TreeSpan: Efficiently Computing Similarity All-Matching

Gaoping Zhu[#] Xuemin Lin[#] Ke Zhu[#] Wenjie Zhang[#] Jeffrey Xu Yu[†]

[#] University of New South Wales, Sydney, Australia
{gzhu, lxue, kez, zhangw}@cse.unsw.edu.au
[†] Chinese University of Hong Kong, Hong Kong, China
yu@se.cuhk.edu.hk

ABSTRACT

Given a query graph q and a data graph G, computing all occurrences of q in G, namely exact all-matching, is fundamental in graph data analysis with a wide spectrum of real applications. It is challenging since even finding one occurrence of q in G (subgraph isomorphism test) is NP-Complete. Consider that in many real applications, exploratory queries from users are often inaccurate to express their real demands. In this paper, we study the problem of efficiently computing all approximate occurrences of q in G. Particularly, we study the problem of efficiently retrieving all matches of q in G with the number of possible missing edges bounded by a given threshold θ, namely similarity all-matching. The problem of similarity all-matching is harder than the problem of exact all-matching since it covers the problem of exact all-matching as a special case with $\theta = 0$.

In this paper, we develop a novel paradigm to conduct similarity all-matching. Specifically, we propose to use a minimal set QT of spanning trees in q to *cover* all connected subgraphs q' of q missing at most θ edges; that is, each q' is spanned by a spanning tree in QT. Then, we conduct exact all-matching for each spanning tree in QT to induce all similarity matches. A rigid theoretic analysis shows that our new search paradigm significantly reduces the times of conducting exact all-matching against the existing techniques. To further speed-up the computation, we develop new filtering, computation sharing, and search ordering techniques. Our comprehensive experiments on both real and synthetic datasets demonstrate that our techniques outperform the state of the art technique by 7 orders of magnitude.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems; I.2.8 [**Problem Solving, Control Methods, and Search**]: Graph and tree search strategies

General Terms

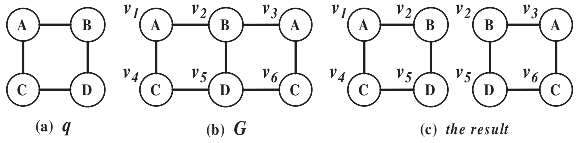
Algorithms, Performance

Keywords

Graph, Similarity All-Matching

1. INTRODUCTION

Recently, graphs have gained much popularity in modeling complex data in many applications, including biology (protein interaction networks), chemistry (chemical compounds), Web (social networks), road networks, etc. Significant research efforts have been made towards many fundamental problems in managing and analyzing graph data. Given a query graph q and a large data graph G, *exact all-matching* [22, 24] returns all occurrences of q in G, called "exact matches" of q in G. Figure 1 (a) and (b) illustrate a query graph q and a data graph G, respectively. The 2 resultant exact matches are depicted in Figure 1 (c). Exact all-matching is very useful for an exploration purpose in many real applications. For example, as shown in [22], in protein-protein interaction (PPI) networks, biologists may want to recognize groups of proteins which match a particular pattern in a large PPI network. Such a pattern could be an interaction network among a number of protein types. Since distinct proteins may share the same protein type (e.g., v_1 and v_3 in Figure 1(b) have label A), it is necessary to retrieve all the occurrences of a particular pattern (query graph) in a PPI network to identify all interactions among the involved proteins following the given pattern. Exact all-matching queries are also useful in a number of other applications [24, 26], such as identifying substructures in community networks, RDF datasets, software programs, etc.



(a) q (b) G (c) *the result*

Figure 1: All-Matching Queries

A common problem is that in many occasions, there could be no result for such an exploratory query issued by users since users often only have approximate goals in minds. For instance, if a user issues the query graph q depicted in Figure 2(a) against the data graph G in Figure 1(b), no results will be returned. Instead of asking users to manually refine a query graph to conduct exact all-matching search again and again, [23] recently proposes to ask the system to generate all *approximate* occurrences of q in G. Specifically, [23] proposes to enumerate all connected subgraphs g of G such that g is at most θ edges away to be identical (*isomorphic*) to q, called "similarity matches" conforming θ. For example, regarding

SIGMOD'12, May 20–24, 2012, Scottsdale, Arizona, USA.
Copyright 2012 ACM 978-1-4503-1247-9/12/05...$10.00.

the query graph q in Figure 2(a) and the data graph G in Figure 1(b), the 2 similarity matches of q in G conforming θ $(= 2)$ are depicted in Figure 1(c).

Note that a similarity match confirming θ could be an exact match. Finding all similarity matches to conform θ is generally harder than the problem of exact all-matching since it covers the problem of exact all-matching as a special case with $\theta = 0$. It is challenging since even finding one exact match of q in G is NP-Complete [5]. A naïve way to compute all similarity matches conforming θ is to enumerate all connected subgraphs q' of q missing at most θ edges and then find all exact matches for each of such subgraphs q'. The recent work [23], SAPPER, proposes to compute all exact matches of the connected subgraphs of q missing θ edges and then induce all other similarity matches from those obtained exact matches. While this effectively reduces the times of conducting exact all-matching from $O(m^\theta)$ to $O(\binom{m}{\theta})$ where m is the number of edges in q, the performance of SAPPER dramatically drops when θ increases to 3. Motivated by this, in this paper we study the problem of efficiently computing all similarity matches conforming θ, namely "similarity all-matching".

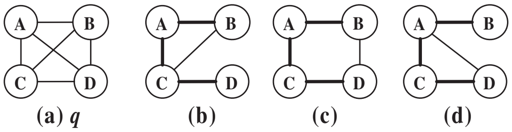


Figure 2: Cover by Trees

Our Approach. Regarding the query graph q in Figure 2(a), each of the 3 subgraphs of q depicted in Figures 2(b)-(d) misses 2 edges from q, respectively. These 3 subgraphs of q share a common spanning tree highlighted by the **bold** lines. Clearly, any exact match of one of these 3 subgraphs must be an exact match of this common spanning tree, and for any exact match $\mathcal{F}$ on the common spanning tree, it can be very efficient to identify (in linear time regarding the number of edges in these subgraphs, respectively) whether $\mathcal{F}$ can be extended to an exact match of one or all of these subgraphs. Consequently, instead of conducting exact all-matching on each of these subgraphs to induce the similarity matches from the exact matches [23] (3 times of exact all-matching in total), we only need to conduct exact all-matching once on the common spanning tree to induce the same set of similarity matches of these 3 subgraphs. Moreover, conducting exact all-matching on a tree is much less expensive than on a general graph.

Based on the above observations, in this paper we propose a novel search paradigm as follows. Firstly, we generate a minimal set QT of spanning trees in q to cover all connected subgraphs of q missing at most θ edges; that is, for each connected subgraph q' of q missing at most θ edges, q' uses one spanning tree in QT as its spanning tree. Then, we generate all exact matches for each spanning tree in QT to induce all similarity matches. Our rigid theoretic analysis shows that the number of spanning trees in QT generated in the worst case is always significantly smaller than the number $|Q_{\mathsf{SAPPER}}|$ of connected subgraphs of q missing θ edges except the two extreme cases; in these two extreme cases, $|QT| = |Q_{\mathsf{SAPPER}}|$. This implies that our algorithm conducts significantly less times of exact all-matching on average than that in the state of the art existing technique, SAPPER.

To further improve the efficiency of our computation, new filtering, computation sharing, and search ordering techniques are developed. We also propose to partially generate QT based on demands to skip unwanted spanning trees in QT regarding a data graph G; that is, based on the current partial mappings from q to G.

Contributions. Our principle contributions in this paper may be summarized as follows.

- We propose a novel search paradigm to conduct similarity all-matching conforming a similarity threshold θ by firstly conducting exact all-matching on a minimal set of spanning trees. Compared with the state of the art technique [23], this not only significantly reduces the times of conducting exact all-matching but also reduces the complexity of exact all-matching from a general graph to a tree.
- To further improve the efficiency of our computation, a set of new techniques are developed, including filtering-based effective search ordering, computation sharing, and adaptive generation of QT.
- We propose to compute all similarity *maximal* matches instead of all similarity matches to further remove computation redundancy.

Comprehensive experiments on real and synthetic datasets show that our techniques significantly outperform the state of the art techniques in [23] by several orders of magnitude.

Organizations. We organize the rest of this paper as follows. Section 2 presents the problem definitions and the framework. Our efficient search algorithms are presented in Section 3. In Section 4, we present our filtering and search ordering techniques. For presentation simplicity and also for the ease of a comparison (with [23]), in Sections 3-4, we present our techniques based on the assumption that no vertices in q are mismatched. In Section 5, we extend our techniques to allow vertices in q to be mismatched. We report the experimental evaluation in Section 6. Section 7 concludes the paper.

Related Work. Extensive research has been conducted in recent years on exact graph structure search. For instance, the problem of *subgraph containment search* [3, 8, 9, 13, 14, 19, 20, 25] is to find the graphs from a given set of data graphs which contain a query graph, while the problem of *supergraph containment search* [2, 21] is to find the graphs from a given set of data graphs which are contained by a given query graph. Driven by recent real applications, the problem of finding data graphs from a given set of data graphs which approximately contain a query graph, namely, *similarity subgraph search*, has been studied in [7, 12, 18, 17]. The problem of exact all-matching has been studied in [22, 24, 26].

Due to the NP-Completeness, most of the above techniques focus on developing effective indexing techniques based on the subgraph mining paradigms [11, 19]. Observing that, in graph structure search, search (i.e., retrieving the actual mappings from a query graph to data graphs) costs play a dominant role, [4, 12, 13, 23] also focus on developing efficient search techniques. While [4, 12, 13] aim to get only one (exact or approximate) mapping from a query to a data graph, [23] is the only work with the aim to efficiently generate all similarity matches by conducting exact all-matching on a set of connected subgraphs of q missing θ edges.

The exact matching from a tree to a graph is widely observed as a much more efficient operation than the exact matching from a graph to another graph. Consequently, trees or spanning trees are mainly used in indexing techniques to quickly prune some non-promising searches; for example, [23] adopts the exact matches of the spanning trees to decide the search root and prune non-promising vertices in data graphs. [4, 12, 13] are the only existing techniques to identify exact or similarity matches based on spanning trees, where [4, 13] propose to conduct the search on one spanning tree of the query graph q to detect whether or not there is an exact subgraph isomorphic mapping from q to a data graph G, and [12] proposes to detect whether or not there is a *similarity-based* subgraph isomorphic mapping from q to G based on all *feasible* spanning trees of q. Nevertheless, [4, 12, 13] aim to identify only one such match. The work presented in this paper is the first to propose to generate a *minimal* set of spanning trees of q to cover all its subgraphs missing at most θ edges and then efficiently conduct exact all-matching to share the computation for inducing all similarity (maximal) matches.

All existing filtering techniques for conducting all-matching are developed for the purpose of efficiently obtaining exact matches, including the filtering techniques in [23]. The work presented in this paper is the first to provide filtering technique for similarity all-matching and propose to use the filtering results to determine an effective search order.

A number of other subgraph search problems have also been investigated. For example, TALE [15] returns one similar subgraph in a data graph to an issued query graph with a high matching quality. [16] proposes to locate all DN-graphs, a new dense graph structure, in a large network. Nevertheless, they are inherently different to the problem studied in this paper.

2. BACKGROUND INFORMATION

Graphs studied in this paper are connected undirected graphs without self-loops and multiple edges [6]; that is, simple and connected undirected graphs. Moreover, the paper focuses on *vertex-labeled* graphs; the developed techniques may be immediately extended to edge-labeled graphs. Given a set of labels, Σ_V, a graph is denoted by $G = (V, E, l)$ where V is the set of vertices, $E \subseteq V \times V$ is the set of edges. Here, l is a labeling function: $V \to \Sigma_V$ (i.e., $l(u)$ is the label of a vertex $u \in V$). We denote the vertex set and the edge set of a graph g by $V(g)$ and $E(g)$, respectively. $|V(g)|$ and $|E(g)|$ denote the number of vertices and edges, respectively. For presentation simplicity, a connected undirected vertex-labeled graph is hereafter abbreviated to a graph.

2.1 Problem Statement

The similarity matches defined below are equivalent to the definition of *approximate matches* in [23]. We first define a subgraph isomorphic mapping.

Definition 1. *Given two graphs $g = (V, E, l)$ and $g' = (V', E', l')$, a **subgraph isomorphic** mapping $\mathcal{F} : V \to V'$ is an injective function such that (1) $\forall u \in V$, $\mathcal{F}(u) \in V'$ and $l(u) = l'(\mathcal{F}(u))$; (2) $\forall (u_1, u_2) \in E$, $(\mathcal{F}(u_1), \mathcal{F}(u_2)) \in E'$.*

We also say that g is *subgraph isomorphic* to g' or g' *contains* g or g has an *exact match* in g'.

Definition 2. *Given a similarity threshold θ, two graphs q and G, if there exists a subgraph isomorphic mapping $\mathcal{F}$ from a connected subgraph q' of q to G where q' has at most θ edges missing from q, then $(\mathcal{F}, \mathcal{F}(E(q')))$ is called a **similarity match** of q conforming θ where $\mathcal{F}(E(q')) = \{\, (\mathcal{F}(u), \mathcal{F}(v)) \mid (u, v) \in E(q') \,\}$ is the set of edges in G mapped from q'.*

We use $\mathcal{F}$ together with $\mathcal{F}(E(q'))$ to identify a similarity match since $\mathcal{F}$ gives the mapping information from a vertex $u \in q'$ to a vertex $v \in G$.

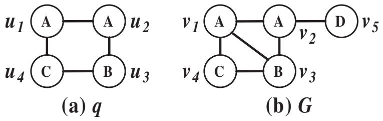


Figure 3: Query and Data Graphs

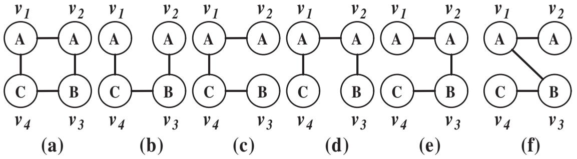


Figure 4: Similarity Matches ($\theta = 1$)

Example 1. *Regarding the two graphs q and G in Figure 3, all the similarity matches of q regarding G conforming $\theta = 1$ are depicted in Figure 4.*

The similarity matches in Figures 4(a)-(e) are generated by $\mathcal{F}_1 = \{u_1 \to v_1, u_2 \to v_2, u_3 \to v_3, u_4 \to v_4\}$ based on different connected subgraphs of q, while the similarity match in Figure 4(f) is generated by $\mathcal{F}_2 = \{u_1 \to v_2, u_2 \to v_1, u_3 \to v_3, u_4 \to v_4\}$. □

In Example 1, the exact match in Figure 4(a) is also regarded as a similarity match conforming $\theta = 1$, and the generated matches in Figures 4(b)-(e) are sub-matches of the exact match. These sub-matches are less interesting to be generated since any connected subgraph of the exact match with one edge missing is a similarity match confirming $\theta = 1$. Therefore, in this paper we focus on generating all the similarity *maximal* matches conforming a given similarity threshold θ.

Definition 3. *A similarity match $(\mathcal{F}, \mathcal{F}(E(q')))$ conforming θ is **maximal** if there does not exist another connected subgraph q'' of q such that q'' is a proper supergraph of q' (i.e, q'' is a supergraph of q' and $q'' \neq q'$), and $\mathcal{F}$ is also a subgraph isomorphic mapping of q''.*

Example 2. *Regarding the two graphs q and G in Figure 3, the two similarity matches in Figures 4(a) and 4(f) are similarity maximal matches conforming $\theta = 1$ generated by $\mathcal{F}_1$ and $\mathcal{F}_2$, respectively.* □

Definition 4 (Similarity Maximal All-Matching). *Given a query graph q, a data graph G, and a θ, find all distinct similarity maximal matches of q in G conforming θ.*

Problem Statement. Given a query graph q, a data graph G, and a θ, this paper studies the problem of efficiently conducting similarity maximal all-matching.

Note that the existing work [23] studies the problem of efficiently retrieving all similarity matches without allowing mismatched vertices in q. For presentation simplicity and the ease of a comparison (with [23]), we present our techniques in Sections 3-4 with the assumption that no vertices in q are mismatched. In Section 5, we show that our techniques can be immediately extended to the general case where vertices in q are allowed to be mismatched. In the rest of the paper, query graphs are abbreviated into queries.

2.2 Framework

As the problem of testing subgraph isomorphism is NP-Complete [5], the problem of similarity maximal all-matching is NP-complete since $\theta = 0$ implies the subgraph isomorphism testing. To reduce the computation costs, our algorithms follow the framework of filtering and search [22, 24]. In the filtering phase, for each vertex u in a query graph q we filter the non-promising vertices in G to generate a set of candidate vertices $C(u)$ in G to be mapped from u. In search phase, we enumerate all similarity maximal matches. Below we first present our novel search paradigm, assuming that $C(u)$ for each vertex in q has already been generated.

3. TREE BASED SPANNING SEARCH

In this section, we present a novel search paradigm to conduct similarity maximal all-matching with the assumption that no vertices in q will be mismatched. We need the following notion.

Definition 5. *Given a graph q_1, a subgraph q_2* **spans** *q_1 if q_2 is connected and no vertex in q_1 is missed in q_2. Here, we also say q_2 is a* **spanning subgraph** *of q_1.*

Definition 5 extends the notion of spanning trees of a graph; that is, any spanning tree of q_1 spans q_1.

Definition 6. *Suppose that q_2 is a spanning subgraph of q_1 and $\mathcal{F}$ is a subgraph isomorphic mapping from q_2 to G. The similarity match of q_1 in G* **induced** *by $\mathcal{F}$ is $(\mathcal{F}, M_{\mathcal{F},q_1})$ where $M_{\mathcal{F},q_1} = \{(\mathcal{F}(u), \mathcal{F}(v)) \mid (u, v) \in E(q_1)$ & $(\mathcal{F}(u), \mathcal{F}(v)) \in E(G)\}$.*

Example 3. *Regarding Example 1, the match depicted in Figure 4(a) can be induced by a subgraph isomorphic mapping on the subgraph, $((u_1, u_2), (u_2, u_3), (u_3, u_4))$, of q in Figure 3(a).*

3.1 Overview of Our Approach

Given a spanning subgraph q' of q and a subgraph isomorphic mapping $\mathcal{F}$ of q', it can be very efficient to compute the induced similarity match of q by $\mathcal{F}$ of q'; this can be done in linear time regarding the number of edges in q. Based on this observation, we use the following 3 phases to conduct similarity maximal all-matching.

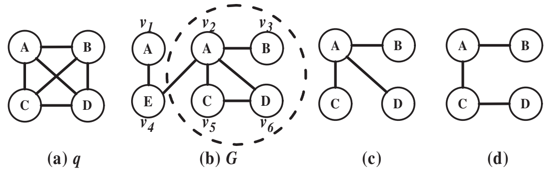
- **Phase 1: Seeding.** Generate a set QT of spanning trees of q to *cover* all spanning subgraphs of q missing at most θ edges from q; that is, any spanning subgraph of q missing at most θ edges uses at least one spanning tree in QT as its spanning tree.
- **Phase 2: Exact All-Matching.** For each spanning tree $T \in QT$, get all subgraph isomorphic mappings (i.e., exact all-matching) from T to G.
- **Phase 3: Inducing Matches.** For each subgraph isomorphic mapping $\mathcal{F}$ of T, induce the similarity match of q from $\mathcal{F}$ conforming θ.

The costs of the 3-phase search are expressed below in (1).

$$C_{seeding} + \sum_{T \in QT} C_{allmatching}(T, G) + C_{inducing} \quad (1)$$

$C_{seeding}$ is the cost of Phase 1, $C_{allmatching}(T, G)$ is the cost of computing all exact matches of T in G, and $C_{inducing}$ is the cost of Phase 3. Phase 2 takes the dominant costs as it takes exponential time in the worst case to conduct exact all-matching due to the NP-Completeness of testing subgraph isomorphism. In (1), $\sum_{T \in QT} C_{allmatching}(T, G)$ can be written as $|QT| C_{allmatching}$ where $C_{allmatching}$ is the average cost of computing exact all-matching for a T in QT.

Duplicates-Free Enforcement. The 3 phases can generate all similarity maximal matches since QT covers all spanning subgraphs of q missing at most θ edges and an exact match of a spanning subgraph q' must be the exact match of any spanning tree of q'. Nevertheless, it is possible that different spanning trees may generate the same similarity maximal match.



(a) q (b) G (c) (d)

Figure 5: Duplicates

Suppose that a query q is given in Figures 5(a). Regarding the data graph G in Figure (b), the two spanning trees of q in Figure 5(c)-(d) will induce the same similarity maximal match in G conforming $\theta = 2$, the subgraph of G circled by the dotted line. In fact, in this example, any spanning tree of q without edges (B, C) and (B, D) will induce the same circled similarity maximal match in G. Duplicates not only require extra effort to be removed but also waste computation costs to be generated.

To enforce that the above 3-phase search always generates *distinct* similarity maximal matches, we attach a set $T.R$ of edges to each spanning tree $T \in QT$ such that $E(T) \cap T.R = \emptyset$, $|T.R| \leq \theta$, and any similarity maximal match induced by an exact match of T must exclude any edge in $T.R$; $T.R$ is called the *edge exclusion set* of T. Regarding the spanning tree T in Figure 5(d), if we make $T.R = \{ (A, D) \}$, then no similarity matches conforming $\theta = 2$ will be induced by an exact match of T.

Speed-Up Techniques. We propose to conduct exact all-matching for spanning trees in QT by sharing the computation of prefix instead of conducting exact all-matching for each $T \in QT$ separately. Moreover, we also propose to enumerate QT partially based on demands (i.e. based on the current mappings obtained) instead of always enumerating the whole QT.

Compared with SAPPER. SAPPER also adopts the above 3-phase approach. In the seeding phase, SAPPER proposes to generate the set Q_{SAPPER} of all spanning subgraphs missing θ edges.

Our paradigm has the following 3 major advantages compared with SAPPER. Firstly, we will show $|QT|$ is significantly smaller than $|Q_{\mathsf{SAPPER}}|$ except that in the two extreme cases, $|QT| = |Q_{\mathsf{SAPPER}}|$. This leads to a significant reduction on the times of conducting exact all-matching; that is, from $|Q_{\mathsf{SAPPER}}|$ to $|QT|$. Our strategy of enumeration on demands further reduces the times of conducting exact all-matching. Secondly, conducting exact all-matching on a spanning tree is much less expensive than on a graph due to a much simpler structure of a spanning tree. Thirdly, we conduct exact all-matching on QT by computation sharing, while SAPPER conducts exact all-matching on each graph in Q_{SAPPER} separately. As a result, our experiment demonstrates that our algorithm significantly improves the performance of SAPPER (by several orders of magnitude).

It is worth mentioning that SAPPER conducts exact all-matching on $(n+1)$ spanning trees of q to do a filtering only. Nevertheless, our algorithm can always generate the results by conducting exact all-matching on at most n spanning trees when $\theta = 1$.

3.2 Generating QT

We present a novel algorithm to effectively generate a *minimal QT* regarding θ. We assume that each edge in q has a weight and a vertex in q is selected as the head; details about selecting edge weights, and the head vertex to improve the efficiency of our search algorithm in Section 3.4 may be found in Section 4.2. Each spanning tree T of q is represented as a sequence of edges, $(T[1], ..., T[n-1])$, where n is the number of vertices of q. We refer the edge $T[h]$ as the hth edge or at h level. For the search efficiency reason (to be stated in Section 4.2), an initial minimum spanning tree T of q is chosen to follow the order of edges selected by the PRIM algorithm [1], namely PRIM order.

PRIM order: for $1 \leq i \leq n-1$, $T[1], ..., T[i]$ are connected, $T[i]$ is the edge in T with the smallest weight to connect $T[1], ..., T[i-1]$ where $T[0]$ is the head vertex.

Our enumeration algorithm to generate QT together with the edge exclusion sets $T.R$ is executed in a *depth-first* search fashion from the lowest level ($h = 1$) to the highest level ($h = n-1$); it is outlined in Algorithm 1 in a recursive fashion. It consists of 2 phases, *go-down* phase (Lines 1-2) and *alternating-reordering* phase (Lines 3-8).

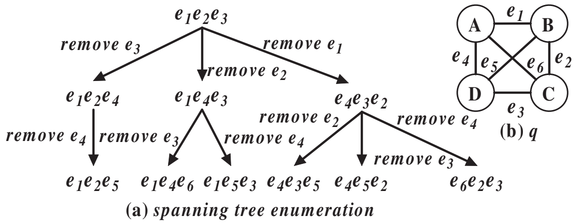
Algorithm 1: EnuQT $(h, T, T.R, \theta, QT)$

```
Input : $h$: current level, initially 1;
        $T$: current spanning tree;
        $T.R$: the edges replaced to get $T$, initially $\emptyset$;
        $\theta$: a given similarity threshold;
        $QT$: the set of spanning trees;
1 if $h < n-1$ then
2     EnuQT $(h+1, T, T.R, \theta, QT)$ ;
3 if $|T.R| < \theta$ & checkReplacing $(T[h])$ then
4     $e :=$ Replacing $(T[h])$;
5     $T' :=$ reOrdering $(T - \{T[h]\} + \{e\})$;
6     $T'.R := T.R + \{T[h]\}$;
7     $QT := QT \bigcup \{T'\}$;
8     EnuQT $(h, T', T'.R, \theta, QT)$ ;
```

Algorithm 1 starts with an initial spanning tree loaded in QT. The *go-down* phase in Algorithm 1 corresponds to the depth-first paradigm.

In each enumerated T, the edge exclusion set $T.R$ records the set of edges replaced during the enumeration process to get T from the initial spanning tree. In the *alternating-reordering phase* at the ith level of T, we use an edge e in $(E(q) - E(T) - T.R)$ to replace $T[i]$ to form another spanning tree together with the remaining edges in T if (1) $|T.R| < \theta$, and (2) at least one edge e' in $(E(q) - E(T) - T.R)$ can replace $T[i]$ to form another spanning tree of q. We check these two conditions in Line 3 where checkReplacing $(T[h])$ returns true if the condition (2) holds.

Replacing $(T[h])$ returns the edge e with the smallest weight among all edges in $(E(q) - E(T) - T.R)$ which can be used to replace $T[h]$ to connect the remaining edges in T to form another spanning tree. reOrdering $(T - \{T[h]\} + \{e\})$ is to reorder the edges in $\{e, T[h+1], ..., T[n-1]\}$ to follow the PRIM order while fixing the order in $T[1], ..., T[h-1]$ (i.e, $T'[1] = T[1], ..., T'[h-1] = T[h-1]$).



(a) *spanning tree enumeration*

(b) q

Figure 6: Spanning Tree Enumeration ($\theta = 2$)

EXAMPLE 4. *We use Figure 6 to illustrate the enumeration process where the query q is depicted in Figure 6(b) and $\theta = 2$. Suppose that the weight of e_i is i. The root in Figure 6(a) gives the initial spanning tree $e_1e_2e_3$. Consecutively conducting the* go-down *phase from the initial spanning tree $e_1e_2e_3$ to drill down to e_3 for executing the* alternating-reordering *phase, e_4 is chosen to replace e_3 to form the next spanning tree $e_1e_2e_4$, and then e_5 is chosen to replace e_4 to form the next spanning tree $e_1e_2e_5$. Note that $e_1e_2e_3.R = \emptyset$, $e_1e_2e_4.R = \{e_3\}$, and $e_1e_2e_5.R = \{e_3, e_4\}$.*

Algorithm 1 may execute the alternating-reordering *phase while drilling down to e_2. Then e_4 is chosen to replace e_2. Since e_1 and e_3 are disconnected,* reOrdering *() gives $e_1e_4e_3$ for further conducting* go-down *and/or* alternating-reordering *phases. Algorithm 1 may execute the* alternating-reordering *phase at e_1 in the original spanning tree. The set QT of spanning trees of q, generated by Algorithm 1, is depicted in Figure 6(a).*

Analysis of Algorithm 1. Immediately, Algorithm 1 does not enumerate two identical spanning trees; and for each $T \in QT$, $E(T) \cap T.R = \emptyset$ and $|T.R| \leq \theta$. Theorem 1 below implies that all generated similarity maximal matches are distinct if each edge exclusion set $T.R$ is enforced to be excluded from the matches induced by exact matches on T.

THEOREM 1. *Given two spanning subgraphs q_1 and q_2 of q, suppose that there are two different spanning trees T_1 and T_2 generated by Algorithm 1 such that q_1 contains T_1 and q_2 contains T_2. Further suppose that $E(q_1) \cap T_1.R = \emptyset$ and $E(q_2) \cap T_2.R = \emptyset$. Then $q_1 \neq q_2$.*

PROOF. We represent each $T.R$ by a sequence of edges, $(T.R[1], ..., T.R[L])$, where $L = |T.R|$, such that in the enumeration process to get T, $TR[1]$ is replaced first from the initial spanning tree, then $T.R[2]$ is replaced from the next spanning tree, and so on. Without loss of generality, we assume that $|T_1.R| \leq |T_2.R|$. Since $T_1 \neq T_2$, it is immediate that $(T_1.R[1], ..., T_1.R[L]) \neq (T_2.R[1], ..., T_2.R[L'])$.

Case 1. For $1 \leq i \leq L$, $T_1.R[i] = T_2.R[i]$. Thus, $L' > L$ since $L' \geq L$ and $T_1.R \neq T_2.R$. According to Algorithm 1, T_2 must be enumerated from T_1 by firstly replacing $T_2.R[L+1]$ from T_1. Consequently, $q_1 \neq q_2$ since q_2 does not contain $T_2.R[L+1]$ but q_1 contains $T_2.R[L+1]$ since q_1 contains T_1 and T_1 contains $T_2.R[L+1]$.

Case 2. Assume that $T_1.R[k]$ and $T_2.R[k]$ are the first edges in $T_1.R$ and $T_2.R$, respectively, such that $T_1.R[k] \neq T_2.R[k]$. Then, there must be a T_3 generated by Algorithm 1 with $T_3.R = (T_1.R[1], ..., T_1.R[k-1])$. It is immediate that both T_1 and T_2 are enumerated from T_3. That is, T_3 contains both of $T_1.R[k]$ and $T_2.R[k]$, T_1 is enumerated from T_3 by first placing $T_1.R[k]$, and T_2 is enumerated from T_3 by first replacing $T_2.R[k]$. Without loss of generality, assume $T_3[i] = T_1.R[k]$, $T_3[j] = T_2.R[k]$, and $j > i$. According to Algorithm 1, T_2 is enumerated from T_3 by iteratively replacing edges at levels not smaller than j. Consequently, T_2 contains $T_1.R[k]$. Therefore, q_2 contains $T_1.R[k]$. Thus, $q_1 \neq q_2$ as q_1 does not contain $T_1.R[k]$. □

The proof of Theorem 1 implies that in two different spanning trees T and T' in QT, there must be one, say T' such that T' contains one edge in $T.R$. Theorem 1 immediately implies the following Theorem with the assumption $\theta \leq m - n + 1$ where $m = |E(q)|$ and $n = |V(q)|$. Note that $\theta > m - (n-1)$ means that any subgraph of q missing θ edges is disconnected; thus we do not need to consider such a case in this section - Section 3.

THEOREM 2. *$|QT| \leq |Q_{\mathsf{SAPPER}}|$ and $|QT| = |Q_{\mathsf{SAPPER}}|$ only when $\theta = 0$ or $\theta = m - (n-1)$.*

PROOF. Clearly, for each spanning tree $T \in QT$, we can generate a spanning subgraph q_T of q missing θ edges such that q_T contains T and q_T does not contain any edge in $T.R$. Theorem 1 immediately implies that $q_T \neq q_{T'}$ for any pair of T and T' in QT. Thus, $|QT| \leq |Q_{\mathsf{SAPPER}}|$.

Note that q_T has $\binom{m-n+1}{\theta}$ choices corresponding to the initial spanning tree T. Consequently, $|QT| = |Q_{\mathsf{SAPPER}}|$ only if $\theta = 0$ or $\theta = m - (n-1)$. □

Clearly, each T in QT leads to $\binom{m-n+1-|T.R|}{\theta-|T.R|}$ distinct spanning subgraphs of q missing θ edges each of which contains T but does not contain any edge in $T.R$.

Theorem 1 immediately implies:

$$|Q_{SAPPER}| \geq \sum_{T \in QT} \binom{m-n+1-|T.R|}{\theta - |T.R|} \qquad (2)$$

Note that $\binom{m-n+1-|T.R|}{\theta-|T.R|} = \binom{m-n+1-|T.R|}{m-n+1-\theta}$. As $|QT|$ is significantly smaller than $\sum_{T \in QT} \binom{m-n+1-|T.R|}{m-n+1-\theta}$ when $\theta \neq 0$ and $m - n + 1 \neq \theta$, $|Q_{\mathsf{SAPPER}}|$ is significantly larger than $|QT|$ except $\theta = 0$ or $m - n + 1 = \theta$. Our experiment also demonstrates that $|Q_{\mathsf{SAPPER}}|$ is much larger than $|QT|$.

Next we show the completeness of QT; that is, every similarity match of q can be induced by a $T \in QT$ even if the edges in $T.R$ are excluded.

THEOREM 3. *For a spanning subgraph q' of q missing at most θ edges, there is a spanning tree $T \in QT$ such that q' contains T and q' does not contain any edge in $T.R$.*

PROOF. Let $S_{q'}$ denote the set of missing edges in q' from q; that is, $S_{q'} = E(q) - E(q')$. If the initial spanning tree T has no edge in $S_{q'}$, then the theorem holds.

Assume that there are k ($k \leq \theta$) edges in $S_{q'}$ which are in T and these k edges are $T[i_1]$, $T[i_2]$, ..., $T[i_k]$ such that $i_1 < i_2 ... < i_k$. As one part of Algorithm 1, we continuously execute Line 2 to reach the level i_1 and then execute the *alternating-reordering* phase (i.e., Line 3 to Line 8) to replace the edge $T[i_1]$ by e to form T'.

Note that e may or may not be in $Sq' - \{T[i_1]\}$ and now we only need to focus on $S_{q'} - \{T[i_1]\}$ against T'. Due to reOrdering () in Algorithm 1, the edges in $(S_{q'} - \{T[i_1]\}) \cap T'$ may change their positions in T. Nevertheless, since we only reorder the edges at the levels not lower than i_1, the lowest level edge $T'[j]$ in $(S_{q'} - \{T[i_1]\}) \cap T'$ must not be lower than i_1; that is, $j \geq i_1$. Therefore, Algorithm 1 goes down to j to replace $T'[j]$. We continue to do this till find a spanning tree T'' (i.e., $T'' \in QT$) such that T'' does not contain any edge in $S_{q'}$ and $T''.R$ is a subset of $S_{q'}$. Thus, q' contains T'' and q' does not contain any edge in $T''.R$. □

Minimality of QT. To enforce our search algorithm in Section 3.4 to generate all *distinct* similarity maximal matches, Algorithm 1 may not give the minimum number of spanning trees as a cover in general. Nevertheless, Theorem 1 immediately implies that QT is *minimal* with the enforcement of the exclusive semantics of edge exclusion sets; that is, removing one T from QT, the maximal similarity matches induced by T to exclude the edges in $T.R$ cannot be induced by another T' in QT to exclude edges in $T'.R$. It can also be immediately verified that any spanning tree T in QT is a minimal spanning tree in $(q - T.R)$ and follows the PRIM order in $(q - T.R)$.

3.3 Effectively Storing QT

We present a data structure, $\mathcal{T}$, to effectively organize QT for prefix sharing search, referred as a DFS Traversal Tree. The basic idea is as follows. When generating T' by replacing $T[h]$ from T, we store their common prefix $T[1], ..., T[h-1]$ only once in $\mathcal{T}$. The remaining spanning edges in T and T' are organized as two different branches.

In a $\mathcal{T}$, each node N represents an edge $T[h]$ of a spanning tree T, while the root R represents the head vertex of the spanning trees in QT. The initial spanning tree is firstly loaded as the left-most path of $\mathcal{T}$. In Algorithm 1, iteratively, if $T[h]$ is replaced by an edge e to form the next spanning tree T', then $T'[h]$ is allocated as the right sibling next to $T[h]$ such that T and T' share the prefix $T[1], ..., T[h-1]$. Note that $T'[h]$ is not always e due to reOrdering (). Clearly, the space requirement of $\mathcal{T}$ is $O(|QT||V(q)|)$.

Regarding the query q in Figure 6(b). The resulted QT of Algorithm 1 depicted in Figure 6(a) is organized in $\mathcal{T}$ as depicted in Figure 7. Here, arrows indicate the order of edges in a spanning tree and a path from the root to a leaf gives a spanning tree.

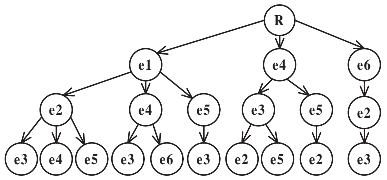


Figure 7: DFS Traversal Tree

3.4 Similarity Maximal All-Matching

Basic Idea. The central idea of our algorithm is to search $\mathcal{T}$ in a depth-first search fashion to generate all extendible exact mappings $\mathcal{F}$ for each spanning tree $T \in QT$. We enforce that the induced match by each $\mathcal{F}$ on T excludes $T.R$ and confirms θ. Theorem 1 and 3 guarantee the correctness.

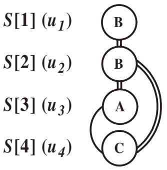
Iteratively, once we detect that a node (corresponding to an edge) in $\mathcal{T}$ does not support the current mapping extension, we immediately terminate the extension to the next level in $\mathcal{T}$ and only focus on alternating the search to the right sibling node.

We adopt a graph encoding technique in [13], called QIsequence, to present our search algorithm.

QIsequence. A QIsequence *seq* of q is determined by a spanning tree T of q; it is represented by a sequence $\{S[1], ..., S[|V(q)|]\}$. Here, each $S[i]$ $(1 \leq i \leq |V(q)|)$ is called an *entry* of *seq* and corresponds to a vertex in q. The first entry $S[1]$ corresponds to the root vertex of T. All edges in T are represented by the *spanning edges* in *seq* such that for $2 \leq i \leq |V(q)|$, each entry $S[i]$ has one and only one spanning edge $(S[i], S[j])$, denoted by $S[i].sEdge$, where $j < i$. Here, we call $S[j]$ the parent of $S[i]$ in T. All other edges in q are called *backward edges* in *seq* and the set of backward edges *incident* to an entry $S[i]$ is denoted by $S[i].bEdges$.

Entry	*sEdge*	*bEdges*
S[1] (u1)	nil	∅
S[2] (u2)	(S[2], S[1])	∅
S[3] (u3)	(S[3], S[2])	∅
S[4] (u4)	(S[4], S[2])	(S[4], S[3])

(*a*)



(b) *seq*

Figure 8: A Sample QIsequence

Example 5. *We depict a QIsequence of a query graph q in Figure 8(b) based on a spanning tree of q. The corresponding entry information is listed in Figure 8(a). Spanning edges and backward edges are depicted by double-lined and single-lined edges, respectively.*

In our search algorithm, the enumerated spanning trees for generating QIsequences are stored in $\mathcal{T}$ as described in Section 3.3. Two ways can be used to conduct similarity maximal all-matching, (1) build an entire $\mathcal{T}$ (i.e., QT), and (2) build $\mathcal{T}$ on the demands; that is, based on the current results. We first present (2).

EnumerateOnDemand. EnumerateOnDemand strategy iteratively enumerates QIsequences (spanning trees) only when it is feasible to extend the current partial mapping vertically (i.e. go-down) or horizontally (i.e., alternating to the next spanning tree). Our search algorithm is outlined below in Algorithm 2 following the *go-down* and *alternating-reordering* phases in Algorithm 1.

```
Algorithm 2: SimSearch $(h, seq, \mathcal{F}, G, \gamma, \theta, \mathcal{T})$

Input : $h$: the current mapping depth; (initially 1)
        $seq$: the current QIsequence;
        $\mathcal{F}$: the current partial mapping;
        $G$: the data graph; $\gamma$: # missing edges;
        $\theta$: a given similarity threshold;
        $\mathcal{T}$: the DFS Traversal Tree;
1  $S[h] := seq.S[h]$;
2  for each $v \in$ getCandiate$(S[h], \mathcal{F}, G)$ do
3      $\alpha :=$ getMissingBackedges$(v, S[h].bEdges, G)$ ;
4      if $\alpha + \gamma \leq \theta$ & Validate$(v, S[h].bEdges, seq.R, G)$ then
5          $\mathcal{F}[h] := v$;
6          if $h < |V(q)|$ then
7              SimSearch $(h+1, seq, \mathcal{F}, G, \gamma+\alpha, \theta, \mathcal{T})$ ;
8          else
9              Output $(\mathcal{F}, M_{\mathcal{F},T_{seq}})$ ;
10 if $\gamma < \theta$ & $h \neq 1$ & checkReplacing $(S[h].sEdge)$ then
11     $S'[h].sEdge :=$ Replacing $(S[h].sEdge)$ ;
12     if Already $(seq, S[h].sEdge, S'[h].sEdge, \mathcal{T})$ then
13         $seq' :=$ get$(seq, S[h].sEdge, S'[h].sEdge, \mathcal{T})$ ;
14     else
15         $seq' :=$ enu$(seq, S[h].sEdge, S'[h].sEdge, \mathcal{T})$ ;
16     SimSearch $(h, seq', \mathcal{F}, G, \gamma+1, \theta, \mathcal{T})$ ;
```

In Algorithm 2, h is the current mapping depth and *seq* is the current QIsequence to be explored against G (initially seq^*, the QIsequence determined by the initial spanning tree in QT). The current partial mapping $\mathcal{F}$ on $seq[1, ..., h-1]$ is a vector $\{\mathcal{F}[1], ..., \mathcal{F}[h-1]\}$ where $\mathcal{F}[i]$ $(1 \leq i \leq h-1)$ is a vertex in G mapped from $S[i]$, and $seq[1, ..., i]$ denotes the prefix of *seq* up to the entry $S[i]$ (i.e., $seq[1, ..., i]$ consists of all the spanning edges and the backward edges of the vertices (entries) $S[j]$ for $1 \leq j \leq i$). γ is the number of missing edges in the current parting mapping $\mathcal{F}$ on $seq[1, ..., h-1]$. $\mathcal{T}$ is the DFS Traversal Tree which initially has only a left-most path representing the spanning tree of seq^*.

Following Algorithm 1, Lines 2-7 of Algorithm 2 execute the *go-down* phase and Lines 10-16 execute the *alternating-reordering* phase. Note that each edge in the spanning tree of *seq* is represented by the spanning edge of an entry and the root $S[1]$ does not have a spanning edge.

go-down. The *go-down* phase (Lines 2-7) of Algorithm 2 needs to check whether or not the current extended partial mapping conforms the θ constraint and excludes the edge exclusion set when it attempts to extend the current partial mapping $\mathcal{F}$ on $seq[1, ..., h-1]$ to $\mathcal{F}$ on $seq[1, ..., h]$ via the spanning edge $S[h].sEdge$ by fixing the mapping on $seq[1, ..., h-1]$. getCandiate iteratively retrieves the next unmapped candidate v in $C(u)$ of G where $C(u)$ is the candidate set of u. If $h = 1$, v is simply a vertex in $C(u)$; otherwise, v is chosen to match the spanning edge $S[h].sEdge$ from q to G; that is, v must also be a neighbor of the vertex in G mapped from the parent of $S[h]$ in *seq*.

getMissingBackedges computes the number of backward edges in $S[h].bEdges$ mismatched by extending to v. Consequently, if $\gamma + \alpha \leq \theta$, the current partial mapping $\mathcal{F}$ on $seq[1, ..., h]$ still conforms the θ constraint. In addition, we use Validate () to ensure that the current induced match does not include any edge in $seq.R$ where $seq.R$ stores the edge exclusion set of the spanning tree of *seq* (i.e., edges replaced in the enumeration process from the spanning tree of seq^* to obtain the spanning tree of *seq*). It checks the induced match on $seq[h].bEdges$; if the induced match maps an edge in $seq.R$ to G, then we stop the current extension. When the current mapping covers all vertices (i.e. $h = |V(q)|$), we output the similarity maximal match $(\mathcal{F}, M_{\mathcal{F},T_{seq}})$ induced by the spanning tree T_{seq} of *seq* in Line 9.

alternating-reordering. Lines 10-16 of Algorithm 2 execute the *alternating-reordering* phase following Algorithm 1. Since the root node $S[1]$ in a *seq* does not have a spanning edge, Algorithm 2 does not execute the *alternating* phase for $h = 1$. checkReplacing $(S[h].sEdge)$ and Replacing $(S[h].sEdge)$ are the same as those in Algorithm 1. Already () checks if the spanning tree of a seq' generated by replacing $S[h].sEdge$ with $S'[h].sEdge$ has been obtained earlier. If yes, it just gets the seq' for the *go-down* phase in Line 16. Otherwise, enu () is executed in the same way as Lines 3-8 of Algorithm 1 to get the spanning tree of seq' from the spanning tree of *seq* by replacing $S[h].sEdge$ with $S'[h].sEdge$; the spanning tree of seq' is then stored in $\mathcal{T}$ in the way described in Section 3.3 for the *go-down* phase in Line 16. $S[h].sEdge$ and all edges in $seq.R$ are added to $seq'.R$.

EnumerateAll. The EnumerateAll strategy firstly enumerates the spanning trees in QT (thus, all corresponding QIsequences) by Algorithm 1, stored in $\mathcal{T}$ as described in Section 3.3. We run the algorithm 2 by excluding Lines 13-14.

Correctness of SimSearch. By Theorems 1 and 3, SimSearch can generate all distinct similarity maximal matches.

3.5 Computing All Similarity Matches

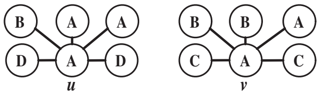
We can compute the similarity matches, as proposed in [23], using our similarity maximal matches as intermediate results. That is, generate all feasible spanning subgraphs from each similarity maximal match. Theorems 1 and 3 ensure that all distinct similarity matches conforming θ will be generated. Our experiments demonstrate that this algorithm for computing all similarity matches achieves a speed-up for up to 5 orders of magnitude against SAPPER algorithm.

4. FILTERING AND ORDERING

In this section, we first present an effective filtering condition to remove non-promising vertices from G for each vertex u in q; that is, generating $C(u)$ to feed Algorithm 2. Then we present an effective edge ordering technique.

4.1 Neighborhood based Pruning

Definition 7 (Neighborhood Aggregates). *Given a set of labels $\Sigma_V = \{L_1, ..., L_m\}$ and a graph g, the* neighborhood aggregates *of each vertex v in g, denoted by $N(v, g)$, is $(x_1, ..., x_m)$ where x_i is the number of vertices with label $L_i \in \Sigma_V$ that can be reached by an edge from v in g. Here, $N(v, g)$ may be regarded as a point in an m-dimensional space and each x_i is called the* hit *of the label L_i or the dimension i.*



(a) q (b) G

Figure 9: Neighborhood Aggregates

Example 6. *Regarding the example in Figure 9, $L_1 = A$, $L_2 = B$, $L_3 = C$, and $L_4 = D$. $N(u, q) = (2, 1, 0, 2)$, and $N(v, G) = (1, 2, 2, 0)$.*

Our filtering technique is based on the neighborhood aggregates. Given two vertices $u \in V(q)$ and $v \in V(G)$ where $l(u) = l(v)$, assume $N(u, q) = (x_1, ..., x_m)$ and $N(v, G) = (y_1, ..., y_m)$. According to Definitions 2 and 3, if $x_i - y_i > \theta$ for any i, u can not be mapped to v because more than θ edges in q will be definitely mismatched. This can be generalized by Theorem 4 below. We first define the *neighborhood distance* $\delta(u, v)$ as $\delta(u, v) = \Sigma_{i=1}^{m} \delta_i$ where for $1 \leq i \leq m$,

$$\delta_i = \begin{cases} x_i - y_i & x_i > y_i \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Definition 4 immediately implies the following theorem.

Theorem 4. *Given a similarity threshold θ, a vertex $u \in V(q)$ can not be mapped to a vertex $v \in V(G)$ by any similarity matches confirming θ, if $\delta(u, v) > \theta$.*

By Theorem 4, in the filtering phase of our techniques, we first precalculate the neighborhood aggregates of each $v \in V(G)$ (a.k.a building the index of G). Once a query q is issued, we compute the neighborhood aggregates of each $u \in V(q)$ and retrieve an initial candidate list $C(u)$ containing all vertices $v \in V(G)$ such that $l(v) = l(u)$. We can filter v from $C(u)$ if $\delta(u, v) > \theta$; the finally obtained $C(u)$ is used as candidate set for Line 2 of Algorithm 2.

The space required to store all neighborhood aggregates is $O(|V(G)||\Sigma_V|)$, while the filtering cost for each $u \in V(q)$ is $O(|C(u)||\Sigma_V|)$.

4.2 Search Order

[13] proposes to assign a weight $\phi(u)$ ($\phi(u_1, u_2)$) to each vertex u (edge (u_1, u_2)) in q such that $\phi(u)$ ($\phi(u_1, u_2)$) is the occurrence of the vertices (edges) in G with the label $l(u)$ ($(l(u_1), l(u_2))$). Then, [13] picks the vertex u with the minimum $\phi(u)$ as the root $S[1]$ for all QIsequences enumerated by Algorithm 2, including the initial spanning tree (QIsequence). The initial spanning tree is generated by the PRIM algorithm [1]; that is, enforce the PRIM order.

The motivation to generate a QIsequence using such an ordered minimum spanning tree is to prune a non-promising search as early as possible; it is shown in [13] that selecting seq^* using the PRIM algorithm (i.e. iteratively enforcing the greedy constraint) is the most efficient way to find a subgraph isomorphic mapping. Although [13] aims to find one exact match for a QIsequence, such motivation is immediately applicable to similarity maximal all-matching since our algorithm and the algorithm in [13] are both based on iteratively extending a subgraph isomorphic mapping on the spanning tree of a QIsequence in a depth-first search fashion.

As discussed in Section 3.2, a generated seq by Algorithm 2 is a minimum spanning tree and follows the PRIM order in $(q - seq.R)$. Therefore, Algorithm 2 always adopts the best available subgraph isomorphism search strategy in [13] while excluding $seq.R$.

Dynamically Weighting. Our algorithm, Algorithm 2, generates alternative QIsequences on the fly and based on the current partial mappings. Below, we propose to assign weights to edges of q based on the results of our filtering technique in Section 4.1 and based on the current partial mapping. Particularly, we choose the vertex u in q with the smallest $|C(u)|$ as $S[1]$ where $C(u)$ is obtained by the filtering technique in Section 4.1 and then the initial QIsequence is chosen in the way described in Section 3.2 (i.e., the same way in[13]).

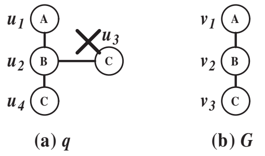
To replace $S[h].sEdge$ to conform the PRIM order (i.e., Line 11 of Algorithm 2), we choose an edge $(u, S[j])$ $(j < h)$ in q such that $\frac{|C(u)| \times \phi(u, S[j])}{\phi(u)}$ is minimized. The observation is as follows. As with [13], we assume that in G (1) the vertices with label $l(u)$ are uniformly distributed in all edges with label $(l(u), l(S[j]))$; (2) each vertex with label $l(u)$ has the same probability to appear in $C(u)$. With the above assumptions, for an edge $(u, S[j])$ $(j < h)$, we can estimate the number of possible mappings of $(u, S[j])$ by $\frac{|C(u)| \times \phi(u, S[j])}{\phi(u)}$ since the mapping on $S[j]$ is already obtained.

5. ALLOWING MISMATCHED VERTICES

Our techniques can be immediately extended to cover the problem of finding all similarity maximal matches allowing mismatched vertices, defined in Section 2. Firstly, our filtering techniques in Section 4.1 are immediately applicable.

Algorithm 2 needs to be carefully extended to deal with mismatched vertices in q since q is allowed to be cut into several disconnected parts by disabling edges in q. The basic idea is to execute the modified Algorithm 2 in multiple rounds. In round-1, we fix $seq^*.S[1]$ as the head and conduct a modified Algorithm 2 to compute all the results with matched $S[1]$, while allowing to mismatch any other $seq^*.S[h]$ $(h > 1)$ as long as the threshold constraint θ still holds. In round-2, we first mark $seq^*.S[1]$ as a *must-missing* vertex and $seq^*.S[2]$ as a *must-matching* vertex, while allowing to mismatch any other $seq^*.S[h]$ $(h > 2)$. In round-3, we enforce that $seq^*.S[1]$ and $seq^*.S[2]$ as *must-missing* vertices and $seq^*.S[3]$ as a *must-matching* vertex. Continuing this till we get all results.

The key part is to carefully deal with a missing vertex u. Simply putting all edges of a missing vertex into $seq.R$ to enforce that the current partial mapping does not include any edge in $seq.R$ will miss results. For example, consider the q and G depicted in Figures 10(a) and (b), respectively. Assume the current QIsequence *seq* is (u_1, u_2, u_4) where u_3 needs to be mismatched and $\theta = 1$. Simply putting (u_2, u_3) into *seq.R* and dealing with *seq.R* in the same way as that in Algorithm 2 will miss the mapping $\{ u_1 \rightarrow v_1, u_2 \rightarrow v_2, u_4 \rightarrow v_3 \}$ and its induced maximal match $\{ (v_1, v_2), (v_2, v_3) \}$. This is because the depth-first search will stop at u_2 after discovering (u_2, u_3) can be mapped into G.



(a) q (b) G

Figure 10: Allowing Vertex Mismatch ($\theta = 1$)

To resolve the above issue, we put edges replaced to obtain the current *seq* in *seq.R* only if they are in *seq*, while the edges cutting *seq* from other parts in q are put in *seq.D*. We check whether or not the current partial mapping can be extended to map an edge $(seq.S[i], u) \in seq.D$ into G only when all vertices in *seq* with the same label of u are exhausted. Due to space limits, we omit the details in this paper and only provide the experiment results to illustrate the efficiency of our extended algorithm.

6. PERFORMANCE EVALUATION

We report our performance evaluation in this section by using the state of the art technique SAPPER as a benchmark algorithm. The following algorithms are implemented:

- TSpan: Our SimSearch algorithm based on the EnumerateOnDemand strategy (i.e., Algorithm 2) in Section 3.4 employing the neighborhood based filtering technique in Section 4.1 and the dynamic weighting strategy in Section 4.2.
- TSpanQI: Running TSpan by the weights of edges and vertices in q as proposed in [13] (also see Section 4.2).
- PrecTSpan: Replacing the EnumerateOnDemand strategy in TSpan by the EnumerateAll strategy (also see Section 3.4).
- NaïveTSpan: Computing similarity maximal matches induced by each spanning tree in QT separately; that is, run PrecTSpan for $|QT|$ times and feed PrecTSpan by one QIsequence every time.
- TSpanNF: Running TSpan without filtering.
- TSpan+: Using TSpan to compute all similarity maximal matches and then enumerate all feasible subgraphs of each maximal match (see Section 3.5).
- TSpanMV: The modified TSpan allowing mismatched vertices (see Section 5).

All algorithms are implemented in C++ and compiled with GNU GCC. We conduct the experiments on a PC with Intel Xeon 2.40GHz CPU and 4GB memory running Debian Linux. We obtain the binary code of SAPPER from the authors of [23].

We evaluate the performance of all algorithms on a real dataset by varying query settings, while synthetic datasets are used to vary data graph settings. Below are the details.

Real Dataset. The Human Protean Interaction Network (HPRD) (http://www.hprd.org/download) is used as the real data graph in this section, denoted by G_H. The data graph contains 9,460 vertices, 37,081 edges and 307 distinct vertex labels generated under the *Gene Ontology Term*.

Query Graphs. Based on G_H, we use the random walk to randomly generate the following query sets from G_H to study the impact of different query settings.

- **Varying** $|V(q)|$: We randomly generate 8 sets of query graphs, denoted by Q_5, Q_{10}, Q_{15}, Q_{20}, Q_{40}, Q_{60}, Q_{80} and Q_{100} where each query in Q_i has i vertices with an average vertex degree 4 (a **default** setting). Each Q_i ($i = 5, 10, ..., 100$) has 100 randomly generated queries which are all subgraphs of G_H.
- **Varying** $avg.deg(q)$: We also randomly generate 4 sets of query graphs, denoted by $Q_{d=3}$, $Q_{d=4}$, $Q_{d=5}$ and $Q_{d=6}$ where each query in $Q_{d=i}$ has the average vertex degree i with 60 vertices (a **default** setting). Each $Q_{d=i}$ ($i = 3, 4, 5, 6$) has 100 randomly generated queries which are all subgraphs of G_H.

Note that $Q_{60} = Q_{d=4}$ with the default setting $|V(q)| = 60$ and $avg.deg(q) = 4$.

Synthetic Dataset. A synthetic data graph G_S is randomly generated as follows. We first randomly generate a spanning tree and then randomly add edges to the spanning tree. Finally, we assign labels randomly to the vertices following the power law distribution [10]. Particulary, we choose *the exponent* of power law to be 3 and randomly map each label to a distinct number $l \in [1, ...|\Sigma_V|]$ to get its weight l^3. We use power law distributions since many real graphs follow a power law distribution.

The **default** settings of the graph G_S are: $|V(G)| = 10k$, $avg.deg(G) = 8$, and $|\Sigma_V| = 50$ (i.e. the number of labels is 50). Note that the smaller the number of labels, the more challenging. The following synthetic data graphs are generated to study the impact of various data graph settings.

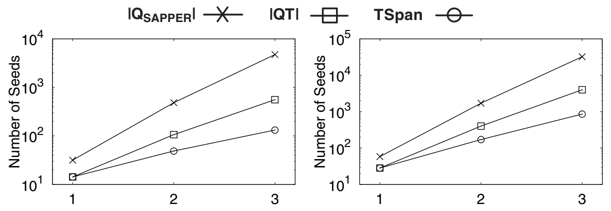
- **Varying** $|V(G)|$: We generate 5 data graphs denoted by G_{5k}, G_{10k}, G_{20k}, G_{40k} and G_{80k} where each G_{ik} has ik vertices with the default settings of $avg.deg(G)$ and $|\Sigma_V|$.
- **Varying** $avg.deg(G)$: We generate 5 data graphs denoted by $G_{d=4}$, $G_{d=8}$, $G_{d=12}$, $G_{d=16}$ and $G_{d=20}$ where each $G_{d=i}$ has an average degree of i with the default settings of $|V(G)|$ and $|\Sigma_V|$.
- **Varying** $|\Sigma_V|$: We generate 4 data graphs denoted by $G_{L=20}$, $G_{L=50}$, $G_{L=100}$, $G_{L=200}$ where each $G_{L=i}$ contains i distinct vertex labels with default settings of $|V(G)|$ and $avg.deg(G)$.

A set of 100 randomly selected subgraphs q of G_S is also generated as the query graphs with the default settings on $|V(q)|$ (60) and $avg.deg(q)$ (4).

Default θ Value. The default similarity threshold is $\theta = 2$.

6.1 Our Search Paradigms Against SAPPER

We evaluate TSpan, NaïveTSpan and TSpan+ against SAPPER on G_H and G_S (with default data graph settings). While TSpan+ and SAPPER compute all similarity matches, TSpan and NaïveTSpan only compute similarity maximal matches. As SAPPER is slow, we randomly generate a set of 100 subgraphs of G_H and G_S as the queries, respectively. Here, each query for G_H and G_S has 20 and 30 vertices, respectively. Note that we use query graphs with 20 vertices against G_H because SAPPER can not terminate in two days for a single query with 30 vertices against G_H.

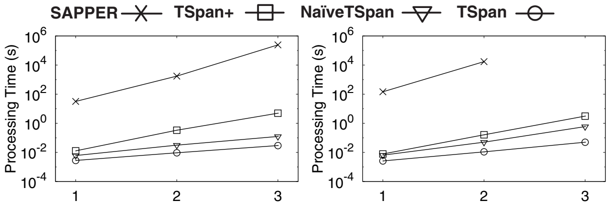


(a) Varying θ on G_H (b) Varying θ on G_S

Figure 11: Number of Seeds

Number of Seed Graphs. While the numbers of seeds generated by SAPPER and NaïveTSpan (i.e, $|Q_{\mathsf{SAPPER}}|$ and $|QT|$) are irrelevant to data graphs, the number of seeds (spanning trees) generated by TSpan depends on data graphs. Figure 11(a) and 11(b) report the average number of enumerated seeds for TSpan, NaïveTSpan and SAPPER, respectively for various θ. Note that $|Q_{\mathsf{SAPPER}}|$ is significantly larger than $|QT|$ (i.e., for up to 7.5 and 8.1 times for query graphs generated from G_H and G_S, respectively). By the EnumerateOnDemand strategy, TSpan can significantly reduce the size of QT; our results demonstrate that TSpan leads to the reduction of $|QT|$ by up to 76% and 79% regarding the queries generated from G_H and G_S, respectively.

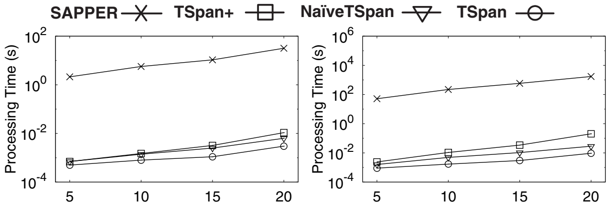
Query Processing Time. Figures 12(a) and 12(b) plot the average query processing time per query regarding various θ. Note that we do not report the results of SAPPER on G_S for $\theta = 3$ as SAPPER fails to terminate in two days. While generating the same results, TSpan+ improves SAPPER by up to 5 orders of magnitude over both G_H and G_S, even excluding the case of failing to terminate. Against SAPPER, TSpan achieves a speed-up by up to 7 orders of magnitude, even excluding the case of failing to terminate.



(a) Varying θ to query G_H (b) Varying θ to query G_S

Figure 12: Total Processing Time

Figures 12(a) and 12(b) also show that TSpan improves NaïveTSpan by up to 4 and 14 times on G_H and G_S, respectively, due to a smaller number of generated seeds to conduct all-matching and computation sharing. Although both NaïveTSpan and SAPPER adopt a naïve strategy to conduct all-matching, NaïveTSpan is 6 orders of magnitude faster than SAPPER, even excluding the case $\theta = 3$ on G_S; this is because (1) fewer seeds are generated, (2) only maximal matches are computed; and (3) conducting all-matching against trees is less expensive conducting it against graphs.



(a) $\theta = 1$ on G_H (b) $\theta = 2$ on G_H

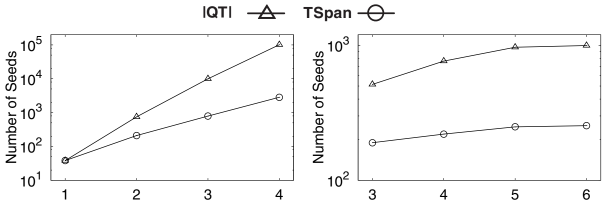
Figure 13: Varying $|V(q)|$ to query G_H

Varying Query Graph Size. To further demonstrate the efficiency of our algorithms, we compare TSpan, NaïveTSpan and TSpan+ with SAPPER when δ is small and $|V(q)|$ is small. We vary $|V(q)|$ from 5 to 20 to query the real data graph G_H. Figure 13(a) and 13(b) report the average query processing time of all algorithms when $\theta = 1$ and 2, respectively. TSpan+ outperforms SAPPER from 3 to 5 orders of magnitude over various $|V(q)|$ settings. It also shows that TSpan and NaïveTSpan further improve the query processing time of TSpan+ for generating maximal matches only.

In the rest of the section, we will exclude the evaluations of SAPPER and NaïveTSpan as they are not competitive against TSpan. We will also exclude the evaluations of TSpan+ as it is always more expensive than TSpan and we aim to compute similarity maximal matches.

6.2 EnumerateOnDemand vs EnumerateAll

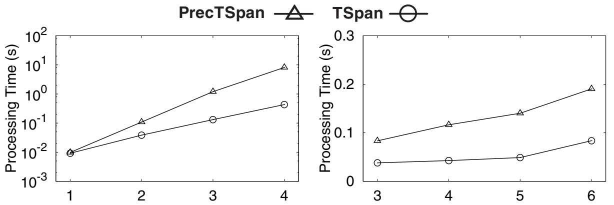
We next evaluate TSpan and PrecTSpan by varying θ and $avg.deg(q)$ to query G_H. The default value for $|V(q)|$ is 60.



(a) Varying θ (b) Varying $avg.deg(q)$

Figure 14: Number of Seeds

Number of Seed Graphs. We report the number of seed graphs of TSpan and PrecTSpan (i.e. generate the whole QT) in Figure 14(a) and 14(b), respectively. Clearly, the difference between TSpan and $|QT|$ increases when $|V(q)|$, θ, and $avg.deg(q)$ increase. For example, TSpan enumerates only 2.7% of the spanning trees in QT for $\theta = 4$.



(a) Varying θ (b) Varying $avg.deg(q)$

Figure 15: Total Processing Time

Query Processing Time. The evaluation results are reported in Figure 15(a) and 15(b). Due to the EnumerateOnDemand strategy, TSpan significantly improves PrecTSpan for up to 26 times over various θ settings. Interestingly, PrecTSpan only doubles the running time of TSpan on various $avg.deq(q)$; this shows that the ratio between TSpan and PrecTSpan is less sensitive to $avg.deg(q)$, compared with θ.

Note that PrecTSpan consumes significantly more memory than TSpan since PrecTSpan generates much more spanning trees than TSpan. In fact, PrecTSpan needs more than 10GB memory for conducting the experiment in Figure 15(a) regarding $\theta = 4$. Thus, we conduct the experiment in Figure 15(a) on another PC with AMD 800MHz CPU and 100GB memory running Ubuntu Linux. We also notice that TSpan is much more scalable regarding the memory usage (e.g., 16MB, 20MB, 36MB, 176MB for $\theta = 1$ to 4, respectively). While TSpan is guaranteed not slower than PrecTSpan, it is significantly faster than PrecTSpan in practice. Thus, we exclude PrecTSpan from further evaluation.

6.3 Evaluating Filtering Technique

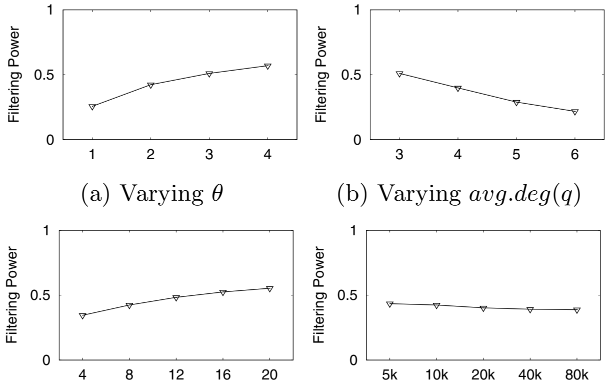
Indexing Cost. The evaluation of index time and size is reported in Table 1 by varying $|V(G)|$ and $avg.deg(G)$. Both index size and time grow linearly with $|V(G)|$ and $avg.deg(G)$ for generating neighborhood aggregates of data graph vertices. They confirm that our index construction costs are low in terms of both index size and time.

Table 1: Index Size and Index Time

avg. deg(G)	4	8	12	16	20
Index Size (MB)	0.30	0.49	0.63	0.74	0.83
Index Time (s)	0.15	0.24	0.30	0.34	0.40

|V(G)|	5k	10k	20k	40k	80k
Index Size (MB)	0.26	0.49	0.89	1.72	3.41
Index Time (s)	0.11	0.24	0.50	1.25	3.12

Filtering Power. The evaluations of the filtering power record the average ratio of size of candidate set for each vertex u in q over that of the set of vertices in G with the same label of u. Lower ratio indicates stronger filtering power.



(a) Varying θ

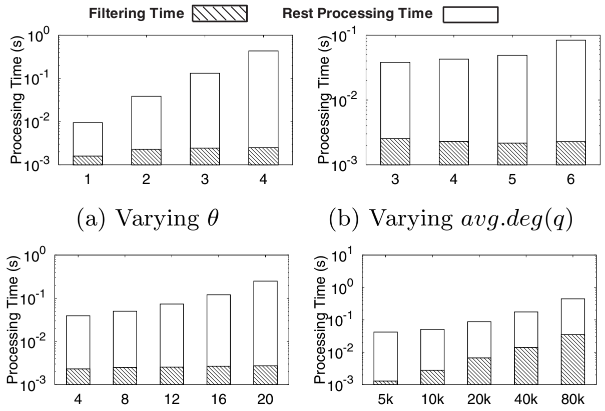
(b) Varying $avg.deg(q)$

(c) Varying $avg.deg(G)$

(d) Varying $|V(G)|$

Figure 16: Filtering Power

Figures 16(a) and 16(b) report our results against G_H, while Figures 16(c) and 16(d) report our results over synthetic data graphs. Clearly, the filtering power degrades when θ and $avg.deg(G)$ increase. However, the filtering power enhances when $avg.deg(q)$ increases. It confirms that the filtering power is not sensitive to the growth of $|V(G)|$.



(a) Varying θ

(b) Varying $avg.deg(q)$

(c) Varying $avg.deg(G)$

(d) Varying $|V(G)|$

Figure 17: Filtering Efficiency

Efficiency. The evaluations of the efficiency of our filtering technique are reported in Figure 17(a) to 17(d). They show the break-down information in TSpan regarding filtering time and the time to conduct Algorithm 2 in TSpan. It shows that the filtering cost increases significantly with the increase of data graph size and becomes a quite significant factor (about 10% of the total costs).

The next experiment shows the efficiency of TSpan on large-scale synthetic data graphs. The default settings of large-scale data graphs are: $|V(G)| = 10M(M = Million)$, $avg.deg(G) = 8$ and $|\Sigma_V| = 200$. The default settings of query graphs are: $|V(q)| = 60$ and $avg.deg(q) = 4$.

Table 2: Processing Time on Large-scale Datasets

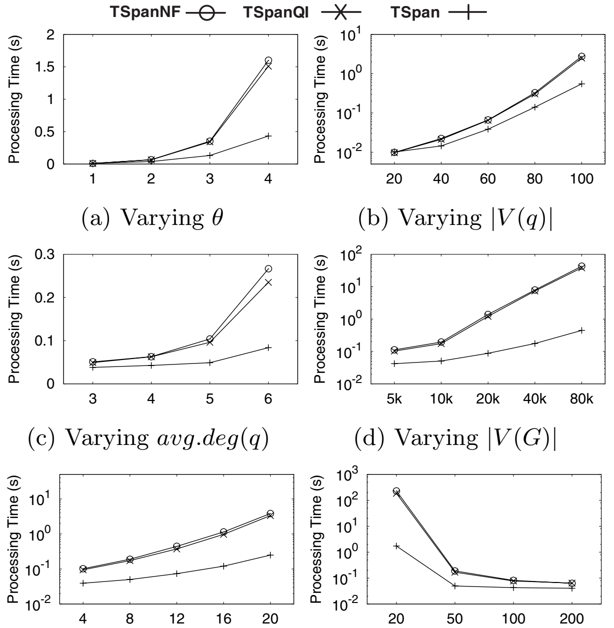
Varying $\|V(G)\|$	0.1M	0.5M	1M	1.5M	
Processing Time (s)	0.06	0.15	0.32	0.51	(a)

Varying avg.deg(G)	4	8	12	16	20	
Processing Time (s)	0.28	0.32	0.45	0.93	2.99	(b)

Varying $\|V(q)\|$	20	40	60	80	100	
Processing Time (s)	0.11	0.21	0.32	0.40	0.57	(c)

Varying θ	1	2	3	4	
Processing Time (s)	0.21	0.32	0.83	6.82	(d)

Table 2(a)-(d) show the query processing time of TSpan by varying various settings. Note that, to be scalable, our algorithms aim to make the total costs of TSpan as linear as possible in practice regarding $|V(G)|$ and $|E(G)|$, respectively, by terminating a false positive match as early as possible and maximizing the computation sharing. Table 2(a)-(b) show that the query processing time of TSpan has almost linear growth when $|V(G)|$ and $avg.deg(G)$ (i.e., $|E(G)|$) increase. In Table 2(c), as $|V(q)|$ increases, TSpan is also efficient and scalable. Table 2(d) demonstrates that the query processing time of TSpan may grow exponentially when θ increases.



(a) Varying θ

(b) Varying $|V(q)|$

(c) Varying $avg.deg(q)$

(d) Varying $|V(G)|$

(e) Varying $avg.deg(G)$

(f) Varying $|\Sigma_V|$

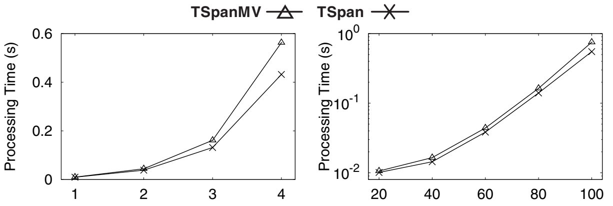
Figure 18: Total Processing Time

Effectiveness of Search Order. Finally, we evaluate the impact of search orders; that is, edge orders in QIsequences (see Section 4.2). We evaluate TSpan (with the dynamical weighting) against TSpanQI (with the weighting strategy in [13]) and TSpanNF (without filtering and thus with the weighting strategy in [13]).

Figures 18(a)-18(c) report our evaluation results against real data graphs, while Figures 18(d)-18(f) report our evaluation results against synthetic data graphs. They show that TSpan significantly outperforms TSpanQI and TSpanNF in all cases, and demonstrate that 1) the search order is very important, and 2) our filtering technique plays a more significant role in determining a search order. Interestingly, TSpanQI and TSpanNF have very close performance though earlier evaluation results show that our filtering technique has a significant filtering power. This shows that the spanning tree based search to iteratively prune a non-promising search can detect and prune non-promising vertices in G as effectively as the proposed filtering technique, especially when $avg.deg(q)$ is not large.

6.4 Allowing Mismatched Vertices

We now evaluate TSpanMV, the extension of TSpan in Section 5 to conduct similarity maximal all-matching by allowing missing vertices. We conduct the evaluation against TSpan over real data graph G_H. The evaluation results are reported in Figures 19(a) and 19(b). They show that the two algorithms perform quite closely though TSpanMV conducts more complex computation than TSpan.



(a) Varying θ (b) Varying $|V(q)|$

Figure 19: Total Processing Time

In summary, our spanning tree based search paradigm significantly outperforms SAPPER. TSpan is the best choice.

7. CONCLUSIONS

In this paper, we propose a novel search paradigm to conduct similarity (maximal) all-matching. It is based on enumerating a minimal set of spanning trees together with the edge exclusion set to generate all distinct similarity maximal matches. Compared with the state of the art technique SAPPER [23], our search algorithm not only leads to significantly fewer times of conducting exact all-matching computation but also reduces the complexity of exact all-matching from graphs to trees. We also present new techniques in filtering, search order, and computation sharing. A comprehensive performance evaluation on both real and synthetic datasets demonstrates that our algorithms outperform SAPPER by several orders of of magnitude.

8. ACKNOWLEDGEMENTS

Xuemin Lin is partially supported by ARC DP0987557, ARC DP110102937, ARC DP120104168 and NSFC61021004. Wenjie Zhang is partially supported by ARC DP120104168 and DE120102144. Jeffrey Xu Yu is partially supported by GRF CUHK/419109 under GRC of Hongkong, SAR.

9. REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. 1983.

[2] C. Chen, X. Yan, P. S. Yu, J. Han, D.-Q. Zhang, and X. Gu. Towards graph containment search and indexing. In *VLDB*, pages 926–937, 2007.

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

[4] L. Cordella, P.Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *3rd Workshop on Graph-based Representations in Pattern Recognition*, pages 149–159, 2001.

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

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

[7] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *ICDE*, page 38, 2006.

[8] H. Jiang, H. Wang, P. S. Yu, and S. Zhou. Gstring: A novel approach for efficient search in graph databases. In *ICDE*, pages 566–575, 2007.

[9] K. Klein, N. Kriege, and P. Mutzel. Ct-index: Fingerprint-based graph indexing combining cycles and trees. In *ICDE*, pages 1115–1126, 2011.

[10] M. E. J. Newman. Power laws, pareto distributions and zipf's law. *Contemporary Physics*, 46:323–351, 2005.

[11] S. Nijssen and J. N. Kok. A quickstart in frequent structure mining can make a difference. In *KDD*, pages 647–652, 2004.

[12] H. Shang, X. Lin, Y. Zhang, J. X. Yu, and W. Wang. Connected substructure similarity search. In *SIGMOD*, pages 903–914, 2010.

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

[14] D. Shasha, J. T.-L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, pages 39–52, 2002.

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

[16] N. Wang, J. Zhang, K.-L. Tan, and A. K. H. Tung. On triangulation-based dense neighborhood graphs discovery. *PVLDB*, 4(2):58–68, 2010.

[17] D. W. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *ICDE*, pages 976–985, 2007.

[18] X. Yan and P. S. Y. J. Han. Substructure similarity search in graph databases. In *SIGMOD*, pages 766–777, 2005.

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

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

[21] S. Zhang, J. Li, H. Gao, and Z. Zou. A novel approach for efficient supergraph query processing on graph databases. In *EDBT*, pages 204–215, 2009.

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

[23] S. Zhang, J. Yang, and W. Jin. Sapper: Subgraph indexing and approximate matching in large graphs. In *VLDB*, 2010.

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

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

[26] L. Zou, J. Mo, L. Chen, M. T. Ozsu, and D. Zhao. gstore: Answering sparql queries via subgraph matching. *PVLDB*, 2011.