

# Optimal Enumeration: Efficient Top-k Tree Matching

Lijun Chang<sup>†</sup>, Xuemin Lin<sup>‡</sup>, Wenjie Zhang<sup>†</sup>, Jeffrey Xu Yu<sup>‡</sup>, Ying Zhang<sup>§</sup>, Lu Qin<sup>§</sup>

<sup>†</sup>University of New South Wales, Australia, {ljchang, lxue, zhangw}@cse.unsw.edu.au

<sup>‡</sup>East China Normal University, China

<sup>‡</sup>The Chinese University of Hong Kong, China, yu@se.cuhk.edu.hk

<sup>§</sup>University of Technology, Sydney, Australia, {Ying.Zhang, Lu.Qin}@uts.edu.au

## ABSTRACT

Driven by many real applications, **graph pattern matching** has attracted a great deal of attention recently. Consider that a twig-pattern matching may result in an extremely large number of matches in a graph; this may not only confuse users by providing too many results but also lead to high computational costs. In this paper, we study the problem of **top-k tree pattern matching**; that is, **given a rooted tree  $T$ , compute its top-k matches in a directed graph  $G$  based on the twig-pattern matching semantics**. We firstly **present a novel and optimal enumeration paradigm based on the principle of Lawler's procedure**. We show that our enumeration algorithm runs in  $O(n_T + \log k)$  time in each round where  $n_T$  is the number of nodes in  $T$ . Considering that the time complexity to output a match of  $T$  is  $O(n_T)$  and  $n_T \geq \log k$  in practice, **our enumeration technique is optimal**. Moreover, the cost of generating top-1 match of  $T$  in our algorithm is  $O(m_R)$  where  $m_R$  is the number of edges in the transitive closure of a data graph  $G$  involving all relevant nodes to  $T$ .  $O(m_R)$  is also optimal in the worst case without pre-knowledge of  $G$ . Consequently, our algorithm is optimal with the running time  $O(m_R + k(n_T + \log k))$  in contrast to the time complexity  $O(m_R \log k + kn_T(\log k + d_T))$  of the existing technique where  $d_T$  is the maximal node degree in  $T$ . Secondly, **a novel priority based access technique is proposed**, which greatly reduces the number of edges accessed and results in a significant performance improvement. Finally, **we apply our techniques to the general form of top-k graph pattern matching problem (i.e., query is a graph) to improve the existing techniques**. Comprehensive empirical studies demonstrate that our techniques may improve the existing techniques by orders of magnitude.

## 1. INTRODUCTION

In many real applications, including social networks, information networks, collaboration networks, XML, web search, biology, biochemistry, etc., data are often modeled as graphs. With the proliferation of graph based applications, significant research efforts have been made towards many fundamental problems in managing and analysing graph data.

The problem of *top-k tree pattern matching* over graph data is investigated in [21]. That is, given a rooted tree  $T$  and a directed

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/3.0/>. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing [info@vldb.org](mailto:info@vldb.org). Articles from this volume were invited to present their results at the 41st International Conference on Very Large Data Bases, August 31st - September 4th 2015, Kohala Coast, Hawaii.

*Proceedings of the VLDB Endowment*, Vol. 8, No. 5  
Copyright 2015 VLDB Endowment 2150-8097/15/01.

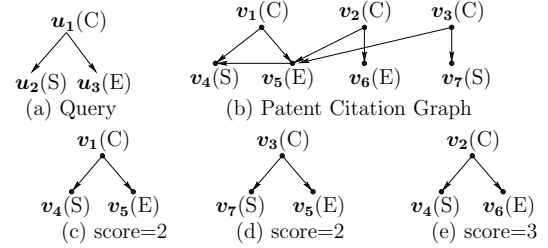


Figure 1: Example

graph  $G$ , find the  $k$  mappings from  $T$  to  $G$  such that in each mapping, the nodes in  $T$  are mapped to the nodes in  $G$  with the same labels, the edges in  $T$  are mapped to the shortest paths in  $G$  connecting the corresponding nodes, and the total lengths (scores) of the mapped shortest paths are minimized. For example, a rooted tree and a data graph are depicted in Figures 1(a) and 1(b), respectively, where the node labels are displayed in brackets. Assuming that each edge in the data graph takes the weight 1, Figures 1(c) and 1(d) give the top-1 and top-2 matches of the query graph with total scores 2 and 2, respectively, while there are totally 5 matches and the largest score is 3 - for example, see the match in Figure 1(e) where the shortest distance from  $v_2$  to  $v_4$  is 2.

The problem of *top-k tree pattern matching* is motivated by *twig-pattern matching queries* [32] over XML/Graph data where a twig query is typically a *rooted tree* [18] which has two types of edges,  $/$  and  $//$ , with the classical XML parent-child ( $/$ ) and ancestor-descendant ( $//$ ) matching semantics [32], respectively. In the example above, the data graph in Figure 1(b) is a tinny portion of a citation graph<sup>1</sup>, where each node  $v_i$  represents a patent and its label represents the discipline category to which the patent belongs. The categories C, E, and S stand for Computer Science, Economy, and Social Science, respectively. Each edge represents a citation from one patent to another (e.g., (C, E) represents that a patent in Computer Science is cited by a patent in Economy). Assume that the two edges in the twig query in Figure 1(a) are of the  $//$  type. Then, the twig query in Figure 1(a) is to find the triples of patents (nodes)  $x$ ,  $y$ , and  $z$  in the data graph with labels C, E, and S, respectively, such that for each triple  $(x, y, z)$  of nodes, there is a path from  $x$  to  $y$  and a path from  $x$  to  $z$ . Intuitively, the closer the citation relationship, the higher the impact; for example, a direct citation is preferred over an indirect citation. Thus, among all the results of the twig query in Figure 1(a), the triple  $(x, y, z)$  of nodes with the smallest total length of matched paths implies that the combination of patents  $y$  and  $z$  gets the highest impact from  $x$ . In the above example, the patent  $v_1$  has a higher impact on the combination of patents  $v_4$  and  $v_5$  (i.e., the match in Figure 1(c)) than the impact of  $v_2$  on the combination of  $v_4$  and  $v_6$  (i.e., the match in Figure 1(e)). Therefore, the top- $k$  tree pattern matching

<sup>1</sup><http://www.nber.org/patents>

can be used to identify the  $k$  results of a twig query with the highest preference/relevance among a possibly exponential number of results. Another example is that to launch a new product, a company may need to assemble a professional team with the people at different levels and having various designated skills. The goal is to assemble such a team so that people can work well with each other in the team. **The top- $k$  tree pattern matching can also serve for this purpose against the data graph based on LinkedIn or Xing (<https://www.linkedin.com/>, <https://www.xing.com/>). In such applications, a query tree may have tens of nodes. Due to space limits, we leave the details in the full paper [4].**

Efficiently computing top- $k$  tree pattern matches not only serves for the purpose of strengthening the relevance/preference of twig-pattern matching results but may also serve for the purpose of top- $k$  graph pattern matching where query is a general graph. As presented in [7], the techniques in [21] for computing top- $k$  tree pattern matches can be used as a key building block to retrieve the top- $k$  graph pattern matches by decomposing a query graph into a set of spanning trees. Motivated by these, in the paper we study the problem of efficient top- $k$  tree pattern matching, namely kTPM.

**Existing Approach.** The authors in [21] present an efficient dynamic programming based algorithm DP-B for kTPM with distinct node labels in a query  $T$ . It iteratively enumerates the top- $k$  matches starting from the top-1 match. To enumerate the top- $i$  ( $i \leq k$ ) match  $M_i$  of  $T$  from the top- $(i-1)$  match, DP-B runs in a *pull-down* fashion to *recursively* compute  $M_i$  to avoid visiting every node in  $G$ . With a priority queue of length (up to)  $k$  maintained at each node of  $G$  for the efficiency purpose, at each iteration DP-B runs in time  $O(d_u^2 + \log k)$  for each node  $u$  in  $T$ , where  $d_u$  is the degree of  $u$  in  $T$ . Thus, DP-B runs in time  $O(n_T(d_T + \log k))$  in each round of enumeration and runs in  $O(m_R \log k + kn_T(\log k + d_T))$  total time, where  $n_T$  is the number of nodes in  $T$ ,  $d_T$  is the maximal node degree in  $T$ , and  $m_R$  is the number of edges in the *run-time* graph that is a very small portion of the transitive closure of a data graph  $G$  induced by the edges in  $T$  (i.e., an edge  $(v_1, v_2)$  in the transitive closure of  $G$  is included in the run-time graph if and only if  $T$  has an edge  $(u_1, u_2)$  such that the labels of  $u_1$  and  $v_1$  are the same, and the labels of  $u_2$  and  $v_2$  are also the same).

**Our Approach.** **We propose a novel enumeration strategy based on the principle of Lawler's procedure.** To enumerate the top- $i$  result from the top- $(i-1)$  result  $M_{i-1}$ , it can be shown that we only need to **replace one node  $v$  in  $M_{i-1}$  with its "sibling" node in  $G$  to generate a new candidate**. Thus, there are totally at most  $n_T$  new candidates generated. Moreover, we can also show that in such  $n_T$  replacements, only one node needs to be replaced by its  $j$ th ( $j \leq k$ ) "best" sibling where  $j$  is dynamically changed (thus,  $O(\log k)$  time), while each of the others only needs to be replaced by its "best" sibling (thus,  $O(1)$  time). The best match (i.e., lowest score) among these newly generated candidates and the candidates generated in earlier rounds is the top- $i$  match, which can be done in  $O(\log k)$  time. Therefore, our enumeration runs in  $O(n_T + \log k)$  time instead of  $O(n_T(\log k + d_T))$  in [21]. Consequently, our algorithm runs in  $O(m_R + k(n_T + \log k))$  time in comparison to  $O(m_R \log k + kn_T(\log k + d_T))$  in [21]. Our algorithm is optimal regarding the worst case, considering that  $n_T \geq \log k$  in practice.

Later in Section 3, we will show that it is immediate  $m_R = \theta n_T$  on average, where  $\theta$  is the average number of edges of the same type in the transitive closure. Two edges,  $(u_1, u_2)$  and  $(v_1, v_2)$ , belong to the same type if and only if the labels of  $u_1$  and  $v_1$  are the same, and the labels of  $u_2$  and  $v_2$  are also the same. Thus, the time complexity of our algorithm is  $O(n_T(\theta + k) + k \log k)$ , while the algorithm DP-B in [21] runs in time  $O(n_T(\theta \log k + k \log k + kd_T))$ . Although  $\theta$  can be quadratic to the number of nodes in  $G$  in the worst case, it is small in practice (e.g., in DBLP data with 1.18 million nodes,  $\theta = 5900$ );

that is, our algorithm behaves linearly regarding  $n_T$ .

[21] also proposes an approach DP-P to run DP-B with a priority order aiming at reducing the number of edges accessed in the run-time graph such that DP-P always extends the partial match with the smallest current score. In this paper, we also extend our techniques by avoiding loading in all edges of a run-time graph. We develop a tighter trigger than that in DP-P to delay a loading of edges in a run-time graph. As a result, our second algorithm is orders of magnitude faster than the techniques in [21].

**Contributions.** Our main contributions are summarized as follows.

- **We propose a novel and optimal enumeration strategy** that leads to an optimal algorithm to compute kTPM in  $O(m_R + k(\log k + n_T))$  time in contrast to the existing techniques in [21] with  $O(m_R \log k + kn_T(\log k + d_T))$  running time.
- **We develop novel pruning techniques to reduce the number of retrieved edges from a run-time graph (i.e., reduce  $m_R$ ) and to compute top- $k$  matches in a priority order, which significantly speed up the computation.**
- While being immediately applicable to obtaining the top- $k$  results of a twig query with unique node labels and '/' semantics, the above techniques can also be immediately extended to support obtaining the top- $k$  results of a general form of twig queries, as well as to significantly improve the performance of the techniques [7] for top- $k$  graph pattern matching.

We conduct empirical studies on large real and synthetic graphs. Extensive performance studies demonstrate that the proposed algorithms significantly outperform the state-of-the-art algorithms in [21] by several orders of magnitude.

**Organization.** The rest of the paper is organized as follows. A brief overview of related work is given below. Section 2 provides the problem definition. Section 3 presents our new enumeration paradigm, while Section 4 presents our second algorithm aiming to reduce the access of a run-time graph. Extensions of our techniques are discussed in Section 5, followed by experimental results in Section 6. Finally, a conclusion is given in Section 7. *Proofs are omitted due to space limits and can be found in the full version [4].*

**Related Work.** Top- $k$  query answering for relational data has been extensively studied. The first fundamental work may be found in [12], and a survey is presented in [25]. Various top- $k$  queries have also been investigated over spatial data, such as kNN [16], RkNN [27], etc. Nevertheless, these techniques are not applicable to computing top- $k$  graph pattern matches [7, 21].

Computing various occurrences (or matches) of a structure  $q$  in a large data graph  $G$  has recently drawn a great deal of attention. It may be classified into three categories: 1) subgraph isomorphism, 2) pattern matching, and 3) graph simulation.

**1. Subgraph Isomorphism.** A subgraph isomorphism of  $q$  in  $G$  is a one-to-one mapping function  $f$  from nodes of  $q$  to nodes of  $G$ , which preserves the label information and the edge information; that is, an edge in  $q$  is mapped to an edge in  $G$  [30, 31, 22]. Subgraph isomorphism is NP-complete, and the existing approaches are based on a backtracking paradigm [31] combining with the techniques of matching ordering, connectivity enforcement, and the neighbourhood based compression [22, 30]. Similarity matching, which finds all approximate occurrences of  $q$  in  $G$  with the number of possible missing edges bounded by a given threshold, has been studied (e.g., [35]). All these techniques do not aim at ranked queries (i.e., results generated do not follow any designated order); thus, they are not applicable to the top- $k$  pattern matching problems by avoiding enumerating all results.

**2. Pattern Matching.** Subgraph isomorphism sometimes may be too restrictive to identify patterns in many other applications [7, 13, 21, 36]. *Pattern matching relaxes subgraph isomorphism by*

mapping edges of  $q$  to paths in  $G$ . The problem of enumerating all matches of a query graph has been studied regarding different types of query graphs, including a tree [5, 20, 34], a directed acyclic graph [5], and a general graph [6]. The problem of retrieving graph pattern matches by restricting each matching path to be within a given threshold is investigated in [36]. The problem of twig-pattern matching over XML is studied in [3, 19]. Nevertheless, the above techniques are not applicable to top- $k$  pattern matching queries since 1) they do not generate results according to any scoring function and/or 2) they only focus on the data graphs with a tree form (e.g., XML data in [3, 19]). The most related works are from [7, 21], where [21] studies the top- $k$  tree pattern matching and [7] studies top- $k$  graph pattern matching. In this paper, we study efficient top- $k$  tree matching and extend our techniques to top- $k$  graph matching, and we will show that our techniques significantly improve the existing techniques in [7, 21].

**3. Graph Simulation.** Another structure matching is bounded graph simulation [14]. In the model of bounded graph simulation, the result is a binary relation,  $R \subseteq V_q \times V_G$  where  $V_q$  and  $V_G$  are the sets of nodes of  $q$  and  $G$ , respectively. Finding the top- $k$  data nodes  $v$  in  $V_G$ , for a designated query node  $u$  in  $V_q$  such that  $(u, v) \in R$ , is studied in [15]. Nevertheless, due to inherently different problem natures, these techniques are inapplicable to our top- $k$  pattern matching problems.

**Others.** Generalizations of graph homomorphism [24] and graph isomorphism [17] may be found in [13], which also map edges in a graph to paths in another graph. Nevertheless, these are based on the whole matching of two graphs rather than the occurrences of one subgraph in another. Another formula of evaluating the top- $k$  twig query on a graph may be found in [29] which maps a rooted tree to a *steiner tree* in data graph and ranks the results based on the total weights of the resulted steiner trees; this is inherently different from our top- $k$  tree matching problem (NP-hard vs polynomial time solvable). Finally, finding the top- $k$  min-cost connected trees over a data graph for keyword search, which is NP-hard [11], is investigated in [2, 11], and a survey can be found in [33]. However, these techniques cannot be used to compute the top- $k$  tree matches defined in this paper since the problems are inherently different: NP-hard vs polynomial time solvable.

**Remark.** Below are the relationships among the above queries. A twig-pattern matching over XML in [3, 19] considers matches between two trees, while the subgraph isomorphism in [30, 31, 22] considers a one-to-one mapping from the nodes and edges of a query graph to the nodes and edges of a data graph. Meanwhile, a pattern matching in [5, 6, 20, 34] considers matches from a tree or a graph to a data graph by allowing edges in a query to be mapped to paths in a graph.

## 2. TREE PATTERN MATCHING

In this paper, we focus on a *node-labeled directed graph*  $G = (V, E, l)$  [18]; hereafter, a data graph always refers to a node-labeled directed graph unless otherwise specified. Here,  $V$  is the set of nodes,  $E$  is the set of edges, and  $l$  is a labeling function assigning each node  $v$  ( $\in V$ ) a label  $l(v)$  ( $\in \Sigma$ ), where  $\Sigma$  is a set of alphabets.

We use  $(v, v')$  to denote an edge from  $v$  to  $v'$  and  $(v, v')$  is referred to as an incoming edge to  $v'$  and an outgoing edge from  $v$ .  $V_G$  and  $E_G$  denote the node set and the edge set of graph  $G$ , respectively. A *rooted tree* [18] is a directed tree with only one node that does not have incoming edges, namely the root, such that there is always a directed path from the root to each leaf node. Figures 2(a) and 2(b) illustrate a rooted tree and a general directed graph, respectively, where node labels are demonstrated in the brackets.

**Definition 2.1: [Tree Pattern Matching]** Given a rooted tree  $T$

Notation	Description
$T/G/G_R$	a query tree / graph / run-time graph
$n_T/n_R/m_R$	#nodes in $T$ / #nodes in $G_R$ / #edges in $G_R$
$d_T/d_R$	maximum node degree in $T$ / in $G_R$
$u \in T/v \in G$	a node in $T$ / in $G$
$l(u)/l(v)$	Label of $u$ / label of $v$
$\delta_{\min}(v, v')$	the shortest distance from $v$ to $v'$ in $G$
$v.children$	the set of all children of $v$ in $G_R$
$v.children_\alpha$	the set of children with label $\alpha$ of $v$ in $G_R$
$M$	a match of $T$ in $G$
$T_u$	subtree of $T$ rooted at $u$
$q(v)$	the node in $T$ that is mapped to $v$ in a match
$bs(v)$	the lowest score of a match of $T_{q(v)}$ containing $v$
$L_{v,\alpha}, H_{v,\alpha}$	data structures to maintain nodes in $v.children_\alpha$

**Table 1: Frequently used notations**

and a data graph  $G$ , a *tree pattern matching*  $f$  is a mapping from  $V_T$  to  $V_G$  such that 1)  $f$  preserves the label information (i.e.,  $\forall u \in V_T, l(u) = l(f(u))$ ); 2)  $f$  preserves the structure information (i.e.,  $\forall (u, u') \in E_T$ , there is a directed path from  $f(u)$  to  $f(u')$  in  $G$ ).  $\square$

In a matching  $f$ , each edge  $(u, u')$  in  $T$  is mapped to a path from  $f(u)$  to  $f(u')$  in  $G$ . Note that there could be many paths from  $f(u)$  to  $f(u')$  and we use the *length of shortest paths* to characterize the relevance between  $T$  and the match  $M_f = \{f(u) \mid u \in V_T\}$  of  $T$  in  $G$ .

**Definition 2.2:[Penalty Score]** Given a tree pattern match  $M_f = \{f(u) \mid u \in V_T\}$  of  $T$  in a graph  $G$ , we compute its *penalty score*  $S(M_f)$  as,

$$S(M_f) = \sum_{(u, u') \in E_T} \delta_{\min}(f(u), f(u')) \quad (1)$$

where  $\delta_{\min}(f(u), f(u'))$  is the shortest distance from  $f(u)$  to  $f(u')$  in  $G$ .<sup>2</sup>  $\square$

**Example 2.1:** Regarding the query and the data graph in Figures 2(a) and 2(b), respectively, the matching  $f$ , mapping  $(u_1, u_2, u_3, u_4, u_5)$  to  $(v_1, v_3, v_6, v_7, v_{10})$ , has the penalty score  $S(M_f) = 6$ , while the matching  $f_1$ , mapping  $(u_1, u_2, u_3, u_4, u_5)$  to  $(v_1, v_3, v_5, v_7, v_9)$ , has the penalty score  $S(M_{f_1}) = 4$ .  $\square$

**Problem Statement.** We use kTPM to denote the problem of top- $k$  tree pattern matching; that is, computing the  $k$  tree pattern matches with the lowest scores. In this paper, we study the problem of efficiently computing kTPM.

Note that as with [21], for presentation simplicity we assume that in a rooted tree (query graph)  $T$ , each node has a unique label (i.e., different nodes have different labels) and there are no wildcard (\*) nodes. This assumption makes a tree pattern matching  $f$  a one-to-one mapping from  $V_T$  to  $V_G$  though generally it is not a one-to-one mapping. In Section 5, we will show that our techniques can be immediately extended to cover a general case in which different nodes may have the same label and a node may be a wildcard node (\*). While the current definition of kTPM only covers the ‘//’ semantics, in Section 5 we will show that our techniques can be immediately extended to include the ‘/’ type edge. Finally, in Section 5 we will also show extensions of our techniques to top- $k$  graph pattern matching [7].

**Frequently used notations** are summarized in Table 1. Most of them will be defined in the following sections.

## 3. OPTIMAL ALGORITHM FOR kTPM

In this section, we present a novel enumeration paradigm for generating the top- $(l+1)$  match from the top- $l$  match. This leads to an optimal algorithm to compute kTPM. The rest of the section is organized as follows. We first present the notation and pre-computation. Then, we present our enumeration techniques, followed by implementation details and the time complexity analysis.

<sup>2</sup>Note that, our techniques directly apply if node weights are also considered in the penalty score.



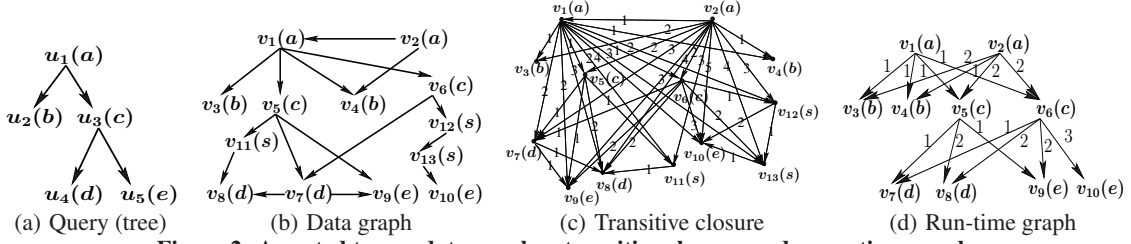


Figure 2: A rooted tree, a data graph, a transitive closure, and a run-time graph

### 3.1 Notation and Pre-Computation

We use  $T$  to denote a query graph which is a rooted tree,  $r(T)$  to denote the root of  $T$ ,  $T_u$  to denote the subtree of  $T$  rooted at  $u \in V_T$ , and  $n_T$  to denote the number of nodes in  $T$ . As stated earlier, in  $T$ , all node labels are distinct.

**Transitive Closure.** We pre-compute a transitive closure  $G_c = (V_c, E_c)$  of  $G = (V, E)$  by the techniques in [9] in  $O(n_G m_G)$  time, where  $n_G$  and  $m_G$  are the number of nodes and the number of edges in  $G$ , respectively. In  $G_c$ ,  $V_c = V$ , and an edge  $(v, v')$  exists if and only if there is a path in  $G$  from  $v$  to  $v'$ . We also record the length of shortest path from  $v$  to  $v'$  in  $G$  as the weight of  $(v, v')$  in  $G_c$ .

**Run-Time Graph.** Instead of loading in  $G_c$  to main-memory, the authors in [21] propose to load in a subgraph of  $G_c$ , called the *run-time graph*  $G_R$  regarding  $T$ .  $G_R$  is defined as follows.  $G_R$  consists of the edges  $(v, v')$  in  $G_c$  induced by  $T$ ; that is, an edge  $(v, v')$  in  $G_c$  is included in  $G_R$  if and only if there is an edge  $(u, u')$  in  $T$  with  $l(u) = l(v)$  and  $l(u') = l(v')$ . We also store the weight (length of shortest paths) of an edge in  $G_R$ . Clearly, finding the top- $k$  matches of  $T$  in  $G$  (or  $G_c$ ) is equivalent to finding the top- $k$  matches in  $G_R$ .

Figure 2(c) shows the transitive closure of the data graph in Figure 2(b), and Figure 2(d) shows the run-time graph regarding  $T$  in Figure 2(a). We denote the number of nodes in  $G_R$  by  $n_R$  and the number of edges in  $G_R$  by  $m_R$ . It is immediate that  $m_R$  is quadratic to  $n_G$  in the worst case but  $m_R = \theta n_T$  on average, where  $\theta$  is the average number of edges of the same type (i.e., with the same pair of labels) in the transitive closure. We denote the set of child nodes of a node  $v$  in  $G_R$  by  $v.children$ . Regarding each non-leaf node  $v \in G_R$  (i.e.,  $v$  has outgoing edges), we group the child nodes of  $v$  in  $G_R$  into different groups such that node labels in each group are the same, and node labels across different groups are different; particularly,  $v.children_\alpha$  denotes the child nodes of  $v$  in  $G_R$  with the label  $\alpha$ . For example, in Figure 2(d),  $v_1.children_c = \{v_5, v_6\}$ .

Note that we do not need to pre-compute  $G_R$ . Instead, a run-time graph  $G_R$  can be identified at query-time from  $G_c$  if edges in  $G_c$  are organized in tables as follows.

**Run-Time Graph Identification.** Similar to [7, 21], for each pair of node labels  $\alpha, \beta \in \Sigma$  in  $G$ , we store in table  $L_{\beta}^{\alpha}$  all the triples  $(v_i, v_j, \delta_{min}(v_i, v_j))$ , where  $l(v_i) = \alpha$ ,  $l(v_j) = \beta$ ,  $v_i$  can reach  $v_j$  in  $G$ , and  $\delta_{min}(v_i, v_j)$  is the shortest distance from  $v_i$  to  $v_j$  in  $G$ . Then the run-time graph can be immediately identified at query-time and loaded in to main memory in linear I/O time regarding the run-time graph size, by reading the tables corresponding to edges in  $T$  from disk to main memory.

Note that we also discuss techniques to avoid computing and storing the entire transitive closure, and to assemble only the needed part of run-time graph on-demand, in Section 5 and Section 4, respectively.

### 3.2 Efficient Enumeration

Our enumeration techniques are based on Lawler's procedure.

**Lawler's Procedure.** The basic idea is as follows. Starting from the entire "solution space", it iteratively divides a solution space into disjoint solution subspaces, where the tree pattern match with

the lowest penalty score in a solution subspace is called the *best match* in the solution subspace. At the  $l$ th round, the tree pattern match  $M_l$  with lowest penalty score among the best tree pattern matches in each remaining solution subspace, respectively, is the top- $l$  match. The iteration continues by dividing the solution subspace, from which  $M_l$  is generated, into disjoint solution subspaces excluding  $M_l$ . Lawler's procedure works as follows for kTPM.

Suppose that  $V_T = \{u_1, u_2, \dots, u_{n_T}\}$  is the node set of  $T$ . For each node  $u_i \in V_T$ , let  $V_i$  denote the set of nodes in  $G_R$  with the label  $l(u_i)$ . The entire solution space, consisting of all tree pattern matches of  $T$  in  $G_R$ , is a subset of  $S = V_1 \times V_2 \times \dots \times V_{n_T}$ . Suppose that  $M_1 = (v_1, v_2, \dots, v_{n_T})$  is the best match (top-1 match) of  $T$  in  $G_R$ ; that is, for  $1 \leq i \leq n_T$ ,  $u_i$  is mapped to  $v_i$ . To obtain the top-2 match,  $S$  is divided into  $n_T$  disjoint subspaces based on  $M_1$ :  $S_1 = (V_1 - \{v_1\}) \times V_2 \times \dots \times V_{n_T}$ ,  $S_2 = \{v_1\} \times (V_2 - \{v_2\}) \times V_3 \times \dots \times V_{n_T}$ ,  $\dots$ ,  $S_{n_T} = \{v_1\} \times \{v_2\} \times \dots \times \{v_{n_T-1}\} \times (V_{n_T} - \{v_{n_T}\})$ . Clearly,  $\{M_1\}, S_1, \dots, S_{n_T}$  are mutually disjoint, and their union is  $S$  which is no longer kept. Then, the top-2 match of  $T$  in  $G_R$  is the match with the lowest score among the best matches in these  $n_T$  subspaces  $\{S_j \mid 1 \leq j \leq n_T\}$ , respectively. Assume that the top-2 match  $M_2$  of  $T$  is obtained from  $S_i$ , then the first  $(i-1)$  nodes in  $M_2$  must be  $(v_1, \dots, v_{i-1})$ . To obtain the top-3 match, the procedure continues to further divide  $S_i = \{v_1\} \times \dots \times \{v_{i-1}\} \times (V_i - \{v_i\}) \times V_{i+1} \times \dots \times V_{n_T}$  into  $(n_T - i + 1)$  disjoint subspaces by fixing  $(v_1, \dots, v_{i-1})$  and dividing  $\{(V_i - \{v_i\}), V_{i+1}, \dots, V_{n_T}\}$  one by one in a similar way to that in obtaining the top-2 match.

Generally, suppose that the top- $l$  match  $M_l$  is obtained from a subspace  $S'_j = \{v_{l_1}\} \times \dots \times \{v_{l_{j-1}}\} \times (V_j - U_j) \times V_{j+1} \times \dots \times V_{n_T}$ , where  $U_j$  denotes the subset of  $V_j$  to be excluded in obtaining  $M_l$  for  $1 \leq i \leq l$ ,  $(v_{l_1}, \dots, v_{l_{j-1}})$  are fixed in the subspace  $S'_j$  and mapped from  $(u_{l_1}, \dots, u_{l_{j-1}})$ . Note that we use  $l_j$  to denote the subscript of a node in  $G_R$ . Suppose that  $M_l$  is  $\{v_{l'_1}, \dots, v_{l'_n}\}$ , then  $v_{l'_j} \notin U_j$  and for  $1 \leq x \leq j-1$ ,  $v_{l'_x} = v_{l_x}$  (i.e.,  $l'_x = l_x$ ) since  $M_l$  is obtained from  $S'_j$ . In order to compute the top- $(l+1)$  match,  $S'_j$  is further divided into  $(n_T - j + 1)$  subspaces:  $\{v_{l_1}\} \times \dots \times \{v_{l_{j-1}}\} \times (V_j - U_j - \{v_{l'_j}\}) \times V_{j+1} \times \dots \times V_{n_T}$ ,  $\{v_{l_1}\} \times \dots \times \{v_{l_{j-1}}\} \times \{v_{l'_j}\} \times (V_{j+1} - \{v_{l'_{j+1}}\}) \times V_{j+2} \times \dots \times V_{n_T}$ ,  $\dots$ ,  $\{v_{l_1}\} \times \dots \times \{v_{l_{j-1}}\} \times \{v_{l'_j}\} \times \dots \times \{v_{l'_{n_T-1}}\} \times (V_{n_T} - \{v_{l'_{n_T}}\})$ . Clearly,  $\{M_l\}$  and these newly generated  $(n_T - j + 1)$  subspaces are disjoint, and their union is  $S'_j$ . Assume there are  $N$  subspaces left in total; that is, the subspaces divided in generating the top-1, 2, ...,  $l$  matches but not used for any of those top- $l$  matches. Then, the top- $(l+1)$  match is the one with the lowest score among the best matches, respectively, in those  $N$  subspaces and the newly divided  $(n_T - j + 1)$  subspaces.

**Example 3.1:** Consider the query tree in Figure 2(a) over the run-time graph in Figure 2(d).  $V_1 = \{v_1, v_2\}$ ,  $V_2 = \{v_3, v_4\}$ ,  $V_3 = \{v_5, v_6\}$ ,  $V_4 = \{v_7, v_8\}$ ,  $V_5 = \{v_9, v_{10}\}$ . The top-1 match of  $T$  is  $(v_1, v_3, v_5, v_7, v_9)$ ; thus, in the expression of  $(v_1, v_2, \dots, v_{15})$ ,  $l_1 = 1, l_2 = 3, l_3 = 5, l_4 = 7$ , and  $l_5 = 9$ . Then, the entire solution space is divided into 5 subspaces,  $S_1 = (V_1 - \{v_1\}) \times V_2 \times \dots \times V_5$ ,  $S_2 = \{v_1\} \times (V_2 - \{v_3\}) \times V_3 \times \dots \times V_5$ ,  $\dots$ ,  $S_5 = \{v_1\} \times \dots \times \{v_7\} \times (V_5 - \{v_9\})$ . The top-2 match of  $T$  is  $(v_1, v_4, v_5, v_7, v_9)$ , thus, in the expression of  $(v_1, v_2, \dots, v_{25})$ ,  $2_1 = 1, 2_2 = 4, 2_3 = 5, 2_4 = 7$ , and  $2_5 = 9$ , which is the best match in  $S_2$ .  $S_2$  is further divided into 4

subspaces,  $S'_1 = \{v_1\} \times (\mathbb{V}_2 - \{v_3, v_4\}) \times \mathbb{V}_3 \times \dots \times \mathbb{V}_5$ ,  $S'_2 = \{v_1\} \times \{v_4\} \times (\mathbb{V}_3 - \{v_5\}) \times \mathbb{V}_4 \times \mathbb{V}_5$ ,  $\dots$ , and  $S'_4 = \{v_1\} \times \{v_4\} \times \{v_5\} \times \{v_7\} \times (\mathbb{V}_5 - \{v_9\})$ . Then, the top-3 match of  $T$  is the one with lowest score among the best matches, respectively, in subspaces  $S_1, S_3, S_4, S_5, S'_2, S'_3, S'_4$  where  $S'_1 = \emptyset$  and is excluded from further consideration; in this case, the top-3 match of  $T$  is  $(v_1, v_3, v_5, v_8, v_9)$  obtained from  $S_4$ .  $\square$

In [28], it shows that Lawler's procedure can correctly generate the top- $k$  solutions in  $O(k(\log k + n_T t(n_T)))$  time, where  $t(n_T)$  is the time to compute the best solution in a subspace. If we use the techniques in [21] to compute the best match in a subspace, then  $t(n_T) = O(m_R)$ ; thus, an immediate application of Lawler's procedure for generating top- $k$  matches runs in time  $O(kn_T m_R)$  and is much more expensive than the techniques of DP-B and DP-P [21]. Below, we show that obtaining the best match in a subspace only needs to replace one node; consequently, we can achieve  $O(n_T t(n_T)) = O(n_T + \log k)$  once the top-1 match is computed.

**Replacing with Connected Nodes Only.** To make an execution more efficient in applying Lawler's procedure, we need the property in Lemma 3.1 below. This requires the nodes in  $T$  to be sorted in a sequence in a top-down and breadth-first fashion. That is, the root node of  $T$  is put the first, followed by its children, then move to the next level. This can be done in linear time,  $O(n_T)$ . For example, the nodes in Figure 2(a) are sub-indexed/ordered in this fashion; that is,  $u_1, u_2, u_3, u_4$ , and  $u_5$  are in such an order. The following lemma, Lemma 3.1, is immediate.

**Lemma 3.1:** Suppose that the nodes,  $u_1, \dots, u_{n_T}$ , of  $T$  are ordered in a top-down and breadth-first fashion. Then, the parent  $u_j$  of  $u_i$  must have the property such that  $j < i$ .  $\square$

Below, we characterize the property of best match in a newly generated subspace where the property in Lemma 3.1 will be used. Note that in Lawler's procedure, the top- $(l+1)$  solution (match) is obtained against two types of subspaces: type 1) the previously divided subspaces that have not contributed to any top- $i$  solution (match) for  $i \leq l$ , type 2) the newly divided subspaces using the top- $l$  solution. For example, in computing the top-3 match in Example 3.1,  $S_1, S_3, S_4, S_5$  are type 1 subspaces, and  $S'_1, S'_2, S'_3, S'_4$  are type 2 subspaces. The top- $(l+1)$  solution (match) is the one with lowest score among all best matches in these type 1 and type 2 subspaces. Assume the best match in each type 1 subspace has already been computed, the key for computing the top- $(l+1)$  match is to efficiently compute the best match in each of these newly generated subspaces; that is, type 2 subspaces.

As presented above, each newly generated subspace (i.e., type-2 subspace) by the top- $l$  match  $M_l = (v_{l_1}, v_{l_2}, \dots, v_{l_{n_T}})$  in Lawler's procedure belongs to one of the following two cases, where  $l > 1$ .

Case 1:  $S' = \{v_{l_1}\} \times \dots \times \{v_{l_{j-1}}\} \times (\mathbb{V}_j - U_j - \{v_{l_j}\}) \times \mathbb{V}_{j+1} \times \dots \times \mathbb{V}_{n_T}$ .

Case 2:  $S'' = \{v_{l_1}\} \times \dots \times \{v_{l_{x-1}}\} \times (\mathbb{V}_x - \{v_{l_x}\}) \times \mathbb{V}_{x+1} \times \dots \times \mathbb{V}_{n_T}$ .

Here,  $j+1 \leq x \leq n_T$ . Note that our enumeration techniques only deal with enumerating the top- $(l+1)$  match from the top- $l$  match; thus  $l \geq 1$ . When  $l = 1$ , all newly generated subspaces fall into Case 2; otherwise, Case 1 exists and  $U_j \neq \emptyset$ . There are only one newly generated subspace in Case 1 and  $(n_T - j)$  subspaces in Case 2. Below, Theorem 3.1 states that for the subspace  $S'$  in Case 1, we need to find the  $(|U_j| + 1)$ th best "sibling" node of  $v_{l_j}$  to replace  $v_{l_j}$  in  $M_l$  to obtain the best match in  $S'$ , while Theorem 3.2 states that for a subspace  $S''$  in Case 2, we only need to find the best "sibling" node of  $v_{l_x}$  to replace  $v_{l_x}$  in  $M_l$  to obtain the best match in  $S''$ . If no such siblings exist for a subspace in Case 1 or 2, then the subspace have no matches; this is stated in Lemma 3.2. Recall that  $v.children_\alpha$  denotes the set of child nodes of  $v$  in  $G_R$  with label  $\alpha$ .

**Lemma 3.2:** Regarding Case 1 above, suppose that  $v_{l_p} \in M_l$  is the parent of  $v_{l_j}$  and  $v_{l_p}.children_{l(v_{l_j})} = U_j \cup \{v_{l_j}\}$ , then there is no tree

pattern match in  $S'$  for  $T$ . Regarding Case 2 above, suppose that  $v_{l_y} \in M_l$  is the parent of  $v_{l_x}$  and  $v_{l_y}.children_{l(v_{l_x})} = \{v_{l_x}\}$ , then there is no tree pattern match in  $S''$  for  $T$ .  $\square$

We need the notations below to present Theorems 3.1 and 3.2. For each node  $v$  in  $G_R$ , we can always use  $q(v)$  to denote the node  $u$  in  $T$  that is mapped to  $v$  in a tree pattern match since node labels in  $T$  are distinct and  $l(u) = l(v)$ . At each node  $v \in G_R$ , we record  $bs(v)$ , the lowest score of a match of  $T_{q(v)}$  containing  $v$  in  $G_R$ , where  $T_{q(v)}$  is the subtree of  $T$  rooted at  $q(v)$ .

**Theorem 3.1:** Regarding Case 1, let  $v_{l_p}$  be the parent of  $v_{l_j}$  in  $M_l$  and there is at least a tree pattern match in  $S'$  for  $T$ , then the best match in  $S'$  is to replace  $v_{l_j} \in M_l$  with the node  $v \in v_{l_p}.children_{l(v_{l_j})}$  such that  $bs(v) + \delta_{\min}(v_{l_p}, v)$  has the  $(|U_j| + 2)$ th lowest value among all nodes in  $v_{l_p}.children_{l(v_{l_j})}$ .  $\square$

**Theorem 3.2:** Regarding Case 2, let  $v_{l_y}$  be the parent of  $v_{l_x}$  in  $M_l$  and there is a tree pattern match in  $S''$  for  $T$ , then the best match in  $S''$  is to replace  $v_{l_x} \in M_l$  with the node  $v \in v_{l_y}.children_{l(v_{l_x})}$  such that  $bs(v) + \delta_{\min}(v_{l_y}, v)$  has the second lowest value among all nodes in  $v_{l_y}.children_{l(v_{l_x})}$ .  $\square$

Regarding Case 2 (Theorem 3.2), due to the connectivity requirement and the ordering of the nodes in  $T$ , there is always such a parent  $v_{l_y}$  of  $v_{l_x}$  in  $M_l$  since the tree has only one root. Nevertheless, for case 1 (Theorem 3.1),  $j$  could be 1; that is, the subspace division starts from  $\mathbb{V}_1$ , in this case the best match in  $S'$  has another root node  $v$  in  $G_R$  such that  $bs(v)$  is the  $(|U_1| + 2)$ th lowest.

Theorems 3.1 and 3.2 give the key observations of our enumeration algorithm. Once the best match in each newly generated subspace is obtained, we select the match with the lowest score among these newly generated best matches and the best matches in type 1 subspaces, respectively. This is the central idea of our top- $k$  enumeration algorithm for computing kTPM, which is outlined in Algorithm 1. The computation of top-1 (best) match in a subspace (Lines 3,10,13), the efficient maintenance of  $Q$ , and the implementation details will be discussed in Section 3.3. The correctness of Algorithm 1 immediately follows from Theorems 3.1 and 3.2. Example 3.2 below demonstrates the algorithm.

---

#### Algorithm 1 kTPM

---

**Input:** A query rooted tree  $T$ , a run-time graph  $G_R$ , and  $k$ .

**Output:**  $\mathcal{M}$ : Top- $k$  matches of  $T$  in  $G_R$ .

---

- 1:  $\mathcal{M} \leftarrow \emptyset; l \leftarrow 0;$
  - 2: Initialize an empty collection  $Q$ ;
  - 3: Compute the top-1 match,  $M$ , in the subspace  $S = \mathbb{V}_1 \times \dots \times \mathbb{V}_{n_T}$ ;
  - 4: Put the pair  $(M, S)$  into  $Q$ ;
  - 5: **while**  $l < k$  **and**  $Q \neq \emptyset$  **do**
  - 6:   Get the entry  $(M', S')$  with the lowest score from  $Q$ ;
  - 7:    $l \leftarrow l + 1$ ; Output  $M'$  as the top- $l$  match;  $\mathcal{M} \leftarrow \mathcal{M} \cup \{M'\}$ ;
  - 8:   Divide( $l, M', S'$ );
  - Procedure** Divide( $l, M', S'$ )
  - 9: Let  $M'$  be  $(v_{l_1}, \dots, v_{l_{n_T}})$  obtained from  $S' = \{v_{l_1}\} \times \dots \times \{v_{l_{j-1}}\} \times (\mathbb{V}_j - U_j - \{v_{l_j}\}) \times \mathbb{V}_{j+1} \times \dots \times \mathbb{V}_{n_T}$ ;  
  { $/*$  if  $l = 1$ , then  $j = 1, U_j = \emptyset$ , and skip Lines 10-11  $*/$ }
  - 10: Compute the best match  $M_j$  in subspace  $S'_j = \{v_{l_1}\} \times \dots \times \{v_{l_{j-1}}\} \times (\mathbb{V}_j - U_j - \{v_{l_j}\}) \times \mathbb{V}_{j+1} \times \dots \times \mathbb{V}_{n_T}$ , as described in Theorem 3.1.
  - 11: Put  $(M_j, S'_j)$  into  $Q$ ;
  - 12: **for**  $x \leftarrow j + 1$  **to**  $n_T$  **do** { $/*$   $x$  starts from 1 if  $l = 1$   $*/$ }
  - 13:   Compute the best match  $M_x$  in subspace  $S'_x = \{v_{l_1}\} \times \dots \times \{v_{l_{x-1}}\} \times (\mathbb{V}_x - \{v_{l_x}\}) \times \mathbb{V}_{x+1} \times \dots \times \mathbb{V}_{n_T}$ , as described in Theorem 3.2;
  - 14:   Put  $(M_x, S'_x)$  into  $Q$ ;
- 

**Example 3.2:** Figure 3(b) shows a running example of kTPM regarding the run-time graph in Figure 3(a), where the query tree  $T$  is shown in Figure 2(a). Each rectangle shows an entry in  $Q$ , where

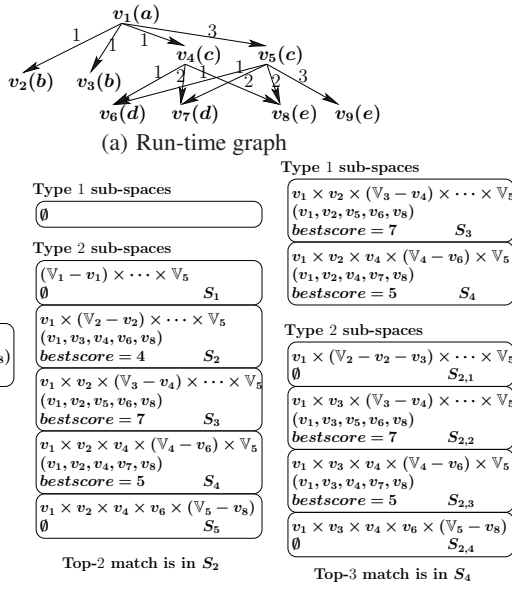


Figure 3: Example of kTPM

the first row represents a subspace, the second row is the best match in it, and the third row is the score of the match. Initially,  $Q$  contains only the entire solution space,  $S = \mathbb{V}_1 \times \dots \times \mathbb{V}_5$ , where  $\mathbb{V}_1 = \{v_1\}$ ,  $\mathbb{V}_2 = \{v_2, v_3\}$ ,  $\mathbb{V}_3 = \{v_4, v_5\}$ ,  $\mathbb{V}_4 = \{v_6, v_7\}$ ,  $\mathbb{V}_5 = \{v_8, v_9\}$ , as shown in the left of Figure 3(b). After  $M_1 = (v_1, v_2, v_4, v_6, v_8)$  is output as the top-1 match,  $S$  is divided into 5 subspaces, Case 1:  $(\emptyset)$ , Case 2:  $(S_1, \dots, S_5)$ , as shown in the middle of Figure 3(b), based on  $M_1$ .  $S_1$  is empty since  $\mathbb{V} = \{v_1\}$ , then  $\mathbb{V}_1 - \{v_1\} = \emptyset$ ; thus  $S_1$  is excluded from further considerations. The best match in  $S_2$  is computed from  $M_1$  by replacing  $v_2$  with a “sibling” node that is connected to  $v_1$ , the parent of  $v_2$  in  $M_1$ , while minimizing the score; in this case only one choice  $v_3$ . Regarding  $S_3$ ,  $v_5$  is the best replacement of  $v_4$  in  $M_1$ , while regarding  $S_4$ ,  $v_7$  is the best replacement of  $v_6$ . Although  $\mathbb{V}_5 - \{v_8\} \neq \emptyset$ ,  $v_9$  is not connected to any node in  $M_1$ ; thus,  $S_5 = \emptyset$  by the connectivity requirement. Therefore, the top-2 match of  $T$  is the top-1 match in  $S_2$ ,  $(v_1, v_3, v_4, v_6, v_8)$ , with score 4.

To compute top-3 match of  $T$ ,  $S_2$  is further divided into 4 subspaces, Case 1:  $(S_{2,1})$ , Case 2:  $(S_{2,2}, S_{2,3}, S_{2,4})$ , as shown in the right of Figure 3(b), which are the type 2 subspaces. Here, the type 1 subspaces are the set of subspaces we have when computing top-2 match of  $T$  excluding  $S_2$  which is divided further. From the two type 1 subspaces and the four type 2 subspaces, we can compute the top-3 match of  $T$ , which is  $(v_1, v_2, v_4, v_7, v_8)$  in  $S_4$ .  $\square$

Next, we will present a minimum priority queue based data structure such that line 13 (corresponding to Theorem 3.2) can be run in constant time, and line 10 (corresponding to Theorem 3.1) can be run in  $O(\log k)$  time. We choose the data structure of minimum priority queue because it can be built in linear time [9], and insertion or deletion can be run in logarithmic time regarding  $k$ . We also show that Line 6 can be done in logarithmic time regarding  $k$ .

### 3.3 Implementation and Complexity

For each distinct label  $\alpha$  of the children of a node  $v$ , we keep a list  $L_{v,\alpha} = \{(v', bs(v') + \delta_{\min}(v, v')) \mid v' \in v.children_\alpha\}$ .  $bs(v)$  can be recursively calculated as follows, assuming  $\Pi_{q(v)}$  is the set of distinct child labels of  $q(v)$  in  $T$  and  $\forall \alpha \in \Pi_{q(v)}$ ,  $L_{v,\alpha} \neq \emptyset$ .

$$bs(v) = \sum_{\alpha \in \Pi_{q(v)}} \min\{bs(v') + \delta_{\min}(v, v') \mid v' \in L_{v,\alpha}\} \quad (2)$$

Theorem 3.1 states that in Line 10 in Algorithm 1, we get the node with the  $(|U_j| + 2)$ th lowest value from  $L_{v, l(v_j)}$  where  $v$  is the

parent of  $v_{l_j}$  in  $M'$ , while Theorem 3.2 states that in Line 13 in Algorithm 1, we get the node with the second lowest value from  $L_{v', l(v_{l_x})}$  where  $v'$  is the parent of  $v_{l_x}$  in  $M'$ . Moreover, in each round of subspace division, Line 10 is only executed once, while Line 13 is executed  $O(n_T)$  times. Motivated by these, we want to 1) execute the replacement in Line 10 of Algorithm 1 by  $O(\log k)$  time, and 2) execute the replacement in Line 13 of Algorithm 1 by constant time. If  $L_{v,\alpha}$  is built as a sorted list on the values of  $(bs(v') + \delta_{\min}(v, v'))$ , then 1) and 2) can be both satisfied. Nevertheless, creating a sorted list  $L_{v,\alpha}$  requires  $O(|L_{v,\alpha}| \log |L_{v,\alpha}|)$  time; this could significantly increase the whole processing cost.

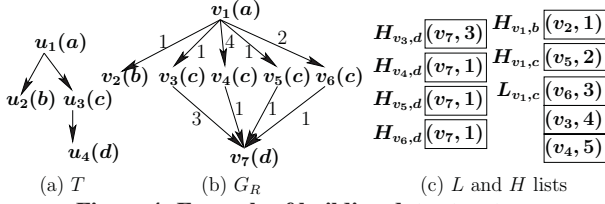
**Data Structure.** We propose to initially create a *minimum priority queue* (say, Binary Heap or Fibonacci Heap) [9] to store  $L_{v,\alpha}$ ; we use a binary minimum priority queue in our implementation. It shows in [9] that creating a binary minimum priority queue  $L_{v,\alpha}$  takes linear time  $O(|L_{v,\alpha}|)$ . It is immediate that retrieving the element with the lowest value from a binary minimum priority queue takes constant time; nevertheless, retrieving the element with the  $i$ th lowest value (the operation in Theorem 3.1) may have to visit  $O(2i - 1)$  elements in a binary minimum priority queue. Therefore, we create a sorted list  $H_{v,\alpha}$  according to a non-decreasing order on the values of  $(bs(v') + \delta_{\min}(v, v'))$ ; then each time when we execute Line 10 in Algorithm 1, we retrieve the top element  $(v', bs(v') + \delta_{\min}(v, v'))$  of  $L_{v,\alpha}$ , and remove it from  $L_{v,\alpha}$  to  $H_{v,\alpha}$ . Note that removing the top to retain a binary minimum priority queue  $L_{v,\alpha}$  takes  $O(\log |L_{v,\alpha}|)$  time. In our implementation, we scan  $L_{v,\alpha}$  once in initialization to get the element with the minimum score, and put it into  $H_{v,\alpha}$ , then organize the remaining elements into  $L_{v,\alpha}$ ; that is, now the top of  $L_{v,\alpha}$  has the second lowest value of  $bs(v') + \delta_{\min}(v, v')$  among all elements in  $L_{v,\alpha} \cup H_{v,\alpha}$ . In the following, we call  $L_{v,\alpha}$  and  $H_{v,\alpha}$  as  $L$  lists and  $H$  lists, respectively.

**Initially Building the Data Structure and Top-1 Match.** We build the data structure in a bottom-up fashion on  $G_R$  as follows. Starting from the lowest level of  $G_R$ , we iteratively create and initialize  $H$  and  $L$  lists. For each node  $v$  of  $G_R$ , we compute  $bs(v)$ ; this can be conducted by combining the top values in  $H_{v,\alpha}$  for each distinct child label  $\alpha$  of  $q(v)$  in  $T$  as shown in Equation 2. Note that if  $v$  has an empty  $H_{v,\alpha}$  for a child label  $\alpha$  of  $q(v)$  in  $T$ , we can safely remove  $v$  from  $G_R$  and then recursively remove its decedents with no parents. Note that, since we build the data structure in a bottom-up fashion, we have already computed  $bs(v')$  for every  $v'$  in  $L_{v,\alpha} \cup H_{v,\alpha}$  when initializing  $H_{v,\alpha}$  and  $L_{v,\alpha}$ . Immediately, the cost of building the data structure is  $O(m_R)$  and the score of the top-1 (best) match of  $T$  can be obtained from the lowest value among all  $bs$  scores at the roots of  $G_R$ . The correctness of the obtained top-1 match can be immediately verified by induction.

**Example 3.3:** Regarding the tree query  $T$  in Figure 4(a) over the run-time graph  $G_R$  in Figure 4(b), the data structure is built for nodes in  $G_R$  level by level. Firstly, for the nodes at level 2 -  $v_3, v_4, v_5, v_6$ , each has only one child with label  $d$ . Thus,  $(v_7, \delta_{\min}(v_i, v_7))$  is put into  $H_{v_i,d}$  for  $3 \leq i \leq 6$  as shown in the left part of Figure 4(c), and the corresponding  $L_{v_i,d}$ s are empty and omitted. For the node  $v_1$  at level 3,  $H_{v_1,b} = \{(v_2, 1)\}$  and  $L_{v_1,b} = \emptyset$ .  $v_1$  has four children with label  $c, v_3, v_4, v_5$ , and  $v_6$ . Among them,  $v_5$  has the lowest  $(bs(v_5) + \delta_{\min}(v_1, v_5))$ ; thus,  $v_5$  is put into  $H_{v_1,c}$  and the other three nodes are put into  $L_{v_1,c}$  as shown in the right part of Figure 4(c). The final data structure is shown in Figure 4(c). As there is only one root,  $v_1$ , in  $G_R$ , the top-1 match of  $T$  is the best match rooted at  $v_1$ , with  $bs(v_1) = 1 + 2 = 3$  obtained from  $H_{v_1,b}$  and  $H_{v_1,c}$ .  $\square$

**Implementing Replacement in Algorithm 1 - Lines 10 and 13.** Regarding the situation in Theorem 3.2 - Line 13 in Algorithm 1, we just use the top in the binary minimum priority queue  $L_{v,\alpha}$ , where  $v$  is the parent of  $v_{l_x}$  in  $M'$  and the label  $\alpha$  defines  $\mathbb{V}_x$ , since





**Figure 4: Example of building data structure**

this is the node with the second lowest value in  $L_{v,\alpha} \cup H_{v,\alpha}$ . Clearly, this takes  $O(1)$  time.

Regarding the situation in Theorem 3.1 - Line 10 in Algorithm 1, there are two cases: 1)  $|U_j| = 1$ , and 2)  $|U_j| > 1$ . When  $|U_j| = 1$ , Line 10 is getting the second lowest value from  $L_{v,\alpha}$  (i.e., the third lowest value in  $L_{v,\alpha} \cup H_{v,\alpha}$ ) and then moving the lowest and second lowest elements in  $L_{v,\alpha}$  to  $H_{v,\alpha}$ . When  $|U_j| > 1$ , Line 10 is getting the top element in  $L_{v,\alpha}$  and then removing it to  $H_{v,\alpha}$ . Note that we treat  $|U_j| = 1$  specially since we do not remove the top of  $L_{v,\alpha}$  when executing Line 13 to avoid increasing the time complexity. Note that to deal with the situation that  $v_{ij}$  is a root, we only need to choose the root with the  $(|U_j| + 2)$ th lowest  $bs$  score among all roots in  $G_R$ , which are organized in a similar way as  $L$  and  $H$  lists, with  $bs$  scores as key.

**Example 3.4:** Consider the tree query  $T$  in Figure 4(a) over the run-time graph  $G_R$  in Figure 4(b). Figure 4(c) illustrates the  $L$  and  $H$  lists constructed.  $\mathbb{V}_1 = \{v_1\}$ ,  $\mathbb{V}_2 = \{v_2\}$ ,  $\mathbb{V}_3 = \{v_3, v_4, v_5, v_6\}$ , and  $\mathbb{V}_4 = \{v_7\}$ . The top-1 match of  $T$  in  $G_R$  is  $M_1 = (v_1, v_2, v_5, v_7)$ . After dividing the entire solution space by  $M_1$ , we get only one nonempty subspace  $S_3 = \{v_1\} \times \{v_2\} \times (\mathbb{V}_3 - \{v_5\}) \times \mathbb{V}_4$ . The best match in  $S_3$  is computed from  $M_1$  by replacing  $v_5$  with the node that has 2nd lowest value ( $bs(v) + \delta_{\min}(v_1, v)$ ) among all nodes adjacent to  $v_1$  with the label  $c$ . This corresponds to the situation (Line 13) in Theorem 3.2. Then,  $v_5$  is replaced by  $v_6$  obtained from the top of  $L_{v_1,c}$ . Therefore, the top-2 match of  $T$  is  $M_2 = (v_1, v_2, v_6, v_7)$ . By dividing  $S_3$ , we get only one nonempty subspace  $S_{3,1} = \{v_1\} \times \{v_2\} \times (\mathbb{V}_3 - \{v_5, v_6\}) \times \mathbb{V}_4$ . The best match in  $S_{3,1}$  is computed from  $M_2$  by replacing  $v_6$ , and this corresponds to the situation (Line 10) in Theorem 3.1. Since  $|U_3| = 1$  ( $U_3 = \{v_5\}$ ), we get the second lowest from  $L_{v_1,c}$  (i.e.,  $v_3$ ), and then remove  $(v_6, 3)$  and  $(v_3, 4)$  from  $L_{v_1,c}$  and put it to  $H_{v_1,c}$ . Therefore, the best match in  $S_{3,1}$  is  $(v_1, v_2, v_3, v_7)$ , and it is the top-3 match of  $T$ . This continues and falls into the case that we only need to use and remove the top of  $L_{v_1,c}$ .  $\square$

**Recovering the Match from Score.** In Algorithm 1, for each top-1 match  $M_j$  computed in a newly generated subspace  $S'_j$ , to save space, we do not store the details of the match. Instead, we record the score and link  $M_j$  to the top- $l$  solution  $M'$  that generates the subspace  $S'_j$ , together with the detailed replacement information, say,  $v_x$  is replaced by  $v_y$ . The score of  $M_j$  can be calculated in  $O(1)$  time after node replacement since it can be calculated as the score of  $M'$  plus the local score difference at the replacement. For example, consider the top-1 match,  $(v_1, v_2, v_5, v_7)$  with score 3, of  $T$  in Example 3.4. After  $v_5$  is replaced by  $v_6$ , the score of the new match  $(v_1, v_2, v_6, v_7)$  is  $3 + (3 - 2) = 4$ .

Once  $M_j$  is selected to be one of the top- $k$  matches of  $T$ , we only need to replace the subtree rooted at  $v_x$  in  $M'$  with the best match of  $T_{q(v_x)}$  rooted at  $v_y$  to generate  $M_j$ ; note  $l(v_x) = l(v_y)$ . To get the best match of  $T_{q(v_x)}$  rooted at  $v_y$ , we only need to iteratively travel down from  $v_y$  to its best decedents following the link information built in the initialization phase when building  $L$  and  $H$  lists. Therefore, recovering the match from score takes  $O(n_T)$  time which is linear regarding the size of the match.

**Computing Top- $k$  Matches from Subspaces.** To control the maintenance complexity of  $Q$  in Algorithm 1, instead of inserting the computed best match in each newly generated subspace into  $Q$ , we

additionally maintain  $k$  binary minimum priority queues  $Q_1, \dots, Q_k$ . In each round of generating a top- $l$  match of  $T$ , we use  $Q_l$  to store the best matches over the newly obtained subspaces (Lines 10 & 11, Lines 13 & 14) except the match with the lowest score which is inserted into  $Q$ ; that is,  $Q_l$  does not contain the match with the lowest score against these newly generated subspaces. While creating  $Q_l$  takes  $O(|Q_l|)$  time and popping-up the best match from  $Q_l$  for inserting into  $Q$  takes  $O(\log |Q_l|)$  time, inserting a match into  $Q$  takes  $O(\log |Q|)$  time.

Moreover, when the top match  $M$  in  $Q$  is popped as a top- $l$  match, we need to pop up the top of  $Q_j$ , to which  $M$  belongs, and insert it into  $Q$ . This takes  $O(\log |Q| + \log |Q_j|)$  time.

**Complexity.** It can be shown that the time complexity of Algorithm 1 is  $O(m_R + k(\log d_R + n_T + \log n_T + \log k))$  where  $d_R$  is the maximum degree of nodes in  $G_R$ . If  $k$  is given, then the time complexity of Algorithm 1 is reduced to  $O(m_R + k(n_T + \log k))$  that is optimal, while DP-B [21] runs in  $O(m_R \log k + kn_T(d_T + \log k))$  time. Our algorithm takes space  $O(\min\{m_R, kn_R d_T\} + kn_T)$ , while DP-B [21] takes space  $O(\min\{m_R, kn_R d_T\} + kn_T d_T)$ , where  $d_T$  is the maximum node degree in  $T$ .

For details of complexity deduction, please refer to the section of “Complexity Analysis” in Section 3.3 in the full paper [4]. As discussed in Sections 1 and 3.1,  $m_R = \theta n_T$  on average, where  $\theta$  is the average number of edges of the same type in the transitive closure. Therefore, the space complexity of our algorithm is linear regarding  $n_T$  (query size) if  $\theta$  is bounded.

## 4. PRIORITY BASED ALGORITHM

Following the framework of Algorithm 1, in this section we present a priority order based algorithm to reduce the access of unnecessary information from a run-time graph  $G_R$ ; that is, reduce the number of edges to be loaded in from disk. In comparison to DP-P in [21], our pruning bound is tighter and may achieve a speed-up towards orders of magnitude.

The rest of the section is organized as follows. We first present the framework and data structures. Then, we introduce our algorithm for computing the top-1 match, followed by our enumeration techniques. Finally, we analyse the complexities of our algorithms.

### 4.1 Framework and Data Structures

To represent the closure  $G_c$  of a data graph  $G$ , we further organize each table  $L_\beta^\alpha$  (Section 3.1) into different groups based on the nodes pointed to; that is, for each node  $v \in G_c$ , the incoming edges to  $v$  are grouped according to the labels of the parents of  $v$  such that the incoming edges  $(v', v)$  with the same label  $l(v') = \alpha$  are put in  $L_v^\alpha$ . Edges in the group  $L_v^\alpha$  are stored together in a non-decreasing order based on their shortest distances by (possibly) multiple blocks and exclusively from edges in other groups. Note that, for an edge in  $L_v^\alpha$ , we only need to store in  $L_v^\alpha$  the adjacent node  $v'$  coming to  $v$  and the corresponding shortest distance  $\delta_{\min}(v', v)$ . In practice, an  $L_v^\alpha$  may contain many nodes; for example, an  $L_v^\alpha$  contains over 1700 nodes in DBLP data with one million nodes and two million edges. To reduce the access of a run-time graph, for each node  $v$  in  $G_c$ , we also keep in  $E_v$  the subset of outgoing edges from  $v$  per distinct label with the minimum value of the shortest distances to  $v$ .

Regarding each  $L_v^\alpha$ , we also keep the information  $d_v^\alpha$ , the minimum value of the shortest distances from nodes in  $L_v^\alpha$  to  $v$  (i.e.,  $d_v^\alpha = \min\{\delta_{\min}(v', v) \mid v' \in L_v^\alpha\}$ ); note that  $d_v^\alpha$ s are not stored with  $L_v^\alpha$ .

**Example 4.1:** For the data graph in Figure 2(b), we store its closure (Figure 2(c)) as follows. For example, for  $v_5$ ,  $L_{v_5}^\alpha = \{(v_1, 1), (v_2, 2)\}$ , where  $d_{v_5}^\alpha = \delta_{\min}(v_1, v_5) = 1$ ;  $E_{v_5} = \{(v_5, v_7, 1), (v_5, v_9, 1), (v_5, v_{11}, 1)\}$  since  $v_5$  reaches 3 distinct labels  $d (= l(v_7))$ ,  $e (= l(v_9))$ , and  $s (= l(v_{11}))$  where  $\delta_{\min}(v_5, v_7) = 1$ ,  $\delta_{\min}(v_5, v_9) = 1$ , and  $\delta_{\min}(v_5, v_{11}) = 1$ .

Similarly,  $L_{v_6}^a = \{(v_1, 1), (v_2, 2)\}$ ,  $d_{v_6}^a = 1$ , and  $E_{v_6} = \{(v_6, v_{12}, 1), (v_6, v_7, 1), (v_6, v_9, 2)\}$ .  $\square$

In our algorithm, all  $d_v^\alpha$ s are stored based on  $\alpha$  and the label of  $v$ . For a pair of node labels  $\alpha$  and  $\beta$ , all  $d_v^\alpha$ s with  $l(v) = \beta$  are allocated in the same group  $D_\beta^\alpha$ . In  $D_\beta^\alpha$ , for each  $d_v^\alpha$ , we record  $v$  and the value of  $d_v^\alpha$ . Note that, most of the values of  $d_v^\alpha$ s are 1; thus, we only store such  $d_v^\alpha$ s with values greater than 1 to save storage space and the costs to be loaded in to main-memory in initialization. Similarly, all  $E_v$ s are also grouped into  $\mathbb{E}_\beta^\alpha$  such that edges  $(v, v')$  in  $E_v$  with  $l(v) = \alpha$  and  $l(v') = \beta$  are put in  $\mathbb{E}_\beta^\alpha$ ; that is,  $\mathbb{E}_\beta^\alpha$  consists of the entries  $(v, v', \delta_{\min}(v, v'))$  with  $l(v) = \alpha$  and  $l(v') = \beta$ . Note that for each pair of  $\alpha$  and  $\beta$ , we put  $D_\beta^\alpha$  in one data block and allocate more blocks if one is not large enough;  $\mathbb{E}_\beta^\alpha$  is physically stored in the same way. The data structures are summarized in the following table.

$D_\beta^\alpha = \{(v, d_v^\alpha) \mid l(v) = \beta\};$	$L_\beta^\alpha = \{(v, L_v^\alpha) \mid l(v) = \beta\}$
$\mathbb{E}_\beta^\alpha = \{(v, v', \delta_{\min}(v, v')) \mid l(v) = \alpha, l(v') = \beta, \text{ and } (v, v', \delta_{\min}(v, v')) \in E_v\}$	

For example, in Example 4.1, since  $d_{v_7}^c = 1$ ,  $d_{v_7}^c$  is not stored in  $D_d^c$ . Consequently,  $D_d^c$  stores only one element  $(v_8, 2)$ ; that is,  $D_d^c = \{(v_8, 2)\}$ . In  $\mathbb{E}_d^c$ , we store  $\{(v_5, v_7, 1), (v_6, v_7, 1)\}$ , while in  $\mathbb{E}_e^c$  we store the information of  $\{(v_5, v_9, 1), (v_6, v_9, 2)\}$ .

**Initialization.** Although the priority order based algorithm presented in this section follows the framework of Algorithm 1, different than Algorithm 1, we aim to load in as few edges as possible to minimize I/Os and to achieve scalability (i.e., allow large graphs to be processed in main-memory). Therefore, instead of loading in  $G_R$  in the initialization phase, we only load in to main-memory the  $D_\beta^\alpha$ s and  $\mathbb{E}_\beta^\alpha$ s such that for each loaded  $D_\beta^\alpha$ , there must be an edge  $(u, u')$  in  $T$  with  $l(u) = \alpha$  and  $l(u') = \beta$ , and for each loaded  $\mathbb{E}_\beta^\alpha$ , there must be an edge  $(u, u')$  in  $T$  with  $l(u) = \alpha$ ,  $l(u') = \beta$ , and  $u'$  is a leaf. For example, consider the query in Figure 2(a) over the closure in Figure 2(c); in initialization, we load in the blocks of  $D_b^a$ ,  $D_d^a$ ,  $D_d^c$ , and  $D_e^c$ , as well as load in the blocks of  $\mathbb{E}_d^c$ ,  $\mathbb{E}_e^c$ , and  $\mathbb{E}_b^a$ .

After initialization our algorithm follows two steps: Step 1) generate the top-1 match and initialize data structure for enumeration, and Step 2) enumerate the top- $k$  matches one by one.

## 4.2 Computing the Top-1 Match

Different than our Algorithm 1, since  $G_R$  is not fully loaded in, we need to 1) detect when edges need to be loaded in, and 2) detect when we can claim that the top-1 match is already obtained. As a byproduct, we also need to efficiently build the  $L_{v,\alpha}$  and  $H_{v,\alpha}$  lists.

The central idea of our priority order based algorithm for computing the top-1 match of  $T$  is to iteratively maintain a minimum priority queue  $Q_g$ , which controls the access of edges of  $G_R$ . Each element in  $Q_g$  is currently an “active” node (defined below)  $v$  in  $G_R$  associated with a lower bound score  $lb(v)$  of the best match (i.e., lowest score) of  $T$  containing  $v$  in  $G_R$ , and  $Q_g$  uses  $lb(v)$  as the key. Then, we iteratively pop the top element  $(v, lb(v))$  from  $Q_g$  and load in to main-memory the incoming edges to  $v$  in  $L_v^\alpha$  where  $\alpha$  is the parent node label of  $q(v)$  in  $T$  (Since  $T$  is a tree, such  $\alpha$  is unique).

As computing a nontrivial  $lb(v)$  is hard, inspired by the A\* algorithm [23] we store  $(v, \overline{lb}(v))$  in  $Q_g$  with the property that  $\overline{lb}(v)$  is a combination of an *upper-bound on  $bs(v)$*  (i.e.,  $\overline{bs}(v)$ ), and a *lower-bound on the remaining edges* of any match of  $T$  containing  $v$ . Here,  $bs(v)$  is the score of the best match of  $T_{q(v)}$  containing  $v$  in  $G_R$ . As with A\* algorithm, for the top  $(v, \overline{lb}(v))$  of  $Q_g$  at any time we pursue two properties that 1) the score  $bs(v)$  is already calculated (i.e.,  $\overline{bs}(v) = bs(v)$ ) and is contained in  $\overline{lb}(v)$  (i.e.,  $\overline{lb}(v)$  now becomes a lower bound score of the best match of  $T$  containing  $v$  in  $G_R$ ), and 2) the top of  $Q_g$  is popped up in a non-decreasing order of its  $\overline{lb}$  value. Thus, the best match of  $T$  is obtained when a root  $v \in G_R$  becomes the top of  $Q_g$  for the first time. An outline of the

algorithm is shown in Algorithm 2.

### Algorithm 2 ComputeFirst

---

```

1: Initialize: load in  $D_\beta^\alpha$  and  $\mathbb{E}_\beta^\alpha$  as described above;
2:  $\forall$  loaded edges  $(v, v')$ , insert  $(v', \delta_{\min}(v, v'))$  into  $L_{v',l(v)}$ ;
3:  $\forall$  active  $v$ , compute  $\overline{bs}(v)$  &  $\overline{lb}(v)$ , and insert  $v$  into  $Q_g$  with key  $\overline{lb}(v)$ ;
4: while  $Q_g \neq \emptyset$  do
5:    $(v, \overline{lb}) \leftarrow Q_g.\text{POP}()$ ;
6:   if  $v$  has the same label as the root of  $T$  then
7:     Return the score of the best match rooted at  $v$ ;
8:   else
9:     Expand( $v$ );
Procedure Expand( $v$ )
10: Load a block of incoming edges to  $v$ ;
11: for each loaded edge  $(v', v)$  do
12:   Insert  $(v, \overline{bs}(v) + \delta_{\min}(v', v))$  into  $L_{v',l(v)}$ ;
13:   Update  $\overline{bs}(v')$  (and  $\overline{lb}(v')$ ) in  $Q_g$  if  $v'$  is active;
14: if an estimation of the next block of incoming edges to  $v$  still makes  $v$ 
    the top of  $Q_g$  then
15:   goto Line 10;
16: else
17:   Insert  $v$  into  $Q_g$  with an updated key  $\overline{lb}_{\text{new}}(v)$ ;
```

---

**Details of Algorithm 2.** We first present  $\overline{bs}$  and  $\overline{lb}$ . Note that in Algorithm 2, it appears that we insert  $(v, \overline{bs}(v) + \delta_{\min}(v', v))$  into  $L_{v',l(v)}$ , instead of inserting  $(v, bs(v) + \delta_{\min}(v', v))$  into  $L_{v',l(v)}$  as done in Section 3.3. Nevertheless, later we will prove that when we do such an insertion,  $\overline{bs}(v)$  already becomes  $bs(v)$ . Moreover, similar to that in Section 3.3, we also put the element with the lowest value of  $(\overline{bs}(v) + \delta_{\min}(v', v))$  in  $H_{v',l(v)}$  instead of in  $L_{v',l(v)}$ .

**Active Node.** A node  $v$  in  $G_R$  is active if for each child node label  $\alpha$  of  $q(v)$  in  $T$ , the current  $L_{v,\alpha} \cup H_{v,\alpha} \neq \emptyset$ ; that is, at least one edge  $(v, v') \in G_R$  with  $l(v') = \alpha$  is loaded in. Note that due to the initialization in Line 1 of Algorithm 2, the nodes in  $G_R$  whose children are all leaf nodes are active; otherwise inactive. For example, regarding Figure 2(d), if  $(v_5, v_7)$  and  $(v_5, v_9)$  are loaded in then  $v_5$  is active (even if  $(v_5, v_8)$  is not loaded in). On the other hand, if none of  $(v_2, v_3)$  and  $(v_2, v_4)$  is loaded in,  $v_2$  is inactive even if  $(v_2, v_5)$  and  $(v_2, v_6)$  are loaded in. To detect if a node in  $G_R$  is active, we keep at each node  $v$  the number  $n_v$  of distinct child labels of  $q(v)$  in  $T$ ; once the number of nonempty  $L_{v,\alpha} \cup H_{v,\alpha}$  reaches  $n_v$ ,  $v$  is active.

**Upper-bound  $\overline{bs}(v)$ .** For each active node  $v \in G_R$ ,

$$\overline{bs}(v) = \sum_{\alpha \in \Pi_{q(v)}} \min\{\overline{bs}(v') + \delta_{\min}(v, v') \mid v' \in L_{v,\alpha} \cup H_{v,\alpha}\}, \quad (3)$$

where  $\Pi_{q(v)}$  is the set of distinct child node labels of  $q(v)$  in  $T$ .

Unlike Algorithm 1, here  $L_{v,\alpha} \cup H_{v,\alpha}$  may be incomplete; thus,  $\overline{bs}(v)$  can only serve as an upper-bound of the score of the best match of  $T_{q(v)}$  containing  $v$  even if the best match of  $T_{q(v')}$  containing  $v'$  is already computed for each  $v'$  in  $L_{v,\alpha} \cup H_{v,\alpha}$ . Later we will show in Theorem 4.2 that even though  $L_{v,\alpha} \cup H_{v,\alpha}$  is incomplete, we are still able to determine whether  $bs(v)$  can be obtained, i.e., determine whether the current value of  $\overline{bs}(v)$  is  $bs(v)$ .

**Lower-bound on Remaining Edges.** For each node  $u \in T$ , we use  $\overline{L}(u)$  to denote a lower bound of the score of the best match of  $T - (T_u \cup (u_p, u))$  where  $u_p$  is the parent of  $u$  in  $T$ . Let  $e_{v'}$  denote a lower bound of  $\delta_{\min}(v, v')$  for all unloaded incoming edges  $(v, v')$  to  $v'$ , where  $e_{v'} = d_{v'}^{l(v)}$  if none of the incoming edges to  $v'$  is loaded in or  $e_{v'}$  is the maximum weight of the already loaded incoming edges to  $v'$ . Thus, we use  $\overline{lb}(v') = \overline{bs}(v') + e_{v'} + \overline{L}(q(v'))$  to estimate the lowest score of any match containing  $(v, v')$  regarding all unloaded incoming edges  $(v, v')$  to  $v'$ , and put  $(v', \overline{lb}(v'))$  into  $Q_g$  to wait for a pop-up from  $Q_g$  to load in to main-memory the unloaded incoming edges to  $v'$ . Due to incomplete information of  $G_R$ , we are only able



to identify a trivial lower bound  $L(u)$ ; that is,  $L(u) = n_T - 1 - |T_u|$  where  $|T_u|$  is the number of nodes in  $T_u$ . Clearly,  $\overline{lb}(v')$  is neither an upper bound nor a lower bound, and becomes a lower bound when  $v'$  becomes the top of  $Q_g$  as proved in theorem 4.2. In Line 14, we use  $\overline{lb}(v)$  to determine if the incoming edges to  $v$  can be still loaded in (i.e., if  $\overline{lb}(v)$  is not greater than the second top in  $Q_g$ , then the incoming edges to  $v$  can be still loaded in).

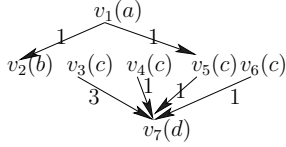
Although  $\overline{lb}$  values in  $Q_g$  may change after more edges are loaded in, we show below that the  $\overline{lb}$  values of popped tops of  $Q_g$  are monotonic.

**Theorem 4.1:** Suppose that  $(v', \overline{lb}(v'))$  is the top after  $(v, \overline{lb}(v))$  has been popped. Then,  $\overline{lb}(v) \leq \overline{lb}(v')$ .  $\square$

Next, we can show that once a node  $v$  is at the top of the queue  $Q_g$ , we have  $\overline{bs}(v) = bs(v)$ ; consequently,  $\overline{lb}(v)$  becomes a lower bound score of the best match of  $T$  containing  $v$ .

**Theorem 4.2:** Suppose that  $(v, \overline{lb}(v))$  is the current top of  $Q_g$  and  $v$  becomes the top for the first time. Then, regarding each child node label  $\alpha$  of  $q(v)$  in  $T$ , the edge  $(v, v')$ , making  $(bs(v') + \delta_{\min}(v, v'))$  the minimum among all children of  $v$  with the label  $\alpha$ , is already loaded in and  $bs(v')$  is already calculated. Consequently,  $bs(v)$  is also calculated.  $\square$

Theorem 4.2 also implies that the  $\overline{bs}$  values that we insert to  $L_{v,\alpha}$  and  $H_{v,\alpha}$  lists in Algorithm 2 are actually  $bs$  values. Based on Theorems 4.1 and 4.2, together with the fact that the nodes in  $G_R$  with the same label as the root of  $T$  do not have incoming edges, it is immediate that the returned score by Algorithm 2 is the score of the top-1 match of  $T$ . We can recover the match corresponding to the score in the same way as described in Section 3.3.



**Figure 5: The part of  $G_R$  that is loaded into main memory**

**Example 4.2:** Consider the tree query  $T$  in Figure 4(a) over the run-time graph  $G_R$  in Figure 4(b). Initially, we load in edges  $(v_1, v_2)$ ,  $(v_3, v_7)$ ,  $(v_4, v_7)$ ,  $(v_5, v_7)$ ,  $(v_6, v_7)$ , and  $H_{v_1,b}$ ,  $H_{v_3,d}$ ,  $H_{v_4,d}$ ,  $H_{v_5,d}$ ,  $H_{v_6,d}$  are initialized as shown in Figure 4(b) while both  $H_{v_1,c}$  and  $L_{v_1,c}$  are empty. Therefore,  $Q_g$  contains four active nodes  $v_3, v_4, v_5, v_6$  with  $\overline{lb}$  values 5, 6, 3, 4, respectively. Then,  $v_5$  is popped up from  $Q_g$ , and  $bs(v_5) = 1$ . Since  $v_5$  is not a root node, the incoming edge  $(v_1, v_5)$  to  $v_5$  is loaded in through Expand. Now,  $v_1$  becomes active with  $\overline{lb}(v_1) = 3$ , and  $Q_g$  contains  $\{v_1, v_3, v_4, v_6\}$ . In the next iteration,  $v_1$  is popped up from  $Q_g$ , and the top-1 match of  $T$  is computed as the best match rooted at  $v_1$  with  $bs(v_1) = 3$ . Here, we compute the top-1 match of  $T$  without loading in the incoming edges to  $v_3, v_4$ , and  $v_6$ , and the part of  $G_R$  that is loaded in is shown in Figure 5.  $\square$

### 4.3 Top- $k$ Enumeration

Our top- $k$  enumeration algorithm for computing top- $k$  matches based on partially available information (i.e.,  $L_{v,\alpha}$  and  $H_{v,\alpha}$ ) also follows the same framework as that in Algorithm 1. Nevertheless, the match obtained in each subspace  $S$  by Theorems 3.1 or 3.2 has no guarantee that it will be the best match in  $S$  due to incomplete information in  $L_{v,\alpha}$  and  $H_{v,\alpha}$ . While we still use the same method to generate the match from each subspace based on the current  $L_{v,\alpha}$  and  $H_{v,\alpha}$ , we will use the current top of  $Q_g$  to determine if the corresponding incomplete  $L_{v,\alpha}$  and  $H_{v,\alpha}$  are enough to guarantee that the obtained match is the best match in the subspace.

As with Algorithm 1, we use a minimum priority queue  $Q$  to

store the candidates for top- $k$  matches. Consider the monotonicity theorem, Theorem 4.1. If the score of an obtained current best match  $M$  in  $S$ , computed from the current partial  $L_{v,\alpha}$  and  $H_{v,\alpha}$  lists, is not greater than the score of the current top of  $Q_g$ , Theorem 4.1 can guarantee that any match obtained in  $S$  involving an unloaded edge in  $S$  will not be better than  $M$ ; consequently,  $M$  is the best match of  $T$  in  $S$ , and we insert  $M$  to  $Q$ . Otherwise, we delay the insertion of  $M$  to  $Q$  and link  $M$  to the top of  $Q_g$  till the top of  $Q_g$  pops and expands to load in more edges. Once the top of  $Q_g$  pops, we update the involved  $M$  and detect if  $M$  can be inserted to  $Q$ . Note that we pop the top of  $Q_g$  only if it has a key value smaller than the top of  $Q$ . That is, we pop the top of  $Q$  as one of the top- $k$  matches of  $T$  only when its score is not larger than the current top of  $Q_g$  and the number of currently popped matches is smaller than  $k$ . Our enumeration algorithm is shown in Algorithm 3. Note that, in Algorithm 3 we conceptually take  $G_R$  as input, however, instead of taking the entire  $G_R$  as input we only load the needed part from disk to main memory as discussed in Section 4.2.

#### Algorithm 3 Enhanced-Top $k$

**Input:** A query rooted tree  $T$ , a run-time graph  $G_R$  stored on disk, and  $k$ .  
**Output:**  $M$ : Top- $k$  matches of  $T$  in  $G_R$ .

```

1:  $M \leftarrow \emptyset$ ;  $l \leftarrow 0$ ;
2: Compute the score ( $lb$ ) of top-1 match of  $T$  in the subspace  $S = \mathbb{V}_1 \times \dots \times \mathbb{V}_{n_T}$  by invoking ComputeFirst (i.e., Algorithm 2);
3: Initialize  $Q$  with  $(S, lb)$ ;
4: while  $l < k$  and  $Q \neq \emptyset$  do
5:    $(M', S') \leftarrow \text{Next}()$ ;
6:    $l \leftarrow l + 1$ ; Output  $M'$  as the top- $l$  match;  $M \leftarrow M \cup \{M'\}$ ;
7:    $\text{Divide}'(l, M', S')$ ;

Procedure Next()
8: if  $Q_g \neq \emptyset$  then
9:   while  $Q = \emptyset$  or the top of  $Q_g$  has a value no less than that of  $Q$  do
10:     Get the top entry  $(v, lb)$  from  $Q_g$ ;
11:     Expand( $v$ ) and update  $M$  of  $S$  that is linked to the top of  $Q_g$ ;
12:   Get the top entry  $(S', lb')$  from  $Q$ ;
13: Return (the current best match  $M'$  of  $T$  in  $S', S'$ );

Procedure Divide'( $l, M', S'$ );
14: Same as  $\text{Divide}(l, M', S')$  in Algorithm 1 except that we might delay the insertion of  $M_j$  and  $M_x$  to  $Q$  at Line 11 and Line 14 of Algorithm 1 as what described in Section 4.3;

```

Unlike Algorithm 1, in Algorithm 3 an empty match  $M$  in a subspace  $S$  computed according to the current  $L_{v,\alpha}$  and  $H_{v,\alpha}$  lists may become nonempty later after loading in more edges. Thus, we assign  $\infty$  as the lowest score of a current empty match in  $S$ . Theorems 4.2 and 4.1 ensure the correctness of Algorithm 3.

### 4.4 Implementation and Complexity

All  $L_{v,\alpha}$  and  $H_{v,\alpha}$  lists are defined and maintained in the same way as in Algorithm 1, as well as  $Q$  and the way to insert the match obtained from each  $S_i$  to  $Q$ . We also maintain  $Q_g$  by a minimum priority queue. The advantages of implementing a minimum priority queue by a Fibonacci heap [9] are that it takes  $O(1)$  time for insertion,  $O(1)$  time for updating if the involved key is decreased, and linear time for building the priority queue, while taking logarithmic time for delete-min.

Regarding time complexity, the following Theorem is proved in Section 4.4 of the full paper [4].

**Theorem 4.3:** Algorithm 3 computes top- $k$  matches of  $T$  in  $O(m'_R + n'_R \log n'_R + k(\log d'_R + \log k + n_T))$  time, where  $m'_R$  is the number of retrieved edges from  $G_R$ ,  $n'_R$  is the total number of active nodes, and  $d'_R$  is the maximum size of a  $L_{v,\alpha} \cup H_{v,\alpha}$  in running Algorithm 3.  $\square$

As with Algorithm 1, Algorithm 3 takes  $O(m'_R + kn_T)$  space. Note that in practice,  $m'_R \ll m_R$  and  $n'_R \ll n_R$ . This is validated by our

experiments. Our experiments also demonstrate that Algorithm 3 can achieve a speedup orders of magnitude over Algorithm 1, DP-B, and DP-P.

## 5. EXTENDING OUR TECHNIQUES

Below we brief various extensions of our techniques; more details can be found in [4].

**Supporting Top- $k$  Twig-Pattern Matching.** The currently presented techniques only cover twig queries with unique node labels and the ‘//’ type edges but without wildcard nodes (i.e., nodes with label \*). Below, we extend them to cover the general case.

Firstly, our algorithms can be immediately extended to cover a twig-pattern query with different nodes having the same label in  $T$ . To extract a run-time graph, for each label  $\alpha$  in  $T$  we make (possibly) multiple copies of a node with the label  $\alpha$  in  $G$  at the levels of  $G_R$  corresponding to the levels of nodes with the label  $\alpha$  in  $T$ . Then our algorithms can be immediately run against such a run-time graph in the same way as the case that each node in  $T$  has a unique label. Thus, the complexity remains the same regarding  $m_R$ ,  $T$ , and  $k$ . In general, the size of a run-time graph is determined by the size of  $T$ ; that is,  $m_R = \theta n_T$  still holds on average. That is, the average performance of our algorithms for such queries will be not worse than that for queries with distinct labels. However, the worst case is that if  $T$  contains all labels in  $G$ , then the number  $n_R$  of nodes in  $G_R$  may be larger than the number of nodes in  $G$ .

Secondly, our techniques can be immediately extended to cover wildcard (\*) nodes as follows. For each non-wildcard node, the treatment is the same as above. For a wildcard (\*) node in  $T$ , in  $G_R$  every node  $v$  in  $G$  may be copied at the level that has the wildcard node in  $T$ . Then we run our algorithms against such a run-time graph by allowing mapping a wildcard (\*) node to any node in  $G_R$  at the same level (i.e., a node with any label); the time complexity also remains the same regarding the run-time graph. Nevertheless, a wildcard node has different semantics: the presence of a wildcard node makes the graph unlabeled; that is, a wildcard node may be mapped to any node in the data graph. This will significantly increase the run-time graph size since in the run-time graph, each wildcard node may require a copy of all nodes in the data graph. The good news is that in practice, there are no (or very few) wildcard nodes; for example, XML benchmark queries<sup>3</sup> have no wildcard nodes. Meanwhile, it could be an interesting problem in theory as our future work - how to deal with wildcard nodes efficiently.

Thirdly, our techniques can be immediately extended to cover label containment. In principle, we can immediately modify our algorithms by testing containment instead of equality. The run-time graph may be constructed in a similar way to the above; that is, a node  $v$  in  $G$  may be duplicated to correspond to nodes of  $T$  with labels contained by  $v$ .

Fourthly, our techniques can also be immediately extended to include the ‘/’ type edges [32], by restricting the retrieval of edges  $(v, v')$  of length 1 (i.e., corresponding to edges in data graph  $G$ ) when retrieving mappings for a ‘/’ type edge. Then, our techniques are immediately applicable with the same complexity.

**Managing Closure Size.** In the worst case, the transitive closure may be extremely large due to possible  $O(n_G^2)$  size. Managing the size of transitive closure of a graph has been studied for decades. One of the promising techniques is the 2-hop node-labeling approach [1, 8, 26], which computes a small in-memory index for efficiently computing the shortest path between any pair of nodes. As proposed in [21], we could pre-compute and store in the transitive closure  $G_c$  only the ‘hot’ lists based on these techniques, while

others may be computed on the fly by using the 2-hop node labeling techniques in [1, 8, 26] to compute shortest paths.

**Supporting Top- $k$  Graph Pattern Matching.** [7] investigates the problem of *top-k graph pattern matching*, namely, kGPM; that is, we replace a rooted tree  $T$  in kTPM with a general labeled undirected graph and replace a labeled directed data graph with a labeled undirected graph, while the others remain the same.

The authors in [7] propose a query decomposition approach to computing top- $k$  matches of a graph pattern  $q$ . The central idea is to decompose  $q$  into a set of spanning trees and then run top- $k$  (undirected) tree pattern matching algorithms. Note that our techniques can be immediately extended to support top- $k$  undirected tree pattern matching by using an undirected tree  $T$  as a query. To do this, we choose a node in  $T$  to be the root node and make  $T$  as a rooted tree. For each edge in the data graph, we make it bidirectional. Thus, our algorithms are immediately applicable. Consequently, such extended techniques for kTPM with an undirected tree can be immediately embedded into the framework proposed in [7].

## 6. EXPERIMENTS

We report the results of our empirical studies. For kTPM with distinct node labels in  $T$ , the following algorithms are evaluated:

- DP-B and DP-P: the two state-of-the-art algorithms [21], used as baseline algorithms.
- Topk: Algorithm 1 in Section 3.
- Topk-EN: Algorithm 3 in Section 4.

For kTPM with different nodes having the same label in  $T$ , we extend Topk-EN as discussed in Section 5, referred to as Topk-GT. Regarding kGPM, we evaluate the following two algorithms:

- mtree: the state-of-the-art algorithm for kGPM [7].
- mtree<sup>+</sup>: as discussed in Section 5, extend our Topk-EN algorithm and embed it into the framework in [7].

Java bytecodes of DP-B and DP-P are obtained from the authors of [21]. We implement our algorithms, Topk, Topk-EN, Topk-GT, in the same environment (i.e., Java 1.5.0) to conduct a fair comparison. C++ source code of mtree is obtained from the authors of [7], and we also implement our mtree<sup>+</sup> in the same codebase in C++. All experiments are conducted on a set of PCs, each with an Intel(R) Xeon(R) 2.66GHz CPU and 4GB memory running Linux. We evaluate the performance of the algorithms on both real and synthetic datasets as follows.

**Real Datasets.** We use *DBLP* as the real dataset in our experiments, which is a computer science bibliography network<sup>4</sup>. In *DBLP*, each node represents a paper, each edge represents a citation relationship between two papers, and the label of a node is either the conference name or the journal name in which the paper appears. There are 1,180,072 nodes, 2,564,678 edges, and 3,136 different labels. From *DBLP*, we randomly extract five connected induced subgraphs by random walks,  $G_{D1}$ ,  $G_{D2}$ ,  $G_{D3}$ ,  $G_{D4}$ , and  $G_{D5}$ , with  $10^4$ ,  $5 \times 10^4$ ,  $10^5$ ,  $2 \times 10^5$ , and  $10^6$  nodes, respectively. We use  $G_{D3}$  as the *default real dataset*.

**Synthetic Datasets.** A synthetic graph is a power-law graph generated from the Boost Graph Library [10] with average out-degree 3, where node labels are randomly assigned from a set of 200 different labels. We generate six connected synthetic graphs  $G_{S1}$ ,  $G_{S2}$ ,  $G_{S3}$ ,  $G_{S4}$ ,  $G_{S5}$ , and  $G_{S6}$ , with  $10^4$ ,  $5 \times 10^4$ ,  $10^5$ ,  $2 \times 10^5$ ,  $10^6$ , and  $2 \times 10^6$  nodes, respectively. The *default synthetic dataset* is  $G_{S3}$ .

**Query Set.** For each real data graph and synthetic data graph, we use random walks to randomly generate five query sets,  $T_{10}$ ,  $T_{20}$ ,  $T_{30}$ ,  $T_{50}$ , and  $T_{70}$ . Each generated  $T_i$  has 100 rooted trees that are

<sup>3</sup><http://www.ins.cwi.nl/projects/xmark/Assets/xmlquery.txt>

<sup>4</sup><http://dblp.uni-trier.de/>

subtrees of the run-time graph, and each query tree has  $i$  nodes with distinct node labels. The *default query set* is  $T_{50}$ . Since in real data graphs, we cannot generate  $T_{100}$ , we generate  $T_{100}$  in addition to the five query sets above for the synthetic data graphs.

$k$  varies from 10 to 100 with *default value* 20. *Unless otherwise specified, default settings are adopted in our experiments.* Note that in the experiments, total time means average total time, including both CPU and I/O time.

## 6.1 Experimental Results

**Eval-I: Pre-Computation Cost of Transitive Closure.** The computation time and sizes of transitive closures (i.e., organized in the form of tables  $L_{\beta}^{\alpha}$  as discussed in Sections 3.1 and 4.1) are shown in Table 2, where the first three columns show that for real datasets and the last three columns for synthetic datasets. Note that the pre-computation is conducted off-line.

Graph	Time of TC (s)	Size of TC (GB)	Graph	Time of TC (s)	Size of TC (GB)
$G_{D1}$	80	0.123	$G_{S1}$	26	0.261
$G_{D2}$	335	0.877	$G_{S2}$	181	1.004
$G_{D3}$	631	1.8	$G_{S3}$	843	2.5
$G_{D4}$	945	4.5	$G_{S4}$	2,438	7.3
$G_{D5}$	39,305	98	$G_{S5}$	23,785	77
			$G_{S6}$	69,688	247

Table 2: Computational costs of transitive closures

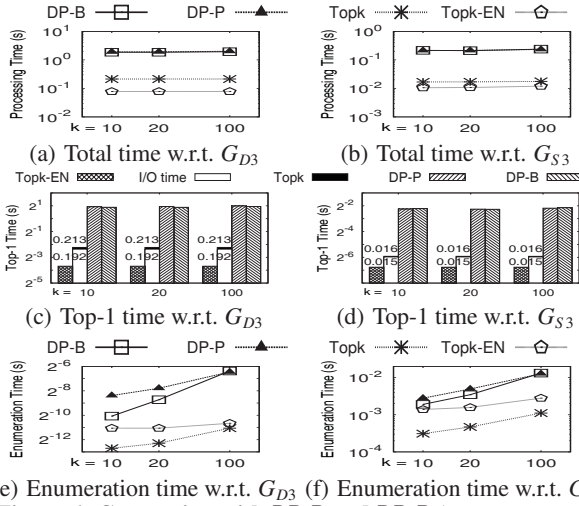


Figure 6: Comparing with DP-B and DP-P ( $T = T_{20}$ , vary  $k$ )

**Eval-II: Evaluating Topk & Topk-EN over DP-B & DP-P.** We use default data graphs and  $T_{20}$  (i.e., query trees with 20 nodes). We do not vary the sizes of query and data graphs since the byte-codes obtained for DP-B and DP-P cannot run for large queries and graphs. Figure 6(a) and Figure 6(b) show the total running time of the four algorithms. As demonstrated, Topk and Topk-EN are significantly faster than DP-B and DP-P, and Topk-EN achieves up to 2 orders of magnitude speed-up against DP-B and DP-P. The experiment also shows that when  $k$  is small, our techniques are not quite sensitive to the values of  $k$ . This is because the query processing time is dominated by the time for computing the top-1 match when  $k$  is small.

Figures 6(c) and 6(d) report the time to generate the top-1 result by the four algorithms, respectively, including I/O time and CPU time. Since DP-B and Topk include the total I/O time (i.e., in loading in the run-time graph) in generating the top-1 result, we use the blank bars in Topk to illustrate the I/O time for loading in the run-time graph. As depicted, our algorithms significantly outperform DP-B and DP-P, and Topk-EN performs the best. Figure 6(e) and Figure 6(f) show the total time for generating the top-1 results

after generating the top-1 result. It shows that DP-P and Topk-EN are slower than DP-B and Topk, respectively. This is because that during the computation, both DP-P and Topk-EN involve I/O costs but DP-B and Topk do not. When  $k$  gets larger, the time of DP-P and Topk-EN is closer to that of DP-B and Topk, respectively. This is because that the CPU costs of DP-P and Topk-EN are lower than those of DP-B and Topk in practice; thus, when  $k$  gets larger, the I/O costs in DP-P and Topk-EN become less significant.

In fact, for small queries (e.g., with 5 - 20 nodes), our techniques Topk and Topk-EN also outperform DP-B and DP-P by up to two orders of magnitude, due to space limits we do not report the results here. Thus, next we will only evaluate the scalability of our techniques and discard comparisons with DP-B and DP-P.

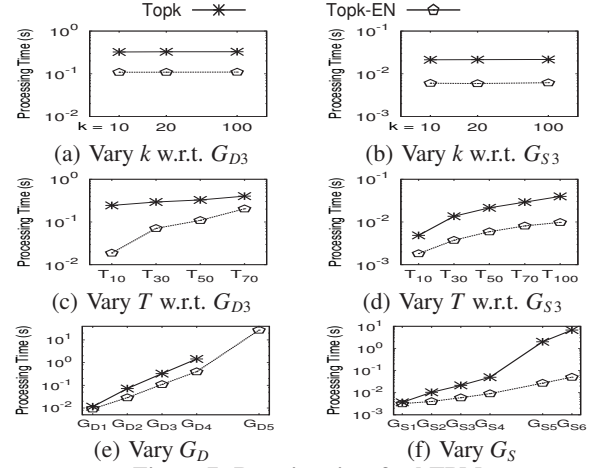


Figure 7: Running time for kTPM

**Eval-III: Scalability Testing for kTPM.** We evaluate the scalability of Topk and Topk-EN against  $k$ , and the sizes of  $T$  and  $G$ .

Figure 7(a) and Figure 7(b) demonstrate the impacts of  $k$  with  $T_{50}$  that are similar to those in Figure 6(a) and Figure 6(b).

Figure 7(c) and Figure 7(d) evaluate the impacts of query sizes. Note that for real data graphs, we are unable to retrieve  $T_{100}$  due to graph characteristics; thus maximal query is  $T_{70}$ . While Topk-EN still significantly outperforms Topk, the running time of Topk-EN grows faster on real data graph than that on synthetic data graph. This is because that when the size of  $T$  increases, the ratio of average degree of run-time graph of real graph over that of synthetic graph increases; subsequently, the ratio of number of edges loaded in by Topk-EN for real graph over that for synthetic graph also increases. Table 3 shows the sizes of run-time graphs for real graph and synthetic graph, respectively.

	$G_{D3}$		$G_{S3}$	
	#nodes of $G_R$	#edges of $G_R$	#nodes of $G_R$	#edges of $G_R$
$T_{10}$	$39.5 \times 10^3$	$1,826 \times 10^3$	$5 \times 10^3$	$32 \times 10^3$
$T_{30}$	$45.5 \times 10^3$	$2,361 \times 10^3$	$15 \times 10^3$	$107 \times 10^3$
$T_{50}$	$51.4 \times 10^3$	$2,840 \times 10^3$	$25 \times 10^3$	$178 \times 10^3$
$T_{70}$	$55.4 \times 10^3$	$3,007 \times 10^3$	$35 \times 10^3$	$250 \times 10^3$
$T_{100}$			$50 \times 10^3$	$394 \times 10^3$

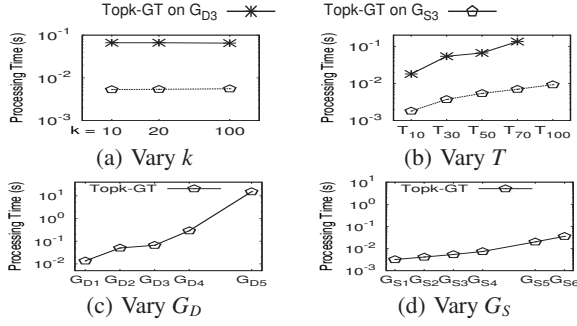
Table 3: Average sizes of run-time graphs (i.e.,  $G_R$ )

Figure 7(e) and Figure 7(f) demonstrate the impacts of data graph sizes. Topk-EN still significantly outperforms Topk. Moreover, Topk even cannot run on  $G_{D5}$  due to running out of memory.

**Eval-IV: General Top-k Twig-Pattern Matching.** Query sets are generated in a similar way except that node labels are not enforced to be distinct. In fact, each query tree generated has multiple nodes with the same label, and the *average label duplication ratios* (i.e.,  $1 - \frac{\text{\#distinct labels}}{\text{\#nodes}}$ ) are 17.2% (for  $T_{10}$ ), 42.3% ( $T_{30}$ ), 50.2% ( $T_{50}$ ), and 54.5% ( $T_{70}$ ) regarding  $G_{D3}$ , and 2.2% (for  $T_{10}$ ), 7.3% ( $T_{30}$ ), 13.8% ( $T_{50}$ ), 18.6% ( $T_{70}$ ), and 24.9% ( $T_{100}$ ) regarding  $G_{S3}$ . Figure 8 re-

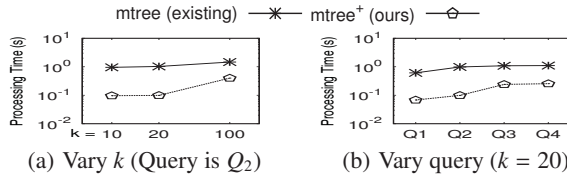


ports the evaluation result where the default settings remain unchanged. Similar trends to Topk-EN are obtained.



**Figure 8: Running time for general twig-pattern matching**

**Eval-V: kGPM.** Figure 9(a) and Figure 9(b) show the evaluation results by comparing *mtree*<sup>+</sup> with *mtree* against different settings. As expected, *mtree*<sup>+</sup> significantly outperforms *mtree*.



**Figure 9: Running time for kGPM**

## 7. CONCLUSION

In this paper, we proposed a novel and optimal enumeration paradigm for kTPM with distinct labels on query nodes. We also proposed a priority order based algorithm, by avoiding loading in the entire run-time graph, to improve the performance of computing top-*k* matches in practice. We have shown that our approaches can be extended to general top-*k* twig-pattern matching, and general top-*k* graph pattern matching. Our extensive experiments demonstrate the efficiency and scalability of our techniques, and our techniques outperform the existing techniques by several orders of magnitude. As possible future work, selecting the “best” node as a root from an undirected tree to use our techniques might be an interesting issue to be investigated. Another interesting issue is to generate the “diverse” top-*k* results.

**Acknowledgements.** Lijun Chang is supported by ARC DE150100563. Xuemin Lin is supported by NSFC61232006, ARC DP120104168, ARC DP140103578, and ARC DP150102728. Wenjie Zhang is supported by ARC DE120102144, DP120104168, ARC DP150103071 and DP150102728. Jeffrey Xu Yu is supported by Research Grants Council of the Hong Kong SAR, China, 14209314 and 418512. Ying Zhang is supported by ARC DE140100679 and DP130103245. Lu Qin is supported by ARC DE140100999.

## 8. REFERENCES

- [1] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proc. of SIGMOD'13*, 2013.
- [2] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *Proc. of ICDE'02*, 2002.
- [3] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal xml pattern matching. In *Proc. of SIGMOD'02*, 2002.
- [4] L. Chang, X. Lin, W. Zhang, J. X. Yu, and Y. Zhang. Optimal enumeration: Efficient top-*k* tree matching. In *UNSW-CSE-TR-201417*, <http://goo.gl/r8CIwa>.
- [5] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *Proc. of VLDB'05*, 2005.
- [6] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang. Fast graph pattern matching. In *Proc. of ICDE'08*, 2008.
- [7] J. Cheng, X. Zeng, and J. X. Yu. Top-*k* graph pattern matching over large graphs. In *Proc. of ICDE'13*, 2013.
- [8] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *Proc. of SODA'02*, 2002.
- [9] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.
- [10] B. Dawes and D. Abrahams. Boost c++ libraries. <http://www.boost.org/>.
- [11] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-*k* min-cost connected trees in databases. In *Proc. of ICDE'07*, 2007.
- [12] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. of PODS'01*, 2001.
- [13] W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu. Graph homomorphism revisited for graph matching. *PVLDB*, 3(1), 2010.
- [14] W. Fan, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractable to polynomial time. *PVLDB*, 2010.
- [15] W. Fan, X. Wang, and Y. Wu. Diversified top-*k* graph pattern matching. *PVLDB*, 6(13), 2013.
- [16] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2), 1998.
- [17] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.
- [18] A. Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, 1985.
- [19] G. Gottlob, C. Koch, and R. Pichler. The complexity of xpath query evaluation. In *Proc. of PODS'03*, 2003.
- [20] G. Gou and R. Chirkova. Efficiently querying large xml data repositories: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(10), 2007.
- [21] G. Gou and R. Chirkova. Efficient algorithms for exact ranked twig-pattern matching over graphs. In *Proc. of SIGMOD'08*, 2008.
- [22] W.-S. Han, J. Lee, and J.-H. Lee. Turbo<sub>iso</sub>: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proc. of SIGMOD'13*, 2013.
- [23] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems Science and Cybernetics*, 4(2), 1968.
- [24] P. Hell and J. Nešetřil. *Graphs and Homomorphisms*. Oxford Lecture Series in Mathematics and Its Applications, 2004.
- [25] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-*k* query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.
- [26] M. Jiang, A. W.-C. Fu, R. C.-W. Wong, and Y. Xu. Hop doubling label indexing for point-to-point distance querying on scale-free networks. *PVLDB*, 7(12), 2014.
- [27] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *Proc. of SIGMOD'00*, 2000.
- [28] E. L. Lawler. A procedure for computing the *k* best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7), 1972.
- [29] Y. Qi, K. S. Candan, and M. L. Sapino. Sum-max monotonic ranked joins for evaluating top-*k* twig queries on weighted data graphs. In *Proc. of VLDB'07*, 2007.
- [30] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 1(1), 2008.
- [31] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1), 1976.
- [32] W3C. Xml path language (xpath) version 1.0. <http://www.w3.org/TR/xpath>, 1999.
- [33] J. X. Yu, L. Qin, and L. Chang. *Keyword Search in Databases*. Morgan & Claypool, 2009.
- [34] Q. Zeng, X. Jiang, and H. Zhuge. Adding logical operators to tree pattern queries on graph-structured data. *PVLDB*, 5(8), 2012.
- [35] G. Zhu, X. Lin, K. Zhu, W. Zhang, and J. X. Yu. Treespan: efficiently computing similarity all-matching. In *Proc. of SIGMOD'12*, 2012.
- [36] L. Zou, L. Chen, and M. T. Özsu. Distancejoin: Pattern match query in a large graph database. *PVLDB*, 2(1), 2009.