This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.2980257, IEEE Transactions on Knowledge and Data Engineering

Subgraph Matching with Effective Matching Order and Indexing

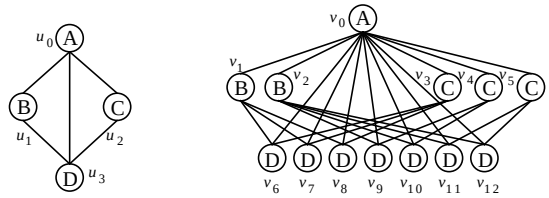
Shixuan Sun and Qiong Luo

Abstract—Subgraph matching finds all embeddings from a data graph that are identical to a query graph. Recent algorithms work by generating a tree-structured index on the data graph based on the query graph, ordering the vertices root-to-leaf path-by-path in the tree, and enumerating the embeddings following the matching order. However, we find such path-based ordering and tree-structured index based enumeration inherently limit the performance due to the lack of consideration on the edges among the vertices across tree paths. To address this problem, we propose an approach that generates the matching order based on a cost model considering both the edges among query vertices and the number of candidates. Furthermore, we create a bigraph index for candidate vertices and their selected neighbors in the data graph, and use this index to perform enumeration along the matching order. Our experiments on both real-world and synthetic datasets show that our method outperforms the state of the art by orders of magnitude.

Index Terms—Graph, Graph Query, Subgraph Matching.

1 Introduction

Given a graph q as the query, and another graph G, usually much larger than q, as the data graph, subgraph matching finds all embeddings in G that are isomorphic to q. For example, given q and G in Figure 1, $\{(u_0, v_0), (u_1, v_1), (u_2, v_3), (u_3, v_6)\}$ is a match.



a Query q. b Data graph G.

Fig. 1: Example graphs.

The subgraph matching problem is NP-complete [6], and a variety of algorithms [3], [5], [9], [11], [20], [23], [28] have been proposed. These algorithms focus on generating effective matching orders and designing powerful filtering strategies to minimize the number of candidates in the data graph. QuickSI [23] designs the *infrequent-edge-first ordering* technique, which sorts the edges of the query graph in the ascending order in terms of the frequency appeared in the data graph. GraphQL [11] adopts the *left-deep-join ordering* strategy, which models the enumeration procedure as the join problem. SPath [28], TurboIso [9] and CFL [3] propose the *path-based ordering* method, which generates a matching order by decomposing the query graph into several paths and ordering the paths based on the estimated number of embeddings of each path. Except the ordering strategies, the state-of-the-art algorithms, such as TurboIso [9] and CFL [3], employ a *tree-based framework*, which constructs a light-weight tree-structured index to minimize the number of candidates, and enumerates all matches based on the index instead of the original data graph. Although these techniques have led to significant advances, we find several inherent problems in them.

First, existing ordering strategies only consider the number of candidates of every vertex (or path), but miss the edges among query vertices.

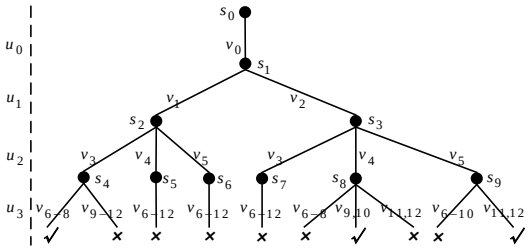


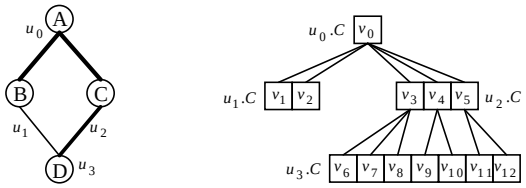
Fig. 2: The enumeration process with $\pi = (u_0, u_1, u_2, u_3)$.

Example 1.1. Given q and G in Figure 1, existing ordering strategies, which include the infrequent-edge-first ordering, the left-deep-join ordering and the path-based ordering, obtain the matching order $\pi = (u_0, u_1, u_2, u_3)$ based on the number of candidates in the data graph: $u_0.C = \{v_0\}$, $u_1.C = \{v_{1-2}\}$, $u_2.C = \{v_{3-5}\}$ and $u_3.C = \{v_{6-12}\}$. During enumeration, u_1, u_2 and u_3 obtain their candidate data vertices based on u_0. Figure 2 visualizes the recursive enumeration process, which expands a partial result by mapping a data vertex to a query vertex along the matching order. Each node s_i $(i = 0, ..., 9)$ denotes a partial result such as s_2 : $\{(u_0, v_0), (u_1, v_1)\}$, and an edge corresponds to a mapping between a query vertex and a data vertex such as $e(s_1, s_2) : (u_1, v_1)$. In this matching order, 35 out of a total of 42 embeddings are not ruled out until the end of enumeration. In contrast, if we swap the matching order of u_2 and u_3 to match in the order of (u_0, u_1, u_3, u_2), we can prune the invalid intermediate results at an early stage, because there is an edge between u_1 and u_3 in q.

- *Shixuan Sun and Qiong Luo are with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, China.*
  E-mail: ssunah@cse.ust.hk, luo@cse.ust.hk.

Furthermore, if u_2 and u_3 obtain their candidates based on u_3 and u_1 respectively instead of u_0, the search space can be reduced further. Figure 5d presents the enumeration process with these two improvements, which is much more efficient than that in Figure 2.

Second, in a tree-based framework, the matching order is generated with tree-structured indices [3], [9], which only maintain edges along paths. As a result, even if there is a more efficient order, the index cannot support the enumeration. In other words, the tree-based framework inherently limits the generation of the matching order.



a Query q. b Tree-structured index G_t.

Fig. 3: An example of the index.

Example 1.2. Given q in Figure 3a and G in Figure 1b, q_t is a BFS (Breadth-first search) tree rooted at u_0 (depicted by thick lines). The tree-structured index G_t in Figure 3b has the same structure as q_t, and stores the candidate set for each query vertex as well as edges between candidates in the original data graph. G_t cannot support the enumeration along (u_0, u_1, u_3, u_2), because it does not keep the edges between $u_1.C$ and $u_3.C$.

Additionally, we find that existing work could benefit from stronger filtering strategies to minimize the size of candidate sets, because they can both further reduce the search breadth of partial results and provide more accurate statistics for the matching order generation.

Our Approach. Motivated by these observations, we propose a new subgraph matching algorithm **VC** (Vertex Connectivity). Different from the tree-based framework, which first constructs the index and then generates a matching order based on the index, VC performs subgraph matching with a new process, which contains four steps: (1) extract a candidate set for each query vertex; (2) generate a matching order based on the statistics of the candidates; (3) construct a bigraph index BI, which stores the edges among data vertices in the candidate sets; and (4) enumerate all results based on BI. The first three steps are the *indexing* phase, and the last step *enumeration*.

During candidate extraction, VC first generates a candidate set $u.C$ for each query vertex u along an order of query vertices, called the *indexing order*. Then, VC refines the candidate sets in the reverse order of the indexing order with a *pseudo star isomorphism constraint* and a *ping-pong filtering* strategy. Next, VC generates an order of query vertices, called the *matching order*, with a ***vertex-based ordering*** strategy, which considers both the sizes of candidate sets and the vertex connectivity among query vertices. When generating the matching order, VC also obtains a *pivot dictionary*, which is a set of pairs of query vertices to determine the edges in the data graph to be stored in the index. After that, we further construct a bigraph (i.e., bipartite graph) index BI: for every pair of vertices (u, u') in the pivot dictionary, we retrieve the candidate sets $u.C$ and $u'.C$ as two sets of vertices and add edges between these two sets as they appear in the data graph. Finally, we enumerate all matches by expanding partial results recursively along the matching order with the assistance of BI.

VC generates the matching order by considering both the sizes of the candidate sets and the edges among query vertices, which addresses the problem that existing ordering strategies only consider the number of candidates. By redesigning the process of performing subgraph matching and proposing the bigraph index, we break the constraint on the generation of matching orders in the tree-based framework, because the edges maintained in the bigraph index of VC rely on the pivot dictionary, which is obtained when generating the matching order, whereas the matching order generated in the tree-based framework depends on the tree-structured index. Moreover, the refinement of the candidate sets based on the pseudo star isomorphism constraint and the ping-pong filtering strategy can further minimize the sizes of the candidate sets.

Contributions. In summary, we make the following contributions in this paper.

- We propose a new subgraph matching algorithm VC, which differs from existing algorithms on the process of performing subgraph matching.
- We minimize the candidate sets of query vertices with a pseudo star isomorphism constraint and a ping-pong filtering strategy.
- We generate a matching order with the vertex-based ordering strategy, which considers both the number of candidates and the edges among query vertices.
- We design a bigraph index for enumeration, whose space complexity is $O(|E(G)| \times |V(q)|)$ and time complexity of construction is $O(|E(G)| \times |E(q)|)$.

Finally, we compare VC with a variety of existing algorithms with different ordering strategies on both real and synthetic datasets with new experiment metrics from a latest performance study [13]. The results show that VC significantly outperforms previous algorithms.

Paper Organization. Section 2 introduces the background. Section 3 gives an overview of our VC algorithm. The construction of indices and the generation of matching orders are presented in Sections 4 and 5 respectively. We evaluate VC in Section 6 and conclude in Section 7.

2 BACKGROUND

2.1 Preliminaries

In this paper, we focus on the vertex-labeled undirected graph $g = (V, E, \Sigma, L)$, where V is a set of vertices, E is a set of edges, Σ is a set of labels, and L is a function that associates a vertex v with a label $L(v) \in \Sigma$. The query graph q is connected. Next, we give a formal definition of subgraph matching and related preliminaries used in this paper, and summarize the frequently used notations in Table 1.

Definition 1. Subgraph Isomorphism (Match): Given a query graph $q = (V, E, \Sigma, L)$ and a data graph $G = (V', E', \Sigma', L')$, a subgraph isomorphism (match) is an injective function $f : V \to V'$ that satisfies:

(1). $\forall u \in V, L(u) = L'(f(u))$;

(2). $\forall e(u, v) \in E, \exists e(f(u), f(v)) \in E'$.

We call a subgraph isomorphism a *match* for simplicity.

Definition 2. Subgraph Matching: Given q and G, find all matches from q to G.

Definition 3. Vertex Induced Subgraph: Given a graph $g = (V, E, \Sigma, L)$ and $V^{'} \subseteq V$, a vertex induced subgraph of g constructed on V' is denoted as $g[V'] = (V', E', \Sigma, L)$ where $E' = \{e(u,v)|e(u,v) \in E \; and \; u, v \in V'\}$.

Definition 4. Star: Given a graph g and a vertex $u \in V(g)$, a star rooted at u, denoted as $ST(u)$, is the tree of depth 1 rooted at u that contains all neighbor vertices of u in g.

Definition 5. Order of Vertices: Given a query graph q, an order Γ is a permutation of the vertices in q. $\Gamma[i]$ is the ith vertex in Γ and $\Gamma[i:j]$ is the set of vertices from index i to j in Γ. We call an order of vertices for matching a matching order, denoted as π, and one for indexing an indexing order, denoted as π'.

Definition 6. Connected Order: Given a query graph q and an order Γ, Γ is connected if given any $1 \leqslant i \leqslant |\Gamma|$, the vertex induced subgraph $q[\Gamma[1:i]]$ is connected.

Definition 7. Backward Neighbors: Given a query graph q, an order Γ and a vertex $u = \Gamma[i]$, the backward neighbors of u, denoted as $BN_q^{\Gamma}(u)$, are the neighbors of u that are positioned before u in Γ.

Definition 8. Complete Candidate Set: Given q and G, a candidate set $u.C$ for $u \in V(q)$ is complete if $u.C$ satisfies: If a mapping (u, v) exists in any matches from q to G where $v \in V(G)$, then $v \in u.C$. If $\forall u \in V(q)$, $u.C$ is complete, then we say C is complete.

Definition 9. Pivot and Pivot Dictionary: Given a query graph q and a connected matching order π, the pivot of a query vertex u is a query vertex u' in $BN_q^{\pi}(u)$. The pivot dictionary, denoted as $\mathcal{P}$, records the pivot of each query vertex, and $\mathcal{P}[u]$ denotes the pivot of u. $\pi[1]$ has no pivot, as it has no backward neighbor.

The notion of a core was first introduced by Seidman, which measures the local density of a graph [22]. The definition of *k-core* is as follows.

Definition 10. k-core: Given a graph g, a k-core of g is a maximal connected subgraph g' of g that satisfies $\forall u \in V(g'), d(u) \geqslant k$, where $d(u)$ is the degree of u in g'.

Definition 11. Core Value: Given a graph g and a vertex $u \in V(g)$, the core value of u, denoted as $u.core$, is c if u belongs to a c-core but not any (c+1)-core.

Batagelj [2] et al. proposed an $O(|E(g)|)$ algorithm to calculate the core values of all vertices in g, and proved that there is exactly one 2-core in a connected graph. We use this algorithm to calculate $u.core$. CFL [3] defined *core structure* as follows.

Definition 12. Core Structure: Given a query graph q, the core structure of q is the 2-core of q.

We further define the core degree in Definition 13. When generating the indexing order and the matching order, we prioritize the vertices by their core values and core degrees to exclude the effect of the vertices not in the core structure.

TABLE 1: Notations.

Notations	Descriptions
g, q and G	graph, query graph and data graph
$V(g)$ and $E(g)$	vertex set and edge set of g
$d(u)$, $L(u)$ and $N(u)$	degree, label and neighbors of u
$e(u,v)$	edge between u and v
$u.core$, $u.core\_degree$	core value and core degree of u
$N(V)$	neighbors of the vertices in V
$g[V]$	induced subgraph of g given V
$ST(u)$	star rooted at u
π and π'	matching order and indexing order
C and $\mathcal{P}$	candidate set and pivot dictionary
$u.C$	candidates of u in C
$BN_q^{\Gamma}(u)$	backward and forward neighbors of u
BI	bigraph index
$BI_u^{u'}$	bigraph between $u.C$ and $u'.C$
$BI_u^{u'}(v)$	neighbors of v in $u.C$ where $v \in u'.C$

Definition 13. Core Degree: Given a query graph q, let V_C be the set of vertices in the core structure of q. The core degree of u is $|N(u) \cap V_C|$, denoted as $u.core\_degree$. If a vertex $u \notin V_C$, $u.core\_degree$ is 0.

2.2 Related Work

To put our research in context, we categorize the related work by the problem addressed: labeled subgraph matching and unlabeled subgraph matching. Our paper focuses on the labeled subgraph matching problem.

Labeled Subgraph Matching. Labeled subgraph matching aims to find all matches in a single labeled data graph. Based on execution phases of the search process, we categorize the algorithms into direct-enumeration and indexing-enumeration. The direct-enumeration approaches, such as Ullmann [26], VF2 [5], QuickSI [23], GraphQL [11] and SPath [28], do not construct an index given a query graph before enumeration, but obtain the candidates of each query vertex individually based on filters such as the neighborhood signature [28]. Due to the lack of accurate information to estimate cost, the matching order can be ineffective and there may be many false positive candidates. Lee et al. [18] provided an extensive discussion on these algorithms and showed that these algorithms have problems in their matching order selection, and the signature-based filters are only effective for some datasets.

To address the problems in the direct-enumeration algorithms, researchers proposed to divide the search process into two phases, first of which constructs an index given a query graph and second conducts enumeration based on the index. The index not only reduces the number of candidates but also provides accurate cost estimation to generate an effective matching order. TurboIso [9] designed a tree-structured index, candidate region, and generated the matching order by the path-based ordering strategy. Moreover, TurboIso [9] proposed the neighborhood equivalent class to compress the similar vertices in the query graph. CFL [3], the state-of-the-art algorithm, accelerated the enumeration by postponing cartesian products with a matching order that prioritizes the query vertices in the core structure, and proposed a new tree-structured index CPI, which reduces the space complexity from $O(|V(G)|^{|V(q)|})$ of TurboIso to $O(|E(G)| \times |V(q)|)$. Both TurboIso and CFL achieved impressive speedups over the direct-enumeration algorithms.

Additionally, previous research improved labeled subgraph matching by exploiting the vertex relationship in data

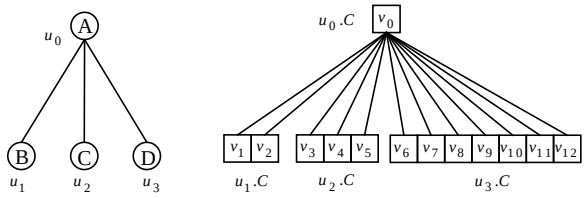
graphs [20], utilizing the computing results among multiple queries [21] and parallelizing the algorithms on a single machine [13], [15] or the distributed environment [25].

Unlabeled Subgraph Matching. Unlabeled subgraph matching is to find all matches in unlabeled graphs. Due to the lack of label, unlabeled subgraph matching is more challenging than labeled subgraph matching. Moreover, unlabeled subgraph matching is required to handle automorphism to eliminate any duplicates in the matching results [7]. The latest research on this problem focuses on designing parallel approaches such as Afrati [1], TwinTwig [16], SEED [17], PSgL [24], Crystal [19] and DualSim [14].

2.3 Tree-based Frameworks

The latest subgraph matching algorithms CFL [3] and TurboIso [9] employ a tree-based framework to conduct subgraph matching. In the following, we use CFL, the state-of-the-art algorithm, as a representative to illustrate the general idea of the tree-based framework.

Given q and G, CFL first obtains a BFS (breadth-first search) tree q_t from q by selecting a root vertex u and performing a BFS traversal from u. Next, CFL constructs a tree-structured index CPI on G based on q_t. Specifically, CPI generates a complete candidate set for each query vertex level-by-level along the top-down order of q_t. During generation, CPI conducts backward pruning at each level. After the generation, CPI performs a bottom-up refinement level-by-level along q_t to minimize the sizes of candidate sets. Both the generation and the refinement are based on the observation: Given $v \in u.C$, if (u, v) exists in a match from q to G, then $\forall u' \in N(u), N(v) \bigcap u'.C \neq \emptyset$. Additionally, in the generation and refinement processes, CPI not only maintains the candidate vertices, but also stores edges among candidate vertices. After constructing CPI, CFL adopts the path-based ordering strategy to generate a matching order. In particular, CFL sorts the paths of q_t based on the estimated number of embeddings of each path in CPI. Finally, CFL enumerates all matches along the matching order based on CPI.



a BFS tree q_t. b The index CPI of CFL.

Fig. 4: A running example of CFL on the graphs in Figure 1.

Example 2.1. Given q and G in Figure 1, CFL obtains the BFS tree q_t rooted at u_0 as shown in Figure 4a. Next, CFL constructs the tree-structured index CPI in Figure 4b, which has the same structure as q_t. CPI stores the candidate set for each query vertex as well as edges between candidates in the original data graph. q_t contains three paths, which are $p_1 = (u_0, u_1)$, $p_2 = (u_0, u_2)$ and $p_3 = (u_0, u_3)$. CFL sorts the three paths by the number of embeddings in CPI, which is (p_1, p_2, p_3). Therefore, CFL obtains the matching order $\pi = (u_0, u_1, u_2, u_3)$. Finally, CFL enumerates all matches along π with the assistance of CPI, which is the same as the enumeration procedure in Figure 2.

Compared with the tree-based framework, VC has four differences: (1) VC redesigns the process of performing subgraph matching; (2) VC simplifies the candidate extraction to the forward and backward stages and refines the candidate sets with stronger pruning strategies; (3) VC adopts the vertex-based ordering method instead of the path-based ordering; and (4) VC models the indices as multiple bigraphs among candidate sets.

3 ALGORITHM OVERVIEW

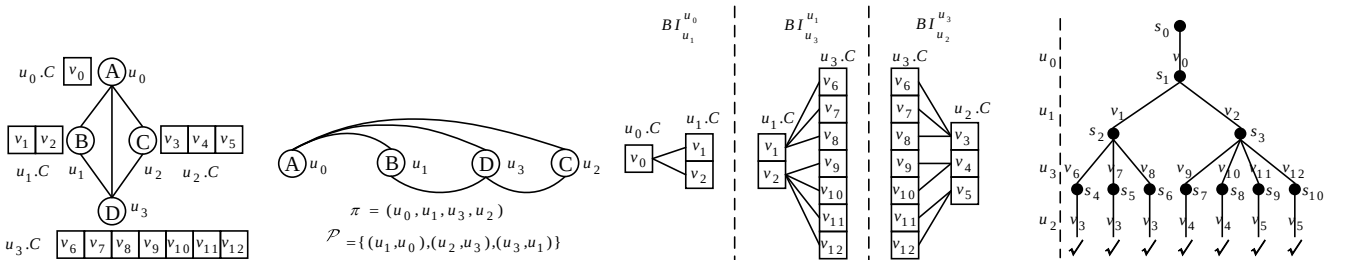
Given q and G, VC recursively expands partial results by mapping query vertices to their candidates along the matching order. Therefore, VC requires a complete candidate set C and a matching order π. Furthermore, we construct two auxiliary structures, a pivot dictionary $\mathcal{P}$ and a bigraph index BI, to speed up the enumeration.

When expanding a partial result by mapping a query vertex u to a data vertex v, we only need to consider the data vertices that are the neighbors of the data vertices mapped to the backward neighbors of u and belong to the candidate set $u.C$ according to Definitions 1 and 8. Therefore, we select a backward neighbor u' of u to maintain the relationship between $u'.C$ and $u.C$ to facilitate the enumeration. The selected backward neighbor u' of u is the pivot of u, and the pivot dictionary $\mathcal{P}$ records the pivot of each query vertex except $\pi[1]$, because $\pi[1]$ has no backward neighbor. We construct the bigraph index BI containing bigraphs for each pair of vertices in $\mathcal{P}$. A bigraph in BI has two vertex sets $u.C$ and $u'.C$ where (u, u') is a pair of vertices in $\mathcal{P}$ and u' is the pivot of u. The bigraph between $u.C$ and $u'.C$ is denoted as $BI_u^{u'}$.

Algorithm 1 presents our VC algorithm, which takes q and G as input, and outputs all matches from q to G. We first extract the candidates in G for each query vertex (Line 2). Next, we generate the matching order π and the pivot dictionary $\mathcal{P}$ (Line 3). Line 4 constructs the bigraph index BI. We call the three steps the indexing phase.

Following the indexing phase, the enumeration phase expands the partial results vertex-by-vertex recursively along π based on BI. If all query vertices have been mapped, we output the result and return (Line 11). Otherwise, we obtain the next query vertex u in π and the pivot of u (Line 12). We expand the partial result M by mapping a candidate to u (Lines 13-15). We get the candidates of u based on $BI_u^{u'}$ and the data vertex mapped to the pivot of u (i.e., $BI_u^{u'}(M[u'])$). For a candidate v of u, line 14 checks whether v is matched, and has edges to the data vertices mapped to the backward neighbors of u. If so, we invoke *Enumerate* recursively the same as in lines 7-9. The *Validate* function does not check the edge between v and the data vertex mapped to the pivot p of u, because v is from $BI_u^p(M[p])$, so the edge exists. During enumeration, the original data graph is only used to check the existence of edges between data vertices (Line 18).

Example 3.1. Figure 5 shows the VC algorithm running on the graphs in Figure 1. The results of candidate extraction are illustrated in Figure 5a. Each candidate set $u.C$ stores the data vertices that can be mapped to u. Figure 5b shows the matching order and pivot dictionary generated by VC. Take u_3 as an example:



a The candidate set C. b π and $\mathcal{P}$. c The bigraph index BI. d The enumeration process of VC.

Fig. 5: A running example of VC on the graphs in Figure 1.

Algorithm 1: VC algorithm

Input: a query graph q and a data graph G
Output: all matches from q to G

```
1  begin
       /* The indexing phase.                                   */
2      C ← ExtractCandidates(q, G);
3      (π, P) ← GenerateMatchingOrder(q, G, C);
4      BI ← ConstructIndex(C, P);
       /* The enumeration phase.                                */
5      l ← 1, u ← π[l], M ← {};
6      foreach v ∈ u.C do
7          M[u] ← v, v.visited ← true;
8          Enumerate(G, π, P, M, BI, l + 1);
9          v.visited ← false, remove (u, v) from M;
10 Procedure Enumerate(G, π, P, M, BI, l)
11     if l = |π| + 1 then Output M, return;
12     u ← π[l], u' ← P[u];
13     foreach v ∈ BI_u^{u'}(M[u']) do
14         if v.visited is false and Validate(G, M, u, v, u') is
             true then
15             Same as Lines 7-9;
16 Function Validate(G, M, u, v, p)
17     foreach u' ∈ BN_q^π(u) with u' ≠ p do
18         if e(M[u'], v) ∉ E(G) then return false;
19     return true;
```

$BN_q^\pi(u_3) = \{u_0, u_1\}$, and u_1 is selected as its pivot. The start vertex u_0 in the matching order has no pivot. For each pair of vertices (u, u') in $\mathcal{P}$, we construct a bigraph to maintain the relationship between $u.C$ and $u'.C$, which is shown in Figure 5c. Specifically, $BI_u^{u'}$ maintains the edges between data vertices in $u.C$ and $u'.C$ in the original data graph. Take v_0 in $u_0.C$ and v_1 in $u_1.C$ as an example: $e(v_0, v_1)$ exists in $E(G)$, then there is an edge between v_0 and v_1 in $BI_{u_1}^{u_0}$. Moreover, in $BI_{u_1}^{u_0}$, $BI_{u_1}^{u_0}(v_0) = \{v_1, v_2\}$. Finally, the enumeration process along π based on BI is illustrated in Figure 5d.

4 BIGRAPH INDEX

In this section, we propose a bigraph index BI, which is constructed in two steps, candidate extraction and index construction.

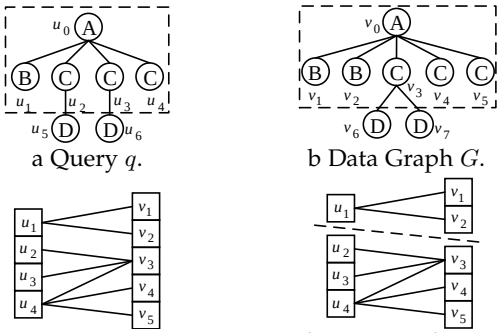
4.1 Candidate Extraction

Given q and G, candidate extraction is to obtain a complete candidate set for each query vertex. To generate a complete candidate set as small as possible, we define the *exact star isomorphism constraint* and identify a property between a pair of query and data vertices under a match.

Definition 14. Exact Star Isomorphism Constraint (ESIC): Given q and G, C is the complete candidate set constructed by *LDF*. Given $u \in V(q)$ and $v \in V(G)$, (u, v) satisfies the exact star isomorphism constraint if there exists a match f from $ST(u)$ to $ST(v)$ such that $\forall u' \in N(u), f(u') \in u'.C$.

Lemma 4.1. Given q, G, $u \in V(q)$ and $v \in V(G)$, if a mapping (u, v) exists in a match from q to G, then (u, v) satisfies the exact star isomorphism constraint.

Based on Lemma 4.1, given $u \in V(q)$ and $v \in u.C$, v can be removed from $u.C$ without breaking its completeness if (u, v) does not satisfy ESIC. We use a technique proposed in an approximate subgraph isomorphism algorithm [10] to check whether (u, v) satisfies ESIC: (1) construct a bigraph, denoted as B_v^u, with $N(u)$ and $N(v)$ as bipartitions where an edge is added between $u' \in N(u)$ and $v' \in N(v)$ if $v' \in u'.C$. We call this method the *neighborhood bigraph construction*; and (2) perform maximum bigraph matching [27] to check the existence of a semi-perfect matching in B_v^u (i.e., every vertex in $N(u)$ is matched).



a Query q. b Data Graph G. c $B_{v_0}^{u_0}$. d Partitioned $B_{v_0}^{u_0}$.

Fig. 6: ESIC and PSIC examples.

Example 4.1. Given q in Figure 6a and G in Figure 6b, the complete candidate set C constructed by *LDF* is as follows: $u_0.C = \{v_0\}$, $u_1.C = \{v_{1-2}\}$, $u_2.C = u_3.C = \{v_3\}$, $u_4.C = \{v_{3-5}\}$ and $u_5.C = u_6.C = \{v_{6-7}\}$. $ST(u_0)$ and $ST(v_0)$ are in the dotted line rectangles respectively. As shown in Figure 6c, $B_{v_0}^{u_0}$ is the bigraph constructed using the neighborhood bigraph construction method. As $B_{v_0}^{u_0}$ has no semi-perfect matching, (u_0, v_0) violates ESIC. Then, v_0 can be safely removed from $u_0.C$.

The time complexity of constructing B_v^u is $O(d(u) \times d(v))$. The time complexity of maximum bigraph matching is denoted as $\Phi(d(u), d(v))$ (e.g., $O(n^{2.5})$ for Hopcorft and Karp's algorithm [12] where n is the num-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.2980257, IEEE Transactions on Knowledge and Data Engineering

ber of vertices in B_v^u). Then, the time complexity of checking every query vertex and its candidates is $O(\sum_{u \in V(q)} \sum_{v_c \in V(G)} (d(u) \times d(v) + \Phi(d(u), d(v)))) = O(|E(q)| \times |E(G)| + \sum_{u \in V(q)} \sum_{v_c \in V(G)} \Phi(d(u), d(v)))$. As this checking is expensive, we raise a question:

- *Can we remove the maximum bigraph matching check, while still keeping a competitive filtering power as ESIC?*

To answer this question, we first define *neighborhood label equivalent class* as follows.

Definition 15. Neighborhood Label Equivalent Class: Given a graph g and a vertex $u \in V(g)$, the neighborhood label set of u contains all distinct labels of its neighbor vertices, denoted as $NLS(u)$. Given $l \in NLS(u)$, the neighborhood label equivalent class $NLEC(u, l)$ is $\{u' | u' \in N(u)\ and\ L(u') = l\}$.

Example 4.2. Given u_0 in Figure 6a, $NLS(u_0) = \{B, C\}$, $NLEC(u_0, B) = \{u_1\}$ and $NLEC(u_0, C) = \{u_2, u_3, u_4\}$.

Next, in order to eliminate the maximum bigraph matching check, we define the *pseudo star isomorphism constraint* and prove Theorem 4.1.

Definition 16. Pseudo Star Isomorphism Constraint (PSIC): Given q and G, C is the complete candidate set constructed by *LDF*. Given $u \in V(q)$ and $v \in V(G)$, (u, v) satisfies the pseudo star isomorphism constraint if $\forall l \in NLS(u)$, $X = \Theta[1 : i]$ where Θ is a permutation of vertices in $NLEC(u, l)$ and $1 \leqslant i \leqslant |\Theta|$, the following two conditions hold: (1) $\forall u' \in X, u'.C \cap N(v) \neq \emptyset$; and (2) $|X| \leqslant |\cup_{u' \in X} (N(v) \cap u'.C)|$.

Theorem 4.1. Given q, G, $u \in V(q)$ and $v \in V(G)$, if a mapping (u, v) exists in a match from q to G, then (u, v) satisfies the pseudo star isomorphism constraint.

Proof: Given q and G, C constructed by *LDF* is complete. Suppose that (u, v) exists in a match from q to G where $u \in V(q)$ and $v \in V(G)$. Then, (u, v) satisfies ESIC (Lemma 4.1). Thus, there exists a semi-perfect matching in B_v^u constructed by the neighborhood bigraph construction method. According to Hall's theorem [8], $\forall Y \subseteq N(u), |Y| \leqslant |N(Y)|$ in B_v^u. Given u' and u'' in $N(u)$ where $L(u') \neq L(u'')$, the construction method of B_v^u guarantees that u' and u'' have no common neighbors in B_v^u, because $u'.C \cap u''.C = \emptyset$. So, $\forall Y \subseteq NLEC(u, l)$ where $l \in NLS(u), |Y| \leqslant |N(Y)|$ in B_v^u. As B_v^u has a semi-perfect matching, $\forall u' \in Y, u'.C \cap N(v) \neq \emptyset$. Because $N(Y)$ in B_v^u is identical to $\cup_{u' \in Y}(N(v) \cap u'.C)$ based on the neighborhood bigraph construction method, $|Y| \leqslant |\cup_{u' \in Y} (N(v) \cap u'.C)|$. Therefore, given $X = \Theta[1 : i]$ where Θ is any permutation of vertices in $NLEC(u, l)$ and $1 \leqslant i \leqslant |\Theta|$, $\forall u' \in X, u'.C \cap N(v) \neq \emptyset$ and $|X| \leqslant |\cup_{u' \in X} (N(v) \cap u'.C)|$. Thus, the theorem holds. □

Based on this theorem, given $u \in V(q)$ and $v \in u.C$, we can safely rule out v from $u.C$ if (u, v) does not satisfy PSIC. PSIC checks each $NLEC(u, l)$ respectively instead of $N(u)$, because it can improve the filtering power. We use the following example to illustrate it.

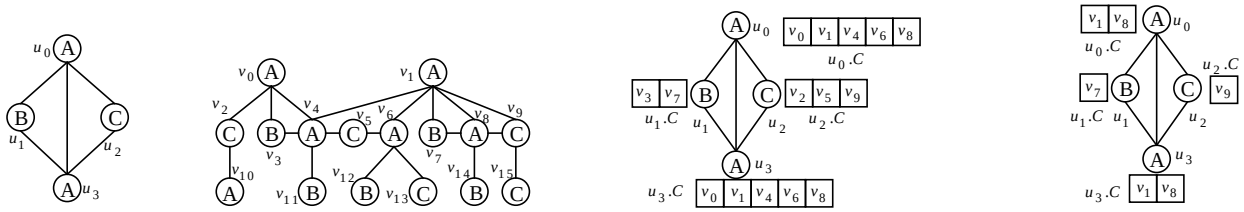
Example 4.3. In Figure 6c, if we check PSIC in terms of $N(u_0)$ with $\Theta(N(u_0)) = (u_1, u_2, u_3, u_4)$, then (u_0, v_0) satisfies PSIC. In contrast, if we check PSIC in terms of $NLEC(u_0, B)$ and $NLEC(u_0, C)$ respectively as shown in Figure 6d and $\Theta(NLEC(u_0, C)) = (u_2, u_3, u_4)$, then (u_0, v_0) does not satisfy PSIC, because $|\{u_2, u_3\}| < |N(\{u_2, u_3\})|$. However, if $\Theta(NLEC(u_0, C)) = (u_4, u_3, u_2)$, then (u_0, v_0) satisfies PSIC.

Algorithm 2: ExtractCandidates

```
Input: a query graph q and a data graph G
Output: the candidate set C
1  begin
2      π' ← GenerateIndexingOrder(q);
       /* The forward stage.                                   */
3      u ← π'[1], set u'.C to empty for all u' ∈ V(q);
4      foreach v ∈ V(G) do
5          if LDF(u, v) is true and NLF(u, v) is true then
6              u.C ← u.C ∪ {v};
7      for i ← 2 to |π'| do
8          u ← π'[i];
9          foreach v ∈ ∩_{u' ∈ BN_q^{π'}(u)} N(u'.C) do
10             if LDF(u, v) is true and NLF(u, v) is true then
11                 u.C ← u.C ∪ {v};
       /* The backward stage.                                  */
12     for i ← |π'| to 1 do
13         u ← π'[i];
14         foreach v ∈ u.C do
15             foreach l ∈ NLS(u) do
16                 X ← ∅, j ← 0;
17                 foreach u' ∈ NLEC(u, l) do
18                     j ← j + 1, Y ← N(v) ∩ u'.C, X ← X ∪ Y;
19                     if Y is ∅ or |X| < j then
20                         u.C ← u.C\{v}, Goto Line 14;
21         foreach u' ∈ N(u) do
22             foreach v ∈ u'.C do
23                 if N(v) ∩ u.C is ∅ then u'.C ← u'.C\{v};
24     return C;
```

The permutation of vertices in $NLEC(u, l)$ may affect the filtering effectivenesses of PSIC. However, our experiment results show that a random order of the vertices is sufficient for PSIC to achieve a competitive filtering power as ESIC. Furthermore, ordering vertices in an $NLEC$ introduces additional cost. Therefore, in this paper we adopt a random order of vertices in $NLEC(u, l)$. Next, we present our candidate extraction algorithm, developed based on Theorem 4.1.

Extract Candidates. Algorithm 2 outlines the process of candidate extraction. We first generate a connected order of query vertices (Line 2). To differentiate from the matching order, we name the order in which we extract candidates as the *indexing order*, denoted as π'. Instead of the naive construction method based on *LDF*, we implement the forward stage based on an observation in CFL: given a query vertex u, $u.C$ can be obtained as the intersection of $N(u'.C)$ for all $u' \in N(u)$ where vertices in the intersection have the same label as u [3]. Lines 4-6 get the candidate set of the start vertex in π', which contains the data vertices that match the *LDF* and *neighborhood label frequency* (*NLF*) filters. The *NLF* filter checks whether the label frequency of the neighbors of the data vertex is greater than or equal to that of the query vertex [3], [9]. Next, we start the forward stage along



a Query q. b Data graph G. c Results of forward stage. d Results of backward stage.

Fig. 7: A running example of candidate extraction.

π' (Lines 7-11). After the forward stage, the backward stage filters candidate sets in the reverse order of π' based on Theorem 4.1 (Lines 12-23). The ping-pong filtering strategy immediately refines the candidate sets of every $u' \in N(u)$ after updating $u.C$ (Lines 21-23). The following proposition holds based on Theorem 4.1.

Proposition 4.1. Given q and G, the candidate set C obtained by Algorithm 2 is complete.

Example 4.4. Figure 7 illustrates Algorithm 2 running on q in Figure 7a and G in Figure 7b. π' is (u_0, u_3, u_1, u_2). We first get $u_0.C = \{v_0, v_1, v_4, v_6, v_8\}$. Next, we start the forward stage along π', and generate $u_3.C = \{v_0, v_1, v_4, v_6, v_8\}$ based on $u_0.C$. As $BN_q^{\pi'}(u_1) = \{u_0, u_3\}$, we obtain $u_1.C = \{v_3, v_7\}$ from the intersection of $N(u_0.C)$ and $N(u_3.C)$. Similarly, we obtain $u_2.C = \{v_2, v_5, v_9\}$. The result of the forward stage is shown in Figure 7c.

We start the backward stage following the reverse order of π'. v_2 is removed from $u_2.C$, as it has only one distinct neighbor in both $u_0.C$ and $u_3.C$. Through the ping-pong filtering strategy, v_0 is removed from $u_0.C$ and $u_3.C$ respectively, since v_0 has no neighbors in $u_2.C$. Similarly, v_3 is removed from $u_1.C$. Based on the updated $u_1.C$, v_4 and v_6 are removed from $u_0.C$ and $u_3.C$. Next, $u_3.C$ has no change, and v_5 is removed from $u_2.C$ based on the ping-pong filtering strategy. $u_0.C$ has no change. The final result is shown in Figure 7d.

Generate Indexing Order. We first prove the following theorem.

Theorem 4.2. Given any connected order, a query vertex not in the core structure has at most one backward neighbor.

Proof: Given q, all query vertices that are not in the core structure form a forest in which each vertex has one parent. Therefore, given any connected order, the vertices that are not in the core structure have at most one backward neighbor. □

Because we generate $u.C$ based on the backward neighbors of u in the forward stage, we prioritize the vertex by the number of backward neighbors to filter the false positive candidates at an early stage. Therefore, when generating the indexing order, we put the query vertices in the core structure before those not, according to Theorem 4.2. In particular, we select the vertex with the maximum core value as the start vertex in the indexing order instead of that with the maximum degree in order to (1) exclude the effect of the query vertices not in the core structure; and (2) start the candidate extraction from the dense part of the core structure. Algorithm 3 presents the details of generating the indexing order.

Algorithm 3: GenerateIndexingOrder

Input: a query graph q
Output: an indexing order π'

1 **begin**
2 $\pi' \leftarrow ()$, set $BN_q^{\pi'}(u)$ to empty for all $u \in V(q)$;
3 $u^* \leftarrow arg\max_{u \in V(q)} u.core$;
4 Add u^* to π';
5 **foreach** $u \in N(u^*) - \pi'$ **do**
6 $BN_q^{\pi'}(u) \leftarrow BN_q^{\pi'}(u) \cup \{u^*\}$;
7 **while** $|\pi'| < |V(q)|$ **do**
8 $u^* \leftarrow arg\max_{u \in V(q) - \pi'} |BN_q^{\pi'}(u)|$;
9 Same as Lines 4-6;
10 **return** π';

Tie Handling. When there are ties in the *max* function (Lines 3 and 8), we pick the vertex in the order of: (1) $u.core$; (2) $u.core\_degree$; and (3) $d(u)$ to lead the candidate extraction to the dense part of q. If there are still ties, we select the vertex with a lower id.

Example 4.5. Given q in Figure 7a, $u.core = 2$ for every $u \in V(q)$. Based on tie handling, u_0 is selected as the start vertex, whose core degree is 3 and id is lower than u_3. At the end of Algorithm 3, we obtain $\pi' = (u_0, u_3, u_1, u_2)$.

Because the core structure of q, which is a connected graph, is connected, Algorithm 3 has the following property.

Proposition 4.2. The indexing order π' generated by Algorithm 3 is connected, and the query vertices in the core structure are positioned before the other vertices in π'.

4.2 Index Construction

After generating a matching order and a pivot dictionary (Section 5.2), we construct an index on these candidate sets to serve the subsequent enumeration. In enumeration, we need to get the candidates of a query vertex u from its pivot u' (Line 13 in Algorithm 1). To facilitate this step, we maintain edges between $u.C$ and $u'.C$ for every pair $(u, u') \in \mathcal{P}$ as a bigraph, denoted as $BI_u^{u'}$, with $u.C$ and $u'.C$ as the two bipartitions. Because we get the candidates of u from u', $BI_u^{u'}$ records the neighbors in $u.C$ of candidates in $u'.C$. The details of index construction are shown in Algorithm 4. Algorithm 4 has the following property.

Proposition 4.3. Every bigraph $BI_u^{u'}$ in BI constructed by Algorithm 4 satisfies that $\forall v \in u.C \cap N(v')$ where $v' \in u'.C$, $v \in BI_u^{u'}(v')$.

4.3 Analysis of BI

We first prove the correctness of VC (Algorithm 1). For brevity, we only present the proof sketch.

Algorithm 4: ConstructIndex

Input: a candidate set C and a pivot dictionary $\mathcal{P}$
Output: the bigraph index BI

```
1  begin
2      foreach (u, u') ∈ P do
3          Set BI_u^{u'}(v) to empty for all v ∈ u'.C;
4          Set v.count to 1 for all v ∈ u.C;
5          foreach v ∈ u'.C do
6              foreach v' ∈ N(v) do
7                  if v'.count = 1 then
8                      BI_u^{u'}(v) ← BI_u^{u'}(v) ∪ {v'}
9          Reset v.count to 0 for all v ∈ u.C;
10     return BI;
```

Correctness of VC. Given q and G, VC can find all matches from q to G based on BI.

Proof: (1) Given M reported by Algorithm 1, BI guarantees M satisfies that $\forall u \in V(q), L(u) = L(M[u])$, and line 14 in Algorithm 1 ensures M satisfies that $\forall e(u,u') \in E(q), \exists e(M[u], M[u']) \in E(G)$ and M does not contain any duplicate data vertices. Therefore, M reported by VC is a match from q to G. (2) According to Proposition 4.1 and 4.3, the enumeration on BI is equivalent to G. Therefore, the search space of VC contains all matches from q to G. (3) There are no duplicate candidates and edges in BI. M does not contain any duplicate data vertices. So, VC will not output any duplicate matches. Thus, the correctness of VC is proved. □

Space Complexity. BI contains two components: the candidate sets and the bigraphs. As there are no duplicate candidates in C and no duplicate edges in BI, the size of the candidate set C is $O(|V(G)| \times |V(q)|)$ and the worst-case size of the bigraphs is $O(|E(G)| \times (|V(q)| - 1))$ (there are $|V(q)| - 1$ bigraphs in BI). As such, the space complexity of BI is $O(|E(G)| \times |V(q)|)$.

Time Complexity. The function *GenerateIndexingOrder* takes time polynomial to the size of query graph, which can be omitted since the query graph is very small compared with the data graph. We implement the forward stage in Algorithm 2 by a counter-based technique proposed in CFL[3], whose time complexity is $O(|E(G)| \times |E(q)|)$. Lines 14-20 in Algorithm 2 take at most $O(\sum_{v \in u.C} d(v) \times d(u))$ time. Similarly, lines 21-23 also take at most $O(\sum_{v \in u.C} d(v) \times d(u))$ time. Therefore, the backward stage (Lines 12-23) spends $O(\sum_{u \in V(q)} \sum_{v \in u.C} d(v) \times d(u)) = O(|E(G)| \times |E(q)|)$ time. Thus, the time complexity of Algorithm 2 is $O(|E(G)| \times |E(q)|)$. Lines 5-8 in Algorithm 4 take $O(|E(G)|)$ time. So, the time complexity of Algorithm 4 is $O(|E(G)| \times |V(q)|)$. Then, the time complexity of constructing BI is $O(|E(G)| \times |E(q)|)$.

Pruning Power Comparison. By the definition of PSIC, we can derive the following lemma.

Lemma 4.2. Given q and G, C is complete. If (u,v) satisfies PSIC where $u \in V(q)$ and $v \in u.C$, then (u,v) satisfies that $\forall u' \in N(u), N(v) \bigcap u'.C \neq \emptyset$.

However, given $v \in u.C$, (u,v) may violate PSIC even if it satisfies that $\forall u' \in N(u), N(v) \bigcap u'.C \neq \emptyset$.

Example 4.6. Take $v_2 \in u_2.C$ in Figure 7c as an example: v_2 satisfies that $N(v_2) \cap u_0.C \neq \emptyset$ and $N(v_2) \cap u_3.C \neq \emptyset$, whereas it violates PSIC because $N(v_2) \cap u_0.C = N(v_2) \cap u_3.C = \{v_0\}$.

Therefore, the pruning power of PSIC in *BI* is stronger than the observation used in CPI of CFL. With the ping-pong filtering strategy, *BI* can further reduce the sizes of candidate sets. However, as *BI* and CPI extract candidates with the orders generated by different heuristic rules, we cannot theoretically prove that the number of candidates obtained by *BI* is smaller than that of CPI. Instead, we experimentally compare the pruning power of *BI* with that of CPI, and the experimental results show that *BI* has stronger pruning power than CPI (see Section 6.4.1).

5 MATCHING ORDER GENERATION

In this section, we present the method that generates a matching order as well as a pivot dictionary.

5.1 State Space Tree based Cost Model

We first define the partial subgraph isomorphism.

Definition 17. **Partial Subgraph Isomorphism:** Given q, G and π, q' is a vertex induced subgraph of q constructed on $\pi[1:i]$ where $1 \leqslant i \leqslant |V(q)|$. A partial subgraph isomorphism (psi), denoted as f', is a subgraph isomorphism from q' to G. Specifically, when $i = 0$, we name the psi as the initial psi, denoted as f_r and $f_r = \{\}$.

State Space Tree Exploration. The enumeration phase of Algorithm 1 is based on the recursive backtracking technique, which conceptually constructs a state space tree H on the fly. It starts from the initial psi and always extends the most recently generated psi by mapping a query vertex to a data vertex along the matching order for the following step. In other words, it explores H by the depth-first search order. The initial state of H is f_r and the internal states are the partial subgraph isomorphisms generated during enumeration. The edges of H correspond to mappings between a query vertex in the matching order to a candidate in BI. To differentiate the edges in H from that in graphs, we call an edge in H an *action*. The leaves of H are terminations of search paths originated from f_r, which can be categorized into two classes: the success leaves with the *if-condition* at line 11 of Algorithm 1 as $true$ and the failure leaves with the *if-condition* at line 14 as $false$. All success leaves are the solutions of subgraph matching. Except the failure leaves, there is a one-to-one relationship from a state in H to a psi. So, we call a non-failure state in H a psi state and denote it as S.

Example 5.1. The search tree in Figure 5d can be viewed as the state space tree H generated by VC on graphs in Figure 1. Take node s_2 as an example. It corresponds to the psi $\{(u_0,v_0),(u_1,v_1)\}$, and the actions under s_2 map u_3 to each candidate in $BI_{u_3}^{u_1}(v_1) = \{v_6, v_7, v_8\}$. VC explores H by the DFS order, and finds 7 matches finally.

Cost Model. Given q and G, VC explores H to find all matches. So, the cost of VC, denoted as T_{iso}, can be estimated by the total number of actions in H, assuming the cost of each action is similar. The maximum depth of H, denoted as n, is equal to $|\pi|$. We regard the depth of f_r, the initial psi, as 0. T_i denotes the number of actions under the psi states at depth i. Then, T_{iso} can be computed as follows.

$$T_{iso} = \sum_{i=0}^{n-1} T_i. \quad (1)$$

N_i denotes the number of the psi states at depth i, b_j^i is the search breadth of the jth psi state at depth i and the average search breadth of the psi states at depth i is b^i. Then, $T_0 = N_1$ since the psi states at depth 1 are generated by lines 6-7 of Algorithm 1, where no candidates are pruned, and $T_i = \sum_{j=1}^{j=N_i} b_j^i = N_i \times b^i$ where $1 \leqslant i \leqslant n-1$. Given an action at depth i where $1 \leqslant i \leqslant n-1$, if it can pass the *if-condition* at line 14 of Algorithm 1, then a new psi state at depth $i+1$ is generated. We call such an action a valid action. N_{i+1} is equal to the number of the valid actions at depth i. We define valid factor α_i as the fraction of valid actions over the total number of actions at depth i. So, $N_{i+1} = T_i \times \alpha_i$ and T_i can be computed as follows.

$$T_i = \begin{cases} N_1 & if\ i = 0. \\ N_1 b^1 & if\ i = 1. \\ N_1 b^i \prod_{j=1}^{i-1} \alpha_j b^j & if\ 2 \leqslant i \leqslant n-1. \end{cases} \quad (2)$$

Based on Equation 1 and 2, the total cost of enumeration can be computed as follows.

$$T_{iso} = N_1(1 + b^1 + \sum_{i=2}^{n-1} b^i \prod_{j=1}^{i-1} \alpha_j b^j). \quad (3)$$

5.2 The Vertex-based Ordering

Estimation of T_{iso}. To minimize T_{iso}, we first give an estimation of the parameters in T_{iso} by examining the factors in enumeration, which affect the parameters.

Given q, G, π, $\mathcal{P}$ and BI, let S_j^i be the jth psi state at depth i. u is the $(i+1)$th vertex in π, whose pivot is u'. In S_j^i, u' has been mapped to v'. In Algorithm 1, procedure *Enumerate* tries to extend S_j^i by mapping u to $v \in BI_u^{u'}(v')$. The search breadth b_j^i of S_j^i is equal to $|BI_u^{u'}(v')|$. So, b^i can be estimated as $\frac{|E(BI_u^{u'})|}{|u'.C|}$, denoted as $\widetilde{b}^i$. In Algorithm 1, function *Validate* plays the key role in terminating the invalid search paths by checking the existence of edges between a candidate v and the mapped data vertices of $BN_q^\pi(u)$. v has a higher probability to be pruned if u has more backward neighbors. So, α_i is estimated as $\frac{1}{|BN_q^\pi(u)|^2}$, denoted as $\widetilde{\alpha}_i$. N_1 is equal to the number of candidates of $\pi[1]$. By replacing the parameters in Equation 3 with our estimations, we obtain the estimation $\widetilde{T}_{iso}$ of the total cost, and optimize $\widetilde{T}_{iso}$ instead of T_{iso}. $\widetilde{\alpha}_i$ is determined by the actual matching order π. We call the corresponding $\widetilde{\alpha}_i$ of every vertex in π an assignment. As the number of assignments is equal to the number of permutations of query vertices ($|V(q)|!$), it is too expensive to compute the optimal value on the fly. So, we propose a greedy approach to minimize $\widetilde{T}_{iso}$.

Greedy Approach. The general idea is to select the query vertex with a minimum $\widetilde{b}^i \times \widetilde{\alpha}_i$ value as the next vertex in π to prune the invalid search paths at an early stage and minimize the number of psi states at the next depth. As the matching order is generated vertex by vertex, we call this strategy the vertex-based ordering. Algorithm 5 outlines the matching order generation.

Algorithm 5: GenerateMatchingOrder

Input: a query graph q, a data graph G and the candidate set C
Output: a matching order π, a pivot dictionary $\mathcal{P}$ and the backward neighbors BN_q^π

1 **begin**
2 $q^w \leftarrow GenerateWeightedGraph(q, G, C)$;
3 $V_C \leftarrow \{u \in V(q) | u.core \geqslant 2\}, V_{NC} \leftarrow V(q) - V_C$;
4 $\mathcal{P} \leftarrow \{\}$, $w^*[u] \leftarrow |V(G)|$ for all $u \in V(q)$;
5 Set $BN_q^\pi(u)$ to empty for all $u \in V(q)$, set UN to empty;
6 $u^* \leftarrow arg\min_{u \in V_C} \frac{|u.C|}{u.core}$, $\pi \leftarrow (u^*)$;
7 **foreach** $u \in N(u^*) - \pi$ **do**
8 $BN_q^\pi(u) \leftarrow BN_q^\pi(u) \cup \{u^*\}$;
9 **if** $w(u^*, u) \leqslant w^*[u]$ **then**
10 $w^*[u] \leftarrow w(u^*, u)$, $\mathcal{P}[u] \leftarrow u^*$;
11 Add u to UN if $u \notin UN$;
12 **while** $|\pi| < |V_C|$ **do**
13 $u^* \leftarrow arg\min_{u \in UN \cap V_C} \frac{w^*[u]}{|BN_q^\pi(u)|^2}$;
14 Add u^* to π, remove u^* from UN;
15 Same as Lines 7-11;
16 **while** $|\pi| < |V(q)|$ **do**
17 $u^* \leftarrow arg\min_{u \in UN \cap V_{NC}} \frac{w^*[u]}{d^2(u)}$;
18 Add u^* to π, remove u^* from UN;
19 Same as Lines 7-11;
20 **return** $(\pi, \mathcal{P}, BN_q^\pi)$;

Line 2 first generates a directed weighted graph q^w from q as follows: for each edge $e(u, u') \in E(q)$, we generate two edges $e(u, u')$ and $e(u', u)$ in q^w with $w(u, u') = \frac{m}{|u.C|}$ and $w(u', u) = \frac{m}{|u'.C|}$ where m is the number of edges between the data vertices in $u.C$ and $u'.C$ in the data graph. Hence, $w(u, u')$ is the value of $\widetilde{b}^i$ if we select u as the pivot of u'.

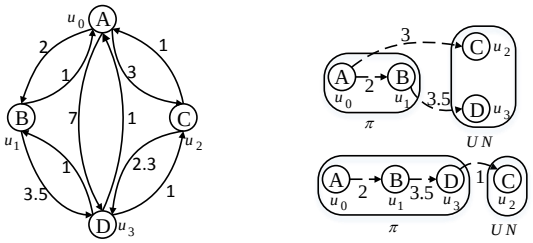
V_C contains the vertices in the core structure, whereas V_{NC} stores the other vertices (Line 3). We initialize the pivot dictionary $\mathcal{P}$ as empty and use $w^*[u]$ to store $\min_{u' \in BN_q^\pi(u)} w(u', u)$ (Line 4). Based on Theorem 4.2, we prioritize the query vertices in the core structure when generating the matching order. Moreover, we prefer starting enumeration from the dense part of the core structure, and the start vertex has a small number of candidates. Therefore, we pick a vertex with the smallest $\frac{|u.C|}{u.core}$ value from the core structure as the start vertex (Line 6). If V_C is empty, we replace it with V_{NC}. Whenever modifying the value of $w^*[u]$, we update the pivot of u to keep that $w(\mathcal{P}[u], u)$ (i.e., the estimated search breadth $\widetilde{b}^i$) has the smallest value among $w(u', u)$ where $u' \in BN_q^\pi(u)$ (Line 10).

We process the vertices in V_C and pick the vertex with the minimum $\widetilde{b}^i \times \widetilde{\alpha}_i$ value (Lines 12-15). Lines 16-19 process the vertices that are not in the core structure. For every $u \in UN \cap V_{NC}$, $|BN_q^\pi(u)| = 1$, $u.core = 1$ and $u.core\_degree = 0$, we use $d(u)$ instead of $|BN_q^\pi(u)|$.

Time Complexity. The time complexity of *GenerateWeightedGraph* is $O(|E(G)| \times |E(q)|)$ with the same technique in Algorithm 4 and the time complexity of the remainder of Algorithm 5 is $O(|E(q)|)$. So, the time complexity of Algorithm 5 is $O(|E(G)| \times |E(q)|)$.

Tie Handling. When there are ties in the *min* function at line 13, we prefer leading enumeration to the dense part of q. So, the denominator $|BN_q^\pi(u)|$ is replaced by the following attribute with priority in order of: (1) $u.core$; (2) $u.core\_degree$; and (3) $d(u)$. If there are still ties, the vertex

with a lower id is selected. Specifically, when $|BN_q^\pi(u)| = 1$ for every $u \in UN \cap V_C$, we use the tie handling to pick the vertex because the *Validate* function in Algorithm 1 always returns $true$ if $|BN_q^\pi(u)| = 1$. When there are ties at line 5 and 17, the vertex with a lower id is selected.



a q^w. b The generation of π and $\mathcal{P}$.

Fig. 8: The running example of Algorithm 5.

Example 5.2. Figure 8a shows q^w generated from the running example in Figure 5. Take $e(u_0, u_1) \in E(q)$ as an example. We convert it to $e(u_0, u_1)$ and $e(u_1, u_0)$ in $E(q^w)$ with $w(u_0, u_1) = 2$ and $w(u_1, u_0) = 1$ since there are two edges between $u_0.C$ and $u_1.C$ where $|u_0.C| = 1$ and $|u_1.C| = 2$. Figure 8b illustrates the process of picking a vertex. The start vertex of an edge is the pivot of the end vertex, and the weight is the w^* value of the end vertex. In the top subfigure of Figure 8b, because $\frac{w^*[u_3]}{|BN_q^\pi(u_3)|^2} = 0.875$ is less than $\frac{w^*[u_2]}{|BN_q^\pi(u_2)|^2} = 3$, u_3 is picked, and the pivot of u_3 is u_1. After that, $w^*[u_2]$ is set to 1 since $w(u_3, u_2)$ is less than the previous $w^*[u_2]$.

Algorithm 5 has the following property.

Proposition 5.1. The matching order π generated by Algorithm 5 is connected, the query vertices in the core structure are positioned before the other vertices in π. All pairs of vertices in the pivot dictionary $\mathcal{P}$ form a spanning tree of the query graph q.

6 EXPERIMENTS

In this section, we conduct extensive experiments to evaluate the performance of VC on both real-world and synthetic datasets.

6.1 Experimental Setup

Algorithms Under Study. We evaluate the following algorithms in our experiments, which have different indices and ordering strategies.

- VC: our algorithm. It has the bigraph index BI and the vertex-based ordering strategy.
- CFL [3]: the state-of-the-art algorithm. It has the tree-structured index CPI and the path-based ordering strategy.
- GQL [11]: GraphQL algorithm. It has the neighborhood signature filter and the left-deep-join ordering strategy.
- QSI [23]: QuickSI algorithm. It does not have any indices and adopts the infrequent-edge-first ordering strategy.

Additionally, we compare VC and CFL with two other algorithms SPath [28] and TurboIso [9], which adopt the path-based ordering strategy. We do not involve SPath and TurboIso in all of our experiments, because they utilize the same ordering strategy as CFL, and CFL outperforms them. We do not consider the Boost technique [20], which accelerates subgraph matching by compressing data graphs, because according to previous experiments [3] the compression has overhead, and the performance of CFL outperforms CFL-Boost.

Experimental Environment. We obtain the source code of CFL from its author, and implement the other algorithms whose source code is unavailable. All involved algorithms are implemented in C++ and compiled with g++ 4.9.3 with the -O3 flag. We perform all experiments on a 64-bit Linux machine equipped with an Intel Xeon E5-2660 v2 processor and 32GB RAM.

Data Graphs. We evaluate the performance on the following real and synthetic datasets.

Real Datasets. We select six real datasets, which have been widely used in previous work [3], [20], [25]. Yeast, Human, HPRD and WordNet originally contain labels. The other two datasets have no labels, to each of which we randomly assign distinct labels. Table 2 lists the detailed information.

TABLE 2: Properties of real datasets where d is the average degree, LF is the label frequency and LF.SD is the standard derivation of LF.

Dataset	$\|\mathbf{V}\|$	$\|\mathbf{E}\|$	$\|\mathbf{\Sigma}\|$	d	LF	LF.SD
Yeast	3,112	12,519	71	8.04	43.83	102.94
Human	4,674	86,292	44	36.91	106.23	146.80
HPRD	9,460	34,998	307	7.39	30.81	92.32
WordNet	76,853	120,399	5	3.13	15,370.6	23,960.25
Youtube	1,134,890	2,987,624	25	5.27	45,395.6	194.46
US Patents	3,774,768	16,518,948	20	8.75	188,738.4	549.6

Synthetic Datasets. We generate synthetic datasets using the RMAT model [4] and randomly assign labels to vertices. To examine the scalability of the algorithms on a variety of data graphs, we vary $|V(G)|$, $d(G)$ and $|\Sigma|$ respectively. The default configuration is $|V(G)| = 100K$ ($K = 10^3$), $d(G) = 16$ and $|\Sigma| = 30$.

- Vary $|V(G)|$: We generate 4 data graphs with the number of vertices as $100K$, $150K$, $200K$ and $250K$ respectively.
- Vary $d(G)$: We generate 4 data graphs with the degrees as 8, 16, 24 and 32 respectively.
- Vary $|\Sigma|$: We generate 4 data graphs with the number of distinct labels as 20, 30, 40 and 50 respectively.

Query Sets. Following previous research [3], [20], [25], we generate query graphs by selecting subgraphs from a data graph randomly to guarantee that at least one match exists. For each data graph, we generate eight query sets as default queries to evaluate the performance of the competing algorithms, and four query sets with large query graphs to examine their scalability. Each query set contains 200 query graphs with the same number of vertices. Each query graph is connected. Specifically, six query sets contain non-sparse query graphs (i.e., average degree $\geqslant 3$), while the other six query sets contain sparse query graphs (i.e., average degree < 3). The query sets of each data graph are shown in Table 3, where q_{iN} represents a set of i-vertex

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.2980257, IEEE Transactions on Knowledge and Data Engineering

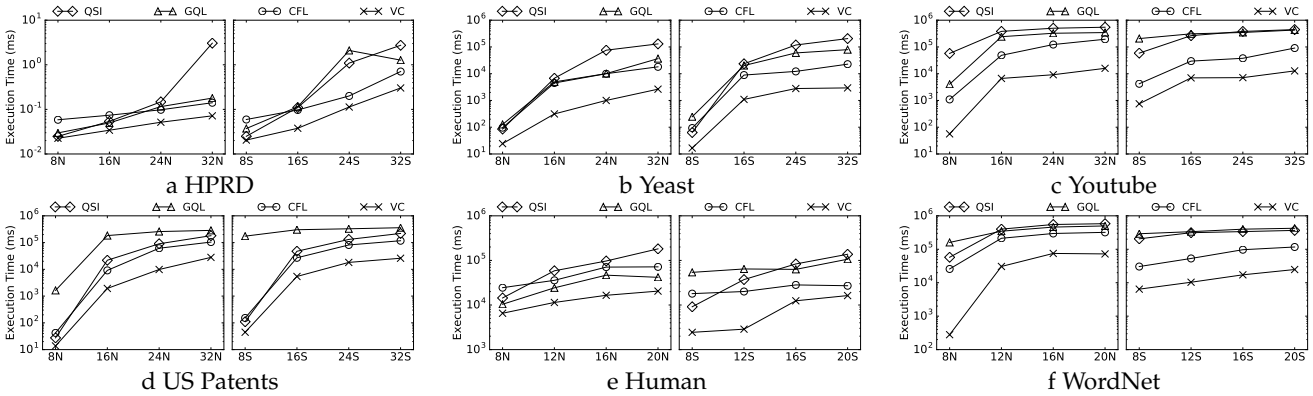


Fig. 9: Execution time on the real datasets with $|V(q)|$ varied.

non-sparse query graphs and q_{iS} denotes a set of i-vertex sparse query graphs. The query graphs for WordNet and Human are smaller, because 63,098 of the 76,853 vertices (i.e., $\geqslant 80\%$) in WordNet have the same label and Human is very dense. As a result, subgraph matching on these two graphs is harder than the other four graphs.

TABLE 3: Query sets information.

Dataset	Default Queries	Large Queries
Yeast, HPRD, Youtube, US Patents, Synthetic	$q_{8N}, q_{16N}, q_{24N}, q_{32N}$, $q_{8S}, q_{16S}, q_{24S}, q_{32S}$	q_{64N}, q_{128N}, q_{64S}, q_{128S}
Human, WordNet	$q_{8N}, q_{12N}, q_{16N}, q_{20N}$, $q_{8S}, q_{12S}, q_{16S}, q_{20S}$	q_{25N}, q_{30N}, q_{25S}, q_{30S}

Metrics. The evaluation metrics of previous algorithms, which only examine the average execution time of query sets and set time limit for processing the entire query set, can hide the details of the execution time per query due to "stragglers" [13]. Therefore, following [13], we examine query graphs in query sets individually. The time limit for processing a query graph is 10 minutes (i.e., 6×10^5 ms). If an algorithm cannot finish within the time limit, we record the elapsed time as 10 mins for comparison purpose. The detailed metrics are as follows.

- Execution Time: The average time of processing a query graph in a query set, which excludes the time of loading the data from the disk. It consists of the indexing time and the enumeration time.
- Relative Performance: The relative performance of an algorithm a on a query is $\frac{t_a}{\min_{a' \in A} t_{a'}}$ where t_a is the execution time of a on the query and A is the set of algorithms involved. We report the average relative performance of a query set. The minimum value of this metric is 1, indicating that the algorithm is very competitive on each query graph in the query set.
- Execution Time Category: Following the settings in [13], we categorize queries that finish within 2 seconds into the *easy* category, and that finish between 2 seconds to 10 mins into the *median* category. The term *completed* refers to all queries that finish within 10 mins (*easy* and *median*). The queries that are terminated due to time limit are called *hard* queries.

To compare different indices, we use the following metrics.

- Indexing Time: The average time spent on the indexing phase for processing a query graph.
- Index Size: The average number of candidates in the index for processing a query graph, which is used to evaluate the filtering power of the index.

#Embeddings. We vary the number of embeddings to be reported from 10^3 to 10^9. As subgraph matching aims to find all embeddings, we set the default number of embeddings to 10^9 to cover as much search space as time allows.

6.2 Comparison with Existing Algorithms

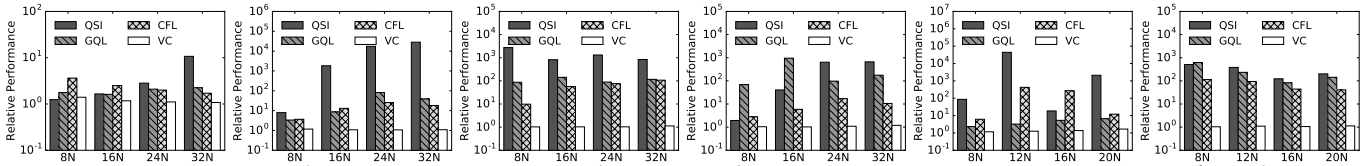
Evaluation on the execution time. In Figure 9, all algorithms generally spend more time on larger queries. The performance of QSI, GQL and CFL varies on different data graphs. CFL outperforms GQL and QSI on Yeast, Youtube, US Patents and WordNet, but performs worse than GQL on non-sparse query graphs on HPRD and Human (CFL was not compared with GQL in previous research). VC consistently outperforms the other algorithms on all data graphs and improves upon existing algorithms by over an order of magnitude (Figure 9b, 9c and 9f), which demonstrates the efficiency and robustness of VC. The performance gap among GQL, CFL and VC is small on HPRD, because the large number of distinct labels makes HPRD an easy dataset for queries. In Figure 9c and 9f, the curves of QSI, GQL and CFL appear insensitive to the increase of query size. It is because a large number of queries cannot complete within the time limit (see Figure 11) and their execution times are replaced with 10 minutes. As such, the actual performance gap between VC and existing algorithms is greater than that shown in Figure 9, as the number of hard queries of VC is much fewer than that of the other three algorithms.

As the performance of the algorithms has similar trends on sparse query graphs and non-sparse query graphs, next, we only report the results of relative performance and execution time category on non-sparse query graphs.

Evaluation on the relative performance. As shown in Figure 10, the relative performance of VC is very close to 1 on all datasets, and outperforms the others by up to two orders of magnitude in some cases (Figure 10c and 10f). In comparison, the other three algorithms all have a relative performance slower than 1, because they process a large number of false positive candidates and generate a less effective matching order than VC.

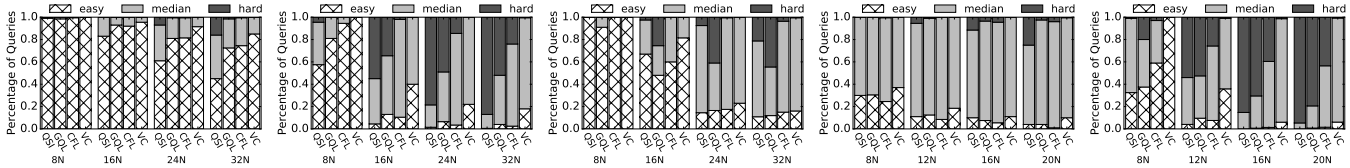
Evaluation on the execution time category. All algorithms complete queries on HPRD within two seconds, so

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.2980257, IEEE Transactions on Knowledge and Data Engineering



a HPRD b Yeast c Youtube d US Patents e Human f WordNet

Fig. 10: Relative performance on the real datasets with $|V(q)|$ varied.

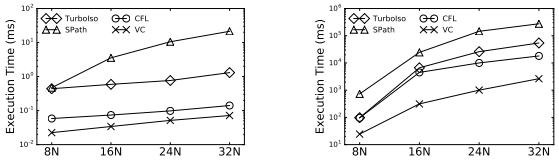


a Yeast b Youtube c US Patents d Human e WordNet

Fig. 11: Execution time category on the real datasets with $|V(q)|$ varied.

we omit the result on HPRD. Figure 11 presents the execution time category on the other five datasets. The portion of *median* and *hard* queries generally increases with the query size. VC has more *easy* queries and fewer *hard* queries than the others, especially when the query sizes are large. This result also reflects the efficiency and robustness of VC.

Comparison with SPath and TurboIso. In this experiment, we compare VC and CFL with SPath and TurboIso on HPRD and Yeast. As shown in Figure 12, CFL outperforms TurboIso, and TurboIso runs faster than SPath, which is consistent with the previous experiment results [3], [9]. Moreover, VC performs the best among the four algorithms.



a HPRD b Yeast

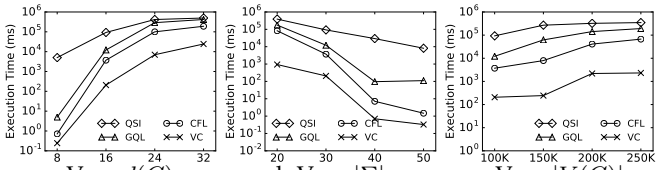
Fig. 12: Comparison with SPath and TurboIso.

For brevity, we only report results on execution time of q_{16N} for Human and WordNet and q_{24N} for the other datasets in the following experiments unless we state that we use different queries.

6.3 Scalability Evaluation

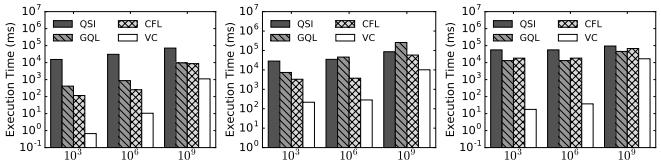
Evaluation on the synthetic datasets. Figure 13a shows the execution time with $d(G)$ varied. All algorithms take longer time with the increase of $d(G)$, because the growth of $d(G)$ results in the increase of the search breadth. VC consistently outperforms the others, and the performance gap between VC and others narrows down when $d(G) = 32$, because existing algorithms failed to complete on a large number of queries when $d(G) = 32$. In Figure 13b, the execution time of involved algorithms decreases with the increase of $|\Sigma|$. CFL, GQL and QSI have severe performance issues when $|\Sigma|$ is small, as the search breadth is large then. Figure 13c shows the experiment results with $|V(G)|$ varied. Compared with $d(G)$ and $|\Sigma|$, $|V(G)|$ does not have a significant impact on the execution time.

Evaluation with #embeddings varied. As shown in Figure 14, the execution time of all algorithms increases when more embeddings are reported, and VC consistently outperforms the other algorithms.



a Vary $d(G)$ b Vary $|\Sigma|$ c Vary $|V(G)|$

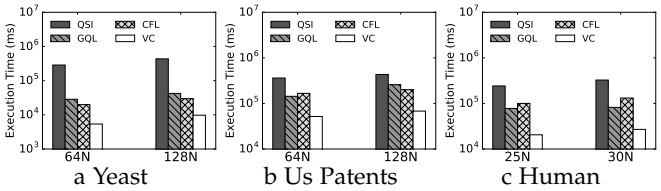
Fig. 13: Execution time on synthetic datasets.



a Yeast b Us Patents c Human

Fig. 14: Execution time with #embeddings varied.

Evaluation on the large queries. Figure 15 shows the execution time on query sets that contain the large query graphs. VC runs much faster than the other three algorithms, which shows that VC scales better than the other algorithms on large queries.



a Yeast b Us Patents c Human

Fig. 15: Execution time on large queries.

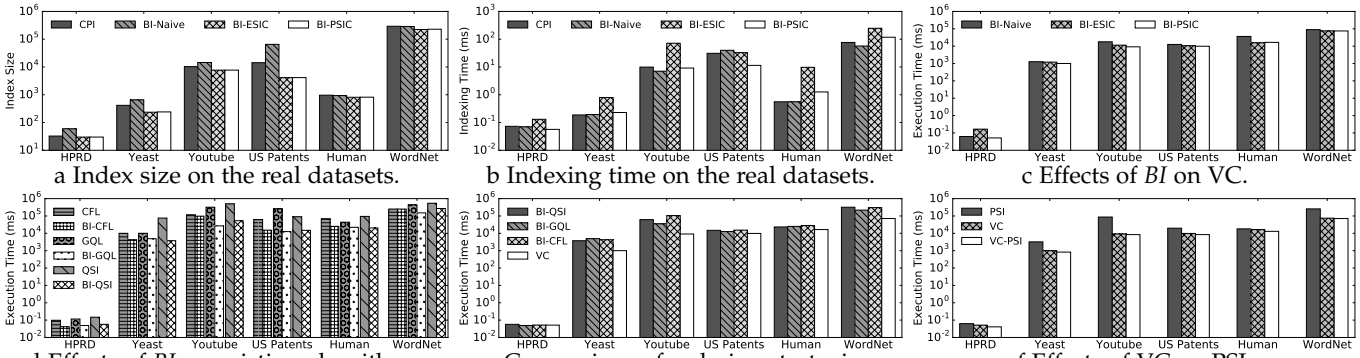
6.4 Effectiveness of BI and the Vertex-based Ordering Strategy

We proposed two techniques in VC: *BI* index and the vertex-based ordering strategy. In this section, we evaluate these techniques respectively with real datasets.

6.4.1 Effectiveness of BI

To evaluate the effectiveness of *BI*, we compare four kinds of indexing strategies in terms of index size, indexing time and index memory cost. Since the space complexity of all these strategies is $O(|E(G)| \times |V(q)|)$, the memory cost of indices is very small in practice and all of them consume less than 10MB on each of the six datasets. Thus, we omit the experiment results on index memory cost.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.2980257, IEEE Transactions on Knowledge and Data Engineering



a Index size on the real datasets. b Indexing time on the real datasets. c Effects of *BI* on VC.

d Effects of *BI* on existing algorithms. e Comparison of ordering strategies. f Effects of VC on PSI.

Fig. 16: Experiment results on effectiveness of techniques proposed in VC.

- CPI: the tree-structured index in CFL [3]. It is constructed by the top-down generation and bottom-up refinement.
- BI-Naive: *BI* index constructed by the forward generation.
- BI-ESIC: *BI* index constructed by the forward generation and the backward filtering based on the exact star isomorphism constraint and the ping-pong filtering strategy.
- BI-PSIC: *BI* index constructed by the forward generation and the backward filtering based on the pseudo star isomorphism constraint and the ping-pong filtering strategy.

Evaluation on the index size. As shown in Figure 16a, BI-Naive generates more candidates than the others, since it does not have any filtering processes. BI-PSIC and BI-ESIC both obtain fewer candidates than CPI on all datasets, which demonstrates that they have stronger filtering power than CPI. BI-PSIC's number of candidates is almost identical to BI-ESIC. Because Human is dense and most vertices in WordNet have an identical label, BI-PSIC and BI-ESIC reduce around 15% and 21% candidates respectively over BI-Naive and CPI.

Evaluation on the indexing time. Figure 16b shows the experiment results on indexing time. CPI, BI-Naive and BI-PSIC have the same time complexity, and BI-Naive generally runs faster than the other strategies, since it has the forward generation process only. However, both CPI and BI-PSIC outperform BI-Naive when the number of candidates is significantly reduced (i.e., on US Patents), because the time spent on extracting edges between candidate sets decreases with a small number of candidates. BI-PSIC outperforms BI-ESIC by up to 8X in terms of indexing time.

Evaluation of the effects of BI on VC. We integrate BI-Naive, BI-ESIC and BI-PSIC with VC respectively to evaluate the effects of different indexing strategies on the performance of VC. Figure 16c shows the experiment results on the execution time. Comparing the results in Figure 16b and 16c, we observe that the indexing time dominates the execution time on HPRD, whereas the enumeration time, which is the time spent on the enumeration process, dominates on the other datasets. Due to the poor indexing time efficiency of BI-ESIC, BI-PSIC outperforms it by 3.2X on HPRD. On the other datasets, BI-ESIC and BI-PSIC spend almost the same amount of execution time, and both outperform BI-Naive. Although BI-PSIC and BI-ESIC significantly reduce the number of candidates compared with BI-Naive, the performance improvement is not as significant (around 1.26X-2.19X). This difference is because the vertex-based ordering strategy in VC is robust to the number of false positive candidates as it considers both the connectivity among query vertices and the number of candidates.

In summary, BI-PSIC's filtering power is as strong as BI-ESIC, and its time efficiency on index construction is much faster. As a result, BI-PSIC outperforms BI-ESIC when the indexing time dominates execution time, and achieves similar performance improvements when the enumeration time dominates.

Evaluation of the effects of BI on existing algorithms. In order to evaluate the benefits of *BI* (*BI* with PSIC) further and show its generability, we integrate CFL, QSI and GQL with *BI*, which are denoted as BI-CFL, BI-QSI and BI-GQL respectively. As shown in Figure 16d, the performance of all integrated algorithms are improved compared with their original versions, because (1) *BI* reduces the number of false positive candidates to provide more accurate information for matching order generation; and (2) *BI* reduces the search breadth. Moreover, to the best of our knowledge, *BI* is the first index that can be integrated with different kinds of ordering strategies, whereas the previous tree-structured index supported the path-based ordering strategy only.

6.4.2 Effectiveness of the vertex-based ordering strategy

To study the effect of our proposed vertex-based ordering strategy, we integrate ordering strategies in QSI, GQL and CFL with *BI*. Figure 16e shows the experiment results on different ordering strategies. Because the indexing time dominates the execution time on HPRD, these algorithms have similar performance on HPRD. On the other datasets, the performance of BI-QSI, BI-GQL and BI-CFL varies, and there is no single winner on all datasets among them, including the path-based ordering strategy, which is regarded as an effective technique in previous research. In contrast, VC consistently outperforms the other algorithms, which demonstrates the efficiency and robustness of our proposed vertex-based ordering strategy.

6.5 Integration with Existing Acceleration Techniques

Instead of designing new subgraph matching algorithms, some researchers [13] tried to accelerate subgraph matching by utilizing existing algorithms based on the observation that it is difficult to design a subgraph matching algorithm

that outperforms others for all queries on all datasets. They proposed a framework called PSI, which executes a variety of algorithms in parallel simultaneously (each algorithm instance executes in serial) and returns the results when one algorithm instance completes. To evaluate the effects of VC on PSI, we compare the following algorithms: PSI (contains QSI, GQL and CFL, 3 threads), VC (1 thread) and VC-PSI (contains BI-QSI, BI-GQL, BI-CFL and VC, 4 threads). As shown in Figure 16f, VC outperforms PSI by 1.21-9.47X, and VC-PSI achieves 1.05-1.27X speedups over VC, which shows that VC significantly improves the performance of PSI. Additionally, we examine the number of stragglers of VC that are accelerated by VC-PSI. Specifically, we regard a query as a straggler of VC if VC-PSI runs over five times faster than VC on this query. As shown in Table 4, VC has a small number of stragglers (200 queries in total). Nevertheless, as VC outperforms others on most cases, the benefit of PSI is limited for VC. Furthermore, PSI consumes more computing resources than VC.

TABLE 4: Number of stragglers accelerated by VC-PSI.

HPRD	Yeast	Youtube	US Patents	Human	WordNet
0	2	0	1	5	1

In addition to parallelization, PSI [13] designs the *query rewriting* technique to further accelerate subgraph matching, which keeps the structure of the query graph, but reassigns the *id*s of vertices following some heuristic rules, such as the descending order of degrees. This technique will affect the matching order when there are ties during the generation of matching orders and the integrated algorithms do not handle the ties. However, because VC has well-designed tie handling strategies, the query rewriting has little effect on our vertex-based ordering strategy. We omit the experiment results of the query rewriting for brevity.

7 CONCLUSION

In this paper, we propose a new subgraph matching algorithm VC. By partitioning the indexing phase into candidate extraction, matching order generation and index construction, we break the limitations in the tree-based frameworks [3], [9]. We construct a bigraph index BI along the indexing order with the pseudo star isomorphism constraint and the ping-pong filtering strategy, which possesses better filtering power than index structures in previous work [3]. Most importantly, by abstracting the enumeration into the exploration of a state space tree, we propose the vertex-based ordering strategy to address the serious performance issues caused by existing ordering strategies. Detailed experiments on the real and synthetic datasets show that VC achieves significant performance improvements over existing algorithms.

REFERENCES

[1] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using map-reduce. *ICDE*, 2013.
[2] V. Batagelj and M. Zaversnik. An o (m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049*, 2003.
[3] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. *SIGMOD*, 2016.
[4] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. *SDM*, 2004.
[5] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *TPAMI*, 2004.
[6] M. R. Garey and D. S. Johnson. Computers and intractability: a guide to the theory of np-completeness. *WH Free. Co.*, 1979.
[7] J. A. Grochow and M. Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. *RECOMB*, 2007.
[8] M. Hall. *Combinatorial theory*. John Wiley & Sons, 1998.
[9] W.-S. Han, J. Lee, and J.-H. Lee. Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. *SIGMOD*, 2013.
[10] H. He and A. K. Singh. Closure-tree: An index structure for graph queries, 2006.
[11] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. *SIGMOD*, 2008.
[12] J. E. Hopcroft and R. M. Karp. A n5/2 algorithm for maximum matchings in bipartite. *Annual Symposium on Switching and Automata Theory*, 1971.
[13] F. Katsarou, N. Ntarmos, and P. Triantafillou. Subgraph querying with parallel use of query rewritings and alternative algorithms. *EDBT*, 2017.
[14] H. Kim, J. Lee, S. S. Bhowmick, W.-S. Han, J. Lee, S. Ko, and M. H. Jarrah. Dualsim: Parallel subgraph enumeration in a massive graph on a single machine. *SIGMOD*, 2016.
[15] R. Kimmig, H. Meyerhenke, and D. Strash. Shared memory parallel subgraph enumeration. In *IPDPS*, 2017.
[16] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce. *PVLDB*, 2015.
[17] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang. Scalable distributed subgraph enumeration. *VLDB*, 2017.
[18] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 2012.
[19] M. Qiao, H. Zhang, and H. Cheng. Subgraph matching: on compression and computation. *PVLDB*, 2017.
[20] X. Ren and J. Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *PVLDB*, 2015.
[21] X. Ren and J. Wang. Multi-query optimization for subgraph isomorphism search. In *PVLDB*, 2016.
[22] S. B. Seidman. Network structure and minimum degree. *Social networks*, 1983.
[23] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 2008.
[24] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu. Parallel subgraph listing in a large-scale graph. *SIGMOD*, 2014.
[25] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 2012.
[26] J. R. Ullmann. An algorithm for subgraph isomorphism. *JACM*, 1976.
[27] D. B. West et al. *Introduction to graph theory*. Prentice hall Upper Saddle River, 2001.
[28] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 2010.

Shixuan Sun received his M.S. and B.S. in Computer Science from Tongji University, Shanghai, China, in 2014 and 2011 respectively. He is currently working toward the PhD degree in the Department of Computer Science and Engineering, Hong Kong University of Science and Technology. His research interests are in the area of graph query processing.

Qiong Luo received her Ph.D. in Computer Science from the University of Wisconsin-Madison in 2002, her M.S. and B.S. in Computer Science from Beijing (Peking) University, China in 1997 and 1992 respectively. She is currently an Associate Professor at the Department of Computer Science and Engineering, Hong Kong University of Science and Technology. Her research interests are in big data systems, parallel and distributed systems, and scientific computing.