labeled undirected graph* $g = (V, E, l, \Sigma)$. Here, V is the set of vertices, $E \subseteq V \times V$ is the set of edges, Σ is the set of labels, and l is a labelling function that assigns each vertex $v \in V$ a label in Σ (denoted $l_g(v)$). The number of vertices and the number of edges in g are denoted by $|V(g)|$ and $|E(g)|$, respectively. The set of neighbors of $v \in V(g)$ in g is denoted by $N_g(v) = \{v' \in V(g) \mid (v, v') \in E(g)\}$, and the degree of v, denoted $d_g(v)$, is the number of neighbors of v (i.e., $d_g(v) = |N_g(v)|$). Given a subset V_s of V, the subgraph of g induced by V_s is $g[V_s] = (V_s, \{(u, v) \in E \mid u, v \in V_s\}, l, \Sigma)$. In the following, for ease of presentation we simply refer a vertex-labeled undirected graph as a graph. Note that, our techniques can be readily extended to handle edge-labeled and directed graphs.

Definition 2.1: Given graphs $q = (V(q), E(q), l, \Sigma)$ and $G = (V(G), E(G), l, \Sigma)$, q is **subgraph isomorphic** to G if and only if there exists an *injective* mapping M from $V(q)$ to $V(G)$ such that $\forall u \in V(q)$, $l_q(u) = l_G(M(u))$ and $\forall (u, u') \in E(q)$, $(M(u), M(u')) \in E(G)$, where $M(u)$ is the vertex to which u is mapped. □



(a) Query q (b) Data graph G

Figure 3: Subgraph Matching

We call an injective mapping from vertices of q to vertices in G as a *subgraph isomorphic embedding* of q in G. For example, consider the graph q in Figure 3(a) and the graph G in Figure 3(b) where {A, B, C, D, E} is the set of vertex-labels, q is subgraph isomorphic to G since there is a subgraph isomorphic embedding $M(u_1 \rightarrow v_0, u_2 \rightarrow v_2, u_3 \rightarrow v_3, u_4 \rightarrow v_5, u_5 \rightarrow v_6)$.

Problem Statement. Given a query q and a large data graph G, in this paper we study the problem of *subgraph matching* which efficiently extracts all subgraph isomorphic embeddings of q in G.

For example, for the query graph q in Figure 3(a) and the data graph G in Figure 3(b), there are three subgraph isomorphic embeddings of q in G, which maps $(u_1, u_2, u_3, u_4, u_5)$ to $(v_0, v_2, v_1, v_5, v_4)$, $(v_0, v_2, v_1, v_5, v_6)$ and $(v_0, v_2, v_3, v_5, v_6)$, respectively.

In the remaining of this paper, we use the term "embedding" to refer to "subgraph isomorphic embedding" for simplicity when there is no ambiguity, and we may use *embedding* and *mapping* interchangeably. We also simplify $l_g(v)$, $N_g(v)$ and $d_g(v)$ as $l(v)$, $N(v)$ and $d(v)$, respectively, when the context is clear. We assume that both the query graph q and the data graph G are connected.

Notation	Description
q and G	Query and data graph
$V(q)$ and $E(q)$	Vertex set and edge set of q
$V(G)$ and $E(G)$	Vertex set and edge set of G
$l_g(v)$, $N_g(v)$ and $d_g(v)$	Label, neighbors and degree of v in g
V_C, V_T and V_I	Core-set, forest-set and leaf-set of q
q_T and r	Rooted (BFS) spanning tree of q and its root
CPI	Auxiliary data structure (compact path-index)
$u.C$	Candidates of a query vertex u in CPI
$N_u^{u'}(v)$	Adjacency list of v regarding (u', u) in CPI

Table 1: Notations

Frequently used notations are summarized in Table 1.

2.1 Existing Subgraph Matching Algorithms

The study of subgraph matching is initiated by Ullmann's backtracking algorithm [19], which iteratively maps vertices one by one from a query graph q to a data graph G by following the input order of query vertices. To enhance the performance, later algorithms in [4, 8, 14, 15] all enforce connectivity of the matching sequence/order. That is, given a spanning tree q_T of q, the matching order, $(u_1, \ldots, u_n)$, of q_T is constructed such that, for each vertex u_i except u_1, its parent $u_i.p$ in q_T is always before u_i in the matching order. This is called a *connected matching order*. For example, assume the spanning tree of the query graph in Figure 3(a) consists of edges (u_1, u_2), (u_2, u_4), (u_1, u_3), and (u_3, u_5), then a possible connected matching order is $(u_1, u_2, u_3, u_4, u_5)$ where $u_4.p = u_2$.

A subgraph matching algorithm grows an embedding M by mapping each query vertex u_i of q, according to the matching order, to a data vertex in G. When mapping u_i, the algorithm iteratively tries each data vertex v that is adjacent to the mapping $M(u_i.p)$ of $u_i.p$ as a candidate mapping of u_i; v is a *successful mapping* of u_i if 1) v has not been used in the current *partial embedding* and 2) it satisfies all the connection requirements specified by *non-tree edges* in the query (i.e., for each $(u_j, u_i) \in E(q)$ with $j < i$, $(M(u_j), v) \in E(G)$). A *full embedding* is obtained if every query vertex is mapped to a data vertex. For example, consider the above matching order for q and the data graph G in Figure 3(b). Assume a partial embedding M maps u_1, u_2, u_3, and u_4 to v_0, v_2, v_1, and v_5, respectively. Then, for mapping u_5, the algorithm iteratively tries v_4 and v_6 which are adjacent to $M(u_5.p)$ (= $M(u_3) = v_1$), to extend M.

Cost Model for Subgraph Matching Algorithms. In this paper, we adopt the cost model in [15] for computing our matching order; that is, the total cost of a backtracking algorithm for subgraph matching is $T_{iso} = B_1 + \sum_{i=2}^{n} \sum_{j=1}^{B_{i-1}} d_i^j (r_i + 1)$. Here, B_i (called the *search breadth*) is the total number of embeddings in G for the subgraph of q induced by $\{u_1, \ldots, u_i\}$, d_i^j is the number of neighbors of $M_{i-1}^j(u_i.p)$ in G with the same label as u_i where M_{i-1}^j is the j-th embedding in G for the subgraph induced by $\{u_1, \ldots, u_{i-1}\}$ and $M_{i-1}^j(u_i.p)$ is the vertex to which the parent $u_i.p$ of u_i in q_T maps, and r_i is the number of non-tree edges between u_i and vertices before u_i in the matching order. Intuitively, a partial embedding M_{i-1}^j of $(u_1, \ldots, u_{i-1})$ in G is extended by mapping u_i to each vertex v in G that is adjacent to $M_{i-1}^j(u_i.p)$ and has the same label as u_i (there are d_i^j such vertices); v is a successful mapping of u_i if it satisfies all connection requirements specified by the r_i non-tree edges of u_i.

Example 2.1: Given the matching order $(u_1, u_2, u_3, u_4, u_5)$ of q in Figure 3(a), $r_3 = 0$ while $r_4 = 1$. $M_2^1 = \{u_1 \to v_0, u_2 \to v_2\}$, then the neighbors of $M_2^1(u_3.p)$ (= v_0) are v_1 and v_3 and thus $d_3^1 = 2$. □

3. A NEW FRAMEWORK

In this paper, we propose a new framework for subgraph matching aiming at postponing the Cartesian products. We first decompose a query graph into three substructures, and then conduct subgraph matching in a substructure-by-substructure manner. In the following, we first define the core-forest-leaf decomposition.

Core-Forest-Leaf (CFL) Decomposition. The core-forest-leaf decomposition consists of core-forest decomposition and forest-leaf decomposition.

Core-Forest Decomposition. Edges of q can be categorized into two categories regarding a spanning tree q_T of q: edges in q_T are called *tree edges* while edges of q that are not in q_T are called *non-tree edges* regarding q_T. Our core-forest decomposition is to compute a small dense subgraph containing all non-tree edges regarding any spanning tree, which is defined as follows.

Definition 3.1: Given a query q, the **core-forest decomposition** of q is to compute the *minimal connected* subgraph g of q that contains all non-tree edges of q regarding any spanning tree of q; g is called the *core-structure* of q. □

The subgraph of q consisting of all other edges not in the core-structure is called the *forest-structure* of q, denoted T. We prove in Lemma 3.1 that the core-structure of q is exactly the 2-core of q, where a 2-core of q is the maximal subgraph of q such that every vertex in the subgraph has at least two neighbors in the subgraph. Note that, 2-core of q is a vertex-induced subgraph of q, and it is connected and unique [1].

Lemma 3.1: *The core-structure of q is the* 2*-core of q.* □

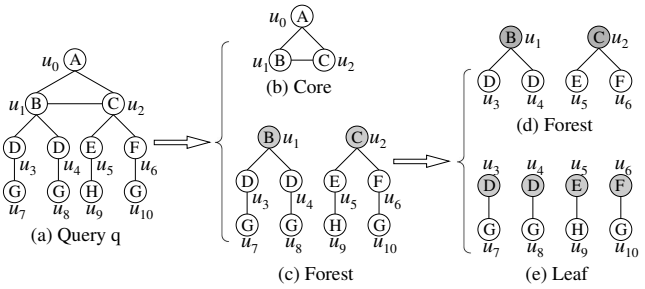


Figure 4: Core-Forest-Leaf Decomposition

We call the vertex set of the core-structure as the *core-set* and denote it as V_C; then, the core-structure is $q[V_C]$. One most important feature of the core-forest decomposition is that the core-structure is the minimal connected subgraph containing all non-tree edges of q regarding any spanning tree of q. As illustrated by *Challenge 1* in Introduction, a good matching order for q needs to conduct all non-tree edge checkings as early as possible, which will not only prune unpromising partial mappings but also reduce the total number of non-tree edge checkings. Thus, we put all vertices of V_C at the beginning of the matching order. Note that, the core-structure is a connected subgraph which is required to generate a connected matching order; this is vital for efficient subgraph matching [4, 15].

Compute Core-Forest Decomposition. Following Lemma 3.1, we compute the core-set V_C by *iteratively* removing all degree-one vertices from q, and the final set of remaining vertices is V_C. This process of iteratively removing degree-one vertices can be implemented in linear time regarding the query size (i.e., in $O(|E(q)|)$ time) [1]. For example, Figures 4(b) and 4(c) show the result of the core-forest decomposition for the query in Figure 4(a). Initially, $\{u_7, u_8, u_9, u_{10}\}$ is the set of degree-one vertices, and its removal creates new degree-one vertices $\{u_3, u_4, u_5, u_6\}$ which are then also removed. Finally, the core-set is $V_C = \{u_0, u_1, u_2\}$ as shown in Figure 4(b). Note that if q itself is a tree, the core-set is simply the root vertex of q, whose selection is discussed in Section A.6 in Appendix; it is possible that the entire query q is the core-structure.

One thing to notice is that, *the forest-structure consists of a set of connected trees and each connected tree in the forest-structure T shares exactly one vertex with the core-structure*. The shared vertex acts as the connection vertex between the tree and the core-structure. For example, u_1 and u_2 are shared by the core-structure and the two connected trees in Figure 4(c), respectively.

Forest-Leaf Decomposition. As illustrated by *Challenge 1* in Introduction, a good matching order also needs to postpone the Cartesian products caused by the candidates of all leaf vertices. Thus, we further decompose the forest-structure T, obtained by the core-forest decomposition, to a *forest-set* V_T and a *leaf-set* V_I, and put all vertices of V_I to the end of the matching order.

Definition 3.2: Given the forest-structure T, the **forest-leaf decomposition** is to compute the set V_I of leaf vertices of T by rooting each tree of T at its connection vertex. The set of other vertices of q not in $V_C \cup V_I$ is called the *forest-set* of q, denoted V_T. □

Thus, $V(q) = V_C \cup V_T \cup V_I$ and $V_C \cap V_T = V_C \cap V_I = V_T \cap V_I = \emptyset$. For example, Figures 4(d) and 4(e) are the results of the forest-leaf decomposition for the forest-structure T in Figure 4(c). In the following, we simply call the subgraph of T after removing all vertices of V_I and their associated edges from T the *forest-structure*.

The CFL-Decomposition based Framework. From the above core-forest-leaf decomposition for a query q, we obtain three vertex sets, the core-set V_C, the forest-set V_T, and the leaf-set V_I. We define a macro ordering of query vertices of q as (V_C, V_T, V_I); that

is, we first map vertices of V_C, then vertices of V_T, and finally vertices of V_I, to data vertices. Thus, we propose a new framework towards efficient and scalable subgraph matching, and its pseudocode is shown in Algorithm 1, denoted CFL-Match.

Algorithm 1: CFL-Match

Input: a query q and a data graph G
Output: the set $\mathcal{M}$ of all embeddings of q in G

1 $(V_C, V_T, V_I) \leftarrow$ CFL-Decompose(q);
2 CPI $\leftarrow$ CPI-Construct(q, G, V_C);
3 $\mathcal{M} \leftarrow \emptyset$;
4 **for each** *core embedding* M_C *in* Core-Match(V_C, CPI, G) **do**
5 **for each** *forest embedding* M_T *in* Forest-Match(V_T, CPI, M_C) **do**
6 $\mathcal{M} \leftarrow \mathcal{M} \cup (\{M_C\} \times \{M_T\} \times$ Leaf-Match(V_I, CPI, $M_C \cup M_T$));
7 **return** $\mathcal{M}$;

Given a query q, we first compute its core-forest-leaf decomposition (V_C, V_T, V_I) (Line 1), which has been discussed in above. Then, we build an auxiliary data structure, called compact path-index (CPI), for the query q regarding the data graph G (Line 2); the details of the CPI construction will be discussed in Section 5. Finally, based on the constructed CPI, we conduct core-match (Line 4), forest-match (Line 5), and leaf-match (Line 6), respectively; the details of these matching algorithms will be discussed in Section 4. Note that, each time when we invoke Core-Match or Forest-Match or Leaf-Match, it returns the next embedding; that is, to save memory space, only one embedding is generated each time.

Benefits of the CFL-Decomposition based Framework. Consider the query q in Figure 1(a) and the data graph G in Figure 1(b) in Introduction. For the matching order $(u_1, u_2, u_3, u_4, u_5, u_6)$ with $u_5.p = u_1$, the search breaths (see Section 2.1) are $B_1 = 1$, $B_2 = 1$, $B_3 = 100$, $B_4 = 100$, and $B_5 = 100$. Since only u_5 has a non-tree edge, the total cost (see Section 2.1) of subgraph matching regarding this matching order is $T_{iso} = B_1 + B_1 \times 1 \times 1 + B_2 \times 100 \times 1 + B_3 \times 1 \times 1 + B_4 \times 1000 \times 2 + B_5 \times 1 \times 1 = 200302$. Our core-forest-leaf decomposition based framework will generate the matching order $(u_1, u_2, u_5, u_3, u_4, u_6)$ which conducts the non-tree edge checkings at an earlier stage, then the cost becomes $T'_{iso} = 2302$ which is significantly smaller than T_{iso}. Moreover, it is immediate that the Cartesian products caused by leaf query vertices are put to the end of the subgraph matching process by our new framework.

4. OUR APPROACHES

In this section, we propose a new auxiliary data structure, called compact path-index (CPI), for efficiently conducting subgraph match. In the following, we first define the compact path-index (CPI) in Section 4.1, and then give our CPI-based core-match, forest-match, and leaf-match in Sections 4.2, 4.3, and 4.4, respectively, while the CPI construction algorithm will be presented in Section 5.

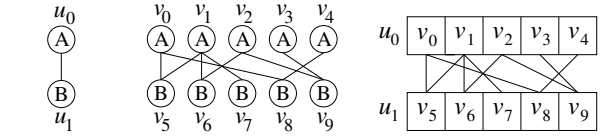
4.1 Auxiliary Data Structure

To compactly encode all possible embeddings of a query in a data graph, we propose a new auxiliary data structure, called compact path-index (CPI). CPI not only prunes false-positive candidates of query vertices, but also serves for the purpose of computing an effective matching order (see Section 4.2).

CPI Structure. Given a query q and a data graph G, CPI is defined regarding a *BFS tree* q_T of q and has the same structure as q_T. To differentiate the vertices of CPI from the vertices of q and G, we call vertices of CPI as *nodes*. Each node u in CPI carries the same label (i.e., $l_q(u)$) as in q_T. Similar to the parent-child relationships in q_T, any two adjacent nodes in CPI also have a parent-child relationship. The structure of CPI is as follows.

- Each node u of CPI has a *candidate set*, denoted $u.C$, which stores all vertices of G to which u can be mapped.
- There is an edge between $v \in u.C$ and $v' \in u'.C$ for adjacent nodes u and u' in CPI if and only if (v, v') exists in G.

Thus, constructing CPI is equivalent to computing a candidate set $u.C$ for every query vertex u of q.



(a) Query q (b) Data Graph G (c) CPI

Figure 5: Example CPI

For example, Figure 5(c) shows the CPI constructed for the query q in Figure 5(a) over the data graph G in Figure 5(b). The candidate sets of u_0 and u_1 are $u_0.C = \{v_0, \ldots, v_4\}$ and $u_1.C = \{v_5, \ldots, v_9\}$, respectively. The edges between vertices in $u_0.C$ and vertices in $u_1.C$ are exactly the same as in the data graph, as shown in the CPI in Figure 5(c). One thing to notice is that, a data vertex of G may appear in the candidate sets of multiple nodes in CPI. Our storage representation of CPI is discussed in Section A.2 in Appendix.

Soundness. Our goal is to directly use the CPI, constructed for a query q over a data graph G, for computing all embeddings of q in G. The data graph G is only probed for non-tree edge checkings (i.e., check whether there exists an edge in G between the mappings of the two end-points of a non-tree edge). To achieve this, the CPI must satisfy the following *soundness requirement*:

- *For every node u in* CPI*, if there is an embedding of q in G that maps u to v, then v must be in $u.C$.*

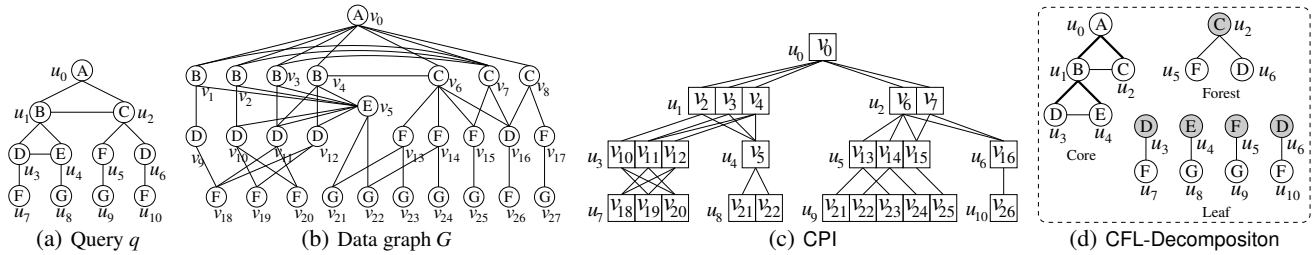
A CPI is *sound* if it satisfies the soundness requirement. Note that, although in the soundness requirement we only consider candidates of query vertices, the edges between candidates of parent-child query vertices are automatically included based on our CPI definition. Regarding a sound CPI, we have the following theorem.

Theorem 4.1: *Given a sound* CPI*, all embeddings of q in G can be computed by traversing only the* CPI *while G is only probed for non-tree edge checkings.* □

The Size of A Sound CPI. *A naive sound* CPI *can be constructed by letting $u.C$ to be the set of all vertices of G with label $l_q(u)$.* The naive CPI will contain a lot of false-positive candidates for query vertices, which will greatly affect the running time of a subgraph matching algorithm due to generating many partial embeddings that are eventually pruned. One natural goal is to build a minimum and sound CPI; that is, the total size of the candidate sets of query vertices in the CPI is minimum. However, this is NP-hard as shown in the lemma below.

Lemma 4.1: *It is NP-hard to build a minimum and sound* CPI. □

Nevertheless, the worst-case size of the CPI built for a query q over a data graph G is $O(|V(q)| \times |E(G)|)$, though there can be exponential number of embeddings of q in G. This is because, i) the size of the candidate set for a query vertex is at most $|V(G)|$, ii) for any pair (u, u') of parent-child nodes in CPI, the size of all the adjacency lists $N^u_{u'}(\cdot)$ is at most $|E(G)|$ since there is no duplicate edges, and iii) there are $|V(q)|$ query vertices and $(|V(q)| - 1)$ pairs of parent-child nodes. Moreover, for a data graph G with $|\Sigma|$ different labels, the average worst-case size of CPI is $O(\frac{|V(q)| \times |E(G)|}{|\Sigma|^2} + \frac{|V(q)| \times |V(G)|}{|\Sigma|})$. This is because the average size of the candidate set of a query vertex is $|V(G)|/|\Sigma|$, and the average total size of all adjacency lists corresponding to a pair of parent-child query vertices is $|E(G)|/|\Sigma|^2$ since



(a) Query q (b) Data graph G (c) CPI (d) CFL-Decompositon

Figure 6: A Running Example

the probability for an edge of G to be in an adjacency list in CPI is $1/|\Sigma|^2$.

In this paper, we aim to build a *small and sound* CPI, which will be discussed in detail in Section 5. For example, Figure 6(c) shows the CPI constructed by our algorithm for the query in Figure 6(a) over the data graph in Figure 6(b).

Remark. Turbo$_{\text{ISO}}$ [8] also proposes an auxiliary data structure to store all path embeddings in a data graph for all root-to-leaf paths of a spanning tree of the query. However, CPI is inherently different. One major difference is that, the data structure in Turbo$_{\text{ISO}}$ can be of exponential size (i.e., with space complexity $O(|V(G)|^{|V(q)|-1})$), and thus the construction time is also exponential. To resolve this, Turbo$_{\text{ISO}}$ [8] only materializes k embeddings for each root-to-leaf query path, where k is the number of subgraph isomorphic embeddings to be reported, when computing the matching order of query vertices; more path embeddings may be materialized on demand when enumerating subgraph isomorphic embeddings. However, we still have to materialize all path embeddings if we want to retrieve all subgraph isomorphic embeddings; that is, the worst case exponential size is unavoidable in Turbo$_{\text{ISO}}$. As a result, Turbo$_{\text{ISO}}$ cannot scale to large queries or large data graphs. Please see Section A.3 in Appendix for a detailed discussion about these issues.

4.2 CPI-based Core-Match

Given a sound CPI for a query q over a data graph G, all embeddings of q in G can be computed by traversing CPI while G is only used for non-tree edge checkings. Here, for presentation simplicity, we assume q itself is the core-structure; otherwise, we only consider the part of CPI corresponding to the core-structure of q. In the following, we first present our CPI-based matching order selection, and then describe the embedding enumeration approach.

4.2.1 CPI-*based Matching Order Selection*

Our algorithm for computing a matching order of query vertices is a *path-based ordering*. Recall that, CPI corresponds to a BFS tree of q, from which we can get a set of root-to-leaf paths $\{\pi_1, \ldots, \pi_k\}$ with all k paths sharing the root node of CPI. Our goal is to compute an effective ordering of the k paths.

The matching order *seq* of query vertices can be obtained from the ordering of paths as follows. Assume the paths are ordered as $(\pi_1, \ldots, \pi_k)$. We initialize *seq* as π_1, and then iteratively add vertices of π_i to *seq* for $i = 2, \ldots, k$. When adding vertices of $\pi_i = (v_{i_1}, \ldots, v_{i_y})$ to *seq*, it is easy to see that π_i shares a prefix with *seq*; we call the last shared vertex v_{i_x} as the *connection vertex* of π_i, and denoted $\pi_i.p$ in analogy to $u.p$ as discussed in Section 2.1. Then, we add vertices $v_{i_x+1}, \ldots, v_{i_y}$ of π_i to *seq*. For example, for the CPI in Figure 6(c), the BFS tree of the core-structure consists of three paths $\pi_1 = (u_0, u_1, u_3), \pi_2 = (u_0, u_1, u_4)$ and $\pi_3 = (u_0, u_2)$. Assume the ordering of paths is (π_2, π_1, π_3), then the matching order of query vertices is $(u_0, u_1, u_4, u_3, u_2)$.

Cost Analysis of Subgraph Matching via Path-based Ordering. Recall from Section 2.1 that, the total cost of a subgraph matching algorithm based on the matching order $(u_1, \ldots, u_n)$ is $T_{iso} = B_1 + \sum_{i=2}^{n} \sum_{j=1}^{B_{i-1}} d_i^j (r_i + 1)$ [15]. For a path-based ordering $(\pi_1, \ldots, \pi_k)$, assume the matching order is $(u_1, \ldots, u_n)$ and the position of the leaf (i.e. last) vertex of π_i in the matching order is l_i. Then, the matching cost is rewritten as,

$$\begin{aligned} T_{iso} &= B_1 + \sum_{i=2}^{l_1} \sum_{j=1}^{B_{i-1}} d_i^j + \sum_{x=2}^{k} \sum_{i=l_{x-1}+1}^{l_x} \sum_{j=1}^{B_{i-1}} d_i^j (r_i + 1) \\ &\approx \sum_{i=1}^{l_1} B_i + \sum_{x=2}^{k} \sum_{i=l_{x-1}+1}^{l_x} B_i (r_i + 1) \\ &\approx B_{l_1} + \sum_{i=2}^{k} B_{l_i} (r_{l_i} + 1) \end{aligned}$$

where the first equation follows from the fact that there is no non-tree edge among vertices on the same root-to-leaf path in a BFS tree, and the next two equations follow from two assumptions: 1) $\sum_{j=1}^{B_{i-1}} d_i^j \approx B_i$, and 2) $\sum_{i=l_{x-1}+1}^{l_x} B_i (r_i + 1) \approx B_{l_x} (r_{l_x} + 1)$. Note that, with the help of CPI, given a partial mapping $(v_1, \ldots, v_{i-1})$ of query vertices $(u_1, \ldots, u_{i-1})$, the partial mapping is extended by considering each vertex $v \in N_{u_i}^{u_j}(v_j)$ in CPI as a candidate of u_i, where u_j is the parent of u_i in CPI. The first assumption says that each extension of an embedding of $(u_1, \ldots, u_{i-1})$ based on $N_{u_i}^{u_j}(v_j)$ in CPI will lead to an embedding of $(u_1, \ldots, u_i)$; this is based on the fact that the pruning of non-tree edges has already been exploited in building the CPI (see Section 5). The second assumption naturally follows from the path-based ordering strategy; that is, we assume that the largest cost of mapping vertices of a query path determines the cost of mapping the path.

Greedy Approach to Ordering Paths. Note that the number of non-tree edges (i.e., r_i) between u_i and vertices before u_i in the matching order depends on the actual matching order. Since the total number of configurations of r_is is exponential (i.e., $O(|V(q)|!)$), it will be too expensive to optimize T_{iso} on the fly by considering r_i. Thus, we instead minimize an approximation of T_{iso}, $\tilde{T}_{iso} = \sum_{i=1}^{k} B_{l_i}$ (i.e., the total size of the search breadths of leaf vertices of q_T). Nevertheless, this is still a hard problem. We propose a greedy approach for ordering paths aiming to minimize $\tilde{T}_{iso}$.

The first path is the one with the minimum number of embeddings (i.e., $\arg\min_{\pi \in \mathcal{P}} c(\pi)$), where $\mathcal{P}$ is the set of all paths and $c(\pi_1, \ldots, \pi_i)$ is the number of embeddings in CPI for the tree formed by these paths. Given a set of chosen paths P, the next path is the one that together with P have the minimum number of embeddings (i.e., $\arg\min_{\pi \in \mathcal{P} \setminus P} c(P \cup \pi)$, or equivalently $\arg\min_{\pi \in \mathcal{P} \setminus P} \frac{c(P \cup \pi)}{c(P)}$). Here, $\frac{c(P \cup \pi)}{c(P)}$ can be estimated as $\frac{c(\pi^u)}{|u.C|}$, where $u = \pi.p$ is the connection vertex of π to P and π^u is the suffix of π starting from u. Intuitively, each embedding of P can be extended to $\frac{c(\pi^u)}{|u.C|}$ embeddings of $P \cup \pi$.

Estimate $c(\pi)$. Recall that π is a query path in CPI. Although the embeddings of π are not explicitly stored in CPI, we can estimate the number of such embeddings by a dynamic programming algorithm. We illustrate the dynamic programming algorithm by an example. Consider estimating the number of embeddings of the path $\pi = (u_0, u_1, u_3, u_7)$ in Figure 6(c). For each vertex $v \in u.C$ with $u \in \pi$, we compute $c_u(v)$, the number of embeddings in CPI for the subgraph of q induced by the suffix of π starting from u such that

u is mapped to v. Initially, $c_u(v) = 1$ for the leaf vertex u; that is, $c_{u_7}(v_{18}) = c_{u_7}(v_{19}) = c_{u_7}(v_{20}) = 1$. Then, we compute such numbers in a bottom-up fashion, $c_u(v) = \sum_{v' \in N^u_{u'}(v)} c_{u'}(v')$; for example, $c_{u_3}(v_{10}) = c_{u_7}(v_{19}) + c_{u_7}(v_{20}) = 2$. Finally, $c(\pi) = \sum_{v \in u.C} c_u(v)$ where u is the first vertex of π. The running time is linear to the total size of all adjacency lists corresponding to edges of π.

The Greedy Algorithm. The greedy algorithm for ordering all root-to-leaf paths in the BFS tree q_T of q is shown in Algorithm 2. The first path is the path with the minimum number of embeddings discounted by the number of non-tree edges of vertices on the path (Lines 2–3). Then, we iteratively select the next path, which is the one minimizing $\frac{c(\pi^u)}{|u.C|}$ where $u = \pi.p$ is the connection vertex of π to *seq* (i.e., the last shared vertex between π and *seq*) (Lines 4–6).

Algorithm 2: Matching-Order

Input: A BFS tree q_T of q and the corresponding CPI
Output: The matching order of query vertices

1 $\mathcal{P} \leftarrow$ all root-to-leaf paths in q_T;
2 $\pi^* \leftarrow \arg\min_{\pi \in \mathcal{P}} \frac{c(\pi)}{|NT(\pi)|}$; /* $NT(\pi)$: non-tree edges of π */;
3 Add vertices of π^* to *seq*; $\mathcal{P} \leftarrow \mathcal{P} \backslash \{\pi^*\}$;
4 **while** $\mathcal{P} \neq \emptyset$ **do**
5 $\quad \pi^* \leftarrow \arg\min_{\pi \in \mathcal{P}} \frac{c(\pi^u)}{|u.C|}$; /* $u = \pi.p$ */;
6 $\quad$ Add vertices of π^* to *seq*; $\mathcal{P} \leftarrow \mathcal{P} \backslash \{\pi^*\}$;
7 **return** *seq*;

Time Complexity. Let $\bar{N}^u_{u'}$ denote the average total size of adjacency lists corresponding to each pair of parent-child nodes in CPI. Then, the time complexity of Algorithm 2 is $O(\bar{N}^u_{u'} \times (\sum_{\pi \in \mathcal{P}} |\pi|))$, where $\mathcal{P}$ is the set of all root-to-leaf paths in CPI and $|\pi|$ is the number of edges in π. Note that $\sum_{\pi \in \mathcal{P}} |\pi|$ is guaranteed to be at most the number of leaf nodes in CPI multiplied by the height of CPI. This time complexity can be achieved by computing $c(\pi^u)$ for each suffix of $\pi \in \mathcal{P}$ in a bottom-up fashion, as described in *Estimate* $c(\pi)$ in above.

Example 4.1: Consider the CPI in Figure 6(c) and the core-structure in Figure 6(d). There are three root-to-leaf paths, $\pi_1 = (u_0, u_1, u_3)$, $\pi_2 = (u_0, u_1, u_4)$, and $\pi_3 = (u_0, u_2)$, with $c(\pi_1) = 4$, $c(\pi_2) = 3$, and $c(\pi_3) = 2$. Since $|NT(\pi_1)| = |NT(\pi_2)| = 2$ and $|NT(\pi_3)| = 1$, the first path is π_2. Then, $\pi_1.p = u_1$, $\pi_3.p = u_0$, $c(\pi_1^{u_1}) = 4$, and $c(\pi_3^{u_0}) = 2$. Thus, the second path is π_1 with $c(\pi_1^{u_1})/|u_1.C| = 4/3$. The third path is π_3, and the matching order is $(u_0, u_1, u_4, u_3, u_2)$. □

4.2.2 CPI-*based Embedding Enumeration*

Given a CPI and a matching order $(u_1, \ldots, u_{|V(q)|})$, the embeddings of q in G are enumerated by Core-Match, where the pseudocode is shown in Section A.4 (Algorithm 5) in Appendix. We iteratively map each query vertex u_i to a data vertex regarding the matching order. 1) If all query vertices are mapped (i.e., $i = |V(q)|+1$), then we output the embedding. 2) If this is the first query vertex (i.e., $i = 1$), then we map u_1 to a data vertex $v \in u_1.C$ in CPI. 3) Otherwise, u_i has a parent $u_i.p$ in the matching order, and $u_i.p$ has been mapped to $M(u_i.p)$; the candidates of u_i are obtained from the adjacency list $N^{u_i.p}_{u_i}(M(u_i.p))$ in CPI, and we map u_i to each candidate each time and proceed to the next query vertex u_{i+1}.

Remark. We do not compress the core-structure by the query graph compression technique in TurboISO [8]. This is because for randomly generated queries as tested in our experiments, their core-structures can hardly be compressed; for example, the core-structures can only be reduced on average by less than 1 vertex (i.e., most core-structures cannot be compressed); see Table 4 in Appendix.

4.3 CPI-based Forest-Match

Our CPI-based forest-match is similar to the core-match in Section 4.2, except that the forest-structure may consist of multiple connected trees. Also note that the forest-structure has no non-tree edges. Following the path-ordering strategy in Section 4.2, we first estimate the number of embeddings in CPI for each connected tree, and then order the connected trees in increasing order regarding their number of embeddings; the root-to-leaf paths in each connected tree are then ordered by Algorithm 2. In this way, we obtain the matching order of query vertices in the forest-structure.

The embedding enumeration algorithm for forest-structure is similar to Core-Match in Algorithm 5 except that we do not need to conduct non-tree edge checkings. Thus, *the data graph G is not probed for forest-match*. We omit the details of these algorithms.

Remark. We do not compress the forest-structure by the compression technique in TurboISO [8], because the forest-structure in a query q cannot be compressed based on the lemma below.

Lemma 4.2: *The compression technique in* TurboISO *[8] for compressing a query q cannot compress the forest-structure of q. That is, there is no two vertices in the forest-set that have the same labels and the same neighborhoods.* □

4.4 CPI-based Leaf-Match

In this subsection, we propose an efficient technique for enumerating all embeddings of the leaf-set V_I, given an embedding M_C of the core-set V_C and an embedding M_T of the forest-set V_T. Note that, $V_C \cup V_T \cup V_I = V(q)$.

Based on M_C and M_T, we first compute a candidate set $C(u)$ for each query vertex $u \in V_I$. Specifically, $C(u) = N^{u.p}_u(M(u.p)) \backslash (M_C \cup M_T)$; that is, $C(u)$ is the set of vertices in the adjacency list of $N^{u.p}_u(v)$ excluding those vertices being used in $M_C \cup M_T$, where $v = M(u.p)$ is the data vertex to which $u.p$ maps in $M_C \cup M_T$. Note that, $C(u) \subseteq u.C$ where $u.C$ is the set of candidates of u in CPI. For example, in Figure 6, $V_I = \{u_7, u_8, u_9, u_{10}\}$. Assume u_3 (= $u_7.p$), u_4 (= $u_8.p$), u_5 (= $u_9.p$), u_6 (= $u_{10}.p$) are mapped to $v_{12}, v_5, v_{13}, v_{16}$, respectively, then $C(u_7) = \{v_{18}, v_{19}\}, C(u_8) = \{v_{21}, v_{22}\}$, $C(u_9) = \{v_{21}, v_{23}\}, C(u_{10}) = \{v_{26}\}$.

Then, our technique is based on the following lemma.

Lemma 4.3: *For any two query vertices, u and u′, in V_I, if the labels of u and u′ are different (i.e., $l_q(u) \neq l_q(u')$), then $C(u) \cap C(u') = \emptyset$.* □

Thus, we partition query vertices of V_I into label classes depending on their labels. A *label class* with label a, denoted S_a, consists of all vertices of V_I with label a, and we denote the set of label classes of V_I by $\mathcal{S}$. After generating all embeddings of each label class S_a in $\mathcal{S}$, the embeddings of V_I can be obtained as a Cartesian product of embeddings for all different label classes. For example, continue the above example, there are two label classes, $S_G = \{u_8, u_9\}$ and $S_F = \{u_7, u_{10}\}$. After generating embeddings for these two label classes, $\mathcal{M}(u_8, u_9) = \{(v_{21}, v_{23}), (v_{22}, v_{21}), (v_{22}, v_{23})\}$ and $\mathcal{M}(u_7, u_{10}) = \{(v_{18}, v_{26}), (v_{19}, v_{26})\}$, we obtain $|\mathcal{M}(u_8, u_9)| \times |\mathcal{M}(u_7, u_{10})| = 3 \times 2 = 6$ embeddings of V_I.

Generate Embeddings for a Label Class. Firstly, we merge all vertices of S_a that have the same parents into neighborhood equivalence class (NEC) vertices. Note that, these NEC vertices are exactly the same as that obtained by the compression technique in TurboISO [8]; that is, a degree-one vertex can only be in the same NEC with another degree-one vertex. Moreover, all vertices in the same NEC have the same set of candidates. Thus, we will be dealing with this updated label class S_a which contains NEC vertices.

We sort all vertices of S_a in increasing order according to their numbers of candidates (i.e., $|C(u)|$), and then iteratively map each query vertex u to a data vertex in $C(u)$ according to this order. Note that, for an NEC vertex u' of cardinality $|u'|$ (i.e., containing $|u'|$ query vertices), we map u' to a combination, instead of a permutation, of $|u'|$ vertices from $C(u')$. The set of all embeddings can be obtained if needed, by permuting the mappings for query vertices in each NEC vertex.

Compared with the compression technique in $\mathsf{Turbo}_{\mathsf{ISO}}$ [8], we not only consider NEC vertices but also put all leaf query vertices with the same label together (i.e., label class). Note that, only query vertices with the same label may conflict with each other; that is, two query vertices with the same label are not allowed to be mapped to the same data vertex in the same mapping. Thus, by putting query vertices with the same label together, we are able to prune unpromising partial embeddings at early stages.

Remark. Note that, the concept of leaf-set V_I can be generalized to an *independent-set* (i.e., there is no edge between any two vertices in the independent-set) of query vertices of q. Nevertheless, we show that the leaf-set V_I is the maximal possible independent-set of the forest-structure of q; details are in Section A.5 in Appendix.

5. CPI CONSTRUCTION

In this section, we develop efficient techniques for constructing a small and sound CPI. Since constructing the smallest sound CPI is NP-hard as proved in Lemma 4.1, we propose a heuristic approach for constructing CPI in two phases: top-down construction (see Section 5.2) and bottom-up refinement (see Section 5.3), where both tree edges and non-tree edges are exploited to prune false-positive candidates of query vertices (see Section 5.4). In the following, we first present the general idea of our CPI construction algorithms in Section 5.1.

5.1 General Idea

The CPI is constructed regarding a BFS tree q_T of q. The selection of the root vertex r of q_T is discussed in Section A.6 in Appendix. Then, vertices of q are partitioned according to their BFS levels where the BFS level of a vertex of q_T is one plus its distance to r in q_T; edges of q are partitioned into tree edges and non-tree edges. The non-tree edges are further categorized as follows.

Definition 5.1: For a non-tree edge (u, u') in q regarding q_T, if u and u' are at the same BFS levels in q_T, then (u, u') is called a **same-level non-tree edge (S-NTE)**; otherwise (u, u') is called a **cross-level non-tree edge (C-NTE)**. □

For example, consider the query q in Figure 7(a). Assume u_0 is the root vertex, the BFS tree q_T is shown in Figure 7(b), and the non-tree edges are (u_1, u_2) and (u_2, u_3), where (u_1, u_2) is a S-NTE and (u_2, u_3) is a C-NTE.

Following Section 4.1, the general idea of our CPI construction is to compute the candidate set $u.C$ for each vertex u in q, while the induced edges in G between $v \in u.C$ and vertices of $u'.C$ are stored as an adjacency list $N^u_{u'}(v)$ in CPI, regarding the tree edge (u, u') of q_T. Thus, the main issue is to construct a sound CPI such that $u.C$ is as small as possible for each $u \in V(q)$. However, we proved in Lemma 4.1 that this is NP-hard. Thus, we propose a heuristic approach for constructing a small CPI based on the following idea.

- *A data vertex v can be pruned from $u.C$ if there exists a neighbor u' of u in q such that $u'.C$ contains none of the neighbors of v in G (i.e., $u'.C \cap N_G(v) = \emptyset$).*

Equivalently, $u.C$ can be obtained as the intersection of the sets of neighbors, with label $l_q(u)$, of vertices in $u'.C$ for all $u' \in N_q(u)$. Thus, following the above general idea, we propose to construct the CPI in two phases: top-down construction, and bottom-up refinement, as will be discussed in the following two subsections.

5.2 Top-Down Construction

Given a BFS tree q_T of the query q, the CPI is constructed by visiting query vertices level-by-level in a top-down manner, in which we also utilize the pruning power of non-tree edges to prune unpromising candidates. The algorithm is shown in Algorithm 3.

Firstly, we obtain the candidates for the root query vertex r, which are the vertices in G that have label $l_q(r)$ and degree at least $d_q(r)$ and also pass the candidate-verification (CandVerify) (Lines 1–2). CandVerify basically verifies whether a data vertex conforms with the local features of the query vertex, which is discussed in Section A.6 in the Appendix. We mark r as visited and set $v.cnt$ to be 0 for all vertices v in G (Line 3), where $v.cnt$ will later be used to determine whether a vertex is qualified to be a candidate.

Then, we process query vertices level-by-level (Lines 4–28) and for query vertices at the same level, we 1) firstly generate their sets of candidates in the forward processing, 2) then prune unpromising candidates in the backward processing, and 3) finally construct the adjacency lists corresponding to query vertices and their parents in q_T. Assume the set of vertices at level lev is V_{lev}.

1) Forward Candidate Generation. In the forward processing, we process query vertices according to their order in V_{lev}. In processing query vertex u, let $u.N$ denote the set of visited neighbors of u in q, and $u.UN$ denote the set of unvisited neighbors of u in q that are at the same BFS level as u in q_T. $u.UN$ is obtained at Lines 7–9, and will be used in the backward processing (see Line 20). The set $u.C$ of candidates of u is generated from the sets of candidates of vertices in $u.N$ (Lines 11–16). Intuitively, *a data vertex v is in $u.C$ only if for each $u' \in u.N$, there is a data vertex $v' \in u'.C$ that is adjacent to v.* To achieve this, we maintain a counter $v.cnt$ for each data vertex in G to count the number of visited query neighbors of u that have a candidate v' adjacent to v. The counters of vertices (i.e., $v.cnt$) are updated at Lines 11-13, and Cnt records the number of query vertices in $u.N$ (Line 14). The candidate $u.C$ is the set of vertices satisfying $v.cnt = Cnt$ and also passing CandVerify (Lines 15–16), following Lemma 5.1. Then we mark u as visited, and reset $v.cnt$ to be 0 for every vertex v that has a positive count (i.e., $v.cnt > 0$) (Line 17). Note that, to reset $v.cnt$, we only need to access those vertices with positive counts, which are stored when we change $v.cnt$ at Line 13.

Lemma 5.1: *Through lines 6–14 of Algorithm 3, a data vertex v with label $l_q(u)$ has a neighbor in $u'.C$ for every $u' \in u.N$ if and only if $v.cnt = Cnt$ (i.e., Line 15).* □

2) Backward Candidate Pruning. In the backward processing, we apply Lemma 5.1 again to filter candidates for each query vertex u based on its set of unvisited neighbors (i.e., $u.UN$), which were not exploited in the forward processing. In contrast to the forward processing, now we process query vertices V_{lev} in reverse order (Lines 18–23). For each $u \in V_{lev}$, we apply Lemma 5.1; a vertex $v \in u.C$ is pruned if $v.cnt \neq |u.UN|$ (Lines 21–22).

3) Adjacency List Construction. After every query vertex $u \in V_{lev}$ has been assigned a set $u.C$ of candidates, the adjacency lists corresponding to tree edge (u_p, u), where $u_p = u.p$ is the parent of u in q_T, is constructed (Lines 26–28). For each data vertex $v \in u_p.C$, an adjacency list $N^{u_p}_u(v)$ is constructed, which is the set of data vertices in $u.C$ that are connected to v; that is, $N^{u_p}_u(v) = \{v' \in u.C \mid (v, v') \in E(G)\}$ (Lines 27–28). Note that, $u_p.C$ has already been constructed in the iteration for processing query vertices at level $lev - 1$.

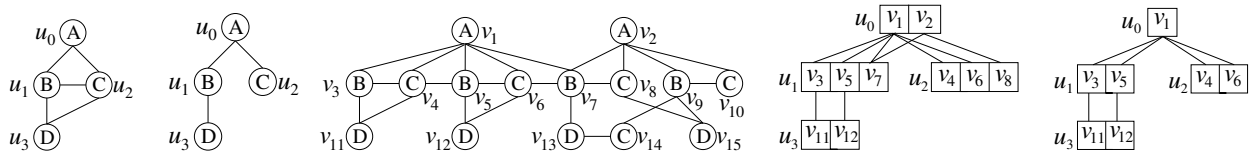


Figure 7: Example CPI Construction

Algorithm 3: Top-Down Construction

Input: A query q and its root vertex r, and a data graph G
Output: the CPI of q over G

1 **for each** *v in G with label $l_q(r)$ and degree at least $d_q(r)$* **do**
2 **if** CandVerify(v, r) **then** $r.C \leftarrow r.C \cup \{v\}$;
3 Mark r as visited; Set $v.cnt \leftarrow 0$ for all v in G;
4 **for each** *level lev from 2 to max_level* **do**
  /* Lines 5-17: Forward Candidate Generation */
5 **for each** *query vertex u at level lev* **do**
6 $Cnt \leftarrow 0$;
7 **for each** *query vertex $u' \in N_q(u)$* **do**
8 **if** *u' is unvisited* ***and*** *(u', u) is a S-NTE* **then**
9 $u.UN \leftarrow u.UN \cup \{u'\}$;
10 **else if** *u' is visited* **then**
11 **for each** *vertex $v' \in u'.C$* **do**
12 **for each** *vertex $v \in N_G(v')$ with label $l_q(u)$ and degree at least $d_q(u)$* **do**
13 **if** $v.cnt = Cnt$ **then** $v.cnt \leftarrow v.cnt + 1$;
14 $Cnt \leftarrow Cnt + 1$;
15 **for each** *vertex v in G with $v.cnt = Cnt$* **do**
16 **if** CandVerify(v, u) **then** $u.C \leftarrow u.C \cup \{v\}$;
17 Mark u as visited; Reset $v.cnt \leftarrow 0$ for all v in G s.t. $v.cnt > 0$;
  /* Lines 18-23: Backward Candidate Pruning */
18 **for each** *query vertex u at level lev in reverse order* **do**
19 $Cnt \leftarrow 0$;
20 **for each** *query vertex $u' \in u.UN$* **do** Same as Lines 11–14;
21 **for each** *vertex $v \in u.C$* **do**
22 **if** $v.cnt \neq Cnt$ **then** Remove v from $u.C$;
23 Reset $v.cnt \leftarrow 0$ for all v in G s.t. $v.cnt > 0$;
  /* Lines 24-28: Adjacency List Construction */
24 **for each** *query vertex u at level lev* **do**
25 $u_p \leftarrow u.p$;
26 **for each** *vertex $v_p \in u_p.C$* **do**
27 **for each** *vertex $v \in N_G(v_p)$ with label $l_q(u)$* **do**
28 **if** $v \in u.C$ **then** $N_u^{u_p}(v_p) \leftarrow N_u^{u_p}(v_p) \cup \{v\}$;
29 **return** CPI;

Example 5.1: Consider the query q in Figure 7(a) with the BFS tree in Figure 7(b) where u_0 is the root vertex. Then, vertices of q are partitioned into three levels, with u_0 at level 1, $\{u_1, u_2\}$ at level 2, and u_3 at level 3. Firstly, we process vertices at level 1 (i.e., u_0). The set of candidates of u_0 is assigned as $u_0.C = \{v_1, v_2\}$.

Secondly, we consider vertices at level 2 (i.e., u_1 and u_2). *1) In the forward processing*, we first process u_1 with $u_1.N = \{u_0\}$ and $u_1.UN = \{u_2\}$; $u_1.C$ is assigned to be the set of vertices with label $l_q(u_1)$ that are adjacent to a vertex in $u_0.C$, and $u_1.C = \{v_3, v_5, v_7, v_9\}$. Then, we process u_2 with $u_2.N = \{u_0, u_1\}$ and $u_2.UN = \emptyset$; $u_2.C$ is assigned as $\{v_4, v_6, v_8\}$. Note that, although v_{10} also satisfies the requirement specified by Lemma 5.1, v_{10} is pruned by the CandVerify due to having no neighbor with label D which is required by u_2. *2) In the backward processing*, v_9 is also pruned from $u_1.C$ due to having no neighbor in $u_2.C$; recall that $u_1.UN = \{u_2\}$. *3) In adjacency list construction*, the adjacency lists corresponding to (u_0, u_1) and to (u_0, u_2) are constructed as shown in Figure 7(d).

Finally, we process vertices at level 3 (i.e., u_3). Since there is only one query vertex at this level, we only have the *forward processing*; $u_3.N = \{u_1, u_2\}$, and $u_3.C = \{v_{11}, v_{12}\}$. Vertices v_{13} and v_{15} are pruned due to having no neighbors in $u_2.C$ or in $u_1.C$, respectively. The adjacency lists corresponding to $(u_3.p, u_3)$ (i.e., (u_1, u_3)) are constructed as shown in Figure 7(d). □

5.3 Bottom-Up Refinement

Note that the top-down construction algorithm in Section 5.2 only considers the ancestors (i.e., parent, parent of parent, $\cdots$) in q_T of a query vertex u to construct $u.C$. Thus, it is possible that a candidate vertex $v \in u.C$ does not have any neighbor in $u'.C$, where u' is a child of u in q_T. For example, in Figure 7(d), v_7 in $u_1.C$ does not have any neighbor in $u_3.C$; or equivalently, $N_{u_3}^{u_1}(v_7) = \emptyset$. In this subsection, we propose a bottom-up refinement approach for further refining the candidates of query vertices.

Algorithm 4: Bottom-Up Refinement

Input: A query q and its root vertex r, a data graph G, and a CPI
Output: The refined CPI

1 **for each** *query vertex u of q in a bottom-up fashion* **do**
  /* Lines 2-7: Candidate Refinement */
2 $Cnt \leftarrow 0$;
3 **for each** *lower-level neighbor u' of u in q* **do**
4 Same as Lines 11–14 of Algorithm 3;
5 **for each** *vertex $v \in u.C$* **do**
6 **if** $v.cnt \neq Cnt$ **then** Remove v from $u.C$ and remove all adjacency lists of v from the CPI;
7 Reset $v.cnt \leftarrow 0$ for all v in G s.t. $v.cnt > 0$;
  /* Lines 8-11: Adjacency List Pruning */
8 **for each** *vertex $v \in u.C$* **do**
9 **for each** *child u' of u in the BFS tree of q* **do**
10 **for each** *vertex $v' \in N_{u'}^{u}(v)$* **do**
11 **if** $v' \notin u'.C$ **then** $N_{u'}^{u}(v) \leftarrow N_{u'}^{u}(v) \backslash \{v'\}$;
12 **return** CPI;

The pseudocode of bottom-up refinement is shown in Algorithm 4. We process query vertices of q in a bottom-up fashion regarding q_T (Line 1). Note that, here the order of query vertices at the same BFS level can be arbitrary since we do not consider the S-NTE in this bottom-up refinement process. Firstly, in *candidate refinement*, similar to Lines 18–23 of Algorithm 3 we exploit the candidate sets of lower-level neighbors of u to prune unpromising candidates from $u.C$ (Lines 2–7). Then, in *adjacency list pruning*, we remove from each adjacency list $N_{u'}^{u}(v)$ those vertices that are not in $u'.C$ (Lines 8–11).

Example 5.2: Continuing Example 5.1, we refine the candidates for query vertices of q in a bottom-up fashion regarding q_T; assume they are processed in the order of u_3, u_2, u_1, u_0. Firstly, u_3 has no lower-level neighbors, and we do nothing. In processing u_2, we refine the candidates $u_2.C$ of u_2 by the candidates $u_3.C$ of u_3; v_8 is pruned from $u_2.C$ since it has no neighbors in $u_3.C$. Next, in processing u_1, v_7 is pruned from $u_1.C$, and the adjacency lists of v_7 are also removed, as shown in Figure 7(e). Finally, we process u_0;

v_2 is pruned from $u_0.C$ due to the same reason, and the adjacency lists of v_2 are removed. Moreover, we also need to remove v_7 from the adjacency list $N^{u_0}_{u_1}(v_1)$ of v_1, since v_7 is no longer a candidate of u_1. The final CPI after refinement is shown in Figure 7(e). □

5.4 Analysis of CPI Construction

Pruning Power of Tree Edges and Non-Tree Edges. In our CPI construction (i.e., Algorithms 3 and 4), we exploit both tree edges and non-tree edges as well as both directions of these edges to refine candidates for query vertices. That is, for a query edge (u, u') in q, we exploit the candidates $u.C$ of u to refine the candidates $u'.C$ of u' as well as exploit $u'.C$ to refine $u.C$; for example, if a candidate $v \in u.C$ has no neighbors in $u'.C$, then v is pruned from $u.C$. The directions of query edges that are utilized for pruning in different processing stages are summarized in Table 2, where the direction (e.g., $\rightarrow$, $\leftarrow$, $\downarrow$, $\uparrow$) indicates the direction of a unidirectional edge regarding an ordered BFS tree; for example, in Figure 7(a), $u_1 \rightarrow u_2$, $u_2 \leftarrow u_1$, $u_1 \downarrow u_3$, and $u_3 \uparrow u_1$. More details and an example are given in Section A.7 in Appendix.

Algorithms	Query Edges	Directions
Top-down construction (Algorithm 3)	Tree edges & C-NTEs	$\downarrow$
	S-NTEs	$\rightarrow, \leftarrow$
Bottom-up refinement (Algorithm 4)	Tree edges & C-NTEs	$\uparrow$

Table 2: Directions of Query Edges Utilized in Pruning

Correctness. We prove the correctness of our CPI construction (i.e., Algorithms 3 and 4) by the following two lemmas.

Lemma 5.2: *For a data vertex $v \in V(G)$ and a query vertex $u \in V(q)$, if there is an embedding M of q in G that maps u to v, then after running Algorithm 3, v is a candidate of u (i.e., $v \in u.C$). That is, the* CPI *constructed by Algorithm 3 is sound.* □

Lemma 5.3: *Given a sound* CPI*, after the bottom-up refinement (i.e., running Algorithm 4), the* CPI *is still sound.* □

Time Complexity. The time complexities of Algorithms 3 and 4 are shown in the following theorem.

Theorem 5.1: *Both Algorithm 3 and Algorithm 4 take time $O(|E(G)| \times |E(q)|)$.* □

6. EXPERIMENTS

We conduct extensive performance studies to evaluate the efficiency of our core-forest-leaf decomposition based framework and our CPI-based matching algorithms. Specifically, the following existing algorithms are evaluated.

- QuickSI: the algorithm in [15].
- Turbo$_{\text{ISO}}$: the state-of-the-art algorithm in [8].
- Turbo$_{\text{ISO}}$-Boost: the Turbo$_{\text{ISO}}$ algorithm boosted by the data graph compression techniques in [14].

We also evaluate the following variants of our algorithms.

- CFL-Match: our core-forest-leaf decomposition based algorithm (see Section 3) with the proposed CPI constructed by Algorithms 3 and 4 (see Section 5) (i.e., *our best algorithm*).
- CFL-Match-Boost: the CFL-Match algorithm boosted by the data graph compression techniques in [14].
- CFL-Match-Naive: the CFL-Match algorithm where the CPI is naively constructed (see Section 4.1).
- CFL-Match-TD: the CFL-Match algorithm where the CPI is constructed by Algorithm 3 in Section 5.2.
- CF-Match: the core-forest decomposition based algorithm (see Section 3) with the proposed CPI (see Section 5); that is, the forest-leaf decomposition is not applied.
- Match: the subgraph matching algorithm without query decomposition and with the proposed CPI; that is, apply the core-match algorithm in Section 4.2 on the entire query q.

All algorithms are implemented in C++ and compiled with GNU GCC with the -O3 flag; source codes of the existing algorithms, QuickSI [15], Turbo$_{\text{ISO}}$ [8], and Turbo$_{\text{ISO}}$-Boost [14], are obtained from their authors, respectively. Experiments are conducted on a machine with an Intel i5 3.20GHz CPU and 8GB memory.

Datasets. We evaluate the performance of the tested algorithms on both real and synthetic graphs as follows.

Real Graphs. We evaluate the algorithms on three real graphs, HPRD[1], Yeast, and Human, which are widely used in existing works [8, 12, 14, 22]. All the three graphs are protein interaction networks where vertex labels are generated under the *Gene Ontology Term*. *HPRD* contains 37,081 edges, 9,460 vertices with an average degree 7.8, and 307 distinct labels. *Yeast* contains 12,519 edges, 3,112 vertices with an average degree 8.1, and 71 distinct labels. *Human* is a dense graph of human protein interactions, which contains 86,282 edges, 4,674 vertices with an average degree 36.9, and 44 distinct labels. We also tested the algorithms on WordNet and DBLP (see Eval-A-II in Appendix), and get similar results.

Synthetic Graphs. We also generate large synthetic data graphs to evaluate the algorithms. We first randomly generate a spanning tree and then randomly add edges to the spanning tree, while vertex labels are added following the power-law distribution. The default settings of synthetic graphs are: $|V(G)| = 100k$ *(i.e.,* 10^5 *vertices),* $d(G) = 8$ *(i.e., the average degree is* 8*), and* $|\Sigma| = 50$ *(i.e. the number of distinct labels is 50)*. Note that the smaller the number of distinct labels, the more challenging. The following synthetic data graphs are generated to test the scalability of our algorithms.

- *Vary* $|V(G)|$: We generate 3 data graphs denoted by G_{100k}, G_{500k}, and G_{1000k}, where each G_{ik} has ik $(= i \times 10^3)$ vertices with the default settings of $d(G)$ and $|\Sigma|$.
- *Vary* $d(G)$: We generate 4 data graphs denoted by $G_{d=4}$, $G_{d=8}$, $G_{d=16}$, and $G_{d=32}$, where each $G_{d=i}$ has an average degree of i with the default settings of $|V(G)|$ and $|\Sigma|$.
- *Vary* $|\Sigma|$: We generate 4 data graphs denoted by $G_{L=25}$, $G_{L=50}$, $G_{L=100}$, $G_{L=200}$, where each $G_{L=i}$ contains i distinct vertex labels with the default settings of $|V(G)|$ and $d(G)$.

Data graphs	Query Sets	Default
HPRD, Yeast, Synthetic	q_{25S}, q_{25N}, q_{50S}, q_{50N} q_{100S}, q_{100N}, q_{200S}, q_{200N}	q_{50S}, q_{50N}
Human	q_{10S}, q_{10N}, q_{15S}, q_{15N} q_{20S}, q_{20N}, q_{25S}, q_{25N}	q_{15S}, q_{15N}

Table 3: Default Query Sets

Query Graphs. A query graph is generated as a connected subgraph of the data graph, by conducting random walk on the data graph. For each data graph, we generate 8 query sets, each containing 100 query graphs of the same size, as summarized in Table 3. In specific, for HPRD, Yeast, and Synthetic graphs, we generate query sets q_{25S}, q_{25N}, q_{50S}, q_{50N}, q_{100S}, q_{100N}, q_{200S}, and q_{200N}, where q_{iS} and q_{iN} denote query sets with i vertices and, respectively, average degree ≤ 3 (i.e., **S**parse) and > 3 (i.e., **N**on-sparse); q_{50S} *and* q_{50N} *are default query sets*. For Human which is a harder data graph for subgraph matching due to a higher average degree and fewer distinct labels, we generate smaller query sets q_{10S}, q_{10N}, q_{15S}, q_{15N}, q_{20S}, q_{20N}, q_{25S}, and q_{25N}, with q_{15S} *and* q_{15N} *being the default.*

[1]http://www.hprd.org/download/

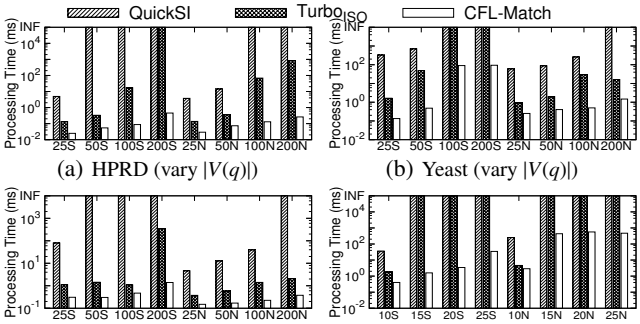
#Embeddings. We vary the number of embeddings to be reported, from 10^3 to 10^5 and 10^8 with *#embeddings* = 10^5 *being the default*. Note that, the total number of embeddings can be much larger.

Metrics. For each testing, we run an algorithm for a query set, containing 100 query graphs, three times, and *report the average CPU time in milliseconds for processing each query graph*. Note that, we set the time limit for processing a query set to 5 hours (i.e., 1.8×10^7 ms). If an algorithm cannot finish within the time limit, then we plot its processing time as "INF".

6.1 Comparing with Existing Techniques

In this subsection, we evaluate CFL-Match against the existing algorithms, QuickSI, Turbo$_{ISO}$, and Turbo$_{ISO}$-Boost. Note that, we also ran Turbo$_{ISO}$-Boost for our queries on the tested graphs using the source code provided by the authors in [14]; however, it cannot finish within the time limit for most of the queries, and even for the cases that it can finish, it is much slower than Turbo$_{ISO}$ (see Figure 21 in Appendix). Apparently, there are some implementation issues in the source code of Turbo$_{ISO}$-Boost. We have communicated with the authors in [14] through email and the problem cannot be solved. Thus we omit Turbo$_{ISO}$-Boost in the following comparisons to be fair to the authors in [14]. Nevertheless, we implemented the techniques in [14] in combination with our techniques as the boosted version of CFL-Match and evaluate it in **Eval-IV**.

Eval-I: Against Existing Algorithms by Varying $|V(q)|$. We evaluate CFL-Match against existing algorithms by varying $|V(q)|$ regarding the *total processing time*, *embedding enumeration time*, and *query vertex ordering time*. Embedding enumeration time is the time to enumerate embeddings after obtaining a matching order of query vertices, while query vertex ordering time is the time to compute the matching order and other auxiliary data structures that are required for computing the matching order.

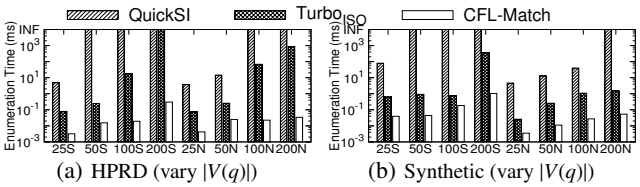


(a) HPRD (vary $|V(q)|$) (b) Yeast (vary $|V(q)|$)
(c) Synthetic (vary $|V(q)|$) (d) Human (vary $|V(q)|$)

Figure 8: Against Existing Algorithms (total processing time)

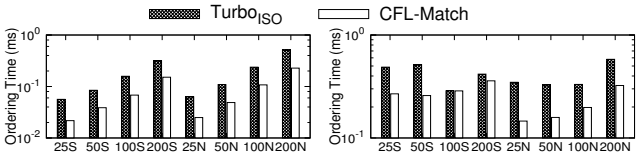
Total Processing Time. Figure 8 shows the average total processing time for each query graph. In general, all three algorithms run slower for larger queries, and QuickSI and Turbo$_{ISO}$ may not finish within the time limit for large queries (denoted as "INF" in the figures). CFL-Match consistently outperforms Turbo$_{ISO}$ which then performs better than QuickSI. This is due to our new framework by postponing the Cartesian products and also our CPI-based effective ordering of queries vertices. Our CFL-Match algorithm improves upon the state-of-the-art algorithm, Turbo$_{ISO}$, by over 3 orders of magnitude (see query q_{200N} in Figure 8(a)), even excluding the cases when Turbo$_{ISO}$ cannot finish within the time limit.

Embedding Enumeration Time. Figure 9 shows the embedding enumeration time of the three algorithms on HPRD and Synthetic graph. We omit the results on Yeast and Human due to QuickSI and Turbo$_{ISO}$ cannot finish within the time limit for most of the queries on these two graphs (see Figure 8). CFL-Match consistently outperforms Turbo$_{ISO}$ across all queries for enumerating embeddings, and the improvement can be over 4 orders of magnitude (see query q_{200N} on HPRD); QuickSI runs the slowest. This confirms the advantage of our new framework, by postponing the Cartesian products (see Section 3), over the existing algorithms.



(a) HPRD (vary $|V(q)|$) (b) Synthetic (vary $|V(q)|$)

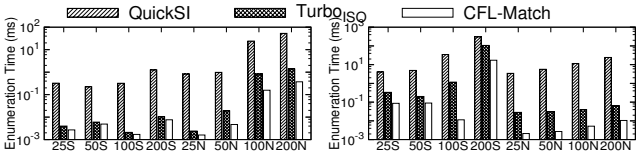
Figure 9: Against Existing Algorithms (enumeration time)



(a) HPRD (vary $|V(q)|$) (b) Synthetic (vary $|V(q)|$)

Figure 10: Against Existing Algorithms (ordering time)

Query Vertex Ordering Time. We illustrate the query vertex ordering time of Turbo$_{ISO}$ and CFL-Match on HPRD and Synthetic graph in Figure 10. Note that, the query vertex ordering time of QuickSI is negligible since the ordering is directly based on the edge frequencies; thus, we omit QuickSI in Figure 10. We can see that the query vertex ordering time of CFL-Match is much smaller than that of Turbo$_{ISO}$, due to the $O(|E(q)| \times |E(G)|)$ time complexity of CFL-Match for CPI construction. Although Turbo$_{ISO}$ has a worst-case exponential time complexity for constructing its data structure due to possibly exponential number of path embeddings, the number of path embeddings in these two graphs are small. Thus, Turbo$_{ISO}$ also performs well regarding query vertex ordering time.



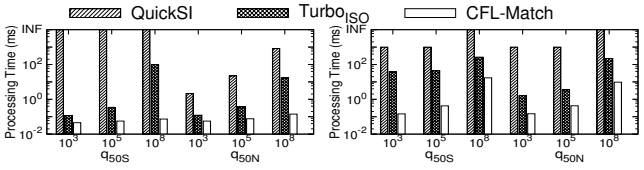
(a) HPRD (enumeration time) (b) Synthetic (enumeration time)

Figure 11: Enumeration Time for Core-Structures (Vary $|V(q)|$)

Eval-II: Evaluating Enumeration Time for Core-Structures of Queries. In this testing, we evaluate the enumeration time of the algorithms for processing core-structures of queries. That is, our core-forest-leaf decomposition based framework has no effect on the running time; CFL-Match is equivalent to the Core-Match algorithm in Section 4.2. The results are shown in Figure 11. Different from Figure 9, both QuickSI and Turbo$_{ISO}$ can now finish within the time limit. This is because, 1) the size of a core-structure is smaller than that of the original query in Figure 9, and 2) the number of embeddings of a core-structure is usually smaller than that of a general query. The embedding enumeration time of CFL-Match is much smaller than that of Turbo$_{ISO}$; this confirms the better matching order computed by our greedy path-ordering approach based on the cost model in Section 4.2.1.

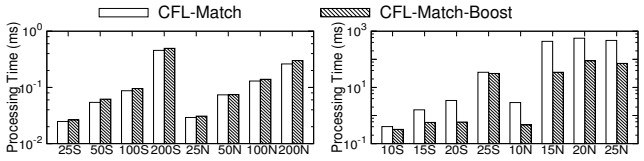
Eval-III: Varying #Embeddings. Figure 12 shows the results of the algorithms by varying *#embeddings*. As expected, the processing time of all three algorithms increases when more embeddings are generated. Nevertheless, CFL-Match consistently outperforms Turbo$_{ISO}$ with QuickSI performing the worst.

Eval-IV: Evaluating The Boost Technique in [14]. The results of applying the boost technique in [14], which compresses the data



(a) HPRD (vary *#embeddings*) (b) Yeast (vary *#embeddings*)

Figure 12: Evaluating *#embeddings*



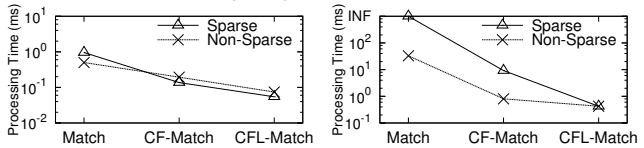
(a) HPRD (vary $|V(q)|$) (b) Human (vary $|V(q)|$)

Figure 13: Evaluating The Boost Technique

graph, to CFL-Match are shown in Figure 13. The boost technique improves CFL-Match on Human due to the high data graph compression ratio (i.e., about 40%). However, CFL-Match-Boost performs a little slower than CFL-Match on HPRD, due to the low compression ratio of HPRD (i.e., < 5%); note that the query-dependent compression has overheads. Thus, *the boost technique may not always help; we omit* CFL-Match-Boost *in the following*.

6.2 Effectiveness of New Framework

In this subsection, we evaluate the effectiveness of our proposed techniques in reducing the overall processing time, and the scalability of our CFL-Match algorithm. We run variants of CFL-Match on different graphs for query sets q_{50S} (average degree ≤ 3, denoted Sparse) and q_{50N} (average degree > 3, denoted Non-Sparse).

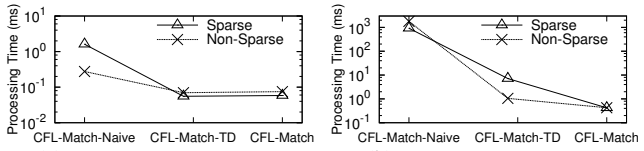


(a) HPRD (vary algorithms) (b) Yeast (vary algorithms)

Figure 14: Evaluating Our Framework

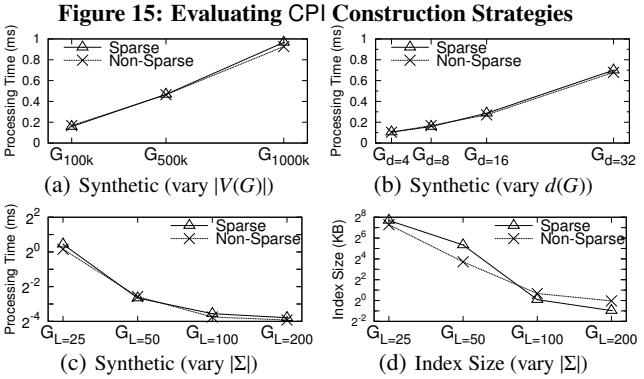
Eval-V: Evaluating Our Framework. We compare CFL-Match with CF-Match and Match to evaluate the effectiveness of our core-forest-leaf decomposition based framework. The results on HPRD and Yeast are shown in Figure 14. We can see that, the core-forest decomposition based framework (i.e., CF-Match) improves upon a non-decomposition based framework (i.e., Match), and the forest-leaf decomposition based framework (i.e., CFL-Match) further improves CF-Match by postponing the Cartesian products. The improvement of the core-forest-leaf decomposition based framework on Yeast is more significant due to more candidates for each query vertex in CPI (thus, more Cartesian products).

Eval-VI: Evaluating the Effectiveness of CPI Construction Strategies. The evaluation results of the effect of different CPI construction strategies on the total processing time of our CFL-Match algorithm are illustrated in Figure 15. The naively constructed CPI (see Section 4.1) significantly degrades the performance of CFL-Match due to lots of false-positive candidates for query vertices in CPI. CFL-Match-TD improves upon CFL-Match-Naive by constructing the CPI in a top-down fashion (see Section 5.2), which also exploits the non-tree edges for candidate pruning. Finally, the bottom-up refinement of CPI (see Section 5.3) further reduces the candidates of query vertices, thus leads to the best performance of CFL-Match. Note that, due to very few candidates of query vertices in CPI constructed by the top-down algorithm for HPRD, the improvement of CFL-Match over CFL-Match-TD is insignificant in Figure 15(a).



(a) HPRD (vary algorithms) (b) Yeast (vary algorithms)

Figure 15: Evaluating CPI Construction Strategies



(a) Synthetic (vary $|V(G)|$) (b) Synthetic (vary $d(G)$)
(c) Synthetic (vary $|\Sigma|$) (d) Index Size (vary $|\Sigma|$)

Figure 16: Scalability Testing of CFL-Match

Eval-VII: Scalability Testing. We test the scalability of CFL-Match on Synthetic graphs by varying $|V(G)|$, $d(G)$, and $|\Sigma|$.

Varying $|V(G)|$. The results of varying $|V(G)|$ are illustrated in Figure 16(a). The processing time of CFL-Match increases linearly with respect to $|V(G)|$. This is because, for Synthetic graphs, the query vertex ordering time dominates the embedding enumeration time as shown in Figures 9(b) and 10(b). The time for CPI construction, $O(|E(G)| \times |E(q)|)$, is the dominating factor in the query vertex ordering time, and it increases linearly regarding $|V(G)|$.

Varying $d(G)$. Figure 16(b) shows the results of varying the average degree $d(G)$ of Synthetic graphs. The processing time of CFL-Match increases almost linearly regarding $d(G)$. This is because the larger $d(G)$, the more edges in CPI, and thus the more running time of CFL-Match.

Varying $|\Sigma|$. Figure 16(c) illustrates the processing time of CFL-Match on Synthetic graphs by varying the number $|\Sigma|$ of distinct labels. The processing time of CFL-Match decreases when $|\Sigma|$ becomes larger, due to fewer candidates for each query vertex as evidenced in Figure 16(d) where the index size (i.e., y-axis) is the size of the CPI constructed by our algorithm.

7. CONCLUSION

In this paper, we proposed a new framework by postponing the Cartesian products based on the core-forest-leaf decomposition of a query to minimize the redundant Cartesian products. We are the first to address the issue of unpromising results by Cartesian products from "dissimilar" vertices. We proposed a new path-based auxiliary data structure, of size $O(|E(G)|\times|V(q)|)$, to generate matching order and conduct subgraph matching, which significantly reduces the exponential size $O(|V(G)|^{|V(q)|-1})$ of the existing data structure in [8]. Extensive empirical studies on real and synthetic graphs demonstrate that our techniques outperform the state-of-the-art algorithm by up to 3 orders of magnitude. As a possible future work, extending our core-forest-leaf decomposition to a hierarchical decomposition of the core-structure (e.g., compute k-core, (k-1)-core, ...) might be an interesting issue to be investigated.

Acknowledgements. Lijun Chang is supported by ARC DE150100563 and ARC DP160101513. Xuemin Lin is supported by NSFC61232006, ARC DP140103578 and ARC DP150102728. Lu Qin is supported by ARC DE140100999, and ARC DP160101513. Wenjie Zhang is supported by ARC DP150103071 and ARC DP150102728.

8. REFERENCES

[1] V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks. *CoRR*, cs.DS/0310049, 2003.

[2] V. Bonnici, A. Ferro, R. Giugno, A. Pulvirenti, and D. Shasha. Enhancing graph database indexing by suffix tree structure. In *Proc. of PRIB'10*, 2010.

[3] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. In *Proc. of SIGMOD'07*, 2007.

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

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[6] A. Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, 1985.

[7] R. Giugno, V. Bonnici, N. Bombieri, A. Pulvirenti, A. Ferro, and D. Shasha. Grapes: A software for parallel searching on biological graphs targeting multi-core architectures. *PLoS ONE*, 8(10), 10 2013.

[8] W. Han, J. Lee, and J. Lee. Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Prof. of SIGMOD'13*, 2013.

[9] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proc. of SIGMOD'08*, 2008.

[10] K. Klein, N. Kriege, and P. Mutzel. Ct-index: Fingerprint-based graph indexing combining cycles and trees. In *Proc. of ICDE'11*, 2011.

[11] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce. *PVLDB*, 8(10), 2015.

[12] J. Lee, W. Han, R. Kasperovics, and J. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2), 2012.

[13] N. Przulj, D. G. Corneil, and I. Jurisica. Efficient estimation of graphlet frequency distributions in protein-protein interaction networks. *Bioinformatics*, 22(8), 2006.

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

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

[16] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu. Parallel subgraph listing in a large-scale graph. In *Proc. of SIGMOD'14*, 2014.

[17] T. A. B. Snijders, P. E. Pattison, G. L. Robins, and M. S. Handcock. New specifications for exponential random graph models. *Sociological Methodology*, 36(1), 2006.

[18] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9), 2012.

[19] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1), 1976.

[20] X. Yan, P. S. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *Proc. of SIGMOD'04*, 2004.

[21] S. Zhang, J. Yang, and W. Jin. SAPPER: subgraph indexing and approximate matching in large graphs. *PVLDB*, 3(1), 2010.

[22] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 3(1), 2010.

[23] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: Tree + delta >= graph. In *Proc. of VLDB'07*, 2007.

[24] G. Zhu, X. Lin, K. Zhu, W. Zhang, and J. X. Yu. Treespan: efficiently computing similarity all-matching. In *Proc. of SIGMOD'12*, 2012.

[25] L. Zou, L. Chen, J. X. Yu, and Y. Lu. A novel spectral coding in a large graph database. In *Proc. of EDBT'08*, 2008.

A. APPENDIX

A.1 Proofs of Lemmas and Theorems.

Proof Sketch of Lemma 3.1. Firstly, we prove that the 2-core of q contains all non-tree edges of q regarding any spanning tree of q. It is easy to see that the set of non-tree edges regarding any spanning tree of q is exactly the set of edges in simple cycles [6] in q, and all vertices in a simple cycle have exactly two neighbors in the cycle. Thus, the 2-core will include all such simple cycles (i.e., include all non-tree edges regarding any spanning tree of q).

Secondly, we prove that the 2-core of q is the minimal connected subgraph of q containing all non-tree edges of q regarding any spanning tree of q, by contradiction. Assume there is a smaller connected subgraph g' of q that contains all non-tree edges of q regarding any spanning tree of q. Denote the 2-core of q as g. Then, there must be an edge (u_1, u_2) of g that is not in g'. Since g' contains all non-tree edges of q regarding any spanning tree of q, there will be no simple cycle of q containing (u_1, u_2), which means that (u_1, u_2) is a bridge of q [6]. Then, the set of all simple cycles in q will form at least two connected components in the subgraph of q by removing (u_1, u_2); otherwise, g will not contain (u_1, u_2). This contradicts that g' is connected and does not contain (u_1, u_2).

Thus, the lemma holds. □

Proof Sketch of Theorem 4.1. Firstly, it is easy to see that there will be no false positives. This is because every edge between two candidate vertices in the CPI is also an edge in the data graph.

Now, we give a constructive proof to show that there is no false negative; that is, every embedding of q in G will be computed by traversing CPI. Consider an embedding $M = (v_0, \ldots, v_n)$ of $q = (u_0, \ldots, u_n)$. Without loss of generality, assume u_0 is the root node in CPI, and the parent node of u_i ($i \neq 0$) in CPI is $u_i.p$ and it is mapped to $v_i.p$ by M. Then, v_0 must be in $u_0.C$, and moreover, for each u_i ($i \neq 0$), v_i must be in both $u_i.C$ and $N^{u_i.p}_{u_i}(v_i.p)$, since CPI is sound. Thus, the embedding $(v_0, \ldots, v_n)$ will be generated by traversing CPI, and it will also survive all non-tree edge checkings.

Hence, the theorem holds. □

Proof Sketch of Lemma 4.1. We reduce the problem of checking subgraph isomorphism to the problem of determining whether a minimum and sound CPI is empty. We prove that, given graphs q and G, q is subgraph isomorphic to G, which is NP-complete [5], if and only if a minimum and sound CPI is non-empty.

($\Longrightarrow$). If q is subgraph isomorphic to G, assume the subgraph isomorphic embedding is M, then for each $u \in V(q)$, the mapping $M(u)$ of u must be in $u.C$ in the sound CPI. Thus, the minimum and sound CPI is non-empty.

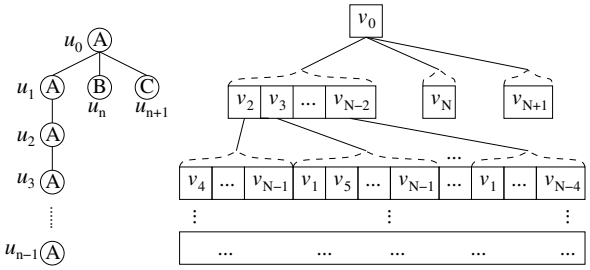
($\Longleftarrow$). It is easy to see that, if a minimum and sound CPI is non-empty, then there must be an embedding of q in G. □

Proof Sketch of Lemma 4.2. Note that, each vertex in the forest-structure and not in V_C has at least two-neighbors in q; recall that, all the degree-one vertices are put in the leaf-set V_I (see Section 3).

We prove the lemma by contradiction. Assume there are two vertices u and u' from the forest-structure that have the same labels and the same neighborhoods, and u_1 and u'_1 are the (common) neighbors of u and u'. Then, (u, u_1, u', u'_1) forms a cycle in q, which means that u and u' cannot be in the forest-structure. We reach a contradiction. Thus, the lemma holds. □

Proof Sketch of Lemma 4.3. This lemma directly follows from the fact that each vertex in G has a single label. Any vertex $v \in C(u)$ with label $l_G(v) = l_q(u)$ is not in $C(u')$ since $l_q(u') \neq l_q(u)$. □

Proof Sketch of Lemma 5.1. We prove the lemma by induction. Obviously, the lemma holds for $|u.N| = 1$ (i.e., $Cnt = 1$). Now, let's consider $u.N = \{u_1, \ldots, u_k\}$. We assume the lemma holds after processed the first $(k-1)$ visited neighbors of u. That is, a data vertex v with label u has a neighbor in $u_i.C$ for $1 \leq i \leq k-1$ if and only if $v.cnt = k-1$; let V_{k-1} denote the set of all such vertices $\{v\}$. Then, after checking $v.cnt = k-1$ at Line 13, only those vertices from V_{k-1} will satisfy this requirement. Moreover, a vertex $v \in V_{k-1}$ have $v.cnt = k$ after processing u_k if and only if there is a vertex from $u_k.C$ that is adjacent to v. Thus, the lemma holds. □



(a) Query q (b) Data structure in $\mathsf{Turbo_{ISO}}$

Figure 17: Example Data Structure in $\mathsf{Turbo_{ISO}}$

Proof Sketch of Lemma 5.2. We prove the lemma by contradiction. Firstly, it is easy to see that, if v is in $u.C$ for every processed query vertex u before running *backward candidate pruning* (i.e., Lines 18–23), then v is still in $u.C$ afterwards. This is because for every query vertex $u' \in u.UN$, $M(u')$ is in $u'.C$.

Now, we assume u is the first query vertex according to the processing order defined by Lines 4–5 of Algorithm 3 such that $v \notin u.C$ with $v = M(u)$. Then, v must satisfy $v.cnt \neq Cnt$ at Line 15. However, since u is the first such query vertex, $M(u')$ must be in $u'.C$ for every $u' \in u.N$. Then, according to Lemma 5.1, $v.cnt = Cnt$. We reach a contradiction. Thus, the lemma holds. □

Proof Sketch of Lemma 5.3. Let's consider an arbitrary embedding M of q in G, we prove that $M(u) \in u.C$ in CPI for every $u \in V(q)$. Since this condition holds before running Lines 2–10 of Algorithm 4, $M(u).cnt = Cnt$ holds afterwards according to Lemma 5.1. Thus $M(u) \in u.C$, and the lemma holds. □

Proof Sketch of Theorem 5.1. Firstly, we consider Algorithm 3, we show that Lines 6–17 for a specific u take time $O(|E(G)| \times |u.N|)$. In particular, for each vertex $v' \in u'.C$, Lines 12–13 take $O(d_G(v'))$ time; thus, for all vertices in $u'.C$, Lines 12–13 take time $O(\sum_{v' \in u'.C} d_G(v')) = O(|E(G)|)$. Similarly, Lines 19–23 for u take time $O(|E(G)| \times |u.UN|)$ time and Lines 25–28 for u take time $O(|E(G)|)$ time. Thus, the total time for a specific u is $O(|E(G)| \times (|u.N| + |u.UN|)) = O(|E(G)| \times d_q(u))$, and the total running time of Algorithm 3 is $O(\sum_{u \in V(q)} |E(G)| \times d_q(u)) = O(|E(G)| \times |E(q)|)$.

The time complexity of Algorithm 4 can be proved similarly. □

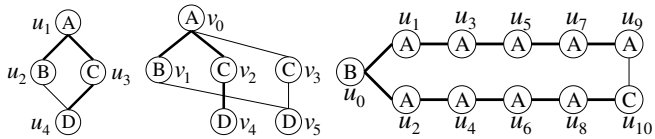
A.2 Storage Representation of CPI

In representing a CPI, we store the candidate set for each node in CPI as an array of vertex ids; for example, $u_0.C = \{v_0, v_1, v_2, v_3, v_4\}$ is stored as an array of five elements in Figure 5(c). For each pair of parent-child (i.e., adjacent) nodes u and u' in CPI with u being the parent, we store the edges between each vertex $v \in u.C$ and vertices of $u'.C$ as an adjacency list of v corresponding to the query edge (u, u'), denoted $N^u_{u'}(v)$. For example, $N^{u_0}_{u_1}(v_0) = \{v_5, v_8\}$.

In order to support fast traversal of a CPI by following vertices' adjacency lists, instead of directly storing vertex ids we store their positions (i.e., offsets) in the corresponding candidate set of the child node, into the adjacency lists. For example, the adjacency list of v_0 becomes $N^{u_0}_{u_1}(v_0) = \{1, 4\}$ meaning that the adjacency list of v_0 contains the first (i.e., v_5) and fourth (i.e., v_8) vertices in $u_1.C$. Consequently, the adjacency lists of v_5 and v_8 can be directly obtained, as the first and forth adjacency lists among all adjacency lists stored in u_1 regarding each child of u_1 in CPI, without the help of a hash-structure. Nevertheless, for presentation simplicity, in the paper we still use vertex ids instead of positions/offsets in the representation of the contents of adjacency lists in CPI.

A.3 Analysis of The Data Structure in $\mathsf{Turbo_{ISO}}$

Exponential Size of The Auxiliary Data Structure in $\mathsf{Turbo_{ISO}}$. The auxiliary data structure in $\mathsf{Turbo_{ISO}}$ [8] can be of exponential size. For example, consider the spanning query tree q in Figure 17(a) and a data graph G with $N + 2$ vertices, where the subgraph of G induced by the first N vertices, $(v_0, \ldots, v_{N-1})$, form a near-clique and all have label A, edges (v_0, v_{N-1}) and (v_i, v_{i+1}) for each $0 \leq i < N-1$ are absent from G, and v_0 is connected to v_N and v_{N+1}, respectively, with labels B and C.[2] The data structure built by $\mathsf{Turbo_{ISO}}$ is shown in Figure 17(b). u_0 has only one candidate, v_0. For u_1, there are $(N-3)$ candidates $\{v_2, \ldots, v_{N-2}\}$; note that v_0 is not a candidate because it has been used in the mapping of u_0, and v_1 and v_{N-1} are also not candidates because they are not connected to v_0 (the map of the parent of u_1, i.e., u_0) in G. For u_2, there are $(N-3) \times (N-4)$ candidates; that is, regarding each candidate v_i of u_1 (i.e., the parent of u_2), there are $(N-4)$ candidates for u_2. In general, u_i has $\prod_{j=1}^{i}(N-j-2)$ candidates for $1 \leq i \leq n-1$ (assume $n < N$). Thus, the total size of the data structure is at least $\prod_{i=1}^{n-1}(N-i-2) > (\frac{N-3}{e})^{n-1}$, which is exponential to the size of the query; here e is a mathematical constant and is approximately equal to 2.71828. Note that, if there is a non-tree edge between u_2 and u_n in q in Figure 17(a), then there will be no embeddings of q in G; that is, all these generated candidates in the data structure will be finally pruned during embedding enumeration.



(a) Query q (b) Data Graph G (c) Another Query q'

Figure 18: Restrict to k (= 1) Path Embeddings

Restrict to k Path Embeddings. To resolve the issue of possible exponential size of the auxiliary data structure, $\mathsf{Turbo_{ISO}}$ only materializes k embeddings for each root-to-leaf query path *in its implementation* when computing the matching order of query vertices, if we only retrieve k subgraph isomorphic embeddings for q. However, such a materialization cannot always guarantee to generate k subgraph isomorphic embeddings for q from the data structure. For example, consider the query q in Figure 18(a) and the data graph G in Figure 18(b), where the BFS tree q_T of q is illustrated by thick edges. Assume $k = 1$, the materialized auxiliary data structure in $\mathsf{Turbo_{ISO}}$ may only contain the embedding of q_T in G as illustrated by the thick edges in Figure 18(b). Then, from the data structure, we cannot generate the embedding of q in G that maps u_1 to v_0.

To generate k embeddings of q in G, $\mathsf{Turbo_{ISO}}$ materializes more path embeddings on demand in enumerating subgraph isomorphic embeddings, when needed. For example, in Figure 18(b), the path embedding (v_0, v_3, v_5) will be materialized in enumerating subgraph isomorphic embeddings. Thus, the size of the data structure in $\mathsf{Turbo_{ISO}}$ can still be exponential in the worst case. We tested $\mathsf{Turbo_{ISO}}$ on the query q' in Figure 18(c) and a data graph with 103 vertices, where the vertices $v_0, \ldots, v_{99}$ with label A form a near-clique as above, v_{100} with label B is connected to v_i for $0 \leq i \leq 99$, and v_{101} with label C is connected to v_0 and also connected to v_{102} with label A; the spanning tree of q is chosen as indicated by thick edges in Figure 18(c), $k = 1$, $\mathsf{Turbo_{ISO}}$ crashes due to the exponential size of the data structure constructed when enumerating subgraph isomorphic embeddings; note that, there actually is no result for this query on this data graph.

Moreover, if we want to retrieve all subgraph isomorphic embeddings, then we have to materialize all path embeddings; that is, the worst case exponential size is unavoidable using $\mathsf{Turbo_{ISO}}$. As a result, $\mathsf{Turbo_{ISO}}$ cannot scale to large queries or large data graphs.

[2]Note that neither the query q nor the data graph G can be compressed by the techniques in [8] or [14].

A.4 Pseudocode of Core-Match

Algorithm 5: Core-Match

Input: Matching index i, matching order $(u_1, \ldots, u_{|V(q)|})$, a CPI, and a data graph G
Output: All embeddings of q in G

1 **if** $i = |V(q)| + 1$ **then** **Output** M as an embedding of q in G;
2 **else if** $i = 1$ **then**
3 **for each** *data vertex* $v \in u_i.C$ **do**
4 $M(u_i) \leftarrow v$; Mark v as visited;
5 Core-Match$(i + 1)$; Mark v as unvisited;
6 **else for each** *data vertex* $v \in N^{u_i.p}_{u_i}(M(u_i.p))$ **do**
7 **if** v *is unvisited* ***and*** ValidateNT(v, u_i, G) **then**
8 Same as Lines 4–6;

The pseudocode of Core-Match is shown in Algorithm 5, where ValidateNT(v, u_i, G) validates non-tree edges. That is, for each non-tree edge $(u_j, u_i) \in E(q)$ with $j < i$ and $u_j \neq u_i.p$ in the matching sequence/order, it checks whether $(M(u_j), v) \in E(G)$. Note that, for simplicity, we present Algorithm 5 in a recursive form by recursively invoking Core-Match; however, for efficiency consideration, our implementation of Algorithm 5 is non-recursive.

A.5 Generalize Leaf-Set to Independent-Set

It is immediate that a leaf-set is an independent-set. However, the reverse is not true. Thus, we can generalize the forest-leaf decomposition in Section 3 to a *forest-IS (independent-set) decomposition*; that is, compute the maximal independent-set of the forest-structure T. Intuitively, we want the independent-set to be as large as possible (or equivalently, the forest-set V_T as small as possible); moreover, we also need the subgraph $q[V_C \cup V_T]$ to be connected to obtain a connected matching order of $V_C \cup V_T$. This is to compute the Connected Minimum Vertex Cover (cMVC) that is also connected to the core-structure $q[V_C]$, for each connected tree in T; that is, we need the connection vertex of the connected tree to be in the cMVC. In general, computing cMVC is NP-hard for a graph [5]. The good news is that we are dealing with trees. Moreover, we require the connection vertex between the tree and the core-structure to be in the cMVC (e.g., u_1 and u_2 in Figure 4(c) need to be in the cMVC). Therefore, it is easy to verify that the set of vertices with degree at least two and the connection vertex is the cMVC; thus, V_T is the set of vertices with degree at least two and excluding the connection vertices, and V_I *is the set of degree-one vertices excluding the connection vertices*. That is, the leaf-set is maximal possible independent-set we can get from the forest-structure T.

Discussion. It is, in principle, also possible to compute an independent set from the core-structure $q[V_C]$ of q and put the independent set at the end of the matching order. However, this will put some of the non-tree edge checkings at the end of the matching, which is not a good idea as shown in the Introduction. This is because we have proved in Lemma 3.1 that the core-structure $q[V_C]$ is the minimal connected subgraph containing all possible non-tree edges regarding any spanning tree of q. For example, consider a query q where the core-structure consists of a cycle, then every edge in the core-structure is a possible non-tree edge. Thus, we do not compute the independent set for the core-structure.

A.6 Root Selection and Candidate Filtering

Root Vertex Selection for a Query. To build the CPI, we first need to construct a BFS tree of the query q, which is to choose a root vertex for q. Recall that, in our core-forest-leaf decomposition based framework (see Section 3), we put all query vertices from the core-structure at the beginning of the matching order. Moreover the root vertex will be the first vertex in a matching order. Thus, the root vertex is chosen from the core-set V_C.

Intuitively, we favor the root vertex to have a small number of candidates and to have a large degree; fewer candidates means fewer partial embeddings being generated, while larger degree means more chance to prune partial embeddings at early stages. Similar to [8], we choose the root vertex r as $r \leftarrow \arg\min_{u \in V(q)} \frac{|C(u)|}{d_q(u)}$ where $C(u)$ is the set of candidates in G of u. There are different strategies for obtaining $C(u)$, and different strategies will have different costs. Thus, we first use a light-weight strategy to obtain $C(u)$ based on only the label $l_q(u)$ and degree $d_q(u)$ of u; that is, $C(u)$ is the set of vertices in G with label $l_q(u)$ and degree at least $d_q(u)$. Top-3 vertices are obtained based on the computed $C(u)$. Then, we compute a reduced (i.e., more accurate) set $C(u)$ of candidates for the selected top-3 vertices, by using the candidate filtering techniques (i.e., CandVefiry) which will be described in below; the vertex with the smallest $|C(u)|/d_q(u)$ is selected as the root vertex. For example, for the query q in Figure 7(a) and the data graph G in Figure 7(c), u_0 is chosen as the root, because $d_q(u_0) = 2$ and $C(u_0) = \{v_1, v_2\}$ and $|C(u_0)|/d_q(u_0) = 1$ is the smallest among all query vertices.

Candidate Filtering. To reduce the size of candidate sets of query vertices, many *candidate filtering techniques* have been proposed [4, 8, 15, 19, 22, 24] by exploiting the local features of query vertices, such as vertex label filter [19], vertex degree filter [19], and Neighborhood Label Frequency (NLF) filter [24]. In Algorithm 3, we retrieve the candidates for a query vertex based on the vertex label filter and vertex degree filter (see Lines 1,12). Applying the NLF filter is time-consuming; that is, verifying one candidate vertex v for a query vertex u takes $O(|L_N(u)|)$ time, where $L_N(u)$ is the set of unique labels of u's neighbors. In order to reduce the number of invokings of the expensive NLF filter, we propose the *maximum neighbor-degree* filter, which can be verified in constant time for each candidate data vertex.

Definition A.1: The **maximum neighbor-degree** of a vertex u in a graph g, denoted $mnd_g(u)$, is the maximum degree of all its neighbors (i.e., $mnd_g(u) = \max_{u' \in N_g(u)} d_g(u')$). □

We verify candidates for query vertices by the lemma below.

Lemma A.1: *Given a query q and a data graph G, a data vertex $v \in V(G)$ is not a candidate of $u \in V(q)$ if $mnd_G(v) < mnd_q(u)$.* □

Proof Sketch: We prove the lemma by contradiction. Assume v is a candidate of u with $mnd_G(v) < mnd_q(u)$; that is, there is an embedding M that maps u to v. Let u' be the neighbor of u with the maximum degree among all neighbors of u (i.e., $d_q(u') = mnd_q(u)$), and v' be the vertex to which u' maps in M. Then, $d_G(v') \leq mnd_G(v) < mnd_q(u) = d_q(u')$, and u' cannot be mapped to v' due to degree violation. We reach a contradiction. Thus, the lemma holds. □

Algorithm 6: CandVerify

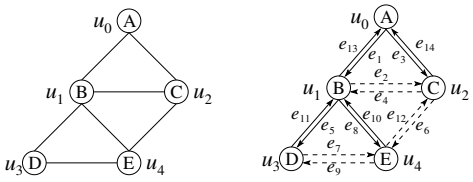
Input: A potential candidate vertex v for a query vertex u
Output: **true** if v is a candidate of u, and **false** otherwise

1 **if** $mnd_G(v) < mnd_q(u)$ **then** **return false**;
2 **for each** *label* $l \in L_N(u)$ **do**
3 **if** $d(v, l) < d(u, l)$ **then**
4 **return false**;
5 **return true**;

Then, CandVerify is shown in Algorithm 6. We check the light-weight (i.e., constant-time) filter (Line 1) before applying the more expensive NLF filter (Lines 2–4), where $d(v, l)$ denotes the number of neighbors of v with label l.

A.7 Pruning Power in CPI Construction

As shown in Table 2 in Section 5, both directions of S-NTEs are exploited in the top-down construction algorithm. The direction of $\rightarrow$ is exploited in the forward processing of Algorithm 3, while the direction of $\leftarrow$ is exploited in the backward processing. The direction of $\downarrow$ (i.e., from upper-level vertices to lower-level vertices) for tree edges and C-NTEs is exploited in the forward processing of Algorithm 3, while the direction of $\uparrow$ is exploited in Algorithm 4. Therefore, both directions of every query edge in q are exploited for reducing the size of candidates of query vertices.



(a) Query q (b) Unidirectional edges

Process order	Vertex	Edges	Algorithm phase
1	u_1	e_1	Algorithm 3 – Forward
2	u_2	e_2, e_3	Algorithm 3 – Forward
3	u_1	e_4	Algorithm 3 – Backward
4	u_3	e_5	Algorithm 3 – Forward
5	u_4	e_6, e_7, e_8	Algorithm 3 – Forward
6	u_3	e_9	Algorithm 3 – Backward
7	u_2	e_{12}	Algorithm 4
8	u_1	e_{10}, e_{11}	Algorithm 4
9	u_0	e_{13}, e_{14}	Algorithm 4

(c) Pruning power of edges exploited

Figure 19: Exploit Both Directions of Query Edges in Pruning

Example 1.1: Figure 19 illustrates exploiting the pruning power of both directions of query edges at different algorithm phases. Figure 19(b) shows the unidirectional edges of the query q in Figure 19(a); each undirected edge of q is replaced by two unidirectional edges, and solid edges are tree edges while dashed edges are non-tree edges. Query vertices of q are processed in the order shown in the first column of Figure 19(c); the last column shows the different phases of the algorithms, while the third shows the set of unidirectional edges exploited in candidates generation or pruning. Thus, each edge is exploited twice, once in each direction. □

A.8 Additional Experimental Results

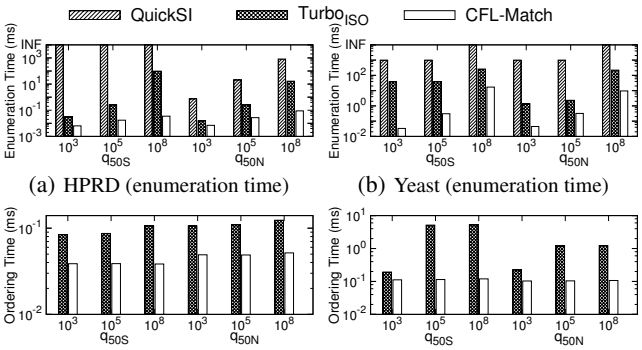
Eval-A-I: Additional Results for Section 6. Firstly, we show some additional results for the testings in Section 6.

Effectiveness of Compressing Core-Structures. Table 4 shows the average number (i.e., Avg) of reduced vertices by the query graph compression technique (NEC) in [8] for compressing core-structures of queries, and the number (i.e., #R) of queries whose core-structures are compressed by NEC for each query set containing 100 queries.

Query	HPRD		Yeast		Synthetic		Human	
	Avg	#R	Avg	#R	Avg	#R	Avg	#R
q_{25S}/q_{10S}	0.09	8	0.41	33	0.02	2	0.62	41
q_{50S}/q_{15S}	0.16	13	0.46	34	0.03	3	0.45	32
q_{100S}/q_{20S}	0.41	28	0.41	33	0	0	0.31	25
q_{200S}/q_{25S}	0.35	27	2.3	83	0.01	1	0.65	39
q_{25N}/q_{10N}	0.1	9	0.41	33	0.01	1	0.83	39
q_{50N}/q_{15N}	0.21	16	0.34	21	0	0	1.37	49
q_{100N}/q_{20N}	0.68	47	0.61	38	0.01	1	1.39	49
q_{200N}/q_{25N}	1.29	67	0.89	44	0	0	0.86	43

Table 4: Avg: average number of reduced vertices by NEC [8]; #R: number of queries compressed by NEC [8]

Enumeration/Ordering Time by Varying #embeddings. Figure 20 shows the results of splitting the total processing time in Figure 12 into enumeration time and ordering time. The query vertex ordering time of CFL-Match remains the same when varying *#embeddings* since our ordering time is independent of *#embeddings*. However, the query vertex ordering time of Turbo$_{ISO}$ increases when more embeddings need to be reported. This is because, Turbo$_{ISO}$ heuristically restricts to only generate and materialize *#embeddings* embeddings in a data graph for query paths (see Section A.3).

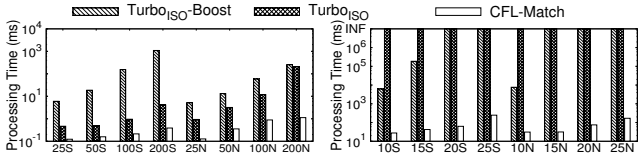


(a) HPRD (enumeration time) (b) Yeast (enumeration time)

(c) HPRD (ordering time) (d) Yeast (ordering time)

Figure 20: Enumeration/Ordering Time (vary *#embeddings*)

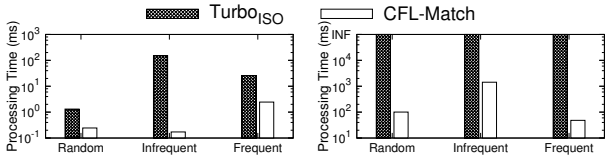
Eval-A-II: Additional Datasets and Queries. In the following, we report the experiment results on another two large real graphs, DBLP[3] and WordNet[4]. WordNet contains 133,445 edges, 82,670 vertices with an average degree 3.3, and 5 distinct labels; DBLP contains 1,049,866 edges, and 317,080 vertices with an average degree 6.6. Vertices in DBLP have no labels, we randomly assign a label out of 100 distinct labels to each vertex.



(a) DBLP (vary $|V(q)|$) (b) WordNet (vary $|V(q)|$)

Figure 21: Evaluating Turbo$_{ISO}$-Boost

Total Processing Time of Turbo$_{ISO}$-Boost. Figure 21 shows the processing time of Turbo$_{ISO}$-Boost against other algorithms on DBLP and WordNet. Turbo$_{ISO}$-Boost is faster than Turbo$_{ISO}$ for queries $q_{10S}, q_{15S}, q_{10N}$ on WordNet, but slower for other settings. Nevertheless, CFL-Match significantly outperforms both algorithms.



(a) DBLP (b) WordNet

Figure 22: Frequent Query vs Infrequent Query

Frequent/Infrequent Queries. In addition to random queries as tested in Section 6, we also consider frequent queries (*#embeddings* $> 10^4$ for DBLP and $> 10^8$ for WordNet), and infrequent queries (*#embeddings* $< 10^3$ for DBLP and $< 10^8$ for WordNet). The results are shown in Figure 22. CFL-Match is much faster than Turbo$_{ISO}$ for all 3 query sets (frequent/infrequent/random queries). CFL-Match is faster for frequent queries than infrequent queries on WordNet since it is easier to reach *#embeddings* results for frequent queries as a result of the permutation effect; however, it runs faster for infrequent queries on DBLP since the number of candidates for query vertices in infrequent queries is much smaller.

[3]https://snap.stanford.edu/data/

[4]http://vlado.fmf.uni-lj.si/pub/networks/data/