Optimal Enumeration: Efficient Top-k Tree Matching

Lijun Chang[1] Xuemin Lin[1] Wenjie Zhang[1]
Jeffrey Xu Yu[2] Ying Zhang[3] Lu Qin[3]

[1] University of New South Wales, Australia
{ljchang,lxue,zhangw}@cse.unsw.edu.au
[2] The Chinese University of Hong Kong, China
yu@se.cuhk.edu.hk
[3] University of Technology, Sydney, Australia
{Ying.Zhang,Lu.Qin}@uts.edu.au

Technical Report
UNSW-CSE-TR-1417
June 2014

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

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.

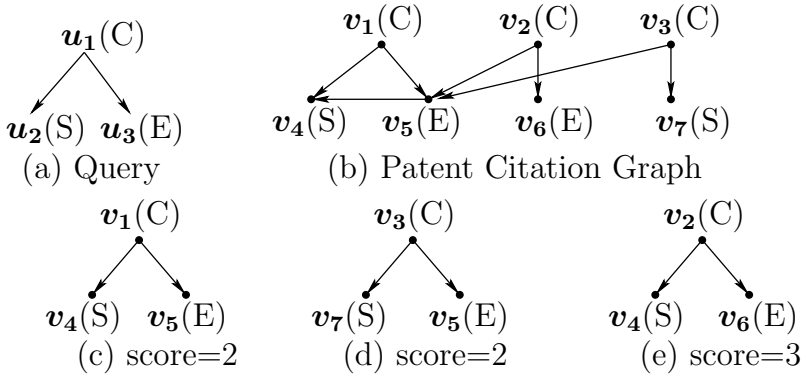


Figure 1.1: Example

The problem of *top-k tree pattern matching* over graph data is investigated in [20]. That is, given a rooted tree T and a directed 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 1.1(a) and 1.1(b), respectively, where the node labels are displayed in brackets. Assuming that each edge in the data graph takes the weight 1, Figures 1.1(c) and 1.1(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 1.1(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* [17] 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 1.1(b) is a tinny portion of a citation graph[1], 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 1.1(a) are of the '//' type. Then, the twig query in Figure 1.1(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 1.1(a), the triple (x, y, z) of nodes with the smallest total length of matched

[1] `http://www.nber.org/patents`

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 1.1(c)) than the impact of v_2 on the combination of v_4 and v_6 (i.e., the match in Figure 1.1(e)). Therefore, the top-k tree pattern matching can be used to identify the k results of a twig query with the highest preference/relevance among a possibly exponential number of results.

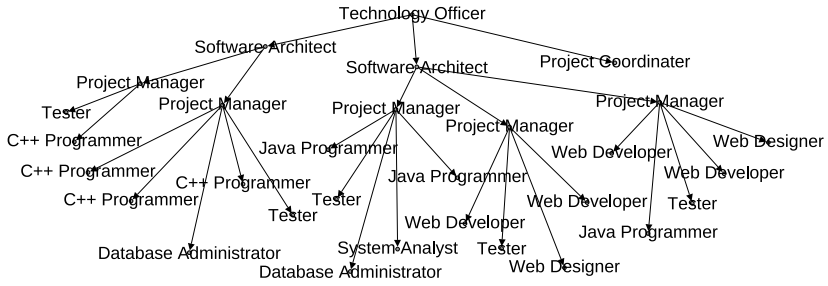


Figure 1.2: Example team finding query

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 a professional social network such as LinkedIn or Xing [2]. Here, each node represents a professional and has a label (or multiple labels) representing the major skill and/or strength of the professional, and each edge represents a relationship between two professionals which can be either directed (e.g., advisor, mentor) or undirected (e.g., collaborator). Note that, in LinkedIn, tags (e.g., advisor, mentor) could be added to relationships, thus edges can be directed. Each edge is possibly assigned a weight to measure the effectiveness of communication between two individuals, and [27] discusses how to assign such weights. Thus, the shortest distance between two nodes may be used to measure the likelihood for two corresponding professionals to collaborate well with each other. Based on such professional social networks, we can find teams of professionals with certain skills by issuing a top-k tree matching query as shown in Figure 1.2. In such applications, a query tree may have tens of nodes.

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 [6], the techniques in [20] 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 [20] 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

[2] `https://www.linkedin.com/`, `https://www.xing.com/`

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 jth ($j \leq k$) "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 [20]. 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 [20]. 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 θ is the average number of edges of the same type in the transitive closure. Here, 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. Therefore, the time complexity of our algorithm is $O(n_T(\theta + k) + k \log k)$, while the algorithm DP-B in [20] runs in time $O(n_T(\theta \log k + k \log k + kd_T))$. Although θ can be quadratic to the number of nodes in G in the worst case, it is small in practice; for example, in DBLP data with 1.18 million nodes, $\theta = 5900$; that is, our algorithm behaves linearly regarding n_T.

[20] 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 [20].

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 with $O(m_R + k(\log k + n_T))$ running time in contrast to the existing techniques in [20] 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 [6] for top-k graph pattern matching.

We conduct empirical studies on large real and synthetic graphs. Extensive perfor-

mance studies demonstrate that the proposed algorithms significantly outperform the state-of-the-art algorithms in [20] 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 of kTPM, 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.

Related Work. Top-k query answering for relational data has been extensively studied. The first fundamental work may be found in [11], and a survey is presented in [24]. Various top-k queries have also been investigated over spatial data, such as kNN [15], RkNN [26], etc. Nevertheless, these techniques are not applicable to computing top-k graph pattern matches [6, 20].

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, 21]. 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 [21, 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 [6, 12, 20, 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 [4, 19, 34], a directed acyclic graph [4], and a general graph [5]. 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, 18]. 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, 18]). The most related works are from [6, 20], where [20] studies the top-k tree pattern matching and [6] 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 [6, 20].

3. Graph Simulation. Another structure matching is bounded graph simulation [13]. 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 [14]. Nevertheless, due to inherently different problem natures, these techniques are inapplicable to our top-k pattern matching problems.

Others. Generalizations of graph *homomorphism* [23] and graph isomorphism [16] may be found in [12], 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 [10], is investigated in [2, 10], 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, 18] considers matches between two trees, while the subgraph isomorphism in [30, 31, 21] 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 [4, 5, 19, 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* $G = (V, E, l)$ [17]; hereafter, a data graph in this paper 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 Σ is a set of alphabets.

We use (v, v') to denote an edge from v to v' and (v, v') is referred 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* [17] 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.

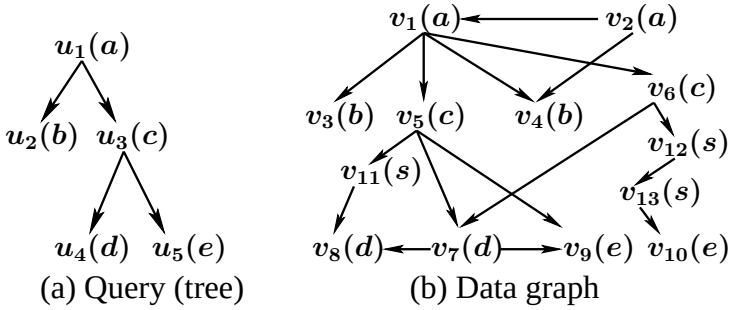


Figure 2.1: A rooted tree and a graph

Figures 2.1(a) and 2.1(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 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, that is, $\forall u \in V_T$, $l(u) = l(f(u))$; 2) f preserves the structure information, that is, for each edge (u, u') in T, there is a directed path from $f(u)$ to $f(u')$ in G. □

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 patten 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,

$$S(M_f) = \sum_{(u,u') \in E_T} \delta_{min}(f(u), f(u')) \tag{2.1}$$

where $\delta_{min}(f(u), f(u'))$ is the shortest distance from $f(u)$ to $f(u')$ in G. [1] □

Example 2.1: Regarding the query and the data graph in Figures 2.1(a) and 2.1(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 $S(M_{f_1}) = 4$. □

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 [20], 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 [6].

Frequently used notations are summarized in Table 2.1. Most of them will be defined in the following sections.

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 α 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 2.1: Frequently used notations

[1] Note that, our techniques directly apply if node weights are also considered in the penalty score.

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.

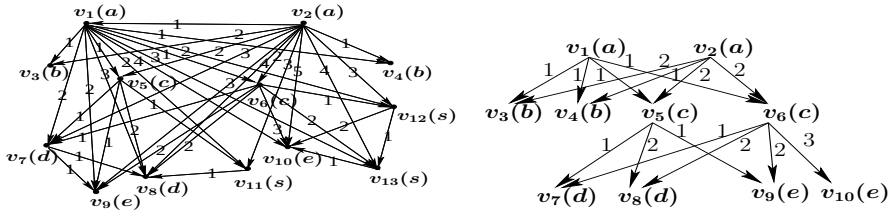
3.1 Notation and Pre-Computation

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 [8] 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 [20] 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 3.1(a) shows the transitive closure of the data graph in Figure 2.1(b), and Figure 3.1(b) shows the run-time graph regarding T in Figure 2.1(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 θ 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 α. For example, in Figure 3.1(b), $v_1.children_c = \{v_5, v_6\}$.



(a) Transitive closure (b) Run-time graph

Figure 3.1: Run-time graph

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 [6, 20], for each pair of node labels $\alpha, \beta \in \Sigma$ in G, we store in table L^α_β 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 lth 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 $V_T = \{u_1, u_2, ..., u_{n_T}\}$ is the node set of T. For each node $u_i \in V_T$, let $\mathbb{V}_i$ 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 $S = \mathbb{V}_1 \times \mathbb{V}_2 \times \cdots \times \mathbb{V}_{n_T}$. Suppose that $M_1 = (v_1, v_2, ..., v_{n_T})$ 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: $S_1 = (\mathbb{V}_1 - \{v_1\}) \times \mathbb{V}_2 \times \cdots \times \mathbb{V}_{n_T}$, $S_2 = \{v_1\} \times (\mathbb{V}_2 - \{v_2\}) \times \mathbb{V}_3 \times \cdots \times \mathbb{V}_{n_T}$, ..., $S_{n_T} = \{v_1\} \times \{v_2\} \times \cdots \times \{v_{n_T-1}\} \times (\mathbb{V}_{n_T} - \{v_{n_T}\})$. Clearly, $\{M_1\}, S_1, \ldots, S_{n_T}$ 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 $(v_1, ..., v_{i-1})$. To obtain the top-3 match, the procedure continues to further divide $S_i = \{v_1\} \times \cdots \times \{v_{i-1}\} \times (\mathbb{V}_i - \{v_i\}) \times \mathbb{V}_{i+1} \times \cdots \times \mathbb{V}_{n_T}$ into $(n_T - i + 1)$ disjoint subspaces by fixing $(v_1, ..., v_{i-1})$ and dividing $\{(\mathbb{V}_i - \{v_i\}), \mathbb{V}_{i+1}, ..., \mathbb{V}_{n_T}\}$ 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 $S'_j = \{v_{l_1}\} \times \cdots \times \{v_{l_{j-1}}\} \times (\mathbb{V}_j - U_j) \times \mathbb{V}_{j+1} \times \cdots \times \mathbb{V}_{n_T}$, where U_j denotes the subset of $\mathbb{V}_j$ to be excluded in obtaining M_i for $1 \leq i \leq l$, $(v_{l_1}, \ldots, v_{l_{j-1}})$ are fixed in the subspace S'_j and mapped from $(u_1, \ldots, u_{j-1})$. Note that we use l_j to denote the subscript of a node in G_R. Suppose that M_l is $\{v_{l'_1}, ..., v_{l'_{n_T}}\}$, 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: $\{v_{l_1}\} \times \cdots \times \{v_{l_{j-1}}\} \times (\mathbb{V}_j - U_j - \{v_{l'_j}\}) \times \mathbb{V}_{j+1} \times \cdots \times \mathbb{V}_{n_T}$, $\{v_{l_1}\} \times \cdots \times \{v_{l_{j-1}}\} \times \{v_{l'_j}\} \times (\mathbb{V}_{j+1} - \{v_{l'_{j+1}}\}) \times \mathbb{V}_{j+2} \times \cdots \times \mathbb{V}_{n_T}$, ..., $\{v_{l_1}\} \times \cdots \times \{v_{l_{j-1}}\} \times \{v_{l'_j}\} \times \cdots \times \{v_{l'_{n_T-1}}\} \times (\mathbb{V}_{n_T} - \{v_{l'_{n_T}}\})$. 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 2.1(a) over the run-time graph in Fig-

ure 3.1(b). $\mathbb{V}_1 = \{v_1, v_2\}, \mathbb{V}_2 = \{v_3, v_4\}, \mathbb{V}_3 = \{v_5, v_6\}$, $\mathbb{V}_4 = \{v_7, v_8\}, \mathbb{V}_5 = \{v_9, v_{10}\}$. The top-1 match of T is $(v_1, v_3, v_5, v_7, v_9)$; thus, in the expression of $(v_{1_1}, v_{1_2}, \ldots, v_{1_5})$, $1_1 = 1, 1_2 = 3, 1_3 = 5, 1_4 = 7$, and $1_5 = 9$. Then, the entire solution space is divided into 5 subspaces, $S_1 = (\mathbb{V}_1 - \{v_1\}) \times \mathbb{V}_2 \times \cdots \times \mathbb{V}_5, S_2 = \{v_1\} \times (\mathbb{V}_2 - \{v_3\}) \times \mathbb{V}_3 \times \cdots \times \mathbb{V}_5, \ldots, S_5 = \{v_1\} \times \cdots \times \{v_7\} \times (\mathbb{V}_5 - \{v_9\})$. The top-2 match of T is $(v_1, v_4, v_5, v_7, v_9)$, thus, in the expression of $(v_{2_1}, v_{2_2}, \ldots, v_{2_5})$, $2_1 = 1, 2_2 = 4, 2_3 = 5, 2_4 = 7$, and $2_5 = 9$, which is the best match in S_2. S_2 is further divided into 4 subspaces, $S'_1 = \{v_1\} \times (\mathbb{V}_2 - \{v_3, v_4\}) \times \mathbb{V}_3 \times \cdots \times \mathbb{V}_5$, $S'_2 = \{v_1\} \times \{v_4\} \times (\mathbb{V}_3 - \{v_5\}) \times \mathbb{V}_4 \times \mathbb{V}_5, \ldots$, 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. □

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 [20] 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 [20]. Below, we show that obtaining the best match in such 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 2.1(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, $u_1, \ldots, u_{n_T}$, 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 $j < i$.* □

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 $M_l = (v_{l_1}, v_{l_2}, \ldots, v_{l_{n_T}})$ in Lawler's procedure belongs to one of the following two cases, where $l > 1$.

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

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

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. When $l > 1$, 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 does not have tree-pattern 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 α.

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.* □

Proof: Note that according to Lemma 3.1, $l_p < l_j$ and $l_y < l_x$; thus, v_{l_p} needs to be fixed in S', and v_{l_y} needs to be fixed in S'' as well. Based on this and the connectivity requirement, the Lemma immediately holds. □

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 $v_{l_p}.children_{l(v_{l_j})}$.* □

Proof: Since any match M in the subspace S' must contain $\{v_{l_1}, \ldots, v_{l_{j-1}}\}$ and $v_{l_p} \in \{v_{l_1}, \ldots, v_{l_{j-1}}\}$, v_{l_p} must be contained in M. The connectivity requirement also requires that each node from $\mathbb{V}_j$ in M must be connected to v_{l_p}. According to Lawler's procedure, any node $v' \in U_j$ is involved in a top-i match ($i \leq l$) that must contain $\{v_{l_1}, \ldots, v_{l_{j-1}}\}$; that is, v' also uses v_{l_p} as the parent. Consequently, $bs(v') + \delta_{min}(v_{l_p}, v') \leq bs(v) + \delta_{min}(v_{l_p}, v)$ (otherwise v shall be selected earlier instead of v').

It is also immediate that in M_l, for any child v' of a node $v_{l_y} \in \{v_{l_1}, ..., v_{l_{j-1}}\}$ such that $v' \in \mathbb{V}_i$ with $i > j$, the score of $bs(v') + \delta_{min}(v_{l_y}, v')$ must be the minimum among all nodes in $v_{l_y}.children_{l(v')}$. Moreover, according to the connectivity requirement, in any match M in S', a node from $\mathbb{V}_i$ must be connected to v_{l_y}.

Therefore, the Theorem holds. □

Similarly, we can prove the following Theorem.

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 $v_{l_y}.children_{l(v_{l_x})}$.* □

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

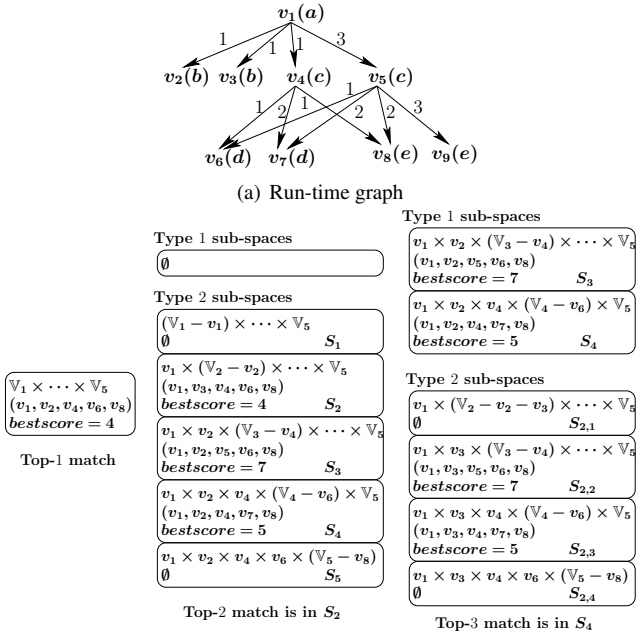
Input: A query rooted tree T, a run-time graph G_R, and k.
Output: $\mathcal{M}$: Top-k matches of T in G_R.

1: $\mathcal{M} \leftarrow \emptyset$; $l \leftarrow 0$;
2: Initialize an empty collection Q;
3: Compute the top-1 match, M, in the subspace $S = \mathbb{V}_1 \times \cdots \times \mathbb{V}_{n_T}$;
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 $(v_{l_1}, \ldots, v_{l_{n_T}})$ obtained from $S' = \{v_{l_1}\} \times \cdots \times \{v_{l_{j-1}}\} \times (\mathbb{V}_j - U_j) \times \mathbb{V}_{j+1} \times \cdots \times \mathbb{V}_{n_T}$; {/* if $l = 1$, then $j = 1$, $U_j = \emptyset$, and skip Lines 10-11 */}
10: Compute the best match M_j in subspace $S'_j = \{v_{l_1}\} \times \cdots \times \{v_{l_{j-1}}\} \times (\mathbb{V}_j - U_j - \{v_{l_j}\}) \times \mathbb{V}_{j+1} \times \cdots \times \mathbb{V}_{n_T}$, 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 $S'_x = \{v_{l_1}\} \times \cdots \times \{v_{l_{x-1}}\} \times (\mathbb{V}_x - \{v_{l_x}\}) \times \mathbb{V}_{x+1} \times \cdots \times \mathbb{V}_{n_T}$, as described in Theorem 3.2;
14: Put (M_x, S'_x) into Q;

Example 3.2: Figure 3.2(b) shows a running example of kTPM regarding the run-time graph in Figure 3.2(a), where the query tree T is shown in Figure 2.1(a). Each rectangle shows an entry in Q, where 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, $S = \mathbb{V}_1 \times \cdots \times \mathbb{V}_5$, 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 part of Figure 3.2(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: $(S_1, \ldots, S_5)$, as shown in the middle part of Figure 3.2(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, that is, $(v_1, v_3, v_4, v_6, v_8)$ with score 4.

To compute top-3 match of T, S_2 is further divided into 4 subspaces, Case 1: $(S_{2,1})$,



(a) Run-time graph

(b) Running example

Figure 3.2: Example of kTPM

Case 2: $(S_{2,2}, S_{2,3}, S_{2,4})$, as shown in the right part of Figure 3.2(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. □

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 [8], 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 α 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 $\forall \alpha \in \Pi_{q(v)}, L_{v,\alpha} \neq \emptyset$.

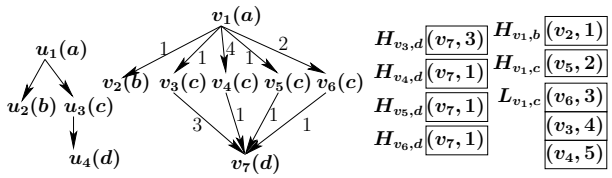
$$bs(v) = \sum_{\alpha \in \Pi_{q(v)}} \min\{bs(v') + \delta_{\min}(v, v') \mid v' \in L_{v,\alpha}\} \tag{3.1}$$

Theorem 3.1 states that in Line 10 in Algorithm 1, we get the node with the $(|U_j| + 2)$th lowest value from $L_{v,l(v_{l_j})}$ 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) [8] to store $L_{v,\alpha}$; we use a binary minimum priority queue in our implementation. It shows in [8] 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 ith 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 α of $q(v)$ in T as shown in Equation 3.1. Note that if v has an empty $H_{v,\alpha}$ for a child label α 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.



(a) T (b) G_R (c) L and H lists

Figure 3.3: Example of building data structure

Example 3.3: Regarding the tree query T in Figure 3.3(a) over the run-time graph G_R in Figure 3.3(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 $3 \le i \le 6$ as shown in the left part of Figure 3.3(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 $(bs(v_5) + \delta_{min}(v_1, v_5))$, therefore, 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 3.3(c). The the final data structure is shown in Figure 3.3(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 $H_{v_1,c}$. □

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 α defines $\mathbb{V}_x$, since 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_{l_j} 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 3.3(a) over the run-time graph G_R in Figure 3.3(b). Figure 3.3(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 $H_{1,c}$. 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 $L_{v_1,c}$. □

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 $T_{q(v_y)}$ rooted at v_y to generate M_j; note $l(v_x) = l(v_y)$. To get the best match of $T_{q(v_y)}$ 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 $Q_1, \ldots, Q_k$. 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 Analysis. The time for creating L and H lists, as well as the top-1 match of T, is $O(m_R)$ as discussed above. Each iteration in generating a top-l match of T takes $O(\log d_R + n_T + \log k + \log n_T)$ time where d_R is the maximum degree of nodes in G_R, since the following non-constant time based operations are executed only once: $O(\log d_R)$ time to remove the top from an L list, $O(\log d_R)$ time to insert an element to an H list and retrieve the jth element in H, $O(\log n_T)$ time to pop up the top of a Q_j list, and $O(\log k)$ time to insert a top of Q_j to Q (note that $|Q| \leq k$). Note that the time complexity also includes the time to output the top-k matches. Therefore, the total costs are $O(m_R + k(\log d_R + n_T + \log n_T + \log k))$ time and $O(m_R + kn_T)$ space. Nevertheless, taking the advantage that k is given (the assumption taken in [20]), we can cut the time complexity to $O(m_R + k(n_T + \log k))$ as follows.

If k is given, it is immediate that we only need to retain no more than k elements with the lowest values in each L list (thus, also H list), as well as only k elements in each Q_i. Note that it is shown in [8] that computing the jth element from an unordered set D takes linear time $O(|D|)$. Consequently, it takes linear time $O(|L_{v,\alpha}|)$ to get the k elements with lowest values and create a binary minimum priority queue on these k elements. Therefore, the time complexity of our algorithm is reduced to $O(m_R + k(n_T + \log k))$ which is optimal, while DP-B [20] 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 [20] takes space $O(\min\{m_R, kn_R d_T\} + kn_T d_T)$, where d_T is the maximum node degree in T, since for each $L_{v,\alpha}$ and $H_{v,\alpha}$ we only need to store at most k elements. As discussed in Section 1 and Section 3.1, $m_R = \theta n_T$ on average, where θ is the average number of edges of the same type in the transitive closure and θ is small in practice. Therefore, the space complexity of our algorithm 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 [20], 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^α_β (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^α_v. Edges in the group L^α_v 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^α_v, we only need to store in L^α_v the adjacent node v' coming to v and the corresponding shortest distance $\delta_{\min}(v', v)$. In practice, an L^α_v may contain many nodes; for example, an L^α_v 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^α_v, we also keep the information d^α_v, the minimum value of the shortest distances from nodes in L^α_v to v (i.e., $d^\alpha_v = \min\{\delta_{\min}(v', v) \mid v' \in L^\alpha_v\}$); note that d^α_vs are not stored with L^α_v.

Example 4.1: For the data graph in Figure 2.1(b), we store its closure (Figure 3.1(a)) as follows. For example, for v_5, $L^a_{v_5} = \{(v_1, 1), (v_2, 2)\}$, where $d^a_{v_5} = \delta_{min}(v_1, v_5) = 1$; $E_{v_5} = \{(v_5, v_7, 1), (v_5, v_9, 1), (v_5, v_{11}, 1)\}$ 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, $L^a_{v_6} = \{(v_1, 1), (v_2, 2)\}$, $d^a_{v_6} = 1$, and $E_{v_6} = \{(v_6, v_{12}, 1), (v_6, v_7, 1), (v_6, v_9, 2)\}$. □

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

$D^\alpha_\beta = \{(v, d^\alpha_v) \mid l(v) = \beta\}$;	$L^\alpha_\beta = \{(v, L^\alpha_v) \mid l(v) = \beta\}$
$\mathbb{E}^\alpha_\beta = \{(v, v', \delta_{min}(v, v')) \mid l(v) = \alpha, l(v') = \beta,$ and $(v, v', \delta_{min}(v, v')) \in E_v\}$	

For example, in Example 4.1, since $d^c_{v_7} = 1$, $d^c_{v_7}$ is not stored in D^c_d. Consequently, D^c_d stores only one element $(v_8, 2)$; that is, $D^c_d = \{(v_8, 2)\}$. In $\mathbb{E}^c_d$, we store $\{(v_5, v_7, 1), (v_6, v_7, 1)\}$, while in $\mathbb{E}^c_e$ 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^α_βs and $\mathbb{E}^\alpha_\beta$s such that for each loaded D^α_β, there must be an edge (u, u') in T with $l(u) = \alpha$ and $l(u') = \beta$, and for each loaded $\mathbb{E}^\alpha_\beta$, 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 2.1(a) over the closure in Figure 3.1(a); in initialization, we load in the blocks of D^a_b, D^a_c, D^c_d, and D^c_e, as well as load in the

blocks of $\mathbb{E}^c_d$, $\mathbb{E}^c_e$, and $\mathbb{E}^a_b$.

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^α_v where α is the parent node label of $q(v)$ in T (Since T is a tree, such α is unique to each node v).

As computing a nontrivial $lb(v)$ is hard, inspired by the A* algorithm [22] 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^α_β and E^α_β as described above;
2:  ∀ loaded edges (v, v'), insert (v', δ_min(v, v')) into L_{v,l(v')};
3:  ∀ active v, compute bs̄(v) & lb̄(v), and insert v into Q_g with key lb̄(v);
4:  while Q_g ≠ ∅ do
5:      (v, lb̄) ← Q_g.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, bs̄(v) + δ_min(v', v)) into L_{v',l(v)};
13:     Update bs̄(v') (and lb̄(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 lb̄_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 α 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 3.1(b), 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 $v \in G_R$,

$$\overline{bs}(v) = \sum_{\alpha \in \Pi_{q(v)}} \min\{\overline{bs}(v') + \delta_{\min}(v, v') \mid v' \in L_{v,\alpha} \cup H_{v,\alpha}\}, \tag{4.1}$$

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 $L(u)$ 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 $\overline{lb}(v') = \overline{bs}(v') + e_{v'} + L(q(v'))$ 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 may change in Q_g 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, $\overline{lb}(v) \leq \overline{lb}(v')$.* □

Proof: To prove this, we prove that once $(v, \overline{lb}(v))$ is expanded, the elements v' in the updated Q_g still have their new $\overline{lb}(v')$ values no smaller than $\overline{lb}(v)$. There are only three

cases below, considering that popping $(v, \overline{lb}(v))$ only changes $\overline{lb}$ values for v, and the parents of v.

Case 1 - Insert $(v, \overline{lb}_{new}(v))$ back to Q_g with updated $\overline{lb}_{new}(v)$. While $\overline{bs}(v)$ and $L(q(v))$ remain unchanged and the new e_v is always not smaller than the old e_v since all incoming edges to v are loaded in non-decreasingly according to their shortest distances. Thus, $\overline{lb}_{new}(v) \geq \overline{lb}(v)$.

Case 2 - The $\overline{lb}$ value of a parent v_1 of v is updated to $\overline{lb}_{new}(v_1)$. Note that v_1 must be an active node to have $\overline{lb}$ value. That is $\delta_{\min}(v_1, v)$ and $\overline{bs}(v)$ are used to calculate $\overline{bs}_{new}(v_1)$ in $\overline{lb}_{new}(v_1)$ while e_{v_1} and $L(q(v_1))$ remain unchanged in $\overline{lb}_{new}(v_1)$. Clearly, $\overline{bs}(v)+\delta_{\min}(v_1, v)+L(q(v)) \leq \overline{bs}_{new}(v_1)+e_{v_1}+L(q(v_1))$ due to the way to calculate $L(q(v))$ and $L(q(v_1))$ values. Given $e_v \leq \delta_{\min}(v_1, v)$ where e_v is a lower bound of $\delta_{\min}(v_1, v)$ used in $\overline{lb}(v)$, $\overline{lb}_{new}(v_1) \geq \overline{lb}(v)$.

Case 3 - A parent v_1 of v becomes active and $(v_1, \overline{lb}_{new}(v_1))$ is inserted to Q_g due to that the edge (v_1, v) is loaded in. Similar to Case 2, we can show that $\overline{lb}_{new}(v_1) \geq \overline{lb}(v)$.

Therefore, the theorem holds. □

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 α 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 α, is already loaded in and $bs(v')$ is already calculated. Consequently, $bs(v)$ is also calculated.* □

Proof: We prove the theorem by contradiction. Suppose that v_1 is the node at the lowest level in G_R that the theorem does not hold for v_1, and α is such a label. This implies that any child v_2 of v_1 follows the theorem and $bs(v_2)$ is already calculated when it is popped from Q_g for the first time. It is immediate from Equation 3.1 that none of the edges (v_1, v_2) with the minimum $(bs(v_2) + \delta_{\min}(v_1, v_2))$ among all children of v_1 with the label α is loaded in before $(v_1, \overline{lb}(v_1))$ pops.

It is clear that every node $v \in G_R$ will eventually be popped from Q_g and every incoming edge to v will be loaded in. Note that $bs(v_2)$ is already calculated when v_2 is popped from Q_g. Consider that $bs(v_2) + \delta_{\min}(v_1, v_2) + L(q(v_2)) < \overline{lb}(v_1)$ since when v_1 the first-time becomes the top, none of its outgoing edges (v_1, v_2) that make $(bs(v_2) + \delta_{\min}(v_1, v_2))$ the minimum among all children of v_1 with label α is loaded in. Consequently, when v_2 the first-time becomes the top of Q_g, $\overline{lb}(v_2) = bs(v_2) + e_{v_2} + L(q(v_2)) < \overline{lb}(v_1)$. According to the monotonic property, $(v_2, bs(v_2) + e_{v_2} + L(q(v_2))$ should be popped before $(v_1, \overline{lb}(v_1))$ being popped. Contradiction. □

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.

Example 4.2: Consider the tree query T in Figure 3.3(a) over the run-time graph G_R in Figure 3.3(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 3.3(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 $\overline{lb}$

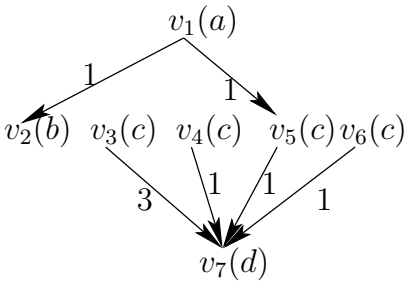


Figure 4.1: The part of G_R that is loaded into main memory

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 into main memory is shown in Figure 4.1. □

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.

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 ∞ as the lowest score of a current empty match in S. Theorems 4.2 and 4.1 ensure the correctness of Algorithm 3.

Algorithm 3 Enhanced-TopK

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

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

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 to use a minimum priority queue with Fibonacci heaps [8] 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.

Let m'_R be the number of retrieved edges from G_R, and n'_R be the total number of active nodes, d'_R be the maximum size of a $L_{v,\alpha} \cup H_{v,\alpha}$. Note that, $m'_R \leq m_R$, $n'_R \leq n_R$, $d'_R \leq d_R$.

To maintain Q_g, there are three operations, 1) delete-min to get the node v with smallest key value from Q_g, 2) insert a new node into Q_g, and 3) decrease-key to update the key value for node v in Q_g. The latter two (i.e., insert and decrease-key) take $O(m'_R)$ time in total, while delete-min takes $O((\frac{m'_R}{\text{BlockSize}} + n'_R) \log n'_R)$ time in total. Each delete-min takes $O(\log n'_R)$ time, and there are at most $O(\frac{m'_R}{\text{BlockSize}} + n'_R)$ delete-min operations, since the same node v will be inserted into Q_g for at most $\frac{d'_v}{\text{BlockSize}} + 1$ times where d'_v is the number of loaded incoming edges to v. Therefore, the total time complexity related to Q_g is $O(m'_R + n'_R \log n'_R)$, since usually $\log n'_R <$ BlockSize.

Secondly, similar to Algorithm 1, the total time complexity related to Q and Q_l (recall that Q_l is used for each round of dividing a subspace into n_T subspaces) is $O(k(\log d'_R + \log k + n_T))$; this also includes the cost $O(kn_T)$ for updating the obtained matches in related subspaces when new edges are loaded in.

Thus, the following theorem is immediate.

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.* □

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

We discuss various ways to extend our techniques.

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 α in T we make (possibly) multiple copies of a node with the label α in G at the levels of G_R corresponding to the levels of nodes with the label α 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 size of run-time graph 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 [1] 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; that is, the labels of a query vertex is contained by the labels of a vertex in the data graph, where each vertex can have multiple labels. 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.

[1] `http://www.ins.cwi.nl/projects/xmark/Assets/xmlquery.txt`

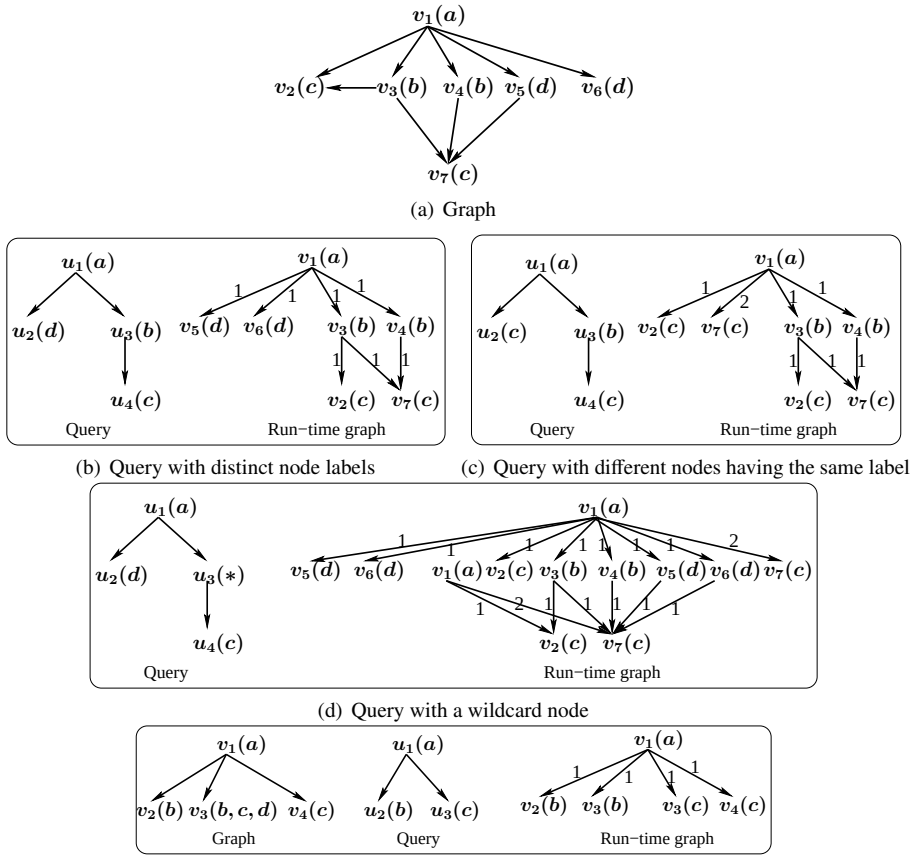


Figure 5.1: Example extensions

Example 5.1: Figure 5.1 shows queries and their corresponding run-time graphs for the extensions discussed above, where Figure 5.1(a) illustrates the data graph. Figure 5.1(b) shows the query with distinct node labels and its corresponding run-time graph, while Figure 5.1(c) shows the query with different nodes having the same label and its corresponding run-time graph; here in Figure 5.1(c), vertices v_2 and v_7 are duplicated at two levels for the two query vertices u_2 and u_4, both of label c. We can see that the queries in Figures 5.1(b) and 5.1(c) are of the same size, so do their corresponding run-time graphs; thus, by allowing different nodes in a query to have the same label does not necessarily increase the run-time graph size and also not the query processing time, on average. A query with a wildcard node and the corresponding run-time graph of the query are shown in Figure 5.1(d); the run-time graph size increases dramatically since each vertex in the data graph has a corresponding copy in the run-time graph for the wildcard (*) node. Figure 5.1(e) shows a query with the label containment semantics; here in Figure 5.1(e), the vertex v_3 in the data graph has three labels, b, c, and d, and it has two copies in the run-time graph to correspond to vertices u_2 and u_3 in the query that have labels b and c, respectively. After obtaining the above run-time graphs, the query processing algorithm remains the same, for different queries. □

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, 7, 25], which can very efficiently compute the shortest path between any pair of nodes. As proposed in [20], 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, 7, 25] to compute shortest paths.

Supporting Top-k Graph Pattern Matching. [6] 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 [6] 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 [6].

Enforcing One-to-One Mapping. In kTPM and kGPM, we can enforce matches to be one-to-one mappings from the node set of a query graph to the node set of a data graph. Such an enforcement makes the problems of kTPM and kGPM harder than their versions without such an enforcement when different nodes in a query may have the same label. In fact, kTPM is NP-hard in this case in contrast to PTIME as discussed in Section 3 for kTPM without the enforcement of one-to-one mapping.

Theorem 5.1: *kTPM by enforcing one-to-one mapping is NP-hard when* $k = 1$. □

Proof: It is well known that Hamiltonian path problem is NP-Complete [16] for directed graphs.

Then, each instance of Hamiltonian path problem can be converted to a special case of kTPM in which each directed graph in Hamiltonian path problem is used as a directed data graph G in kTPM with only one node label A, and the query graph (rooted tree) T is a directed path consisting of n_G nodes (n_G is the number of nodes in G) with only label A at each node. Then, G has a Hamiltonian Path if and only if T has a match with the penalty score ($|V_T| - 1$), where determining whether T has a match with the penalty score ($|V_T| - 1$) is equivalent to finding the top-1 match of T because all matches of T have penalty scores no less than ($|V_T| - 1$). Thus, the theorem holds. □

Although being NP-hard , we could also extend our techniques for computing kTPM with unique node labels in query T to solve kTPM with the enforcement of one-to-one mapping and different nodes in T having the same label in the same way as discussed above for supporting general top-k twig queries. With the enforcement of one-to-one mapping, some matches returned may be *invalid matches* (i.e., multiple nodes with the same label in T are mapped to the same node in G), we report only valid matches and stop once k valid matches are reported. We can speed up the computation by pruning invalid matches earlier. Specifically, for a subspace $S = \{v_1\} \times \cdots \times \{v_{i-1}\} \times (\mathbb{V}_i - U_i) \times \mathbb{V}_{i+1} \times \cdots \times \mathbb{V}_{n_T}$ (refer to Section 3 for definition of subspace), if there are already duplicate nodes in $\{v_1, \ldots, v_{i-1}\}$, then we can remove S from further computations, since all matches in S are invalid.

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 [20], 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 [6].
- mtree$^+$: as discussed in Section 5, extend our Topk-EN algorithm and embed it into the framework in [6].

Java bytecodes of DP-B and DP-P are obtained from the authors of [20]. 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 [6], and we also implement our mtree$^+$ 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[1]. 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 $10^4, 5\times10^4, 10^5, 2\times10^5$, 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 [9] 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 $10^4, 5\times10^4, 10^5, 2\times10^5$, 10^6, and 2×10^6 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 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

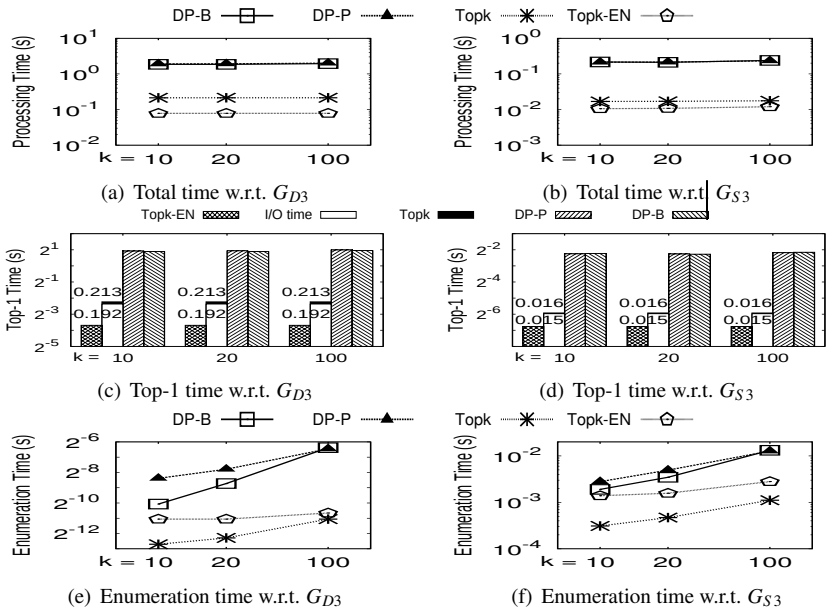
Eval-I: Pre-Computation Cost of Transitive Closure. The computation time and sizes of transitive closures (i.e., organized in the form of tables L^{α}_{β} as discussed in

[1] http://dblp.uni-trier.de/

Sections 3.1 and 4.1) are shown in Table 6.1, 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 6.1: Computational costs of transitive closures



(a) Total time w.r.t. G_{D3} (b) Total time w.r.t. G_{S3}

(c) Top-1 time w.r.t. G_{D3} (d) Top-1 time w.r.t. G_{S3}

(e) Enumeration time w.r.t. G_{D3} (f) Enumeration time w.r.t. G_{S3}

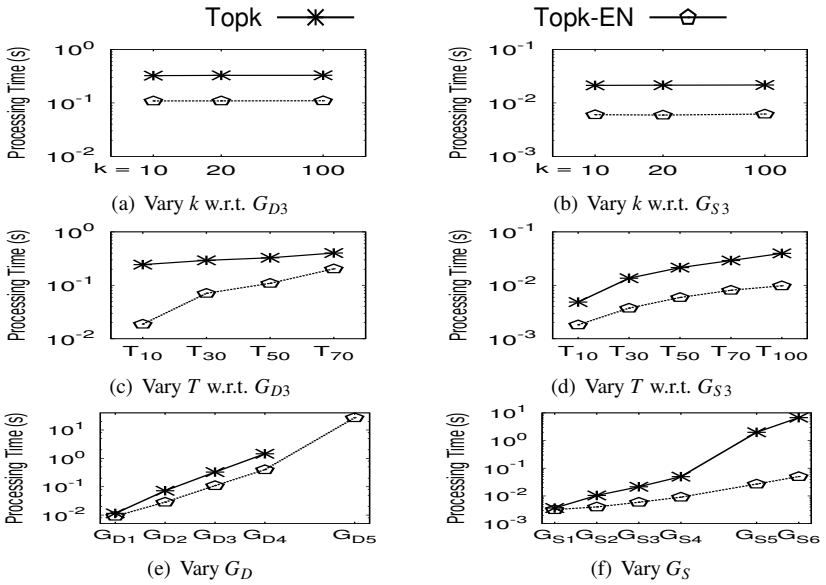
Figure 6.1: Comparing with DP-B and DP-P ($T = T_{20}$, vary k)

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 bytecodes obtained for DP-B and DP-P cannot run for large queries and graphs. Figure 6.1(a) and Figure 6.1(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 6.1(c) and 6.1(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 6.1(e) and Figure 6.1(f) show the total time for

generating the top-l 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.



(a) Vary k w.r.t. G_{D3} (b) Vary k w.r.t. G_{S3}

(c) Vary T w.r.t. G_{D3} (d) Vary T w.r.t. G_{S3}

(e) Vary G_D (f) Vary G_S

Figure 6.2: 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 6.2(a) and Figure 6.2(b) demonstrate the impacts of k with T_{50} that are similar to those in Figure 6.1(a) and Figure 6.1(b).

Figure 6.2(c) and Figure 6.2(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 6.2 shows the sizes of run-time graphs for real graph and synthetic graph, respectively.

Figure 6.2(e) and Figure 6.2(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: Space Cost of Auxiliary Structures. The space costs of auxiliary structures (i.e., size of $L_{v,\alpha}$ and $H_{v,\alpha}$, the size of subspaces' information, and the size of priority

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

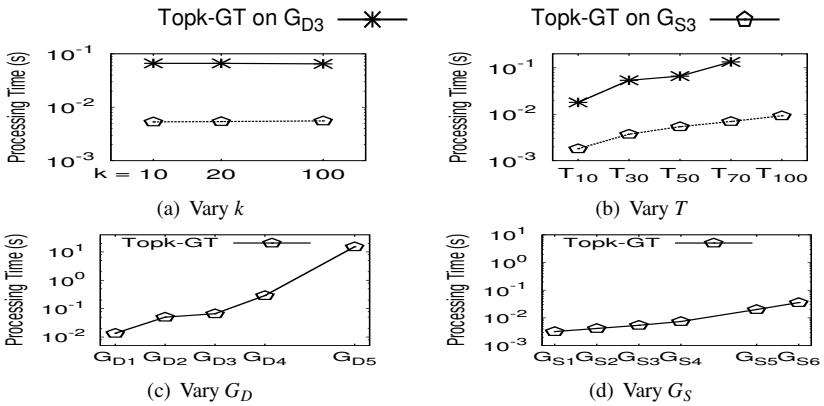
Table 6.2: Average sizes of run-time graphs (i.e., G_R)

queues) used by our Topk algorithm are shown in Table 6.3.

	Space Cost for G_{D3} (Bytes)			Space Cost for G_{S3} (Bytes)		
	$L_{v,\alpha}$ + $H_{v,\alpha}$	Subspace	Q	$L_{v,\alpha}$ + $H_{v,\alpha}$	Subspace	Q
T_{10}	15.61×10^6	1,614	81	0.31×10^6	1,605	80
T_{30}	21.56×10^6	1,639	83	1.02×10^6	1,640	84
T_{50}	28.18×10^6	1,639	83	1.70×10^6	1,640	84
T_{70}	36.41×10^6	1,607	80	2.39×10^6	1,640	84
T_{100}				3.35×10^6	1,638	83

Table 6.3: Space cost of auxiliary structures (Bytes)

Eval-V: 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 6.3 reports the evaluation result where the default settings remain unchanged. Similar trends to Topk-EN are obtained.

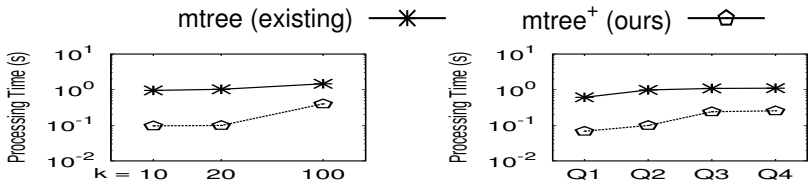


(a) Vary k (b) Vary T

(c) Vary G_D (d) Vary G_S

Figure 6.3: Running time for general twig-pattern matching

Eval-VI: kGPM. Figure 6.4(a) and Figure 6.4(b) show the evaluation results by comparing mtree$^+$ with mtree against different settings. As expected, mtree$^+$ significantly outperforms mtree.



(a) Vary k (Query is Q_2) (b) Vary query (k = 20)

Figure 6.4: 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. Our new enumeration algorithm computes the top-1 match in $O(m_R)$ time; then enumerates each of the remaining top-k matches in $O(\log k + n_T)$ time. This is an optimal algorithm since the lower bound to compute the top-1 match shall be $O(m_R)$ in the worst case without any pre-knowledge of G, the lower bound to enumerate a new match is $O(n_T)$ in the worst case, and n_T is practically a dominant factor in $O(n_T + \log k)$. 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.

Bibliography

[1] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proc. of SIGMOD'13*, 2013.

[2] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *Proc. of ICDE'02*, 2002.

[3] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal xml pattern matching. In *Proc. of SIGMOD'02*, 2002.

[4] Li Chen, Amarnath Gupta, and M. Erdem Kurul. Stack-based algorithms for pattern matching on dags. In *Proc. of VLDB'05*, 2005.

[5] Jiefeng Cheng, Jeffrey Xu Yu, Bolin Ding, Philip S. Yu, and Haixun Wang. Fast graph pattern matching. In *Proc. of ICDE'08*, 2008.

[6] Jiefeng Cheng, Xianggang Zeng, and Jeffrey Xu Yu. Top-k graph pattern matching over large graphs. In *Proc. of ICDE'13*, 2013.

[7] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. In *Proc. of SODA'02*, 2002.

[8] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[9] B. Dawes and D. Abrahams. Boost c++ libraries. http://www.boost.org/.

[10] Bolin Ding, Jeffrey Xu Yu, Shan Wang, Lu Qin, Xiao Zhang, and Xuemin Lin. Finding top-k min-cost connected trees in databases. In *Proc. of ICDE'07*, 2007.

[11] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proc. of PODS'01*, 2001.

[12] Wenfei Fan, Jianzhong Li, Shuai Ma, Hongzhi Wang, and Yinghui Wu. Graph homomorphism revisited for graph matching. *PVLDB*, 3(1), 2010.

[13] Wenfei Fan, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. Graph pattern matching: From intractable to polynomial time. *PVLDB*, 2010.

[14] Wenfei Fan, Xin Wang, and Yinghui Wu. Diversified top-k graph pattern matching. *PVLDB*, 6(13), 2013.

[15] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[16] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[17] Alan Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, 1985.

[18] Georg Gottlob, Christoph Koch, and Reinhard Pichler. The complexity of xpath query evaluation. In *Proc. of PODS'03*, 2003.

[19] Gang Gou and Rada Chirkova. Efficiently querying large xml data repositories: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(10):1381–1403, 2007.

[20] Gang Gou and Rada Chirkova. Efficient algorithms for exact ranked twig-pattern matching over graphs. In *Proc. of SIGMOD'08*, 2008.

[21] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. $\text{Turbo}_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proc. of SIGMOD'13*, 2013.

[22] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems Science and Cybernetics*, 4(2), 1968.

[23] Pavol Hell and Jaroslav Nesetril. *Graphs and Homomorphisms*. Oxford Lecture Series in Mathematics and Its Applications, 2004.

[24] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-*k* query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[25] Minhao Jiang, Ada Wai-Chee Fu, Raymond Chi-Wing Wong, and Yanyan Xu. Hop doubling label indexing for point-to-point distance querying on scale-free networks. *PVLDB*, 7(12), 2014.

[26] Flip Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *SIGMOD Conference*, pages 201–212, 2000.

[27] Theodoros Lappas, Kun Liu, and Evimaria Terzi. Finding a team of experts in social networks. In *KDD*, 2009.

[28] Eugene 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] Yan Qi, K. Selçuk Candan, and Maria Luisa Sapino. Sum-max monotonic ranked joins for evaluating top-k twig queries on weighted data graphs. In *Proc. of VLDB'07*, 2007.

[30] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 1(1), 2008.

[31] Julian 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] Jeffrey Xu Yu, Lu Qin, and Lijun Chang. *Keyword Search in Databases*. Morgan & Claypool, 2009.

[34] Qiang Zeng, Xiaorui Jiang, and Hai Zhuge. Adding logical operators to tree pattern queries on graph-structured data. *PVLDB*, 5(8), 2012.

[35] Gaoping Zhu, Xuemin Lin, Ke Zhu, Wenjie Zhang, and Jeffrey Xu Yu. Treespan: efficiently computing similarity all-matching. In *Proc. of SIGMOD'12*, 2012.

[36] Lei Zou, Lei Chen, and M. Tamer Özsu. Distancejoin: Pattern match query in a large graph database. *PVLDB*, 2(1), 2009.