Turbo$_{ISO}$: Towards UltraFast and Robust Subgraph Isomorphism Search in Large Graph Databases

Wook-Shin Han
School of Computer Science and Engineering
Kyungpook National University, Korea
wshan@knu.ac.kr

Jinsoo Lee
School of Computer Science and Engineering
Kyungpook National University, Korea
jslee@www-db.knu.ac.kr

Jeong-Hoon Lee
School of Computer Science and Engineering
Kyungpook National University, Korea
jhlee@www-db.knu.ac.kr

ABSTRACT

Given a query graph q and a data graph g, the subgraph isomorphism search finds all occurrences of q in g and is considered one of the most fundamental query types for many real applications. While this problem belongs to NP-hard, many algorithms have been proposed to solve it in a reasonable time for real datasets. However, a recent study has shown, through an extensive benchmark with various real datasets, that all existing algorithms have serious problems in their matching order selection. Furthermore, all algorithms blindly permutate all possible mappings for query vertices, often leading to useless computations. In this paper, we present an efficient and robust subgraph search solution, called Turbo$_{ISO}$, which is turbo-charged with two novel concepts, *candidate region exploration* and the *combine and permute* strategy (in short, COMB/PERM). The candidate region exploration identifies on-the-fly candidate subgraphs (i.e, candidate regions), which contain embeddings, and computes a robust matching order for each candidate region explored. The COMB/PERM strategy exploits the novel concept of the *neighborhood equivalence class* (NEC). Each query vertex in the same NEC has identically matching data vertices. During subgraph isomorphism search, COMB/PERM generates only combinations for each NEC instead of permutating all possible enumerations. Thus, if a chosen combination is determined to not contribute to a complete solution, all possible permutations for that combination will be safely pruned. Extensive experiments with many real datasets show that Turbo$_{ISO}$ consistently and significantly outperforms all competitors by up to several orders of magnitude.

Categories and Subject Descriptors

H.3.1 [**Information storage and retrieval**]: Content Analysis and Indexing—*Indexing methods*; H.3.3 [**Information storage and retrieval**]: Information Search and Retrieval—*Search process*

General Terms

Algorithm; Performance

Keywords

Subgraph Isomorphism; Candidate Region Exploration; Neighborhood Equivalence Class; Combine and Permute Strategy

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

1. INTRODUCTION

Graphs are prevalently used for modeling complex structures such as molecules, protein interaction networks, and program dependence. Along with the growing importance of graphs, considerable research efforts have focused on efficient graph data management.

One of the most fundamental graph queries in graph databases is the subgraph isomorphism search. That is, given a query graph q and a data graph g, the subgraph isomorphism problem is to find all distinct mappings (or embeddings) of q in g. Subgraph isomorphism search is actively used in many applications such as protein interaction analysis [21], graph classification [4], computer-aided design of electronic circuits [10], scene analysis [1], and chemical compound search [17]. Although, in theory, subgraph isomorphism search belongs to NP-hard, we can efficiently find embeddings in real graph databases in practice if we exploit a good matching order and powerful pruning techniques.

Although recent efforts [3, 12, 6, 21, 20] have seemed to enjoy some success, a recent study [8] has shown, through an extensive benchmark with many datasets, that all existing algorithms have serious performance problems for some query/data sets, which are summarized as follows: (P1) all methods have serious problems in their matching order selection, showing exponential behavior in some query/data sets, which means no methods show *robust* search performance. However, the good news is that, for each query/data set, there exists at least one method which processes such query set in a reasonable time; (P2) blind exploitation of neighborhood based filtering of GraphQL [6] and SPath [21] often negates their search performance in many query/datasets. In contrast, in those query/data sets, QuickSI [12] often outperforms more recent competitors, such as GraphQL and SPath, when it chooses a good matching order. This finding again reinforces the importance of choosing good matching orders; (P3) parameters of GraphQL and SPath significantly affect search performance. However, finding best parameter values in advance for each query is very difficult in practice.

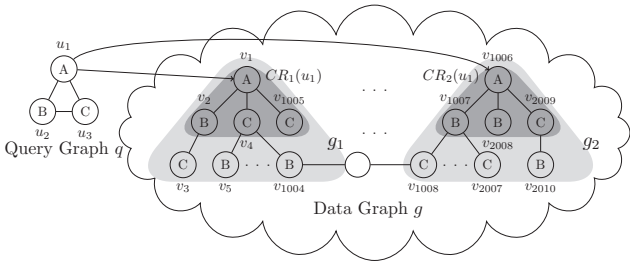
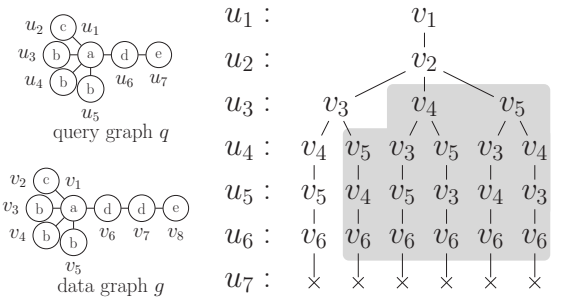


Figure 1: A query graph q and a data graph g.

Our new observation is that a good matching order for one data subgraph in g may be bad for other data subgraphs in g. Consider

the two data subgraphs g_1 and g_2 in Figure 1. If we try to match a query vertex with a data vertex of g_1 using the matching order O_1 =(u_1, u_2, u_3), after three matches (u_1, v_1), (u_2, v_2), (u_3, v_3), we know that there is no embedding in g_1. However, if we try to match using O_2 =(u_1, u_3, u_2), for each child vertex v_c in v_4, we must match u_2 with every child vertex of v_4. Thus, the number of matching attempts using O_2 is 1003 (1 for u_1, 2 for u_3, and 1000 for u_2). Likewise, if we use O_1 for g_2, the number of matching attempts is 1003, while using O_2 requires only three.

In addition, we observe another new problem (P4) in the existing methods; exhaustive enumerations between query vertices and data vertices often lead to useless computations, resulting in significant performance degradation. Consider the query q and data graph g in Figure 2(a). Figure 2(b) shows a recursion tree T_r where each node in the tree corresponds to a match between a query vertex u and a data vertex v. For example, the first three recursive calls find the matches (u_1, v_1), (u_2, v_2), and (u_3, v_3) in this order. Given a matching order (u_1, u_2, $\cdots$, u_7), all existing methods generate a similar recursion tree to T_r. However, when the seventh recursive call fails to match u_7 with v_7, we ensure that the subsequent recursive calls will not contribute to any valid embedding.



(a) a query and data graph. (b) a recursion tree T_r.

Figure 2: An example of useless enumerations.

These findings motivate us to develop a novel technique to efficiently and robustly process all such query/data sets. Our goal is to completely eliminate all these problems of existing methods. To this end, we propose 1) the novel concept of *candidate region exploration* and 2) a novel query processing strategy called *combine and permute* (in short, COMB/PERM). Candidate region exploration is a new approach for solving the matching order problem in subgraph isomorphism search, while COMB/PERM avoids the aforementioned useless enumerations between query/data vertices.

A candidate region for q is a data subgraph which may contain embeddings of the query graph. Additionally, if we perform subgraph isomorphism search on all candidate regions, we ensure that all embeddings can be obtained, avoiding all false negatives. For faster matching, minimizing the size of each data region is obviously important. Figure 1 illustrates two candidate regions, $CR_1(u_1)$ and $CR_2(u_1)$, for q, where both regions contain only four data vertices. The first matched vertex in each region is called the start vertex of each candidate region. v_1 and v_{1006} are the start vertices of $CR_1(u_1)$ and $CR_2(u_1)$, respectively.

In order to minimize the size of each candidate region, we first obtain a BFS search tree T_g from a selected root node u'_s of q so that each leaf is on the shortest path from u'_s. Then, for the start vertex v_s of each target candidate region, candidate region exploration identifies candidate data vertices for each query vertex by simply performing depth-first search using T_g starting from v_s. Next, we sort each path (accordingly, each leaf vertex) in T_g in ascending order of its number of candidate vertices and generate a matching order. Finally, given a candidate region, we find embeddings from candidate vertices for each query vertex (not from a data graph) according to the order obtained. For example, if we obtain a BFS search tree from q in Figure 1, the edge between u_2 and u_3 is removed. In $CR_1(u_1)$, a query vertex u_2 has a candidate vertex v_2, while u_3 has two candidate vertices v_4 and v_{1005}. Thus, a matching order (u_1, u_2, u_3) is obtained, and we next compute embeddings using these candidate vertices only. That is, we can easily determine that there are no embeddings in this region. Note that, for $CR_2(u_1)$, we obtain a different matching order (u_1, u_3, u_2) after candidate region exploration.

To avoid blindly permutating all possible mappings for query vertices u_3, u_4, and u_5 in Figure 2(a), the COMB/PERM strategy only generates combinations for such vertices. If a generated combination contributes to a valid mapping, the strategy permutes all possible mappings for the combination. Otherwise, it prunes out all other permutations for that combination. For example, we generate a combination (v_3, v_4, v_5) from data vertices $\{v_3, v_4, v_5\}$ for u_3, u_4, and u_5. If this combination fails to generate a valid mapping, then we do not permute the other combinations such as (v_3, v_5, v_4) and (v_4, v_3, v_5).

This presents a challenging question. "Which query vertices should be used for generating such combinations?" To answer this question, we propose the novel concept of *neighborhood equivalence class* (NEC). Each query vertex in the same NEC has identically matching data vertices. For example, u_3, u_4, and u_5 have the same matching data vertices $\{v_3, v_4, v_5\}$. Thus, u_3, u_4, and u_5 are in the same class. We need to identify all NECs before executing the subgraph isomorphism search. For this purpose, we propose an efficient polynomial-time algorithm called FINDNEC.

Our contributions are as follows: 1) We propose an efficient and robust subgraph search solution called TurboISO for completely solving the problems of existing methods. 2) We propose candidate region exploration to generate a robust matching order for each candidate region. 3) We propose a new concept of neighborhood equivalence class and the COMB/PERM strategy to generate only necessary enumerations. 4) Extensive experimental results show that TurboISO consistently and significantly outperforms the state-of-the-art methods by up to orders of magnitude.

The rest of this paper is organized as follows. Section 2 reviews subgraph isomorphism search and related work. Section 3 gives an overview of our solution for efficiently and robustly processing subgraph isomorphism queries. In Section 4, we propose the concept of neighborhood equivalence class and present an efficient algorithm to rewrite from an input query graph into an NEC tree. Section 5 presents an algorithm for performing candidate region exploration, and Section 6 presents a detailed matching algorithm based on COMB/PERM. Section 7 presents the results of performance evaluations. Section 8 summarizes and concludes the paper.

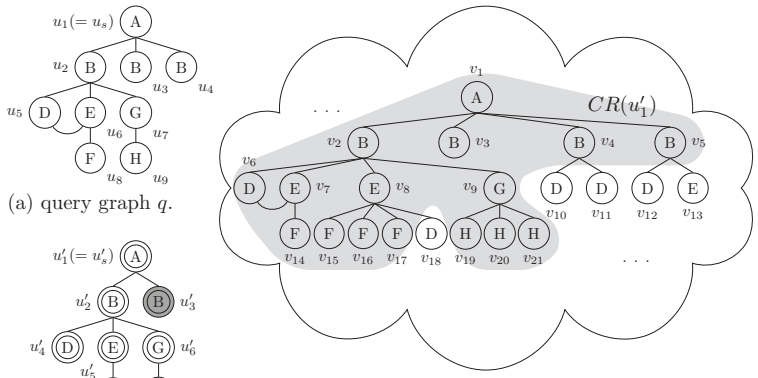
2. BACKGROUND

2.1 Problem Definition

A graph q is defined as (V, E, L) where V is the set of vertices, E ($\subseteq V \times V$) is the set of edges, and L is a label function that maps a vertex to a set of labels. Without loss of generality, all subgraph isomorphism algorithms can be easily extended to handle graphs whose edges have labels.

A graph $q = (V, E, L)$ is *isomorphic to a subgraph* of a data graph $g = (V', E', L')$ if there is an injection (or an embedding) $M : V \rightarrow V'$ such that, $\forall\, u \in V$, $L(u) \subseteq L'(M(u))$, and $\forall$ (u_i, u_j) $\in E$, ($M(u_i)$, $M(u_j)$) $\in E'$.

Now, the subgraph isomorphism problem is defined as follows: "Given a query graph q and a data graph g, find all distinct embeddings of q in g." In practice, however, we stop subgraph isomor-



(a) query graph q.

(b) NEC tree q'.

(c) data graph g.

CR	candidate vertices
$\mathrm{CR}(u'_1, v^*_s)$	$\{v_1\}$
$\mathrm{CR}(u'_2, v_1)$	$\{v_2\}$
$\mathrm{CR}(u'_3, v_1)$	$\{v_2, v_3, v_4, v_5\}$
$\mathrm{CR}(u'_4, v_2)$	$\{v_6\}$
$\mathrm{CR}(u'_5, v_2)$	$\{v_7, v_8\}$
$\mathrm{CR}(u'_6, v_2)$	$\{v_9\}$
$\mathrm{CR}(u'_7, v_7)$	$\{v_{14}\}$
$\mathrm{CR}(u'_7, v_8)$	$\{v_{15}, v_{16}, v_{17}\}$
$\mathrm{CR}(u'_8, v_9)$	$\{v_{19}, v_{20}, v_{21}\}$

(d) candidate subregions.

Figure 3: A running example

phism search after finding the first k embeddings. k is set to 1000 in [6, 21] and to 1024 in [14].

Consider a query graph q and a data graph g in Figure 1. Suppose that an edge between u_2 and u_3 is removed. Then, all embeddings of q in g are $\{(u_1, v_1), (u_2, v_2), (u_3, v_4)\}$, $\{(u_1, v_1), (u_2, v_2), (u_3, v_{1005})\}$, $\{(u_1, v_{1006}), (u_2, v_{1007}), (u_3, v_{2009})\}$, and $\{(u_1, v_{1006}), (u_2, v_{2008}), (u_3, v_{2009})\}$.

2.2 Related work

Most subgraph isomorphism algorithms [16, 3, 12, 6, 21, 20] are based on backtracking [8]. They first obtain a set of candidate vertices $C(u)$ for each query vertex u in a query graph q. For this purpose, each algorithm maintains a mapping table which maps a label to a list of vertices having that label. Given a matching order O, each algorithm invokes the main recursive subroutine SUBGRAPHSEARCH to match one query vertex with one data vertex at a time according to O. Note that the matching order plays a significant role in subgraph isomorphism performance.

In SUBGRAPHSEARCH, if the current query vertex u is selected, each algorithm refines $C(u)$ using its specific pruning rules. Then, for each candidate vertex refined v, each algorithm checks if the edges between u and already matched query vertices in q have corresponding edges between v and already matched data vertices of g. If v is qualified, we match u with v. Each algorithm then invokes SUBGRAPHSEARCH recursively by matching the remaining query vertices. We can finish the subgraph isomorphism search when all possible embeddings are obtained.

The Ullmann algorithm [16] is the first algorithm to tackle this problem. Since it uses the input order of query vertices and has almost no pruning rules to filter out candidate vertices, its performance is obviously the slowest among all existing algorithms. VF2 [3] optimizes Ullmann in the following aspect. In SUBGRAPHSEARCH, Ullmann chooses the current query vertex u arbitrarily, while VF2 chooses the u which is connected from the already matched query vertices. Note that, except for this constraint, VF2 does not define any matching order either. VF2 exploits this constraint to prune out candidate vertices from $C(u)$. The QuickSI algorithm proposes a matching order selection. Its matching order selection method chooses query vertices having infrequent vertex labels and infrequent, adjacent edge labels as early as possible. Although this greedy approach works for some datasets, it has a serious problem in its matching order for other datasets [8]. That is, choosing a more selective query edge can lead to a significantly less selective query path, which frequently occurs in the human protein interaction network. If we maintain statistics for all distinctive paths in the data graph, we can accurately calculate the selectivity of the path. However, this is infeasible in reality due to the huge amount of space and the expensive maintenance cost. GraphQL and SPath focus on minimizing the size of $C(u)$ by exploiting neighborhood-based filtering. GraphQL further minimizes it by performing the pseudo-isomorphism test, which generates BFS search trees for u and $v \in C(u)$, checks if the containment relationship exists between the two, and iterates this process by enlarging the BFS search trees. However, their matching order selection strategies using some simple formulae result in a serious problem in their matching orders. Furthermore, to store neighborhood information for each data vertex, some parameters must be provided by users.

Most recently, a new subgraph isomorphism algorithm was proposed in [14]. This algorithm is implemented in the memory cloud in Trinity [13]. It first decomposes a query into a set of basic query units called STwigs. Then, it obtains all possible intermediate results for each STwig. Here, a large amount of them must be transferred to other servers before join. Next, it performs block-based pipeline joins to merge all intermediate results. To define a join order, it uses a sampling based join optimization. However, it is unclear whether this algorithm is superior to the existing ones, since it has not been empirically compared with them. Its salient advantage is that it does not maintain any auxiliary index structure except the mapping table which maps a label to a list of vertices having that label. Thus, it can gracefully scale to support large graphs.

There are graph-index based subgraph isomorphism algorithms such as gIndex [17, 18], FG-Index [2], Tree+Δ [22], SwiftIndex [12], C-Tree [19], and gCode [23]. These utilize the filter-and-refinement strategy by using graph indexes. By using graph indexes as filters, we may prune out many unnecessary graphs at low cost during query processing. We store pairs of a graph feature and its posting list in the graph index. For a given query graph q, we enumerate all subgraphs of size up to $maxL$ for q and check whether such features are in the index. After finding features associated with their posting lists of graph IDs, we intersect all posting lists to obtain candidate graphs. Thus, the results of the filtering step are a set of graph IDs. If there is only a single, large graph in the database, using such graph indexes may not be useful, since the indexing-based algorithms must perform the subgraph isomorphism test if the query graph is contained in the single, large graph.

GraMi [11] is a frequent subgraph mining algorithm for a single graph g. It finds more generalized subgraph patterns where edges in a pattern correspond to distance-constrained paths in g. If at least σ embeddings of a pattern P exist in g (i.e., the support of $P \geq \sigma$), P is determined to be frequent in g. GraMi proposes a subgraph isomorphism algorithm based on a constraint satisfaction problem solver in order to find frequent patterns. However, the problem set-

ting in [11] is completely different from the traditional subgraph isomorphism problem in that it uses the minimum image-based support (MNI) σ. That is, let M_j be the j-th embedding for P in g. Let ∇_i be $\bigcup_{j=1}^{|M|} M_j(u_i)$, where u_i is the i-th query vertex, and $|M|$ is the number of all possible embeddings. Then, $min_{u_i \in P}|\nabla_i|$ is used to calculate the support of a pattern. Thus, to check whether a given pattern P is frequent under the MNI frequency metric, the core idea is to check if there are at least σ data vertices that are valid assignments for each query vertex without enumerating all possible embeddings of P in g. If not, we can safely determine that P is infrequent. Note that this property can not be used for the traditional subgraph isomorphism problem. Regarding the matching order, GraMi selects the query vertex which has the largest number of query constraints without considering its selectivity. However, this heuristic again would result in similar performance problems as the existing subgraph isomorphism methods.GraMi also uses an optimization under the MNI frequency metric to handle a special case when a query graph itself is completely symmetric. It would be interesting to apply our techniques to this problem to speed up frequent subgraph mining.

Research efforts on approximate subgraph search algorithms [15, 9, 7] are also active. Here, to find approximate embeddings, each algorithm uses its own similarity measure. Note that both exact subgraph search and approximate subgraph search algorithms have their own applications [8].

3. OVERVIEW OF Turbo$_{\text{ISO}}$

In order to integrate the concept of neighborhood equivalence class in both candidate region exploration and the main recursive subroutine, Turbo$_{\text{ISO}}$ first rewrites a query graph q into an NEC tree q' by performing BFS search from a given start query vertex. Note that each vertex in q' corresponds to an NEC. This way, we only need to match one NEC vertex with the corresponding data vertices regardless of how many query vertices correspond to the NEC vertex. Consider a running example in Figure 3, where Turbo$_{\text{ISO}}$ first rewrites q in Figure 3 (a) into q' in Figure 3 (b). From a given start query vertex u_s, we obtain q'. Note that the NEC vertex u'_3 corresponds to two query vertices u_3 and u_4, while the other vertices in q' correspond to a single vertex in q.

Next, the candidate region exploration process identifies candidate regions by performing a depth-first search for the start data vertex of each candidate region. After that, we obtain candidate data vertices for each NEC vertex. Here, candidate region exploration ensures that all candidate vertices for each NEC vertex u' have the same subtree as u'. Next, by ordering the leaf NEC vertices by the number of their candidate vertices, we can obtain a matching order for each candidate region. Here, we need to consider more sophisticated cases where 1) there exist non-tree edges in q, and 2) one NEC vertex corresponds to several query vertices, which will be elaborated in Section 5.3. In the running example, u'_2 has one candidate vertex v_2, while u'_3 contains four candidates $v_2 \sim v_5$. Since the leaf NEC vertices u'_4, u'_7, u'_8, and u'_3 have one, four, three, and four candidate vertices respectively, one possible generated order would be O_3 = (u'_1, u'_2, u'_4, u'_6, u'_8, u'_5, u'_7, u'_3).

The main recursive subroutine, SUBGRAPHSEARCH, generates embeddings using candidate data vertices of the NEC vertices according to the matching order obtained. For example, given a matching order O_3, when we match u'_5 with its candidate vertices v_7 and v_8, v_8 is pruned since it does not have an edge incident to v_6. When we match u'_3 with its candidate vertices, v_2 is pruned since it has already been matched with u'_2. After matching all NEC vertices, we find an embedding m = {(u_1, v_1), (u_2, v_2), (u_3, v_3), (u_4, v_4), $\cdots$, (u_8, v_{14}), (u_9, v_{19})}. Then, with COMB/PERM, we generate another embedding, m - {(u_3, v_3), (u_4, v_4)} $\cup$ {(u_3, v_4), (u_4, v_3)}.

Algorithm 1 outlines our subgraph isomorphism search algorithm, called Turbo$_{\text{ISO}}$. We first choose a start query vertex from q by calling CHOOSESTARTQVERTEX. Then, we obtain an NEC tree q' from q in Line 2. Next, for each start vertex of each region, we perform candidate region exploration from the root vertex u'_s of q' (Lines 3 $\sim$ 4). If we succeed in finding a region, we determine a matching order by calling DETERMINEMATCHINGORDER (Line 6). Now, we can map the start vertex u_s of q to the start vertex v_s of the region found by calling UPDATESTATE (Line 7). Like existing algorithms, we maintain two arrays M and F, where M is a mapping array, and F is an array storing information about which data vertices have been used in the current mapping. We next invoke the main recursive subroutine SUBGRAPHSEARCH to generate all possible embeddings for each region (Line 8). After we find all embeddings for each region, we restore the two arrays by RESTORESTATE (Line 9).

Algorithm 1 Turbo$_{\text{ISO}}$(q, g)

Input: query graph q, data graph g
Output: all embeddings of q in g

```
1: u_s := ChooseStartQVertex(q, g);
2: q' := RewriteToNECTree(q, u_s); // q' : NEC tree
3: for each v_s in {v|(v ∈ V(g)) and (L(u_s) ⊆ L(v))} do
4:     if ExploreCR(u'_s, {v_s}, CR) = FAIL then
5:         continue;
6:     order := DetermineMatchingOrder(q', CR);
7:     UpdateState(M, F, {u_s}, {v_s});
8:     SubgraphSearch(q, q', g, order, 1);/* 1 means "the first query
       vertex to match."*/
9:     RestoreState(M, F, {u_s}, {v_s});
```

4. QUERY REWRITE

In order to choose a start query vertex u_s which matches the start vertex of each region, we rank every query vertex by its value and select top-k query vertices by their ranking values. Here, we use a ranking function $Rank(u) = \frac{freq(g,L(u))}{deg(u)}$, where $freq(g, l)$ is the number of data vertices in g that have label l, and $deg(u)$ means the degree of u. In our experiment, we set k to 3. This ranking function favors lower frequencies and higher degrees. The rationale for this rule is that we want to minimize the number of candidate regions. This ranking function has been also used in [14]. Consider the running example in Figure 3. Since the rank value (= 1/3) of u_1 is the minimum, u_1 is selected as the start query vertex.

Among the top-three query vertices selected u_1, u_2, and u_3, we estimate the number of candidate regions more accurately by exploiting the following two filters whose costs are low. For each query vertex u selected, we obtain a set of candidate data vertices $C(u)$. Then, for each vertex v in $C(u)$, we first check if $deg(u) \leq deg(v)$. If qualified, we next exploit the *neighborhood label frequency filter* (NLF filter), which checks if $|adj(u, l)| \leq |adj(v, l)|$ for every distinct label l of the adjacent vertices of u. Here, $adj(u, l)$ represents a list of vertices having label l which are adjacent to u. Then, we can accurately estimate the number of candidate regions for each start query vertex selected. Lastly, we pick the one having the minimum number of candidate regions. Note that, we exclude any of u_2 or u_3 from applying these filters if its selectivity(=$\frac{|C(u)|}{|V(g)|}$) $> 5\%$. That is, we do not blindly apply these filters to such non-selective, start query vertices.

4.1 Neighborhood Equivalence Class

In this section, we formally define the concept of neighborhood equivalence class.

Definition 1. *Let $\simeq$ be an equivalence relation over all query vertices in q such that, $u_i(\in V(q)) \simeq u_j(\in V(q))$ if for every embedding m that contains (u_i, v_x) and (u_j, v_y) ($v_x, v_y \in V(g)$), there exists an embedding m' such that $m' = m-\{(u_i, v_x), (u_j, v_y)\} \cup \{(u_i, v_y), (u_j, v_x)\}$.*

The equivalence class of a query vertex u is a set of query vertices which are equivalent to (i.e., $\simeq$) u. This class is called *neighborhood equivalence class* (NEC). Consider a query q in Figure 3(a). Here, the query vertices u_3 and u_4 constitute one NEC, while each of the other vertices constitutes one NEC. Regarding the NEC of u_3 (= $\{u_3, u_4\}$), for every embedding m containing (u_3, v_3) and (u_4, v_4), there exists an embedding $m' = m-\{(u_3, v_3), (u_4, v_4)\} \cup \{(u_3, v_4), (u_4, v_3)\}$.

Let N_q be an NEC. Let $\mathcal{P}$ be a permutation of the vertices in N_q, and let M_{all} be a set of all embeddings. If $\{(\mathcal{P}[1], v_{i_1}), \cdots, (\mathcal{P}[|\mathcal{P}|], v_{i_{|\mathcal{P}|}})\} \subseteq m \in M_{all}$, then $\forall\ \mathcal{P}'$ in M_{all}, $m' = m-\{(\mathcal{P}[1], v_{i_1}), \cdots, (\mathcal{P}[|\mathcal{P}|], v_{i_{|\mathcal{P}|}})\} \cup \{(\mathcal{P}'[1], v_{i_1}), \cdots, (\mathcal{P}'[|\mathcal{P}|], v_{i_{|\mathcal{P}|}})\}$ for some $m' \in M_{all}$. Otherwise, $\forall\ \mathcal{P}'$, $\{(\mathcal{P}'[1], v_{i_1}), \cdots, (\mathcal{P}'[|\mathcal{P}|], v_{i_{|\mathcal{P}|}})\} \nsubseteq m'$ for all $m' \in M_{all}$.

We need to identify all neighborhood equivalence classes before executing subgraph isomorphism search. The following lemma states an important property of NEC which allows us to identify NECs by examining query vertices only.

Lemma 1. *Every query vertex u in an NEC N_q has the same label and the same set of adjacent query vertices. Otherwise, every query vertex u has the same label and is adjacent to every other vertex in N_q and has the same set $adj(u)$-N_q, where $adj(u)$ is the set of vertices adjacent to u. That is, all query vertices in N_q themselves form a clique.*

PROOF: We need to prove two conditions: 1) Suppose that every query vertex in N_q has the same label and same set of adjacent query vertices. Then, for any two query vertices u_i and u_j in N_q, it is clear that matching data vertices for u_i must be the same as those for u_j. Thus, for any embedding m, we can always find another embedding by exchanging u_i and u_j in m. 2) Suppose that every query vertex u has the same label and is adjacent to every other vertex in N_q and has the same set $adj(u)$-N_q. Then, $adj(u_i)-\{u_j\}$=$adj(u_j)-\{u_i\}$. Thus, for any two query vertices u_i and u_j, they match v_x and v_y respectively. Therefore, for any embedding m, we can always find another embedding by exchanging u_i and u_j in m. □

4.2 Rewrite to NEC tree

When we rewrite a query graph q into an NEC tree q', we need to achieve two goals: 1) candidate regions matching q' need to be minimized; and 2) all NECs must be efficiently obtained. It is obvious to exploit a BFS search tree from q to minimize the diameter of each candidate region on average. However, it is unclear whether we can efficiently identify all NECs using the BFS search tree. Lemma 2 states an important property of the NEC for this question. That is, we can achieve these two goals by performing a BFS search over the query graph.

Lemma 2. *Let $SP(u_s, u_i)$ be the set of all shortest paths from u_s to u_i. If the two query vertices u_i and u_j belong to the same NEC, for every shortest path $p_1 = p'(=(u_s, \cdots, u_x)) \bullet (u_x, u_i)$ in $SP(u_s, u_i)$, there exists a shortest path $p_2 = p' \bullet (u_x, u_j)$ in $SP(u_s, u_j)$, where $\bullet$ is the path concatenation operator. Thus, $|SP(u_s, u_i)|$=$|SP(u_s, u_j)|$.*

PROOF: According to Lemma 1, u_j must be adjacent to u_x as well, since u_i and u_j belong to the same NEC. Then, by substituting u_i with u_j in p_1, we can obtain p_2 whose length = $|p_1|$. Now, we need to show that p_2 is a shortest path from u_s to u_j. We prove this by contradiction. That is, suppose that p_1 is a shortest path from u_s to u_i, and that p_2 is not a shortest path. Then, there exists another shortest path $p_3 = (u_s, \cdots, u_y, u_j)$ whose length < $|p_2|$. According to Lemma 1, by substituting u_y and u_j with u_x and u_i in p_3, we can find a path $(u_s, \cdots, u_x, u_i)$ whose length = $|p_3|(< |p_1|)$. This contradicts the assumption that p_1 is a shortest path. □

Lemma 3 states the important relationship between the NEC of a vertex u' (denoted as $u'.NEC$) and the NEC of the child vertex u'_c of u' in the NEC tree. We exploit this relationship for both generating an NEC tree and performing subgraph isomorphism search.

Lemma 3. *Given any two NEC vertices u' and u'_c satisfying the parent-child relationship in the NEC tree, every query vertex u in $u'.NEC$ is adjacent to all vertices in $u'_c.NEC$.*

PROOF: We prove by contradiction. Suppose that there exists one query vertex u_j in $u'_c.NEC$ which is not adjacent to some vertex u_p in $u'.NEC$. Since u' and u'_c satisfy the parent-child relationship in the NEC tree, each query vertex in $u'_c.NEC$ must be adjacent to at least one query vertex in $u'.NEC$. Thus, there exists another query vertex u_i ($i \neq j$) in $u'_c.NEC$ which is adjacent to u_p. According to Lemma 2, $|SP(u_s, u_i)|$=$|SP(u_s, u_j)|$. Consider a shortest path $p_1 = (u_s, \cdots, u_p, u_i)$ in $SP(u_s, u_i)$. According to Lemma 2, there also exists $p_2 = (u_s, \cdots, u_p, u_j)$ in $SP(u_s, u_j)$. This contradicts that u_j is not adjacent to u_p. □

According to Lemma 3, in order to create child NEC vertices of a given NEC vertex u', it is sufficient to examine the query vertices in $u'.NEC$. Algorithm 2 shows a rewrite algorithm, REWRITETONECTREE, which converts an input query graph into its NEC tree. REWRITETONECTREE first creates the root vertex of an NEC tree whose NEC is $\{u_s\}$ (Line 2). For enabling BFS search over q guided by the NEC tree being created so far, REWRITETONECTREE uses two lists $V_{current}$ and V_{next} to maintain NEC vertices at the current level and NEC vertices at the next level to visit, respectively. Initially, the NEC vertex that REWRITETONECTREE has to visit in the next level is set to the root node u'_s of the NEC tree (Line 4). By visiting all NEC vertices in the current level, REWRITETONECTREE creates all NECs in the next level (Lines 7 $\sim$ 16). For each NEC vertex u'_c in the current level, we first collect the adjacent and unvisited query vertices of all query vertices in $u'_c.NEC$ (Lines 7 $\sim$ 9). We then group those vertices by the vertex label (Line 12), and, for each group, we invoke FINDNEC to create the child NEC vertices of u'_c (Lines 13 $\sim$ 14). We repeat this process until V_{next} is empty (Line 17).

Algorithm 2 REWRITETONECTREE (q)

```
1: create a root NEC vertex u'_s;
2: u'_s.NEC = [u_s];
3: mark u_s as visited;
4: V_current = []; V_next = [u'_s];
5: repeat
6:    V_current = V_next; V_next = ∅;
7:    for each NEC vertex u'_c in V_current do
8:       C = adj(u'_c.NEC[1]) ∪ ··· ∪ adj(u'_c.NEC[|u'_c.NEC|]);
9:       remove all visited query vertices from C;
10:      mark all query vertices in C as visited;
11:      if C is non-empty then
12:         group C by the vertex label into G_1, ··· , G_k;
13:         for each group G_i do
14:            NECVertices = FINDNEC(G_i);
15:            append NECVertices into V_next;
16:            make u'_c be the parent of all NECVertices;
17: until (V_next is empty )
```

Consider the query graph in Figure 3 (a). REWRITETONECTREE first creates u'_1 whose NEC is $\{u_1\}$. Since the adjacent query vertices of u_1 are $\{u_2, u_3, u_4\}$, each of which has the same label B, we make one group containing all those query vertices. Then,

by calling FINDNEC, we can create two NEC vertices u'_2 and u'_3 where u'_2.NEC =$\{u_2\}$, and u'_3.NEC =$\{u_3, u_4\}$. Now, V_{next} contains u'_2 and u'_3. We next examine all adjacent and unvisited vertices of the query vertices in u'_2.NEC, $\{u_5, u_6, u_7\}$. By grouping those query vertices by the vertex label, we obtain three groups, each of which is a singleton. By calling FINDNEC for each group, we can create three NEC vertices u'_4, u'_5, and u'_6 which are the child NEC vertices of u'_2. If we repeat this process, we can obtain the NEC tree finally as in Figure 3 (b). We explain another example using Figure 1. Here, u_1 is chosen as the start query vertex, and each query vertex corresponds to one NEC vertex. Thus, the resulting NEC tree q' is derived from q by removing the edge between u_2 and u_3. Thus, we have three NEC vertices u'_1, u'_2, and u'_3, which correspond to u_1, u_2, and u_3, respectively.

Now, we explain how to find NECs from a given set of candidate query vertices using FINDNEC. We first group those vertices by their adjacent vertices. According to Lemma 1, if the size of each group is larger than one, i.e., every query vertex in the group has the same label and the same set of adjacent query vertices, then we create an NEC vertex that contains all query vertices in the group. Otherwise, we examine all singleton groups to see if any groups can be merged into one NEC N_q such that every query vertex u_c in N_q has the same label and is adjacent to every other vertex in N_q and has the same set $adj(u_c)$-N_q Note that the worst time complexity of FINDNEC is $O(|G|^2)$.

5. CANDIDATE REGION EXPLORATION

As explained in the introduction, by performing subgraph isomorphism search for candidate regions only, we can obtain all embeddings for a given query graph q. In order to minimize the size of the candidate region, we rewrite q into an equivalent NEC tree q'. Here, by using q' instead of q, we can also accelerate the candidate region exploration process, since many query vertices in the same NEC are merged into one NEC vertex. The following section explains the concept of *candidate subregion* which stores candidate data vertices for each NEC vertex. Note that the union of all candidate subregions in a candidate region is equal to the candidate region.

5.1 Candidate Subregion

Given an NEC tree q' for a query graph q, we identify candidate regions by traversing g by using q'. For this purpose, the candidate region exploration process traverses the subgraph rooted at the start vertex of each candidate region.

During candidate region exploration, we can identify candidate data vertices for each NEC vertex, which will be used for subgraph isomorphism search. In order to store these data vertices, we propose the concept of the candidate subregion denoted as $CR(u', v)$, where u' is an NEC vertex, and v is a data vertex. $CR(u', v)$ contains data vertices such that they match u' and are child vertices of v in the DFS tree. We formally define $CR(u', v)$ in Definition 2.

Definition 2. *$CR(u', v)$ is a set of data vertices $\{v'\}$ such that 1) v' is a child vertex of v in the DFS search tree, 2) all NEC vertices on the path from u'_s to u' matches all data vertices on the path from v_s to v' in the DFS search tree, and 3) each subtree of u' matches the corresponding subtree of v' in the DFS search tree, where u'_s is the root vertex of q', and v_s is the start vertex of the candidate region.*

Figure 4 illustrates the semantics of $CR(u', v)$. To store a candidate vertex (i.e., v_s) for u'_s, we assume that there exists an artificial vertex v^*_s that is the parent of v_s.

Consider an NEC tree and a data graph in Figure 3. Here, $CR(u'_2, v_1) = \{v_2\}$, since v_2 can match u'_2, and the subtrees of u'_2 match those of v_2. Note that $CR(u'_2, v_1)$ does not contain v_3, v_4, or v_5, since their subtrees do not match those of u'_2. Similarly, $CR(u'_3, v_1)$ =$\{v_2, v_3, v_4, v_5\}$. Figure 3(d) illustrates all candidate subregions for this example.

5.2 Algorithm and Data Structure

When we traverse a candidate region, we need to test if the data vertex v' which is adjacent to v can belong to $CR(u', v)$ for an NEC vertex u'. In order to speed up the test and prune out non-promising data vertices efficiently, we first check if $deg(u'.NEC[1]) \leq deg(v')$. Note that here we use the degree of the first query vertex $u'.NEC[1]$ rather than that of u'. For example, consider Figure 3(b), where $u'_1.NEC[1] = u_1$. When we try to match u'_1 with v_1 in Figure 3(c), the degree of u_1 is three while the degree of u'_1 is two. Thus, if the degree of v_1 is less than three, v_1 is safely pruned. Then, we exploit the NLF filter before traversing the subtree of v'. If not qualified, we prune such v' without traversing the subtrees of v'. Again, we invoke $NLF(u'.NEC[1], v')$ for this purpose.

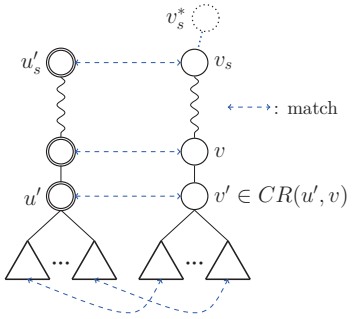


Figure 4: Semantics of $CR(u', v)$.

Algorithm 3 outlines the candidate region exploration process, EXPLORECR, which performs a depth-first search from the start vertex of a candidate region guided by an input NEC query tree. Inputs to EXPLORECR are an NEC vertex u', and a candidate data vertex set V_M for u'. For each unvisited data vertex v', EXPLORECR exploits the two tests to prune out unpromising data vertices (Line 2). If v' is qualified, we traverse the child vertices of v' by recursively calling EXPLORECR (Lines 6 $\sim$ 7). Here, we order child query vertices u'_cs of u' according to the size of $adj(v', \mathrm{L}(u'_c))$ (Line 6). This would increase the probability that v' is to be pruned early if its subtrees do not match the subtrees of u'. If the subtrees of v' do not match the subtrees of u', we clear the candidate subregions for all visited vertices in the subtrees of v' (Lines 8 $\sim$ 9). Thus, we invoke CLEARCR for this purpose. If v' is qualified, we append it into $CR(u', v)$, where v is the parent vertex of v' in the DFS search tree (Line 15). If $|CR(u', v)| < |u'.NEC|$, we cannot map all query vertices in $u'.NEC$ to the vertices in $CR(u', v)$, and thus, we clear any subregions that we have found (Lines 16 $\sim$ 18).

Consider the running example in Figure 3. EXPLORECR performs depth-first search starting from v_1 using the NEC tree q' and identifies candidate vertices for each NEC vertex, which are stored in the corresponding candidate subregions. Note that v_{18} and v_{10} $\sim$ v_{13} are determined to be not in the candidate region by Line 6.

If v' is not qualified, CLEARCR should be invoked for *each subregion* for all visited vertices in the subtrees of v'. To speed up this process, we propose an efficient data structure for representing $CR(u', v)$s, where we maintain one array for each NEC vertex u' so that all $CR(u', v)$s are stored in the array $CR(u', -)$. Each element in $CR(u', -)$ consists of (v', a list L of ranges [s,e]) such that $|L|$ is equal to the number of child NEC vertices of u'. If the i-th child NEC vertex of u' is u'', then the candidate vertices for CR(u'', v') are stored from $CR(u'', -)[L_i.s]$ to $CR(u'', -)[L_i.e]$. By do-

ing this, we only need to invoke CLEARCR for each descendent NEC vertex of u'.

Figure 5 shows an example of the data structure for $CR(u', -)$. Consider $CR(u'_1, -)$ which corresponds to $CR(u'_1, v^*_s)$. There is one element $(v_1, [1:1], [1:4])$ in $CR(u'_1, -)$. Since u'_1 has two child NEC vertices u'_2 and u'_3, the candidate vertex for $CR(u'_2, v_1)$ is stored in $CR(u'_2, -)[1]$, while the candidate vertices for $CR(u'_3, v_1)$ are stored from $CR(u'_3, -)[1]$ to $CR(u'_3, -)[4]$. Similarly, consider $CR(u'_5, -)$. The second element in $CR(u'_5, -)$ is $(v_8, [2:4])$. Thus, the candidate vertices for $CR(u'_7, v_8)$ are stored from $CR(u'_7, -)[2]$ to $CR(u'_7, -)[4]$.

Algorithm 3 EXPLORECR (u', V_M, CR)

```
1: for each unvisited data vertex v' in V_M do
2:   if deg(u'.NEC[1]) > deg(v') or NLF(u'.NEC[1], v') = FAIL
     then
3:     continue;
4:   mark v' as visited;
5:   matched := true;
6:   for each child query vertex u'_c of the u' ordered by the
     |adj(v', L(u'_c))| do
7:     if EXPLORECR(u'_c, adj(v', L(u'_c)), CR) = FAIL then
8:       for each visited child query vertex u''_c of u' except u'_c do
9:         CLEARCR(u''_c, v', CR) ;
10:        matched := false;
11:        break;
12:    mark v' as unvisited;
13:  if matched = false then
14:    continue;
15:  append {v'} into CR(u',v) such that v is the parent vertex of v' in
     the DFS search tree;
16: if |CR(u', v)| < |u'.NEC| then
17:   CLEARCR(u', v, CR);
18:   return FAIL;
19: return DONE;
```

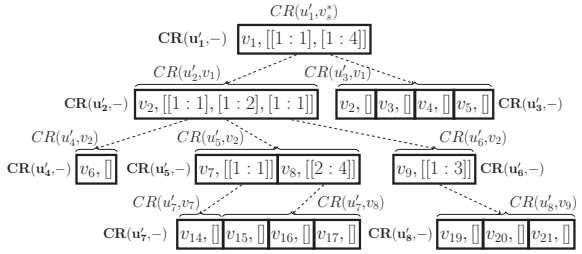


Figure 5: Representation of $CR(u', -)$.

Note that if a query is a path, and the start vertex of the query is at the end of the path, TurboISO can directly generate embeddings without calling the main recursive subroutine while we perform candidate region exploration. This simple optimization is typically effective if the query size is small as we will see in our experiments.

5.3 Matching Order Selection

If a query q has n vertices, u_1, u_2, $\cdots$, u_n, then the size of its search space becomes $|C(u_1)| \times |C(u_2)| \times \cdots \times |C(u_n)|$, where $C(u_i)$ is a set of candidate vertices that match u_i. Thus, we need to find a matching order so that the intermediate search space is minimized. However, this is a very difficult problem since we do not know the join selectivity of each pair of query vertices. Although every method tries to solve this problem using its own estimation formula, it fails to find a good matching order robustly due to serious problems in its estimation formula. For example, SPath finds the most selective path which starts from an already matched query vertex. Here, the selectivity function $sel(p)$ for a given path p is calculated as $\frac{2^{|\mathrm{V}(p)|}}{\prod_{u \in \mathrm{V}(p)} |C(u)|}$, where $\mathrm{V}(p)$ denotes all vertices in a path p. Note that the denominator represents the join cardinality of the candidate sets for all query vertices in p. However, this overestimate leads to significant errors in estimating the join cardinality.

On the other hand, rather than relying on unrealistic estimation formulae, TurboISO performs candidate region exploration in order to count the number of candidate vertices for a given path in an NEC tree. This way, we can obtain almost *exact* selectivity for each query path from the start NEC vertex. Furthermore, we calculate the selectivity for each candidate region rather than for the whole graph. This is how TurboISO achieves more robust matching orders than the existing methods. Thus, we can exploit this precise information in selecting a matching order. Specifically, we sort each leaf vertex u' (each path, accordingly) by $|CR(u', -)|$. That is, we favor a path, such that the number of candidate vertices for the path is the smallest among the paths not selected yet. Consider Figure 6, where we have three query paths $p_1 \sim p_3$. For the first candidate region, through the candidate region exploration, we know that the cardinalities for those paths are 10, 10000, and 5, respectively. For the second candidate region, their cardinalities are 10000, 10, 5, respectively. If we perform the subgraph isomorphism search using p_1 and p_2 first, a lot of partial solutions will be enumerated. However, all of them will be eventually discarded by matching with p_3, since there are no edges between the data vertices having X and those having Z. On the other hand, if we perform the subgraph isomorphism search using p_1 and p_3, no results are generated, and thus, we can prune the entire search space. Accordingly, for the second candidate region, if we perform the subgraph isomorphism search using p_2 and p_3 first, we can prune the entire space. Otherwise, we are bound to enumerate a lot of partial solutions.

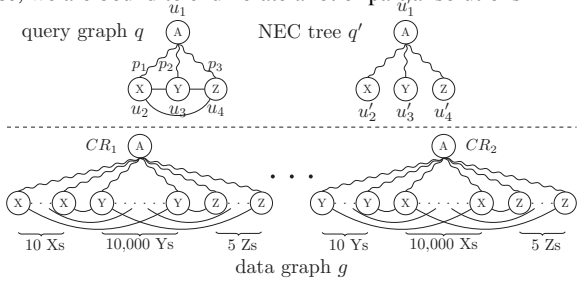


Figure 6: Importance of matching orders in different regions.

Here, we need to consider more sophisticated cases where 1) there exist non-tree edges in q, and 2) one NEC vertex corresponds to several query vertices. When we perform candidate region exploration, we do not consider non-tree edges in the query graph. Thus, we take this into account in estimating the number of candidate vertices for a given NEC vertex. Suppose that there are j non-tree edges incident to an NEC vertex u'. Then, we divide the number of candidate vertices for u' by $j + 1$. This strategy is consistent with the computation of $Rank(u)$ in Section 4. We need to apply this reduction factor to subtrees of u'. For example, in Figure 3, the number of candidate vertices for u'_7 is four. However, since there is a non-tree edge between u_5 and u_6, we divide four by two, thereby our estimated value, two, becomes closer to the actual number, one (i.e., $|\{v_{14}\}|$). Thus, we obtain a matching order O_4 = (u'_1, u'_2, u'_4, u'_5, u'_7, u'_6, u'_8, u'_3).

When a path p in the NEC tree consists of NEC vertices whose NEC sizes are greater than one, we have many corresponding query paths from p in the original query graph q. Since each query path in q has the same number of candidate vertices as that of p, we may still apply our strategy to this case as well. However, when we match the path p in the NEC tree with candidate vertices, we actually match many query paths from p at the same time. Thus, the actual number of candidate vertices for those paths are not the same as that for p. If there are many NEC vertices on p, we cannot

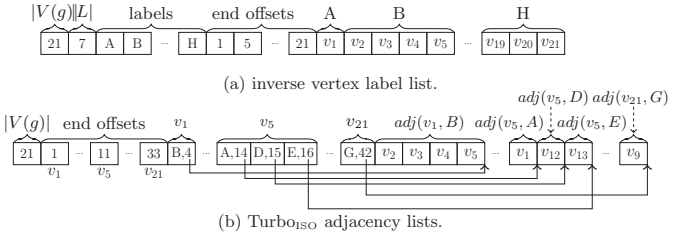
know in advance the number of matches for the query paths without performing an expensive subgraph isomorphism search. The good news is that we can compute the exact number of candidate vertices when there exists only one NEC vertex whose NEC size > 1 at the end of the path, which frequently occurs in the query graph. Consider the NEC vertex u'_3 in Figure 3. For the NEC path $u'_1.u'_3$, we have two corresponding query paths $u_1.u_3$ and $u_1.u_4$. Thus, for the leaf query vertex pair of (u_3, u_4), we have six data vertex pairs, (v_2, v_3), (v_2, v_4), (v_2, v_5), (v_3, v_4), (v_3, v_5), and (v_4, v_5).

Thus, if the path p has an NEC node u' whose NEC size > 1 at the end of p, we compute the number of candidate vertices for the corresponding query vertices for u' by using Equation (1), where $P(u')$ is the parent NEC vertex of u'. For example, the number of candidate vertices for u'_3 ((u_3, u_4), accordingly) is $\binom{|CR(u'_3,v_1)|}{|u'_3.NEC|}$ $\left(=\binom{4}{2}\right)$. Note that the formula $\binom{|CR(u',-)|}{|u'.NEC|}$ may not compute the correct value, since the selected data vertices such as v_3 and v_4 that match u_3 and u_4 respectively may not have the same parent data vertex in the DFS search tree that matches u_1.

$$\sum_{v \in CR(P(u'),-)} \binom{|CR(u',v)|}{|u'.NEC|} \quad (1)$$

6. MATCHING WITH COMB/PERM

Before explaining the subgraph isomorphism process, we explain the disk representation of a data graph. TurboISO represents a graph using two structures: 1) inverse vertex label list that allows access to the ordered vertex ID list by a given vertex label (see Figure 7(a)) and 2) adjacency list (see Figure 7(b)) of each vertex which stores adjacency information. Here, we group adjacent vertices by the label in the adjacency list. This way, we can efficiently compute $adj(v, l)$, which is used to prune unpromising vertices during candidate region exploration.



(a) inverse vertex label list.

(b) Turbo$_{ISO}$ adjacency lists.

Figure 7: Disk representation of the data graph in Figure 3 (c).

6.1 Combine and Permute

In the subgraph isomorphism test, when we match the query vertices in $u'.NEC$ with a set of candidate data vertices C, we actually have to map all query vertices in $u'.NEC$ to some data vertices in C. For this purpose, we propose the combination operator, $NextComb(u', C)$. We elaborate how to find the set C in the next subsection. $NextComb(u', C)$ returns one $|u'.NEC|$-combination from C at a time whenever we invoke this function. Consider the running example in Figure 3. If we invoke $NextComb$ (u'_3, $\{v_3, v_4, v_5\}$), it first returns $[v_3, v_4]$. Thus, we can map u_3 and u_4 to v_3 and v_4, respectively. If we invoke $NextComb$ the next time, we can map u_3 and u_4 to v_3 and v_5, respectively.

Now, we explain the permutation operator $GenPerm(M, q', i)$, where M is the mapping table, q' is an NEC tree, and i is the index to a list of NEC vertices. In the main recursive subroutine of the subgraph isomorphism test, we invoke $GenPerm(M, q', 1)$ to process the first NEC vertex in q'. Repeatedly invoking $NextPerm$ (M, u') generates all possible permutations for the combinations generated by $NextComb$, where each combination is generated from one NEC vertex u'. We invoke $NextPerm(M, V(q')[i])$ to generate one permutation for $V(q')[i].NEC$ and to map the permutation to M accordingly (Line 6), where $V(q')[i]$ is the i-th NEC vertex in q'. Then, we recursively call $GenPerm(M, q', i + 1)$ to generate permutations for the remaining $V(q')[i + 1]$s (Line 7). Recall the running example in Figure 3 again. Suppose that an embedding M containing (u_3, v_3), (u_4, v_4) has been determined as a solution, we can generate all permutations for these query vertices, $[u_3, u_4]$ and $[u_4, u_3]$, and apply these permutations to the embedding, thereby generating two solutions.

Algorithm 4 GENPERM(M, q', i)

Input: M : a mapping table, q' : an NEC tree, i : index

```
1: if i = |V(q')|+1 then
2:     report M and return; /*i.e., we have processed all NEC vertices */
3: if |V(q')[i].NEC| =1 then /*V(q')[i] denotes the i-th NEC vertex in q'*/
4:     GenPerm(M, q', i+1); /*Then, we don't need to generate permutations if the NEC size is one*/
5: else
6:     while NextPerm(M, V(q')[i]) ≠ DONE do
7:         GenPerm(M, q', i+1);
```

6.2 Main Recursive Subroutine

In candidate region exploration, we match an NEC vertex u' with data vertices. However, in subgraph isomorphism search, we have to match all query vertices in $u'.NEC$ with the corresponding data vertices. This presents the following challenging questions: (Q1) Given an NEC vertex u', which data vertices should be considered as candidates to match every query vertex in $u'.NEC$? (Q2) How should edges among the query vertices in $u'.NEC$ be handled? Note that these edges are not represented in the NEC tree.

Suppose that we try to match u' with the data vertices. Note that all query vertices in $P(u').NEC$ have already been matched with the corresponding data vertices, where $P(u')$ is the parent NEC vertex of u'. Then, in the mapping table M, query vertices $P(u').NEC[1]$, $\cdots$, $P(u').NEC[|P(u').NEC|]$ have already been mapped to $v_{M[P(u').NEC[1]]}, \cdots, v_{M[P(u').NEC[|P(u').NEC|]]}$, respectively. The following lemma answers the first question (Q1).

Lemma 4. *Given an NEC vertex u', a data graph g, and a mapping table M, any data vertex $v \in V(g)$ that u' matches must be in every $CR(u', v_{M[P(u').NEC[i]]})$ ($1 \le i \le |P(u').NEC|$).*

PROOF: Proof by contradiction. Suppose that u' matches a data vertex v, but v is not in $CR(u', v_{M[P(u').NEC[i]]})$ for some i. That is, v is not connected to $v_{M[P(u').NEC[i]]}$. Then, some query vertex in $u'.NEC$ is not connected to $P(u').NEC[i]$. However, according to Lemma 2, every query vertex u in $P(u').NEC$ is connected to all vertices in $u'.NEC$. This contradicts the assumption that some query vertex in $u'.NEC$ is not connected to $P(u').NEC[i]$.

Consider an example in Figure 8. The data vertices v_5 and v_6 in $CR(u'_3, -)$ are considered as candidates since they are both in $CR(u'_3, v_2)$ and $CR(u'_3, v_3)$, while v_4 and v_7 are not considered as candidates for the query vertex u_4, since they are in either $CR(u'_3, v_2)$ or $CR(u'_3, v_3)$. That is, v_4 is not connected to v_3, and v_7 is not connected to v_2. However, their corresponding query vertices u_2 and u_3 are connected to u_4.

Now, we explain how to answer the second question. Here, the query vertices in $u'.NEC$ themselves must form a clique. If the query vertices in $u'.NEC$ form a clique, the corresponding candidate data vertices must form a clique as well. Otherwise, we can prune out those data vertices safely. Consider the example in Figure 8, where we match $u'_2.NEC$ with $\{v_2, v_3\}$. Since u_2 and u_3 in $u'_2.NEC$ themselves form a clique, the corresponding data vertices must form a clique. Otherwise, they are pruned.

SUBGRAPHSEARCH outlines the subgraph isomorphism search to generate embeddings. In Line 2, for an NEC vertex u', we obtain a set C of candidate vertices for $u'.NEC$ by intersecting all

$CR(u', v_{M[P(u').NEC[i]]})$s according to Lemma 4. Next, we obtain one $|u'.NEC|$-combination from C for the query vertices in $u'.NEC$ (Line 4). If any data vertex in the combination has already been mapped before, this combination is safely pruned (Lines 7 $\sim$ 8). Note that we push this constraint to generate the combination for speedup. Next, if all query vertices in $u'.NEC$ themselves form a clique, we check if the data vertices in the combination themselves form a clique as well (Lines 9 $\sim$ 11). Otherwise, we prune out these vertices. Next, by calling IsJoinable, we check if, for each query vertex u in $u'.NEC$, the edges between u and already matched query vertices of q have corresponding edges between $V[i]$ and already matched data vertices of g (Lines 13 $\sim$ 18). If every vertex in the combination is qualified, it is mapped to the corresponding query vertex u, and SubGraphSearch updates status information by calling UpdateState (Line 19). Once we examine all NEC vertices (all query vertices, accordingly), we generate embeddings by calling GenPerm (Line 21). Otherwise, SubGraphSearch proceeds to match the remaining NEC vertices by recursively calling itself (Line 23). Finally, all changes done by UpdateState are restored by calling RestoreState.

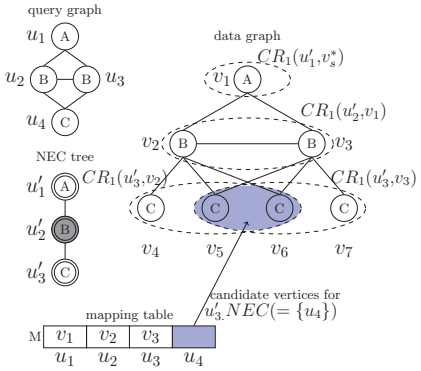


Figure 8: An example for Lemma 4.

Consider the running example in Figure 3. Given the matching order O_4 = (u'_1, u'_2, u'_4, u'_5, u'_7, u'_6, u'_8, u'_3), when we match u'_5 with its candidate vertices, v_8 is pruned by Lines 14 $\sim$ 16. When we match u'_3 with its candidate vertices, v_2 is pruned by Lines 7 and 8. Since $u'_3.NEC$ contains two query vertices u_3 and u_4, we generate combinations from the candidate data vertices for u_3 and u_4 (Line 4) and then generate all possible permutations using those combinations (Line 21).

Unlike all existing algorithms, Turbo$_{\text{ISO}}$ has two phases to process subgraph isomorphism queries. One minor drawback of this approach is that, if we want to find k subgraph isomorphisms only, the existing approach may not access all data vertices in candidate regions for some queries, thus making early termination possible. To solve this problem, Turbo$_{\text{ISO}}$ *interleaves* the candidate region exploration process and the subgraph isomorphism process. That is, for each leaf in the NEC tree, if the number of vertices obtained by candidate region exploration exceeds k, we do not explore more vertices for the NEC vertex. Next, we estimate the number of candidate data vertices for each leaf NEC vertex using data vertices we have explored so far as a sample. Then, we execute the subgraph isomorphism test with the candidate vertices we have found so far. If we are unable to find k subgraph isomorphisms with these candidate vertices, we resume the candidate region exploration to find more candidate vertices. Thus, we can clearly avoid this drawback.

7. EXPERIMENTS

We evaluate the performance of the state-of-the-art algorithms against Turbo$_{\text{ISO}}$. The algorithms considered are as follows: (1) VF2 [3], (2) QuickSI [12] (in short, QSI), (3) GraphQL [6] (in short, GQL), (4) GADDI [20] (in short, GAD), and (5) SPath [21] (in short, SPA). Additionally, we compare Turbo$_{\text{ISO}}$ with the most recent state-of-the-art algorithm [14] in Section 7.5. Our main objective is to show that Turbo$_{\text{ISO}}$ 1) *consistently* outperforms all competing algorithms regardless of query/data sets and 2) *significantly* outperforms the competing algorithms by up to several orders of magnitude for datasets containing large graphs.

Algorithm 5 SubGraphSearch $(q, q', g, order, d_c)$

```
1: u' := order[d_c];
2: C := CR(u', v_{M[P(u').NEC[1]]}) ∩ ··· ∩
                    CR(u', v_{M[P(u').NEC[|P(u').NEC|]]});
3: while true do
4:    C' := NextComb(C, |u'.NEC|) ; // C' is a list
5:    if |C'| = 0 then
6:       return; /*That is, no more combination exists.*/
7:    if ∃v ∈ C' such that F[v] = true then
8:       continue;
9:    if u'.NEC form a clique then
10:      if ∃(v, v'), v, v' ∈ C' such that v ≠ v' ∧ (v, v') ∉ E(g) then
11:         continue ;
12:   matched:=true;
13:   for i := 1 to |u'.NEC| do
14:      if not IsJoinable(q, g, M, u'.NEC[i], C'[i]) then
15:         matched:=false;
16:         break;
17:   if matched = false then
18:      continue;
19:   UpdateState(M, F, u'.NEC, C');
20:   if |V(q')| = d_c then
21:      GenPerm(M, q', 1);
22:   else
23:      SubGraphSearch(q, q', g, order, d_c + 1);
24:   RestoreState(M, F, u'.NEC, C');
```

For comprehensive performance study, we use the same benchmark setting as [8]. For this purpose, we obtain the source code used in [8] and implement Turbo$_{\text{ISO}}$ on top of this framework for fair comparison. Regarding GraphQL and SPath, we use best parameter values for each dataset as in [8]. We use four datasets referred to here as AIDS, NASA, Yeast, and Human. These datasets have different characteristics. The AIDS dataset contains 10,000 small graphs whose average size is about 27 (in terms of the number of edges). The number of unique labels in AIDS is 51. The NASA dataset contains 36,790 trees where the average tree size is about 32, and the number of unique labels is much larger (=117,302) than AIDS. The Yeast dataset contains only one large graph having 3,112 vertices and 12,519 edges, where a vertex and an edge correspond to a protein and an interaction between two proteins, respectively. Since one protein can appear in several cellular components and biological processes, a vertex can have multiple labels. This graph is denser than those in AIDS and NASA. The Human dataset contains a large graph modeling a protein interaction network of 4,675 proteins (i.e., vertices) and 86,282 interactions (i.e., edges), which is larger and denser than Yeast. This graph has a fewer number of vertex labels (=90) and a larger average degree (=36.82) than Yeast (=8.05). The Human dataset also consists of vertices having multiple labels.

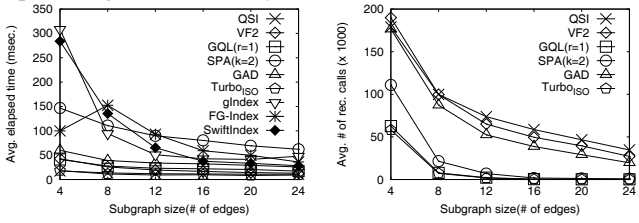
We use the same query sets as in [8]. Specifically, for the AIDS and NASA datasets, we use six query sets (Q_4, Q_8, $\cdots$, Q_{24}), each of which contains 1,000 query graphs of the same size. Note that in these query sets, a small size query graph q_s of size s is constructed from a large size query graph q_l of size s + 4 by removing edges until q_l is connected and contains s edges. Thus, the containment relationship between q_s and q_l is satisfied. For the Yeast and Human datasets, we use 10 query sets by varying the number of query sizes from 1 to 10. For those datasets, we use two more query sets (clique/path queries), generated by a tool used in [6]. These query

sets do not satisfy the containment relationship. Since overall performance trends of the algorithms in the Yeast dataset are similar to those in the Human dataset, and thus, we omit them to save space.

We conducted all the experiments on a Windows Vista machine with Xeon Quad Core 2.27 GHz CPU and 8 GBytes RAM. We set the page size to 8 KBytes. We used the average number of SUBGRAPHSEARCH calls, and the average wall clock time as the performance metrics for query execution. We stopped subgraph isomorphism search after the first 1,000 subgraph isomorphisms were found, as it was done in [6, 21, 8]. For the database build time, since we do not maintain any auxiliary data structures, for AIDS and NASA, our building time is similar to VF2 which has the lowest database building time among all competitors. If there are multiple labels in data vertices as in Yeast and Human, since we group each adjacency list by label, our index building time is about 4-5 times slower than VF2 due to larger adjacency lists, but is still much faster than the other competitors. Since the subgraph isomorphism search is a CPU intensive task, TurboISO significantly outperforms all competitors for those datasets in spite of the increased reading time in Yeast and Human.

7.1 Experiment on the AIDS dataset

Figure 9 shows the experimental results for AIDS. All algorithms behave well since they choose relatively good matching orders for this dataset and prune unpromising candidates early. Here, as we increase the query size, the elapsed time of every algorithm decreases. Due to the containment relationship among the query sets in AIDS, as we increase the query size, the number of embeddings may decrease, and thus, we achieve better performance by pruning unpromising candidates early.



(a) average elapsed time. (b) avg. # of recursive calls.

Figure 9: AIDS dataset.

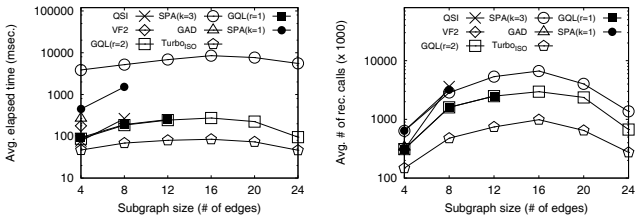
In terms of average number of recursive calls, TurboISO consistently outperforms all competitors for all query sets. Due to good matching order and the COMB/PERM strategy, TurboISO outperforms VF2, QuickSI, GraphQL, GADDI, and SPath by 6.96, 7.28, 1.09, 5.97, and 2.12 times, respectively. Note that the numbers of recursive calls for VF2, GraphQL, SPath, and GADDI are 1.05, 6.7, 3.44, and 1.22 times smaller than QuickSI. Note also that, we use the optimized version of VF2 used in [8].

In terms of average elapsed time, TurboISO also consistently outperforms all competitors for all query sets, although the ranking order (TurboISO, QuickSI, VF2, GraphQL, GADDI, and SPath) is completely different. Since TurboISO has the candidate region exploration before executing the subgraph isomorphism test, this processing time is included in the elapsed time. Despite this overhead, TurboISO consistently outperforms all competitors. Although the average number of recursive calls for VF2 is slightly smaller than QuickSI, the average cost of recursive calls for VF2 is 2.82 times larger than that for QuickSI. This is because ISJOINABLE of TurboISO and QuickSI iterates through adjacent and already matched query vertices of u, while ISJOINABLE of VF2 iterates all adjacent query vertices of u. Regarding GraphQL, the filtering time spent for neighborhood based filtering and pseudo-isomorphism based pruning constitutes about 60 % of the total elapsed time, and the average cost of recursive calls for GraphQL is 1.94 times larger than VF2 due to 1) query optimization overhead and 2) absence of pruning rules in refining candidates. Regarding SPath, the additional times compared with TurboISO and QuickSI are due to 1) the filtering time spent for SPath neighborhood signature (about 62% of the total elapsed time), and 2) the reading times for accessing large SPath neighborhood signatures. In GADDI, due to the expensive NDS distance computation, the average cost of recursive calls for GADDI is about two times larger than VF2.

In addition, we perform experiments with the three best indexing methods in the iGraph package [5], gIndex, FG-Index, and SwiftIndex, for the AIDS dataset. Note that all indexing methods fail to create indexes for the NASA dataset due to the huge memory requirement (more than 50 GBytes) and exponential behavior in finding frequent subgraphs using their own subgraph isomorphism algorithms. Note that these methods report only the IDs of data graphs which contain a query graph along with only *one* embedding for each resulting data graph. Nevertheless, the experiments show surprising results that all these indexing methods perform poorer than TurboISO. We analyze this phenomenon further. 1) For a given query graph q, they enumerate all subgraphs of size up to $maxL$ for q and check whether such subgraphs are in their indexes. However, this step constitutes a considerable performance bottleneck. Furthermore, this cost increases as we increase the query size. 2) After that, they intersect all relevant posting lists to obtain candidate graphs. Finally, they execute a subgraph isomorphism test for each candidate for refinement. However, the non-pruning ratios (or selectivities) using the indexes for $Q4 \sim Q24$ are considerably large, i.e., about 29.6% $\sim$ 0.18%. Considering that the index scan outperforms the sequential scan only when the selectivity is very small, these indexing-based methods cannot outperform the fast subgraph isomorphism algorithms using the sequential scan including TurboISO. This indicates that we must reconsider the graph indexing problem itself.

7.2 Experiment on the NASA dataset

Figure 10 shows the experimental results for NASA. Although the NASA dataset contains only trees, VF2, QuickSI, and GADDI show exponential behaviors at the sizes of query sets 8, 12, and 8, respectively. GraphQL and SPath can finish their query processing in a reasonable time. However, if we use $r = 1$ and $k = 1$, both methods also fail at query sizes 16 and 12. Even if we use the best parameter values for GraphQL and SPath, TurboISO significantly outperforms them.



(a) average elapsed time. (b) avg. # of recursive calls.

Figure 10: NASA dataset.

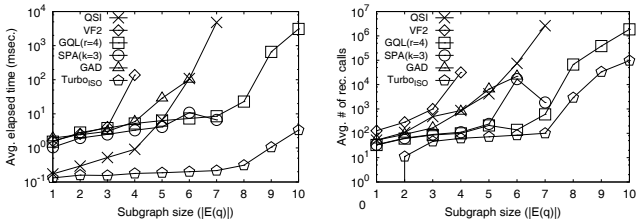
The average numbers of recursive calls for TurboISO and GraphQL increase as we increase the query size up to 16. This is due to search space increments. However, when we increase the query size further, the average numbers of recursive calls decrease, since the matching orders of TurboISO and GraphQL significantly prune candidates, thereby significantly reducing the average number of recursive calls.

In the NASA dataset, there exist star-shaped queries having many of the same vertex labels. In GraphQL, by exploiting the pseudo-isomorphism test and signature-based filtering, it can gracefully handle such non-trivial queries. On the other hand, TurboISO pro-

cesses such queries very efficiently by using the COMB/PERM strategy. In terms of the average elapsed time, Turbo$_{ISO}$ significantly outperforms GraphQL and SPath by up to 3.22 and 119.49 times, respectively. We note that the overhead of executing the candidate region exploration for the NASA dataset is marginal to the overall query processing time.

7.3 Experiment on the Human dataset

Subgraph Queries: Figures 11(a) and 11(b) show the results of subgraph isomorphism search performance using subgraph queries for the Human dataset. Except GraphQL, all competitors fail to finish their query processing in a reasonable time. Specifically, VF2, GADDI, QuickSI, and SPath fail at query sizes 5, 7, 8, and 8. Note that GraphQL uses a large r value (=4) for this experiment. Otherwise, it also fails to complete its query processing.

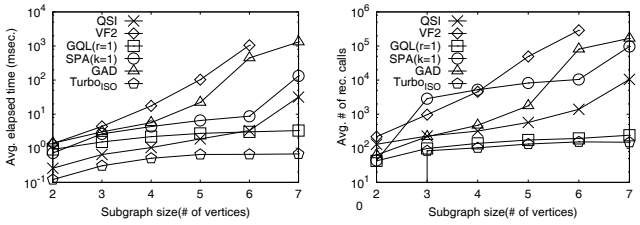


(a) average elapsed time. (b) avg. # of recursive calls.

Figure 11: Human dataset (Subgraph queries).

By using candidate region exploration and the COMB/PERM strategy, Turbo$_{ISO}$ significantly outperforms GraphQL by up to 925.43 times! GraphQL prunes many candidates by using the pseudo-isomorphism test. However, since it uses the large r value, the performance overhead of the pseudo-isomorphism test becomes extremely expensive.

Clique Queries: Figures 12(a) and 12(b) show the performance results for the clique queries. Again, Turbo$_{ISO}$ significantly outperforms all competitors. VF2 fails at query size 7. Since the Human dataset is denser than the Yeast dataset, matching orders generated by QuickSI are worse than those for Yeast. Thus, GraphQL outperforms QuickSI by 2.84 times on average.



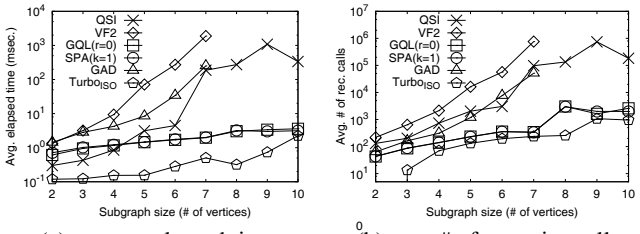
(a) average elapsed time. (b) avg. # of recursive calls.

Figure 12: Human dataset (Clique queries).

Path Queries: Figures 13(a) and 13(b) show the performance results for the path queries. Both VF2 and GADDI fail at query size 8. On average, Turbo$_{ISO}$ is 3.79, 3.86, and 408.70 times faster than SPath, GraphQL, and QuickSI. Note that the performance of QuickSI for the Human dataset is much worse than that for the Yeast dataset due to the serious problem in its matching order selection. GraphQL shows more robust performance than QuickSI in the Human dataset, since it uses signature-based filtering and the pseudo-isomorphism test. Note that it uses a small r value (=0), and its elapsed time is reasonably good since the query type is path.

7.4 Scalability Test

We perform experiments by varying the number of vertices from 0.5 million to 4 million vertices (i.e., 8 datasets). The average degree is 8. That is, the largest graph contains 32 million edges. These



(a) average elapsed time. (b) avg. # of recursive calls.

Figure 13: Human dataset (Path queries).

datasets are generated by exploiting a graph generator provided by the GraphQL package where we make the generator produce multiple labels per vertex. The number of vertex labels is 500, and the average number of labels per vertex is about 8. We use one query set containing 1000 subgraph queries of size 15 for each dataset.

Figure 14 shows the average elapsed time for each query as we vary the number of data vertices. As we increase the number of vertices, the performance linearly increases mainly due to linearly increased graph loading time. If we exclude the loading time from the total elapsed time, the subgraph isomorphism performance of Turbo$_{ISO}$ remains almost the same. This shows the excellent scalability of Turbo$_{ISO}$ when varying the number of vertices.

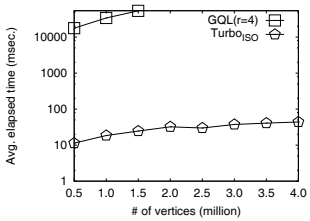


Figure 14: Scalability test using synthetic datasets (Subgraph queries).

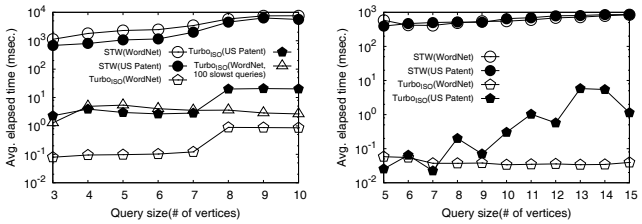
Since GraphQL shows the best performance in the previous experiments, we perform the same scalability test using GraphQL. For the graph having one million vertices, Turbo$_{ISO}$ outperforms GraphQL by 1836 times, finishing all 1000 queries in only 18 seconds, while GraphQL requires about 9.42 hours. The signature-based pruning and pseudo-isomorphism based pruning of GraphQL incur serious performance overhead when the number of vertices is very large. We see that the gap increases further as we increase the number of vertices.

7.5 Comparison with STW [14]

We compare Turbo$_{ISO}$ with the most recent state-of-the-art in [14], referred to here as STW. Since we are unable to re-implement it due to unclarity in sampling-based join optimization of STW, we instead perform Turbo$_{ISO}$ using the same real data sets as used in [14]. For fair comparison, we use a machine having the same CPU and OS as in [14] and also make the data graphs be resident in memory the same way as STW. Note that, like STW, there is no extra overhead in loading graphs into our database, since we do not maintain any auxiliary (index) structures. We stop subgraph isomorphism search after the first 1024 subgraph isomorphisms are found as done in [14]. We use the same two real data sets, US Patent and WordNet as in [14]. The US Patent graph consists of 3,774,768 vertices and 16,522,438 edges, having 418 labels. The WordNet graph contains 82,670 vertices, 133,445 edges and 5 labels. Due to unavailability of the query sets, according to the description in [14], we generate two query sets, referred to here as DFS and Random. While the query sets in [14] contain 100 queries, each of our query sets contains 10,000 queries for a more objective empirical com-

parison. We report the average elapsed time as in [14]. Since both datasets are directed graphs, the edge direction is considered in the adjacency list, the NEC tree rewrite, candidate region exploration, and the main recursive subroutine in TurboISO.

Figure 15 shows the average elapsed time for each query as we vary the number of query vertices using *one* machine. Here, in this figure, we also plotted performance results (using *eight* machines) reported in [14] for comparison. The average elapsed times of STW using one machine for DFS and Random queries are about 21 seconds and 20 seconds, respectively [14]. TurboISO processes these query sets remarkably fast (1046 and 23024 times faster for DFS queries over US Patent and WordNet datasets, respectively!). This is again due to the good matching order selection and the COMB/PERM strategy. One author of STW kindly explains why it takes more than 20 seconds to process one query even if it terminates query processing as soon as it finds 1024 embeddings: From queries having 10 vertices and 20 edges, 6∼8 STwigs are generated. In STW, for each STwig, all partial solutions must be obtained first. Here, a large amount of them must be transferred to other servers. Then, STW performs 5∼7 block-based pipeline joins using the partial solutions for all STwigs. In [14], the block size, which means the number of partial solutions instead of the fixed memory bytes, is set to 8096. Thus, the number of the block-wise join result may grow up to about 64 million(=$(8096)^2$). In a clear contrast to STW which generates and transfers a large amount of intermediate STwig results, TurboISO directly accesses candidate regions without generating intermediate results, obtains good matching orders for them, and finds embeddings using those matching orders and the COMB/PERM strategy, leading to much earlier termination than STW. In addition, we generate 82,670 DFS queries for all vertices in WordNet, choose the top-100 slowest queries, and report the average time for those slowest queries. Nevertheless, TurboISO outperforms STW by up to 2851 times. This additional test *ensures* the superiority of TurboISO against STW.



(a) DFS queries. (b) Random queries.

Figure 15: Comparison with STW.

8. CONCLUSION

In this paper, we presented an efficient and robust subgraph search solution called TurboISO in graph databases. We showed that our solution completely solved the problems of existing algorithms.

We first proposed the novel notion of neighborhood equivalence class (NEC), and then formally derived how to efficiently obtain NECs from a query graph. We then proposed the candidate region exploration process to efficiently identify candidate regions and find robust matching orders which are used for subgraph isomorphism search. We next proposed the COMB/PERM strategy to generate only necessary enumerations. That is, during subgraph isomorphism search, COMB/PERM generated only combinations for each NEC instead of permutating all possible enumerations. Thus, if a chosen combination is determined to not contribute to a complete solution, all possible permutations for the combination will be pruned without any enumeration, thereby significantly avoiding useless computations.

Through extensive experiments, we showed that our solution outperformed all competitors by up to orders of magnitude. Overall, we believe our subgraph isomorphism solution provides comprehensive insight and a substantial framework for future research.

9. ACKNOWLEDGEMENTS

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MEST) (No. 2011-0016520) and Next-Generation Information Computing Development Program through the NRF funded by the Ministry of Education, Science and Technology (No. NRF-2012M3C4A7033342).

10. REFERENCES

[1] R. A. Baeza-Yates and G. Valiente. An image similarity measure based on graph matching. In *SPIRE*, pages 28–38, 2000.

[2] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. In *SIGMOD*, pages 857–872, 2007.

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

[4] M. Deshpande, M. Kuramochi, N. Wale, and G. Karypis. Frequent substructure-based approaches for classifying chemical compounds. *IEEE Trans. Knowl. Data Eng.*, 17(8):1036–1050, 2005.

[5] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu. iGraph: A framework for comparisons of disk-based graph indexing techniques. *PVLDB*, 3(1):449–459, 2010.

[6] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, pages 405–418, 2008.

[7] A. Khan, N. Li, X. Yan, Z. Guan, S. Chakraborty, and S. Tao. Neighborhood based fast graph search in large networks. In *SIGMOD*, pages 901–912, 2011.

[8] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2), 2013, http://www-db.knu.ac.kr/vldb13.pdf.

[9] M. Mongiovi, R. D. Natale, R. Giugno, A. Pulvirenti, A. Ferro, and R. Sharan. Sigma: a set-cover-based inexact graph matching algorithm. *J. Bioinformatics and Computational Biology*, 8(2):199–218, 2010.

[10] M. Ohlrich, C. Ebeling, E. Ginting, and L. Sather. Subgemini: Identifying subcircuits using a fast subgraph isomorphism algorithm. In *DAC*, pages 31–37, 1993.

[11] M. E. Saeedy and P. Kalnis. Grami: Generalized frequent pattern mining in a single large graph.

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

[13] B. Shao, H. Wang, and Y. Li. The trinity graph engine. Technical Report 161291, Microsoft Research, 2012.

[14] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9):788–799, 2012.

[15] Y. Tian and J. M. Patel. TALE: A tool for approximate large graph matching. In *ICDE*, pages 963–972, 2008.

[16] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23:31–42, January 1976.

[17] X. Yan, P. S. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *SIGMOD*, pages 335–346, 2004.

[18] X. Yan, P. S. Yu, and J. Han. Graph indexing based on discriminative frequent structure analysis. *ACM Trans. Database Syst.*, 30(4):960–993, 2005.

[19] S. Zhang, M. Hu, and J. Yang. Treepi: A novel graph indexing method. In *ICDE*, pages 966–975, 2007.

[20] S. Zhang, S. Li, and J. Yang. GADDI: distance index based subgraph matching in biological networks. In *EDBT*, pages 192–203, 2009.

[21] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 3(1):340–351, 2010.

[22] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: Tree + delta >= graph. In *VLDB*, pages 938–949, 2007.

[23] L. Zou, L. Chen, J. X. Yu, and Y. Lu. A novel spectral coding in a large graph database. In *EDBT*, pages 181–192, 2008.