Multimedia Tools and Applications (2024) 83:69875–69896
https://doi.org/10.1007/s11042-024-18328-3

Tps: A new way to find good vertex-search order for exact subgraph matching

Yixing Ma[1,2] · Baomin Xu[2] · Hongfeng Yin[3]

Received: 31 July 2023 / Revised: 14 November 2023 / Accepted: 19 January 2024 /
Published online: 3 February 2024
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2024

Abstract
Exact subgraph matching is fundamental to numerous graph data applications. The inherent NP-completeness of subgraph isomorphism poses significant challenges in developing efficient matching algorithms. Current methods show limited success, particularly in their filtering and verification stages. Selecting an optimal vertex-searching order can greatly improve a subgraph searching algorithm's effectiveness, yet a comprehensive theory for this selection is lacking. In this paper, we introduce the Multistage Graph Search Model (MGSM), a novel approach addressing this gap. MGSM provides insights for identifying the most efficient vertex-searching order and offers a systematic framework for evaluating existing algorithms. Using MGSM, we identify two main challenges in optimizing the search order and present a new matching algorithm "Tps" noted for its strategic vertex-searching order. Extensive experiments demonstrate the superior performance of Tps and its effectiveness and soundness in optimizing search orders.

Keywords Exact subgraph matching · Subgraph query · Optimization · Query processing

1 Introduction

Graphs are widely used as data representations in various fields, denoted as G(V, E, L) for labeled graphs. Here, 'V' represents the set of vertices, typically interpreted as entities; 'E' denotes the edges, signifying relationships between entities; and 'L' comprises the labels, indicating attributes of entities. Graph data are notably applied in areas like information networks, exemplified by the semantic web [1], where they depict sets of entities and their interconnections. In bioinformatics, for instance, the protein–protein interaction

Yixing Ma
yxgma@ucdavis.edu

Hongfeng Yin
hfyin@czjtu.edu.cn

1 College of Letters and Science, University of California, Davis, USA

2 School of Computer and Information Technology, Beijing Jiaotong University, Beijing, China

3 School of Computer and Information Technology, Cangzhou Jiaotong College, Cangzhou, Hebei, China

(PPI) network [2] is effectively represented as a graph. Due to its capability to encapsulate complex structures and relationships, graph theory has found increasing application across numerous disciplines. A primary challenge in this realm is identifying a small query graph within a larger target graph or graph database. This task is inherently challenging, given the NP-completeness of subgraph isomorphism, which precludes any algorithm from ensuring a solution in polynomial time. As graph data grow in complexity and size, the development of efficient subgraph matching algorithms becomes increasingly crucial.

Most exact subgraph matching algorithms [3–9] employ a filtering-verification framework. In the filtering phase, a large portion of impractical matches are eliminated, yielding a more manageable set of candidates. The verification phase then conducts a subgraph isomorphism search on this set to identify exact matches for the query, typically using the Depth-First-Search (DFS) based Ullmann algorithm [10]. The exact subgraph matching problem can be categorized into two types: one involves finding all graphs in a database containing a specific query, and the other aims to locate all subgraphs within a large target graph isomorphic to a given query.

For the first type, graph indexing methods [3–5] are commonly used, quickly filtering out unsuitable target graphs, thereby allowing early matching techniques to focus on refining indexing strategies. However, for the second type, traditional index-based approaches become ineffective as they rely on substructures like paths or trees for indexing, leading to exponentially growing index sizes with larger target graphs, rendering them impractical [11]. In scenarios with small target graphs, these methods might directly apply the Ullmann algorithm without optimization in the verification stage. Yet, for larger target graphs, this verification process requires significant enhancements.

Responding to the growing scale of real-world graphs, several algorithms addressing the second type of problem have emerged [6–8]. Despite some successes, they still face challenges in terms of efficiency and robustness, as highlighted in [12]. Two main issues are evident:

1) While some filtering methods [6, 7] designed for single large graphs have been introduced, comprehensive performance analyses are lacking. Reports [12] indicate that these methods were less effective than QuickSI [8], which does not include any specific filter for large single graphs. This contrasts with the efficiency of index methods for the first type of problem, suggesting a disparity in filtering strategies for large target graph searches.
2) Improving the Ullmann algorithm necessitates an apt vertex-searching order for the query graph. Existing studies [6–9] often rely on intuitive justifications for their chosen orders, lacking a theoretical framework or solid rationale. While some algorithms show promising experimental results, the soundness of their reasoning is not always clear. Conflicting results from various algorithms, due to different experimental datasets, make drawing definitive conclusions challenging.

In this paper, our contributions are twofold: 1) We introduce the Multistage Graph Search Model (MGSM), a theoretical model to guide the selection of vertex-searching orders, a previously unaddressed issue, and define the optimal order for the Ullmann search. We reformulate the order selection problem into an optimization challenge. 2) Employing MGSM, we propose a pseudo-tree method to estimate the time cost of a given search order. Building on this, we develop a novel exact matching algorithm named “Tps”. In our experimental evaluation with large target graph and over 600 query graphs, Tps

demonstrates enhanced efficiency by outperforming TurboISO in benchmarks. Furthermore, Tps is grounded in a robust theoretical framework.

2 Related work

In the field of subgraph matching, initial studies predominantly addressed problems where the target set consisted of numerous small graphs, emphasizing swift filtering methods based on indexing. Shasha et al.'s GraphGrep [4] employed paths as indexing features but faced limitations as the number of paths increased with larger target graphs. Yan et al.'s gindex [3] improved upon this by using subgraphs as features, focusing on their frequency and discriminative power [21] for more effective indexing.

Subsequently, innovations like TreePi [4] and Tree+delta [5] emerged, striking a balance between path-based and subgraph-based methods by utilizing subtrees as features. These approaches, based on the premise that the pruning power of the subtree is more effective than that of paths yet less costly than full subgraphs, represented a significant advancement in the field [22]. Other noteworthy works include FG-index [5] and gcode [15], introducing innovative verification approaches and no-feature index techniques, respectively [23].

As researchers turned their attention to larger single target graphs, a second problem type emerged [24]. Ullmann's algorithm [10] was pioneering in this area, using a DFS strategy to search large target graphs for a sequence of induced subgraphs. This approach was further refined by GraphQL [6] and SPath [7], which introduced neighbor signatures as effective filters and proposed estimation functions to optimize the vertex-searching order [13].

More recent developments have seen algorithms like QuickSI [8] and TurboISO [9] demonstrate exceptional performance for this second problem type. QuickSI transformed the order selection problem into a minimal spanning tree problem, thus bypassing the need for explicit estimations of search space size [25]. TurboISO introduced an efficient path filter, further avoiding the need for explicit order selection estimations [26].

In contrast, our Multistage Graph Search Model (MGSM) and Tps algorithm present a novel approach to optimizing the vertex-searching order in subgraph matching. Unlike the previous methods which primarily focus on indexing techniques or employ path and subtree features for efficiency, our MGSM approach analyzes the selection of vertex-searching order through a multistage graph representation. This allows for a more nuanced and dynamic optimization of the search order compared to the static approaches of earlier methods [16, 17].

Furthermore, the Tps algorithm, developed based on the theoretical foundations of MGSM, prioritizes finding an optimal vertex-searching order for both filtering and verification steps in subgraph matching. This contrasts with algorithms like QuickSI and TurboISO, which either simplify the order selection problem or avoid explicit estimations altogether. Our approach not only addresses the efficiency of the matching process but also bridges the theoretical gap in order selection strategies, marking a significant advancement in the field [18].

Inexact subgraph matching, another significant area of research [13, 15, 19, 20], employs different similarity measures for matching. Algorithms such as Ness [15], SAGA [19], and SIGMA [20] in this domain have greatly inspired our work, particularly in the development of versatile and adaptable matching strategies.

3 Preliminaries

3.1 Subgraph matching problem

In graph representation, we denote a graph G as G(V, E, L), where V represents the vertex set, E is the edge set, and L is the label set associated with the vertices.

Definition 1. (Graph Isomorphism) For two graphs G1(V_1, E_1, L_1) and G_2(V_2, E_2, L_2), if they are isomorphic, then a bijective function f: $V_1 \rightarrow V_2$ exists such that: 1) for every edge(v_1, v_2)$\in E_1$, a corresponding edge($f(v_1)$, $f(v_2)$)$\in E_2$; and 2) for all vertices v, $L(v) = L(f(v))$. If G_1 is subgraph isomorphic to G_2, it implies that G_2 contains a subgraph G_3 isomorphic to G_1.

Definition 2. (Exact Subgraph Matching) Given a query graph Q and a target graph G, the objective of exact subgraph matching is to identify all subgraphs within G that are isomorphic to Q.

The exact subgraph matching problem, known for its NP-completeness, primarily employs a filtering-verification framework. The filtering phase eliminates infeasible matches, leading to a refined candidate set. The subsequent verification phase involves a subgraph isomorphism search on this candidate set to locate all embeddings of Q. Effective filtering techniques for large single target graphs include the neighbor signature filter, pseudo subgraph isomorphism search, and the path filter, as exemplified in TurboISO. The verification phase typically utilizes the Ullmann framework, characterized by a depth-first-search (DFS) process. A key challenge in optimizing this process is determining an effective vertex-search order.

3.2 Ullmann algorithm and vertex-searching order

To contextualize the Ullmann algorithm and vertex-searching order, we introduce:

Definition 3. (Induced Subgraph) Given a graph $Q = (V, E, L)$. Let V' be a subset of V, $E' = \{(v, u) \mid \text{all } v, u \in V'\}$, L' is the label set associated with V'. The subgraph $q_s = (V', E', L')$ is an induced subgraph of Q.

The Ullmann algorithm serves as a fundamental verification framework for the exact subgraph matching problem. It employs DFS to identify induced subgraphs q_1, $q_2 \ldots q_i$ of Q within G. During the DFS, vertices of Q are included in the search according to the vertex-searching order.

Figure 1 demonstrates the search process following the vertex-searching order of v_1-v_2-v_3-v_4. The Ullmann algorithm, using this order, examines a series of induced subgraphs. Considering the varying frequencies of subgraph q_i in G, an optimal vertex-searching order can reduce the number of subgraphs requiring inspection during DFS. This, in turn, decreases the number of recursive calls, making the identification of an effective vertex-searching order a crucial factor in enhancing the efficiency of Ullmann-based search algorithms.

To establish an efficient vertex-searching order, algorithms like GraphQL and SPath have developed estimation functions to evaluate proposed orders. QuickSI utilizes the

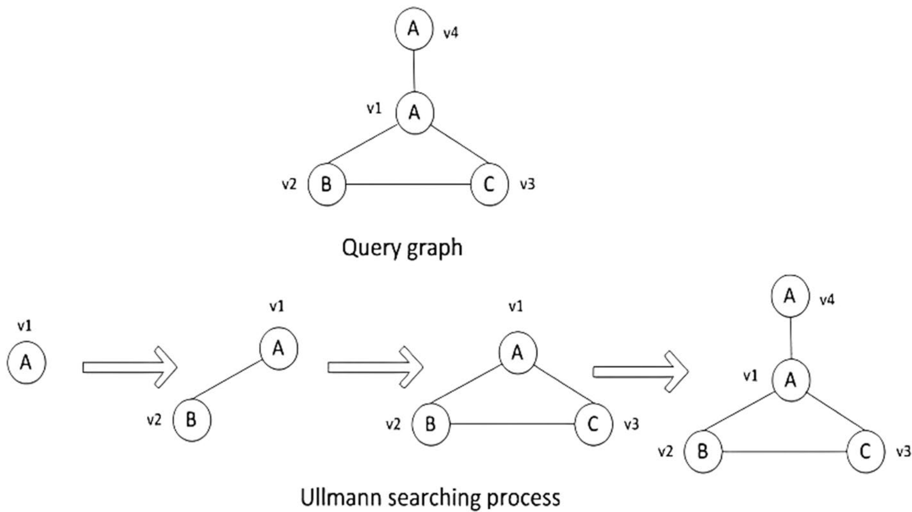


Fig. 1 Example of Ullmann search process

frequency of query edges as weight, determining the vertex-searching order through the Prim algorithm. Conversely, TurboISO sorts the paths of the spanning tree based on the frequency of leaves. The next section will delve into identifying the most effective vertex search order.

4 Multistage-graph search model

In this section, we introduce the **M**ultistage-**G**raph **S**earch **M**odel (MGSM) to analysis the problem of selecting vertex-searching order.

Definition 4. (Multistage graph) A multistage graph M can be represented by a directed graph M (s, t, V, D, W). V is the set of vertices. D is the set of directed edges. W is the set of weights associated with vertices. s is the source vertex, and t is the terminal vertex. Vertices are divided into several stages. Given a directed edge like (u, v), stage(v) = stage(u) + 1.

Definition 5. (MGSM) Given a query $Q = (V_1, E_1, L_1)$ and a target graph $G = (V_2, E_2, L_2)$. Using a new vertex u to denote an induced subgraph q_s of Q, and let U be the set of u for all induced subgraph of Q. Let $D = \{$directed edge $(u_1, u_2) | u_1, u_2 \in U$, $q(u_1)$ is an induced subgraph of $q(u_2)$ that has one more vertex than $q(u_1)\}$. Let $w(u) = |q(u)|$. $|q(u)|$ denotes the frequency, i.e., the number of matches of $q(u)$ on G. $W = \{w(u) \mid u \in U\}$. The source vertex u_0 denotes the empty set, terminal u_t denotes Q itself. Now we have a multistage graph M (u_0, u_t, U, D, W). To reflect the relationship between M, Q and G, we rewrite it as M (u_0, u_t, U, D, $W \mid Q$, G), and we call M the MGSM of Q and G.

Figure 2 shows an example of induced subgraphs and MGSM. We can see that a vertex-searching order corresponds to a path on the multistage graph. Given an order v_1-v_2-v_3-v_4, the sequence of induced subgraphs been tested is $q_{1.1}$, $q_{2.1}$, $q_{3.1}$, q_4. So, the corresponding path is u_0-*1.1–2.1–3.1*-u_t.

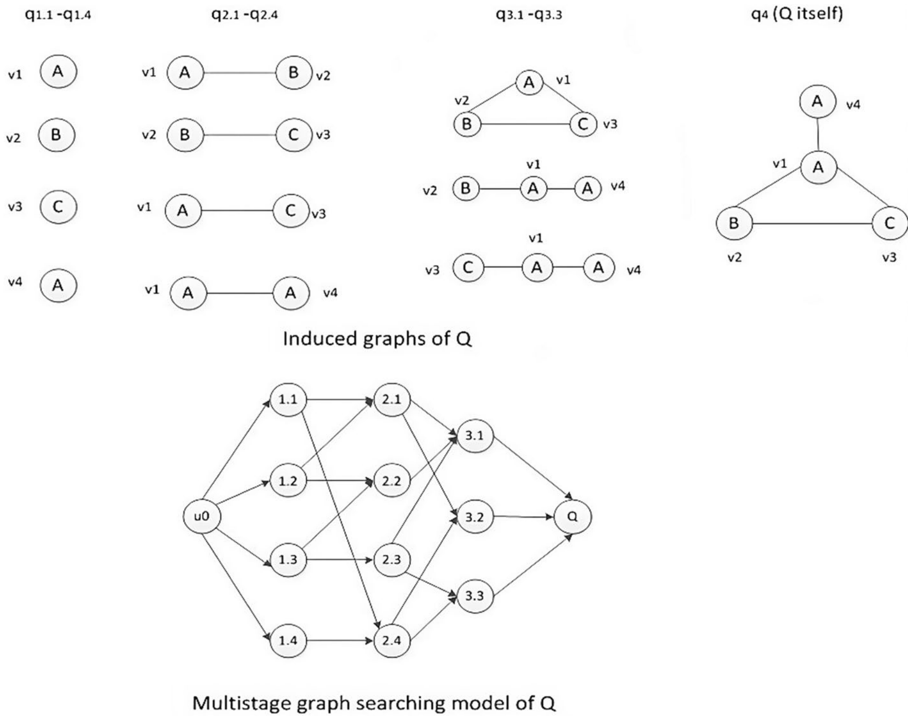


Fig. 2 MGSM example

Theorem 1 *Given query Q, target G and M (u_0, u_t, U, D, W | Q, G). The best vertex-searching order is the minimum weight path on M, from u_0 to u_t.*

Proof The weight of vertex on M denotes the frequency of induced subgraph of Q. So, the minimum weight path on M means if the Ullmann algorithm follows this search order, it tests the least induced subgraphs. So, the number of recursions calls of Ullmann algorithm is minimized.

4.1 ES-feature

Based on Theorem 1, the optimization of the vertex-searching order can be reformulated as follows:

Problem 1 Given query graph Q (V_1, E_1, L_1), target graph G (V_2, E_2, L_2) and M (u_0, u_t, U, D, W | Q, G), find a path p from u_0 to u_t on M, which minimizes.

$$\sum_{u_i \in P} W\left(u_i\right)$$

$w(u)$ denotes the weight of u, which is equal to the frequency of induced subgraph represented by u. Obviously, it is extremely hard to solve this problem directly using methods

such as dynamic programming, because according to the binominal theory, $|U|$ is exponential based on $|V_1|$, besides, $w(u)$ is usually unknown.

However, if the precise frequencies of certain substructures such as edges or paths are pre-calculated, these could assist in estimating w(u). These pre-counted substructures are referred to as es-features.

For instance, GraphQL and SPath can determine the frequency of each query vertex using neighbor signature filtering, hence their es-feature is the vertex. QuickSI obtains the frequency of each query edge through indexing, hence its es-feature is the edge. TurboISO, through path filtering, can acquire both the frequency of vertices and paths.

It's straightforward to deduce that the greater the number of edges an es-feature encompasses, the more accurate the estimation. Being significantly larger than a vertex or edge, the path as an es-feature is one reason why TurboISO outperforms others. On the contrary, the pseudo subgraph isomorphism test, acting as a global filter, cannot offer large es-features comparable to the path filter. The filtering step, therefore, not only reduces the candidate set but also provides prior information for the optimization of the verification step.

4.2 Estimation problem of induced subgraph

Given a vertex u on M (u_0, u_t, U, D, W | Q, G), the induced subgraph $q(u)$ usually contains more than two es-features. So, we must consider how to estimate $w(u)$ by frequencies of es-features. According to the estimating approach, recent algorithms can be divided into two groups:

1) The first group includes GraphQL and SPath. They have designed estimation function $E(u)$ to estimate w(u). Since they choose vertex as es-feature, $E(u)$ is proportional to $\prod|C(v_q)|$. $C(v_q)$ is the candidate set of query vertex v_q. So, the order optimization problem (1) are converted into an approximate form.

Problem 2 Given a query graph Q, a target graph G and M (u_0, u_t, U, D, W |Q, G), find a u_0- u_t path p on M, which minimizes.

$$\sum_{u \in P} E(u)$$

This kind of estimation function has three shortages: (1) It uses vertex as es-feature, whose structural complexity is far less than subgraph. (2) If an induced subgraph q_s has more than one es-feature, $E(q_s)$ is obviously unrealistic. Forexample, suppose q_s can be divided into two segments q_1, q_2. Even if $E(q_1)$ and $E(q_2)$ are exactly equal to $|q_1|$ and $|q_2|$, $E(q_s)$ will be proportional to $E(q_1)$ *$E(q_2)$, which can hardly reflect the true value of $|q_s|$. (3) The optimization problem deduced from this estimating function cannot be solved in polynomial time.

2) QuickSI and TurboISO belong to the second group, which claims no need to estimate the $w(u)$. But, the second group uses greedy strategy to get vertex-search-order. However, we have found that both have similar approach to estimate $w(u)$ implicitly. They just take the frequency of a most frequent es-feature, such as most frequent edge or path, as

the approximate value of $w(u)$. So, the order optimization problem (1) is converted into an approximate form.

Problem 3 Given a query graph Q, a target graph G and M (u_0, u_t, U, D, W |Q, G), find a u_0- u_t path p on M, which minimize.

$$\sum_{u \in P} maxF(u)$$

max$F(u)$ denotes the largest frequency of es-feature contained in u. The following examples show how the two algorithms, QuickSI and TurboISO, can be derived from similar ways.

Example 1 Consider how to select a spanning tree T on Q and take the edge set of T as es-feature set. It means that given an induced subgraph $q(u)$ and the most frequent edge $e_f \in Edge(q(u)) \cap Edge(T)$, we use the frequency $|e_f|$ upon target graph to be the value of $w(u)$. Then problem (1) naturally converts into minimal spanning tree problem. So, we get the basic order-selection strategy of QuickSI.

Example 2 Given a spanning tree T on Q, we take the *r-l* path (Given a spanning tree of a query graph, an r-l path is a path from root to a leaf.) on T as es-feature. It means that we use the most frequent *r-l* path p_f within $q(u)$ to estimate $w(u)$. Then problem (3) can be solved by sorting all *r-l* paths based on their frequency. Now we get the basic order-selection strategy of TurboISO.

QuickSI and TurboISO use most frequent es-feature to estimate $|q_s|$, and it is true that their estimations may be very accurate. The $|E(q_s)$-$|q_s||$ can be much smaller. However, $|E(q_s)$-$|q_s||$ does not determine whether an estimation is reasonable or not. Imagine that if we multiply $E(q_s)$ with a large number such as 2^{10}, $|E(q_s)$-$|q_s||$ will increased dramatically. However, it has no influence on the selection of vertex-searching order. When q_s contains more than one es-feature, this estimation will overlook the complexity of graph.

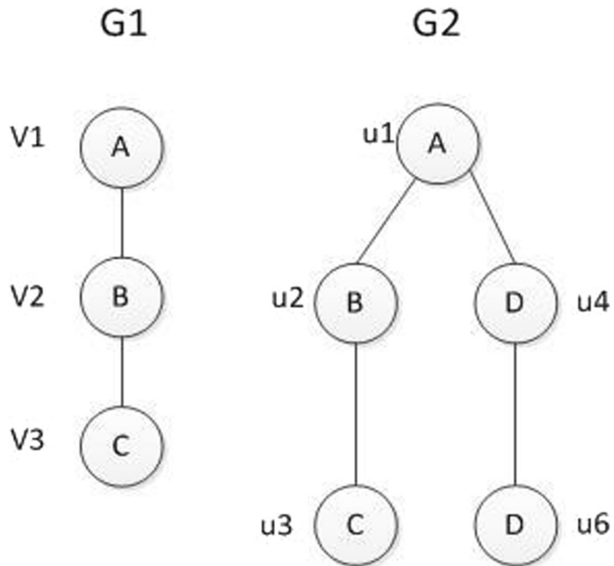
For example, as shown in Fig. 3, G_1 is a path with frequency of 100. G_2 is a tree having two *r-l* paths each with a frequency of 50 and $|u_1|=1$. The most frequent path estimation of G_1 is 100 and the estimation of G_2 is 50. But the real frequency of G_2 is 2500 which is much more than G_1. To solve this problem, this estimation must choose more complex structure to be es-feature, for example, a tree. However, mining tree will increase the cost of filtering.

5 Tps algorithm

In this section, we introduce Tps, a novel subgraph matching algorithm that emphasizes the importance of identifying an optimal vertex-searching order. Drawing from our previous analysis, Tps seeks to address two key questions:

1) **Feature Selection for es-features:** What attributes should be selected as es-features, and how can we ascertain their frequency? We suggest that TurboISO's approach, which

Fig. 3 Irrationality of most frequent path estimation



advocates for paths as a suitable choice, provides a solid answer. This concept will be further elaborated in Section 4.1.

2) **Frequency Estimation of Induced Subgraphs:** How can we estimate the frequency of each induced subgraph of a query based on the selected es-feature? Our approach tackles this in two stages: During the filtering phase, we calculate the approximate frequency of each subtree using the exact frequency of each path, as detailed in Section 4.2. Post-filtering, we estimate the frequencies of certain induced subgraphs based on the exact frequencies of paths and the approximate frequencies of subtrees, discussed in Section 4.3.

5.1 Path-feature filtering

Traditional index-based filtering methods, such as the G-index, become ineffective with large target graphs due to the exponential increase in index size. Recently, non-index filtering approaches, like the pseudo subgraph isomorphism test, have been proposed to reduce the candidate set size. However, these methods often overlook that the filtering step also provides critical information for determining an effective search order in the verification phase.

Additionally, these approaches can be time-consuming, sometimes yielding only a modestly reduced candidate set. For instance, while GraphQL can attain the smallest candidate set in certain scenarios using a pseudo subgraph isomorphism test, its verification phase often takes longer than other algorithms like QuickSI. Furthermore, since the pseudo subgraph isomorphism test primarily yields the exact frequency of query vertices, it falls short in estimating the frequency of induced subgraphs. Its filtering runtime often exceeds that of other methods, as the pseudo subgraph isomorphism test can be more complex than the actual subgraph isomorphism verification process.

Therefore, an effective filtering method must meet three criteria: Firstly, it should significantly reduce the candidate set, thereby narrowing the verification step's search

space. Secondly, it should aid in determining frequencies for larger subgraphs, facilitating the development of an efficient vertex-searching order for verification. Lastly, it should be executable within a reasonable timeframe. To achieve these objectives, we have devised the following filtering strategy.

Filtering ruler 1 Given a spanning tree *T* of query graph *Q*, we execute depth-first search (DFS) to find all mapping functions f: $V_Q \rightarrow V_G$ that satisfies: for each leaf vertex v_{leaf} on *T*, the path v_{root}-v_1-v_2-…v_{leaf} is isomorphic to $f(v_{root})$-$f(v_1)$-$f(v_2)$-…$f(v_{leaf})$. Then $f(v)$ will be add into v's candidate set $C(v)$.

During the DFS, we adopt an additional filtering ruler to speed up this searching process and reduce the size of $C(v)$.

Filtering ruler 2 Let $NL(v)$ denotes the label set of v's neighbors. If $u \in C(v)$, v then u must satisfy: $label(v) = label(u)$, degree$(v) \leq degree(u)$ and $NL(v) \subset NL(u)$.

We combine these two rulers as a filtering method to get refined candidate set $C(v)$ for each query vertex, and then store $C(v)$ in a tree structure. Given a query vertex v and its parent v_p on *T*, and a target vertex $u \in C(v_p)$, we use a node structure $cNode(v, u)$ to store a candidate match set $\{u_v \in C(v) \cap neighbor(u)\}$. For the target graph *G* and a spanning tree *T* shown in Fig. 4, all *cNode* (v, u) can make up a tree structure *cTree* shown in Fig. 5. *cTree* is built along with DFS filtering.

From the common view of filtering method, ruler 2 is a local filter, i.e., it only considers the local information (such as vertex label, degree, neighbor label). And ruler 1 is a global filter, since it takes all the vertices and paths on *T* into consideration. This combination can effectively reduce the size of candidate set.

Form the view of MGSM, ruler 1 implies that we choose the root-leaf paths on *T* as es-features. This is because searching for mapping functions which satisfy ruler 1 needs to execute subgraph isomorphism searching for these paths. So, the outputs of this filtering method include not only a refined candidate set *cTree*, but also the exact frequency of each root-leaf path of *T*. One of the reasons we consider path as es-feature is because the filtering procedure can be time-efficient.

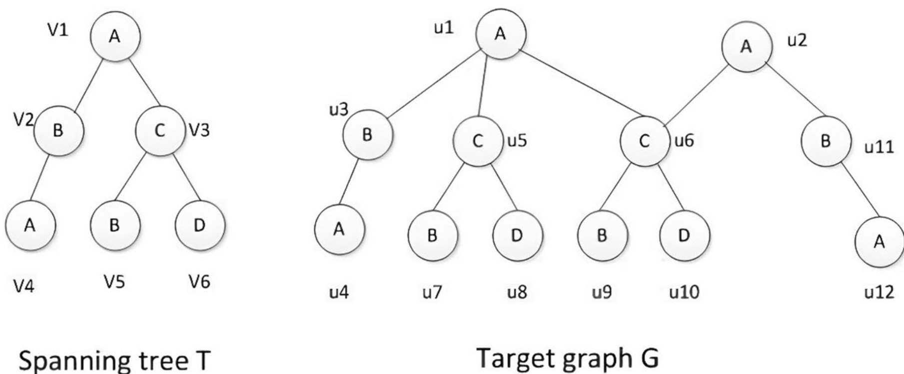


Fig. 4 Example about spanning tree and target graph

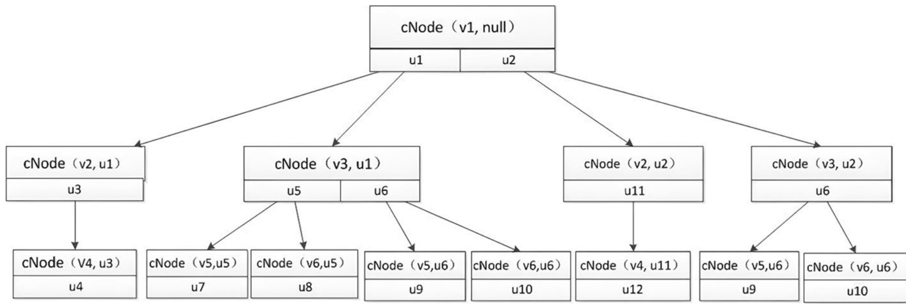


Fig. 5 The example of cTree (candidate set)

Theorem 2 *Searching for mapping functions satisfy ruler 1 can be done in polynomial time*.

Proof Given a path p, we use $F_{iso}(p)$ to denote its isomorphism embedding set. For a mapping function f_i satisfies ruler 1, $\cup f_i(p)$ is only a subset of $F_{iso}(p)$. It is already known that subgraph isomorphism search for path query can be done in polynomial time and each embedding path $f_i(p)$ is scanned once because f_i is not injective. Thus, our filtering step can also be done in polynomial time.

If we take more complex substructures such as subtree or subgraph as es-feature, Theorem 2 does not apply and the filtering will take a long time. It is interesting to know that if f_i is changed to be injective, the ruler 1 is equal to searching the exact matches of whole T. Maybe we can use the exact frequencies of paths to estimate the frequency of subtree.

5.2 Pseudo-tree estimation

We now try to guess the approximate frequency of some subtrees of Spanning tree T. The kind of subtree we are interested in is defined as:

Definition 6 (full subtree): Given a tree T and a vertex v on T, the full subtree $st(v)$ is the subtree composed by v and all v's descendants.

Given a query vertex v and a target graph vertex u, the term $e_a(v|u)$ denotes the estimated local frequency of $st(v)$, when v is matched to u. If v is a leaf, $e_a(v|u)$ is set to be 1. Otherwise, $e_a(v|u)$ is calculated as Eq. (1) and Eq. (2). The term $e_t(v|u)$ denotes the estimated local frequency of $st(v)$, when v' parent is matched to u.

$$e_a(v|u)\prod_i e_t\left(v_i|u\right), v_i \epsilon\ child(v) \tag{1}$$

$$\mathrm{e}_t\left(v_i|u\right) = \sum_j e_a\left(v_i|u_j\right),\ u_j \in c\ Node\left(v_i, u\right) \tag{2}$$

Given a target graph G, a query graph Q and its spanning tree T, we estimate the global frequency, on G, of every T's full subtree $st(v)$ by.

$$E_p(st(v)) = \sum_k e_a\left(v|u_k\right),\ u_k \in C(v) \tag{3}$$

This estimating approach is called pseudo-tree estimation, because of the following Theorem:

Theorem 3 *Given a tree t and a graph G, if any two vertices v_1 and v_2 on t without ancestor–descendant relationship have different labels, $E_p(t)$ is equal to the exact number of t 's embedding of G, which have been refined by both of the two Filtering rulers in Section 4.1.*

Proof According to Filtering ruler 1, for any two query vertices v_1 and its descendant v_2, the path v_1-v_2 is isomorphic to $f(v_1)$-$f(v_2)$. Then for a query vertex v_3 that doesn't have ancestor–descendant relationship and same label with v_1, we have $f(v_1) \neq f(v_3)$. So, if any two vertices on tree t without ancestor–descendant relationship have different labels, the tree $f(t)$ is isomorphic to t. As a further deduction from Eq. 1, $e_{at}(u\text{p})$ equals to the exact number of embedding of $st(v)$ on target graph when $f(v) = u$. Obviously, if $e_a(v|u)$ is exact, $E_p(st(v))$ must be exact too.

Although pseudo-tree estimation is based on the output of path filtering, it can be performed during path filtering instead of waiting for the filtering step to be finished. The algorithm 1 shows that the DFS procedure combines path filtering with pseudo-tree estimation. Suppose that there are two query vertices, v and v's child v_i, and two target vertices, $u = f(v)$ and u 's neighbor u_j. When u_j is added into *cNode* (v_i, u), $e_t(v_i|u)$ will be updated by Eq. (2) (line 11). When *cNode* (v_i, u) is appended to *cTree*, $E_p(st(v_i))$ and $e_a(v|u)$ will be updated (line 17 and 18). If all neighborhoods of u cannot match v_i, then the candidate matches stored in the subtree of *cNode* (v_i, u) are all false matches (line 12). In this case, a function *clean* will be called to cut the subtree, i.e., delete all the false matches and undo the update of $E_p(st(v_k))$ for each v_k (line 13), where v_k is a descendant of v_i.

Since the pseudo-tree estimation does not add any recursion calls to DFS, the filtering step can still be done in polynomial time.

```
1 If u and v satisfy the Filtering ruler 2
2  u.visited=1;
3  For each vi∈v's children
5    Create cNode(vi, u);
6    matched=0;
7    For each uj∈u's neighbor and is uj.visited=0;
8      If filtering (vi, uj, cNode(vi, u) )==false
9        continue;
10       matched =1;
11     et(vi | u)+= ea(vi | uj);
12   If matched=0
13     Clean(v, u);
14     Return false;
16   append cNode(vi, u) under cNode(v, u');  // a new embedding of path vroot-vi is found.
17   |C(vi)|++;  // an approximate embedding of tree st(vi) is found.
18   Ep(st(vi)) +=et(vi | u);
19   ea(v | u)*=et(vi | u);
20  u.visited=0;
21  Add u to cNode(v, u')
22  Return true;
23 Return false;
```

Algorithm 1 filtering (query vertex v, target vertex u, cNode(v, u')).

```
1 O_v =empty set;
2 Adjust(T.root);
3 Get O_v by depth-first walk on T;
4 Return O_v;
```

Algorithm 2 order_selection(T).

```
1 For each v_i∈ v's children
2   Adjust(v_i);
3 For each possible permutation of v's children
4   If the current children order can minimize E(st(v), O_c)
5     return ;
```

Algorithm 3 Adjust(vertex v).

5.3 Order selection

To find a good vertex-searching order for verification step, we first constrain the order selection within the scope of depth-first-walk order of spanning tree T. Setting this limited scope can help us make reasonable choice, since in this way we can make use of the frequencies of paths and subtrees outputted by filtering step.

By adjusting the visiting order of full subtrees of T, we can have different depth-first-walk orders. As suggested by MGSM, a given searching order corresponds to a certain sequence of induced subgraphs. To estimate the frequencies of these subgraphs, they will be categorized into two groups.

1) The induced subgraph g contains only root-leaf paths but no full subtree. Suppose that the most frequent path is p_m (denote its frequency as $freq(p)$), We estimate the approximate frequency of g as.

$$E(g) = freq\left(p_m\right) \tag{4}$$

2) The induced subgraph contains full subtrees. Suppose the most frequent path is p_m, and the most frequent (approximately) full subtree is t_m. We estimate the approximate frequency of g as.

$$E(g) = max\left(freq\left(p_m\right), E_p\left(t_m\right)\right) \tag{5}$$

Based on pseudo-tree estimation, the optimization problem 1 is converted into the following.

Problem 4 Given a query graph Q, a target graph G, Q's spanning tree T, find a depth-first-walk order O_v, which minimizes.

$$cost\left(T, O_v\right) = \sum_i E\left(g_i\right),\ i \in \left\{1, 2, \dots, \left|O_v\right|\right\} \tag{6}$$

where g_i is the induced graph composed by first i vertices of O_v.

This problem can be exactly solved by algorithm 2 and 3. First, for T's root and each of its child v_c, adjust the visiting order within $st(v_c)$ to get a depth-first-walk order O_c that minimizes $E(st(v_c), O_c)$ (algorithm 2 line 2, algorithm 3 line 1). Then numerate all

Algorithm 4 verification(query vertex v, candidate set C, vertex-searching order Ov, current match f, match set M).

```
1 For u∈C(v) and u.matched==0
2  If joint-test(v, u, f)=1
3    Add (v, u) to f;
4    u.matched=1;
5    If (Ov.next=null)
6      add f to M;
7      Remove (v, u) from f;
8      u.matched=0;
9      Continue;
10   v′=Ov.next;
11   C(v′)=cNode(v′, f (parent of v′));
12   verification(v′, C(v′), Ov, f, M );
13   Remove (v, u) from f;
14   u.matched=0;
15 return ;
```

Algorithm 5 Tps(query Q, target graph G).

```
1 Get spanning tree T;
2 v0=T's root;
3 for each v0's candidate match u
4   build cTree by filtering(v0, u, cNode(v, null));
5 Ov=order_selection(T);
6 Verification(v0, cNode(v, null), Ov, ϕ, ϕ );// ϕ means empty set.
```

visiting sequence of these full subtrees to find the sequence that minimizes $E(T, O_v)$ (algorithm 3 line 3), so the whole spanning tree is adjusted. At last, the vertex-searching order O_v is achieved by DFW on T (algorithm 2 line 3). But in experiment, we implement an approximate version based on greedy strategy with lower cost: perform DFW on T directly. At each recursion step, choose the unvisited child with largest $E(g)$ to perform the next recursion step. The visiting order O_v is the vertex-searching order we want. This approximate way can usually get the same result as the exact one.

5.4 Subgraph matching

The verification step is based on Ullmann framework. As shown in algorithm 4, DFS verification on candidate set is executed, and the searching order is the output of our order selection method. The difference between the verification procedure and naive Ullmann algorithm is that during each recursion step, we do not need to test the whole $C(v)$, but the $cNode(v, f$ (parent of v)), a subset of $C(v)$ at the local region around the current matches of v's parent (algorithm 5, line 12).

Based on these, we proposed a subgraph matching algorithm called Tps. As shown in algorithm 5, Tps includes three step: Step 1, convert query graph Q into spanning tree T (algorithm 5 line 1). Step 2, perform path filtering to achieve refined candidate set cTree and perform pseudo-tree estimation to achieve vertex-searching order O_v (algorithm 5 line 2–5). Step 3, perform Ullmann algorithm based on O_v and find all embedding of Q (algorithm 5 line 6).

6 Optimization of filtering step

Comparing Ullmann algorithm and algorithm 1, it is easy to see Ullmann algorithm and path filtering have one thing in common: both are based on DFS. So, it is reasonable to assume that a good vertex-searching order can also reduce the number of times for recursive calls of filtering. We have already given the method for optimizing the searching order of Ullmann algorithm, now let's address the same question for path filtering.

To find a good vertex-searching order for filtering is equal to generate a good spanning tree and determine the visiting order of subtrees. Unlike pseudo-tree estimation, we have little prior information such as frequency of path. So, the main approach is first to index the target vertices with some pruning conditions. Then, by searching on index, we can get the initial candidate set $C_{initial}(v)$ for each query vertex v. $|C_{initial}(v)|$ will be chosen as es-feature, and we use most frequent es-feature to estimate the frequency of induced subgraph. Then, convert the optimization problem 2 into corresponding form. The spanning tree T will be achieved by solving the optimization problem. At last, perform depth-first walk on T to get the vertex-searching order for filtering step.

6.1 Index

If u is a feasible match of v, it is easy to draw a simple necessary condition: label(v) = label(u), degree(v)≦degree(u). If v and u satisfy this condition, we add u to the initial candidate set $C_{initial}(v)$. It is easy to index all target vertices according to their labels and degrees. The space complexity of this index is $O(n)$, which is much more compact than the feature-based indexes.

Figure 6 shows the structure of our three-level index. The third level of index has three fields: *vertex label*, *degree threshold*, and *amount*. The *amount* stores the total number of vertices who share the same label and whose degrees are less than *degree threshold*. Given a query vertex v, $|C_{initial}(v)|$ is equal to the value of *amount* in the index entry whose *vertex label* is label(v) and *degree threshold* is equal to *degree*(v).

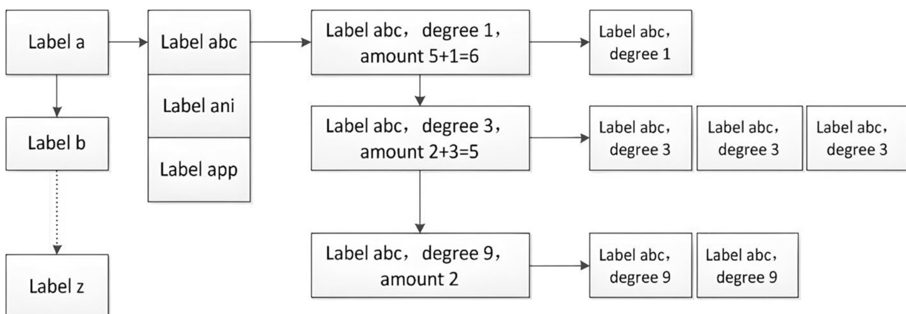


Fig. 6 Three-level index

```
1 Generating a minimum spanning tree T by Prim algorithm;
2 Adjust(T. root);
3 Get O_f by depth-first walk on T
4 Return O_f;
```

Algorithm 6 Spanning_tree_generation(query Q).

```
1 For v_c ∈ v's children
2   Adjust(v_c);// minimize SumV(st(v_c))
3 For each permutation of v's children
4   if the current children order can minimize SumV(st(v));
5    return
```

Algorithm 7 Adjust(vertex v).

6.2 Spanning tree generation

Choosing $|C_{initial}(v)|$ to be es-feature and adopting most frequent feature estimation, we get a new form of problem 2.

Problem 5 Given a query Q, find a spanning tree T and determine its subtree visiting order, minimize:

$$sumV(T) = \sum_i maxV(i) \tag{7}$$

$maxV(i)$ is the largest $|Cinitial(v)|$ of the first i vertices during the depth-first walk. In fact, to find such T exactly is very costly. We propose an approximate solution: if we have already get T, it is easy to adjust the visiting order of the subtrees of T in order to minimize $sumV(T)$.

As shown in algorithm 6, we firstly find a minimal spanning tree by using Prim algorithm (algorithm 6 line 1). Then for each root's children v_c, adjust the visiting order within full subtrees $st(v_c)$ recursively to minimize $sumV(st(v_c))$ (algorithm 7 line 1). After that, enumerate all the permutation of the subtrees to get the best subtree-visiting order which minimizes $sumV(T)$ (algorithm 7 line 3). At last, perform DFW on T to get the vertex-searching order O_f for filtering. Algorithm 7 is very similar with algorithm 4 (adjust function of pseudo-tree estimation), the only difference is that the optimization target is replaced by $sumV(T)$ since we adopt a different estimation function.

7 Experiments

In this section, we present experimental results obtained from a large synthetic graph to demonstrate the effectiveness of the Tps algorithm. For comparative analysis, we also include results from TurboISO in our tests. Note that the "combine and permute" strategy typically employed in TurboISO is removed for three primary reasons: 1) This strategy shows significant benefits mainly for specific types of query graphs, such as clique queries, which do not represent our general graph queries. 2) In scenarios where this strategy is

effective, Tps can be adapted to function similarly to TurboISO, thus not markedly affecting the comparative outcomes. 3) Our research primarily focuses on optimizing vertex-searching order, a factor more influential to query algorithm performance than other optimization methods. Discussions on alternative optimization approaches are reserved for future studies.

7.1 Comparison for filtering step

Our target graph encompasses 3,000 vertices and 40,000 edges, generated using the method adopted by GraphQL [14]. This involves creating 'n' vertices and 'm' edges with randomly selected end vertices, with each vertex assigned a label. To ensure fairness in comparison, every query was designed to have as many embeddings as possible, with the label set comprising 40 distinct labels. The query set was generated following the procedure proposed by [6]. We began with a small 5-edge graph as the seed and randomly added edges, resulting in a 6-edge query set with each query having over 50 matches. We continued this process up to 11-edge queries, resulting in a set of 643 queries.

7.2 Comparison for filtering step

Figure 7 and 8 illustrate the number of recursions calls and the running time for the filtering step, respectively. Although both Tps and TurboISO implement path filtering, Tps optimizes the vertex-searching order within this process as outlined in Section 5. In contrast, TurboISO randomly selects the spanning tree and its corresponding order. Figure 7 shows that Tps consistently requires fewer recursion calls than TurboISO, indicating the influence of optimized vertex-searching order in filtering, albeit less pronounced than in the verification phase. This is partly due to limitations in estimating the most frequent es-feature, especially when the es-feature is overly simplistic (e.g., a vertex). Figure 8 reveals that the filtering running times of both algorithms are similar, attributable to the marginal

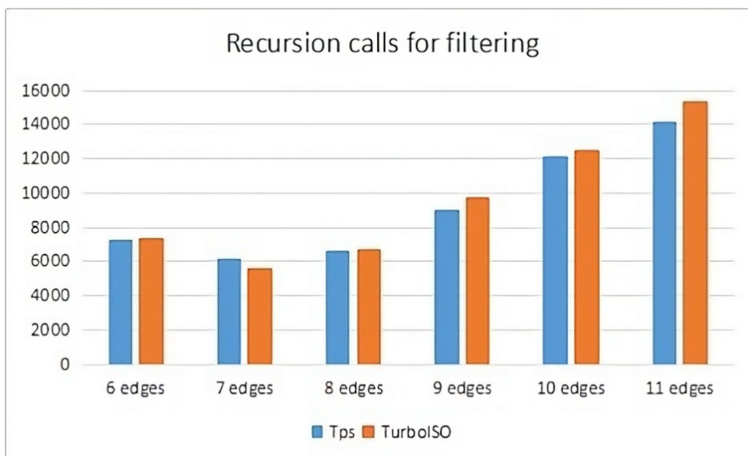


Fig. 7 Recursion calls for filtering step

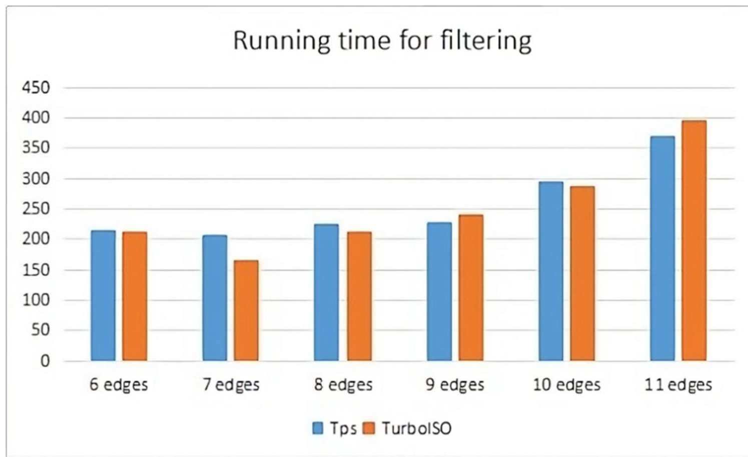


Fig. 8 Running time for filtering step

improvement in vertex-searching order and the additional time incurred by Tps for pseudo-tree estimation during path filtering.

7.3 Comparison for verification step

Our comparative analysis of the subgraph isomorphism verification step for Tps and TurboISO focused on two aspects: query graph size and frequency. Figure 9 and 10 show the average number of recursions calls and running times for verification, respectively. The performance disparity between the two algorithms is minimal for queries with less than 7 edges. However, as the query size increases, the difference becomes more significant. For queries exceeding 10 edges, Tps's recursion calls are only about one-third of those

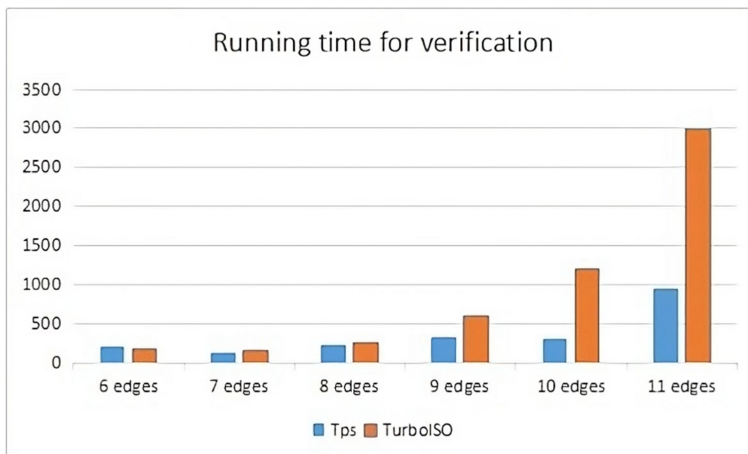


Fig. 9 Running time for verification step (view of query size)

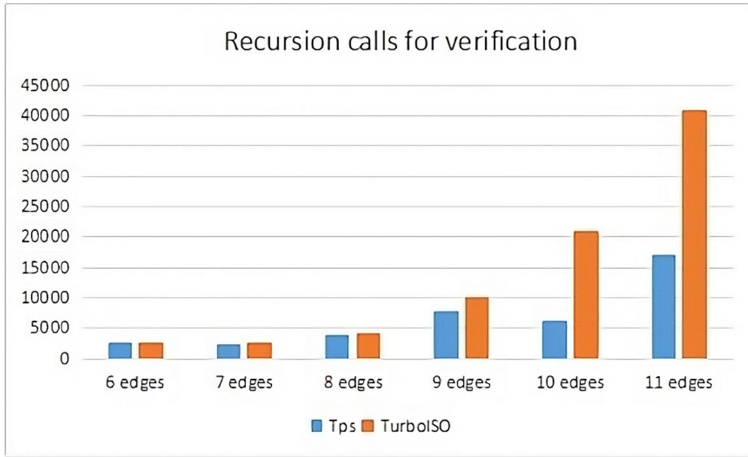


Fig. 10 Recursion calls for verification step (view of query size)

in TurboISO, demonstrating the effectiveness of pseudo-tree estimation over the most frequent es-feature. Similarly, Tps's average running time outperforms TurboISO by up to four times.

Figures 11, 12, 13 and 14 present the results segmented by query frequency. Queries with over 1,000 matches were classified into the high-hit group, comprising 335 queries, and further divided into four subsets with step sizes of 2,000. The remaining queries fell into the low-hit group, divided into five subsets with 200 step sizes. For queries with frequencies below 800, Tps's average running time closely matches that of TurboISO, yet with significantly fewer recursion calls. The performance gap widens with increasing query frequency. Notably, for frequencies above 7,000, Tps outperforms TurboISO by nearly a factor of 5.

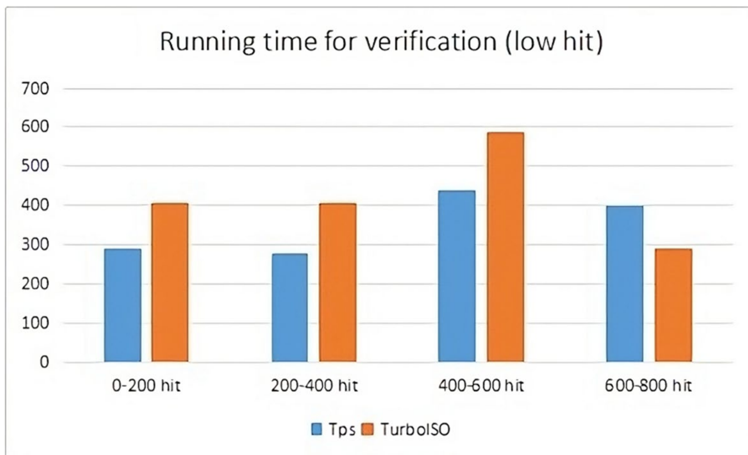


Fig. 11 Running time for verification step (low hit)

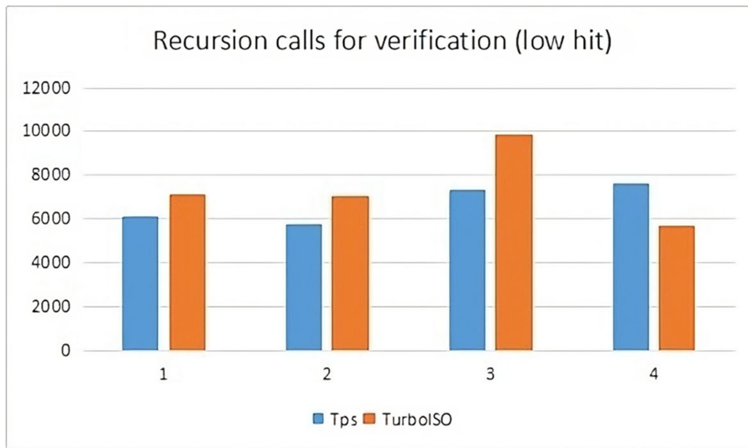


Fig. 12 Recursion calls for verification step (low hit)

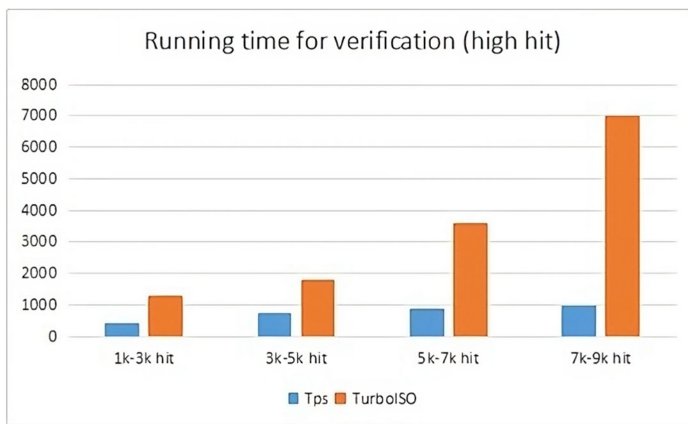


Fig. 13 Running time for verification step (high hit)

8 Conclusions

In this study, we present the Multistage Graph Search Model (MGSM), a novel theoretical framework for analyzing vertex-search order selection in graph matching. Our findings reveal that the optimal search order aligns with the minimum weight path in a multistage graph. Leveraging this insight, we developed Tps, an innovative matching algorithm that effectively optimizes search orders for both filtering and verification phases. A key aspect of our approach is the introduction of the pseudo-tree estimation method which improves the order selection process. The primary contributions of this work include the development of the Tps algorithm and the establishment of a theoretical foundation for order selection in graph matching.

Looking forward, our research aims to further refine these findings. We plan to construct a more comprehensive query set for balanced comparative analysis. Additionally,

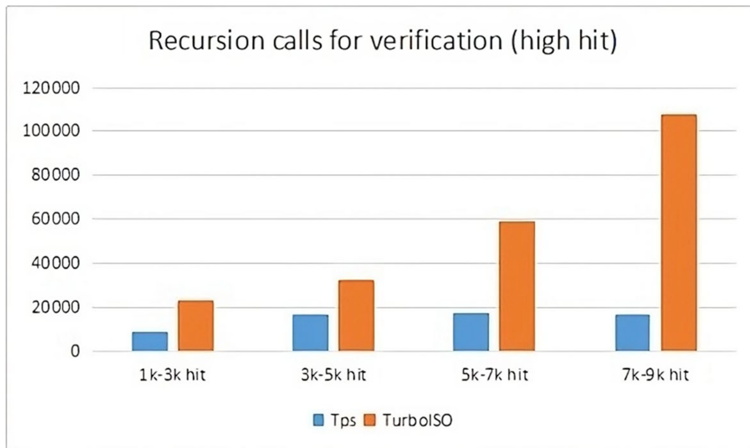


Fig. 14 Recursion calls for verification step (high hit)

we intend to conduct extensive comparisons with existing algorithms, not only to assess overall performance but also to evaluate their order selection methodologies and estimation techniques. We hope future work will advance our understanding of graph matching algorithms and their practical applications.

Acknowledgements This research was funded by Science Research Project of Hebei Education Department [QN2023256]. I would like to thank Xin Dai for his contribution on this paper.

Data availability Data sharing is not applicable to this article as no new data were created or analyzed in this study.

Declarations

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

1. Gandon F (2018) A survey of the first 20 years of research on semantic Web and linked data. Ingenierie des Systemes d'Information 23(3–4):11–56
2. Xin H , Xuejun C (2012) A Visualize Method for Protein-Protein Interaction Network Based on Extended Clique. Bulletin of Science and Technology
3. Houbraken Maarten et al (2013) The Index-Based Subgraph Matching Algorithm with General Symmetries (ISMAGS): Exploiting Symmetry for Faster Subgraph Enumeration. Plos One 9(5):e97896–e97896
4. Xu, Xiang, Wang, Xiaofang, Kitani, Kris M. (2018) Error Correction Maximization for Deep Image Hashing. British Machine Vision Conference (BMVC)
5. Coming DS, Staadt OG (2008) Velocity-Aligned Discrete Oriented Polytopes for Dynamic Collision Detection. IEEE Trans Visualization and Computer Graphics 14(1):1–12. https://doi.org/10.1109/TVCG.2007.70405
6. Wickramaarachchi, Charith, et al. (2016) Distributed Exact Subgraph Matching in Small Diameter Dynamic Graphs. IEEE International Conference on Big Data
7. H. Goto, Y. Hasegawa, and M. Tanaka. (2007) Efficient Scheduling Focusing on the Duality of MPL Representation. Proc. IEEE Symp. Computational Intelligence in Scheduling 57–64. https://doi.org/10.1109/SCIS.2007.367670

8. Xiang Xu, Megha Nawhal, Greg Mori, Manolis Savva.MCMI: (2007) Multi-Cycle Image Translation with Mutual Information Constraints. https://arxiv.org/abs/ 2007.02919
9. Kush, Deepanshu; Rossman, Benjamin. Tree-depth and the Formula Complexity of Subgraph Isomorphism. https://arxiv.org/abs/2004.13302
10. Cibej, Uros, Mihelic, et al. (2015) Improvements to Ullmann's Algorithm for the Subgraph Isomorphism Problem. International Journal of Pattern Recognition & Artificial Intelligence
11. J. Ingraham, V. Garg, R. Barzilay, T. Jaakkola. (2019) Generative Models for Graph-based Protein Design. In Neural Information Processing Systems (NeurIPS)
12. J.M.P. Martinez, R.B. Llavori, M.J.A. Cabo, and T.B. Pedersen. (2007) Integrating Data Warehouses with Web Data: A Survey. IEEE Trans. Knowledge and Data Eng., preprint, 21 https://doi.org/10.1109/TKDE.2007.190746
13. Kim, H., Choi, Y., Park, K., Lin, X., Hong, S. H., Han, W. S. (2021). Versatile equivalences: Speeding up subgraph query processing and subgraph matching. In Proceedings of the 2021 International Conference on Management of Data (pp. 925–937)
14 Nabieva E, Jim K, Agarwal A et al (2005) Whole-proteome Prediction of Protein Function Via Graph-theoretic Analysis of Interaction Maps. Bioinformatics 21(suppl 1):i302–i310
15. Lyu X , Wang X , Li Y F , et al. (2015) GraSS: An Efficient Method for RDF Subgraph Matching. International Conference on Web Information Systems Engineering. Springer, Cham
16. Baomin Xu, Tinglin Xin, Yunfeng Wang, Yanpin Zhao. (2013) Local Random Walk with Distance Measure. Modern Physics Letters B.27(8) 1–9
17. J. Cheng, Y. Ke, W. Ng, A. Lu. (2007) Fg-index: Towards Verification-free Query Processing on Graph Databases. In Proceedings of the ACM SIGMOD international conference on Management of data, pages 857–872
18. Khan, Arijit, Nan Li, Xifeng Yan, Ziyu Guan, Supriyo Chakraborty, Shu Tao. (2011) Neighborhood Based Fast Graph Search in Large Networks. In Proceedings of the ACM SIGMOD International Conference on Management of data,901–912
19. Liu J, Baomin Xu, Xiang Xu, Xin T (2016) A Link Prediction Algorithm Based on Label Propagation. Journal of Computational Science 16:43–50
20. Foggia P, Percannella G, Vento M (2014) Graph Matching and Learning in Pattern Recognition in the Last 10 Years. Int J Pattern Recognit Artif Intell 28(01):1450001
21. Kim H, Choi Y, Park K, Lin X, Hong SH, Han WS (2023) Fast subgraph query processing and subgraph matching via static and dynamic equivalences. VLDB J 32(2):343–368
22. Wang X, Zhang Q, Guo D, Zhao X (2023) A survey of continuous subgraph matching for dynamic graphs. Knowl Inf Syst 65(3):945–989
23. Ge, Y., Bertozzi, A. L. (2021). Active learning for the subgraph matching problem. In 2021 IEEE International Conference on Big Data (Big Data) (pp. 2641–2649). IEEE
24. Wang, H., Zhang, Y., Qin, L., Wang, W., Zhang, W., Lin, X. (2022, May). Reinforcement learning based query vertex ordering model for subgraph matching. In 2022 IEEE 38th International Conference on Data Engineering (ICDE) (pp. 245–258). IEEE
25. Zhao, K., Yu, J. X., Li, Q., Zhang, H., Rong, Y. (2023). Learned sketch for subgraph counting: a holistic approach. The VLDB Journal, 1–26
26. Zhang H, Bai Q, Lian Y, Wen Y (2022) A twig-based algorithm for top-k subgraph matching in large-scale graph data. Big Data Research 30:100350

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.