Finding Patterns in a Knowledge Base using Keywords to Compose Table Answers

Mohan Yang[1] * Bolin Ding[2] Surajit Chaudhuri[2] Kaushik Chakrabarti[2]
[1]University of California, Los Angeles, CA, USA
[2]Microsoft Research, Redmond, WA, USA
yang@cs.ucla.edu, {bolind, surajitc, kaushik}@microsoft.com

ABSTRACT

We aim to provide table answers to keyword queries using a knowledge base. For queries referring to multiple entities, like "Washington cities population" and "Mel Gibson movies", it is better to represent each relevant answer as a table which aggregates a set of entities or joins of entities within the same table scheme or *pattern*. In this paper, we study how to find highly relevant patterns in a knowledge base for user-given keyword queries to compose table answers. A knowledge base is modeled as a directed graph called knowledge graph, where nodes represent its entities and edges represent the relationships among them. Each node/edge is labeled with type and text. A pattern is an aggregation of subtrees which contain all keywords in the texts and have the same structure and types on node/edges. We propose efficient algorithms to find patterns that are relevant to the query for a class of scoring functions. We show the hardness of the problem in theory, and propose path-based indexes that are affordable in memory. Two query-processing algorithms are proposed: one is fast in practice for small queries (with small numbers of patterns as answers) by utilizing the indexes; and the other one is better in theory, with running time linear in the sizes of indexes and answers, which can handle large queries better. We also conduct extensive experimental study to compare our approaches with a naive adaption of known techniques.

1. INTRODUCTION

Users often look for information about sets of entities, *e.g.*, in the form of tables [27, 41, 35]. For example, an analyst wants a list of companies that produces database software along with their annual revenues for the purpose of market research. Or a student wants a list of universities in a particular county along with their enrollment numbers, tuition fees and financial endowment in order to choose which universities to seek admission in.

To provide such services, some works leverage the vast corpus of HTML tables available on the Web, trying to interpret them, and return relevant ones in response to keyword queries [27, 41, 35, 44]. There are also two such commercial table search engines: Google Tables [4] and Microsoft's Excel PowerQuery [3]. Our work is complementary to this line, and aims to compose tables in response to keyword queries from patterns in *knowledge bases* when the desired tables are not available or of low quality in the corpus.

There are abundant sources of high-quality structured data, called knowledge bases: DBPedia [2], Freebase [6], and Yago [9] are examples of knowledge bases containing information on general topics, while there are also specialized ones like IMDB [7] and DBLP [8]. A knowledge base contains information about individual *entities* together with *attributes* representing relationships among them. We can model a knowledge base as a directed graph, called *knowledge graph*, with *nodes* representing *entities* of different *types* and *edges* representing relationships, i.e., *attributes*, among entities.

We can find the subtrees of the knowledge graph that contain all the keywords and return them in ranked order (refer to Yu *et al.* [46] and Liu *et al.* [32] for comprehensive surveys, and Section 6 for detailed discussion). However, it is not adequate when the user's query is to look for a table of entities. As has been noticed in [42], the returned subtrees with a heterogeneous mass of shapes might correspond to different interpretations of the query, and the subtrees corresponding to certain desired interpretation may not appear contiguously in the ranked order. If the user wants to explore all subtrees of the desired interpretation, she has to examine all the returned subtrees and manually gather those corresponding to the interpretation. This is extremely labor intensive. So we propose to automatically aggregate the subtrees that contain all the keywords into distinct interpretations and produce a ranked list of such aggregations. Structural pattern of a subtree together with the mapping from the keywords to its nodes/edges represents an interpretation of the query, called *tree pattern*. We aggregate the subtrees based on tree patterns. Our work sharply contrasts earlier works on ranking subtrees. To the best of our knowledge, this is the first work on finding aggregations of subtrees on graphs for keyword queries.

In this paper, we propose and study the problem of finding relevant aggregations of subtrees in the knowledge graph for a given keyword query. Each answer to the keyword query is a set of subtrees – each subtree containing all keywords and satisfying the same tree pattern. Such an aggregation of subtrees can be output as a table of entity joins, where each row corresponds to a subtree. When there are multiple possible tree patterns, they are enumerated and ranked by their relevance to the query.

EXAMPLE 1.1. (Motivation Example) *Figure 1(a)-(c) is a small piece of a knowledge base with three entities. For each entity (*e.g.*, "SQL Server", "Microsoft", and "Bill Gates"), we know its type (*e.g.*, Software, Company, and Person, respectively), and a list of attributes (left column in Figure 1(a)-(c)) together with their values (right column). The value of an attribute may either refer to another entity,* e.g.*, "Developer" of "SQL Server" is "Microsoft",*

*Work done while visiting Microsoft Research

Proceedings of the VLDB Endowment, Vol. 7, No. 14

SQL Server	(**Software**)
Developer:	Microsoft
Genre:	Relational Database
Written in:	C++
…:	…

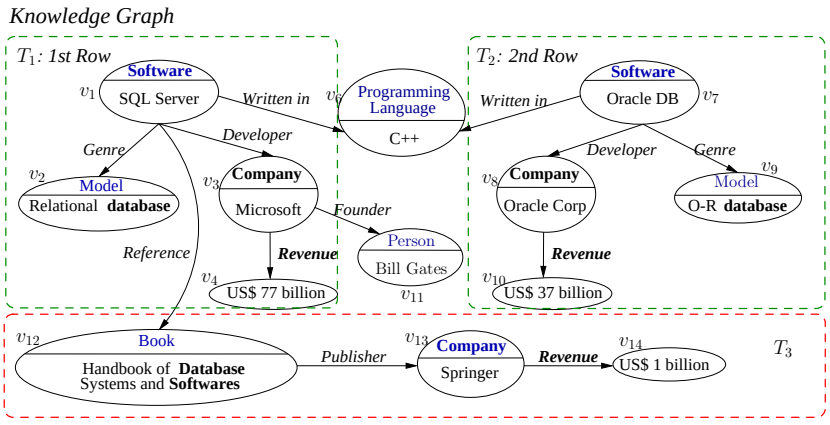
(a) Entity "SQL Server"

Microsoft	(**Company**)
Founder:	Bill Gates, Paul Allen
Products:	Windows, Bing, …
Revenue:	US$ 77 billion
…:	…

(b) Entity "Microsoft"

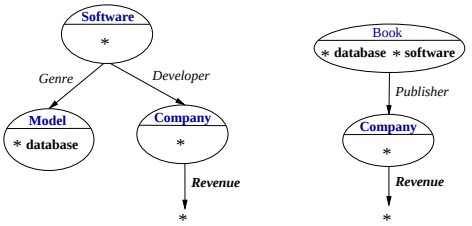
Bill Gates	(**Person**)
Alma mater:	Harvard University
Residence:	Medina, WA, US
Spouse:	Melinda Gates
…:	…

(c) Entity "Bill Gates"



(d) Part of a knowledge graph derived from the knowledge base in (a)-(c), and subtrees (T_1-T_3) matching to query "database software company revenue"

Figure 1: (a)-(c) Entities/Attributes in Knowledge Base, (d) Knowledge Graph, Query, and Subtrees



(a) Tree pattern P_1 (b) Tree pattern P_2

Figure 2: Tree patterns for (a) $\{T_1, T_2\}$ and (b) $\{T_3\}$

or be plain text, e.g.*, "Revenue" of "Microsoft" is "US$ 77 billion". Such a knowledge base can be extracted from the Web like infoboxes in Wikipedia [5], or from datasets like Freebase [6].*

Knowledge graph. *A knowledge base can be modeled as a direct graph and Figure 1(d) shows part of such a knowledge graph. Each entity corresponds to a node labeled with its type. Each attribute of an entity corresponds to a directed edge, also labeled with its attribute type, from the entity to some other entity or plain text.*

Queries, Subtrees, and Tree Patterns. *Consider a keyword query "database software company revenue". Three subtrees (T_1, T_2, and T_3) matching the keywords are shown using dashed rectangles in Figure 1(d). In subtrees T_1 and T_2, "database" is contained in the names of the some entities; "software" and "company" match to the types' names; and "revenue" matches to an attribute. Also, the structures of T_1 and T_2 are identical in terms of the types of both nodes and edges, so they belong to the same pattern in Figure 2(a). Similarly, T_3 belongs to the tree pattern in Figure 2(b).*

Tree patterns as answers. *A tree pattern corresponds to a possible interpretation of a keyword query, by specifying the structure of subtrees as well as how the keywords are mapped to the nodes or edges. For example, the tree pattern P_1 in Figure 2(a) interprets the query as: the* revenue *of some* company *which develops* database software*; and P_2 in Figure 2(b) means: the* revenue *of some* company *which publishes books about* database software*. Subtrees of the same tree pattern can be aggregated into a table as one answer to the query, where each row corresponds to a subtree. For example, subtrees (T_1 and T_2) of the pattern in Figure 2(a) can be assembled into the table (the first and second rows) in Figure 3.*

Contributions. First, we propose the problem of *finding relevant tree patterns in a knowledge graph*. We define *tree patterns* as answers to a keyword query in a knowledge graph. A class of scoring functions is introduced to measure the relevance of a pattern.

There are usually many tree patterns for a keyword query. We need efficient algorithms to enumerate these patterns and find the top-k. We then analyze the hardness of the problem in theory. The hardness comes from "counting the number of paths between two nodes in the graph", which inspires us to design two types of path-pattern based inverted indexes: paths starting from a node/edge containing some keyword and following certain pattern are aggregated and materialized in the index in memory. When processing an online query, by specifying the word and/or the path pattern, a search algorithm can retrieve the corresponding set of paths.

Two algorithms for finding the relevant tree patterns for a keyword query are proposed based on such indexes.

The first one enumerates the combinations of root-leaf path patterns in tree patterns, retrieves paths from the index for each path pattern, and joins them together on the root node to get the set of subtrees satisfying each tree pattern. Its worst-case running time is exponential in both the index size and the output size: when there are m keywords and each has p path patterns in the index, we need to check all the p^m combinations in the worst case; but it is possible that there is no subtree satisfying any of these tree patterns. Although join operations are wasted on such "empty patterns", the advantage of this algorithm is that no online aggregation is required, as all subtrees with the same tree pattern are generated at one time. So it performs well in practice most of the time.

The second algorithm tries to avoid unnecessary join operations by first identifying all *candidate roots* with the help of path indexes. Each candidate root reaches every keyword through at least one path pattern, so there must be some tree pattern containing a subtree with this root. Those subtrees are enumerated and aggregated for each candidate root. The running time of this algorithm can be shown to be linear in the index size and the output size. To further speed it up, we can sample a random subset of candidate roots (*e.g.*, 10% of them), and obtain an estimated score for each pattern based on them. Only for the patterns with the highest top-k estimated scores, we retrieve the complete set of subtrees, and compute the exact scores for ranking. Note that when we apply such sampling techniques, there might be errors in the top-k tree patterns. But we will show that the error can be bounded in theory, and demonstrate the effectiveness of this sampling technique in experiments.

We compare our algorithms with a straightforward adaption of previous techniques on finding subtrees in database graphs (*e.g.*, [11, 13, 18, 25]) in experiments. We adapt their algorithms to enumerate all subtrees each containing all keywords as the first step.

Software	Genre	Company	Revenue
SQL Server	Relational **database**	Microsoft	US$ 77 billion
Oracle DB	O-R **database**	Oracle	US$ 37 billion
...	...	...	...

Figure 3: Example of a table aggregating subtrees of the tree pattern in Figure 2(a)

The second step is to aggregate those subtrees into a ranked list of tree patterns. Note that no ranking is required for the first step so the adapted enumeration algorithm is efficient, but the bottleneck lies on the second step. Efforts along this line are not helpful in solving our problem because they aim to find highly relevant subtrees while we aim to find highly relevant tree patterns.

Organization. Section 2 formally defines the concept of tree patterns as answers to keyword queries, and gives the problem statement. A baseline approach and hardness result are given at the end of Section 2. In Section 3, we introduce the index structures inspired by the hardness result. Two search algorithms based on the proposed path indexes are introduced in Section 4. Experimental results and discussions are in Section 5, followed by the discussion of related work in Section 6, and conclusion in Section 7.

2. MODEL AND PROBLEM

We first formally define the graph model of a knowledge base used in this work, called *knowledge graph*. The model itself is not new but it servers as a general platform where our techniques introduced later can be applied. We then define *tree patterns*, each of which is an answer to a keyword query and aggregates a set of *valid subtrees* in the knowledge graph. We also introduce the class of scoring functions we use to measure the relevance of a *tree pattern* to a query. Finally, we formally define the problem of *finding top-k tree patterns in a knowledge base using keywords*.

2.1 Knowledge Graph

A *knowledge base* consists of a collection of *entities* $\mathcal{V}$ and a collection of *attributes* $\mathcal{A}$. Each *entity* $v \in \mathcal{V}$ has values on a subset of *attributes*, denoted by $\mathcal{A}(v)$, and for each attribute $A \in \mathcal{A}(v)$, we use $v.A$ to denote its value. The value $v.A$ could be either another entity or free text. Each entity $v \in \mathcal{V}$ is labeled with a *type* $\tau(v) \in \mathcal{C}$, where $\mathcal{C}$ is the set of all types in the knowledge base.

It is natural to model the knowledge base as a *knowledge graph* $\mathcal{G}$, with each entity in $\mathcal{V}$ as a node, and each pair (v, u) as a directed edge in $\mathcal{E}$ iff $v.A = u$ for some attribute $A \in \mathcal{A}(v)$. Each node v is labeled by its entity type $\tau(v) = C \in \mathcal{C}$ and each edge $e = (v, u)$ is labeled by the attribute type A iff $v.A = u$, denoted by $\alpha(e) = A \in \mathcal{A}$. So we denote a *knowledge graph* by $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \tau, \alpha)$ with τ and α as *node type*s and *edge type*s, respectively. There is *text description* for each entity/node type C, entity/node v, and attribute/edge type A, denoted by $C.\mathsf{text}$, $v.\mathsf{text}$, and $A.\mathsf{text}$, respectively. In the rest of this paper, w.l.o.g., we assume that the value of an entity v's attribute is always an entity in $\mathcal{V}$, because if $v.A$ is plain text, we can create a *dummy entity* with text description exactly the same as the plain text.

EXAMPLE 2.1. (Knowledge Graph) *Figure 1(d) shows part of the knowledge graph derived from the knowledge base in Figure 1(a)-(c). Each node is labeled with its type $\tau(v)$ in the upper part, and its text description is shown in the lower part. For nodes derived from plain text, their types are omitted in the graph. Each edge e is labeled with the attribute type $\alpha(e)$. Note that there could be more than one entity referred in the value of an attribute,* e.g.*, attribute "Products" of entity "Microsoft". In that case, we can create multiple edges with the same label (attribute type) "Products" pointing to different entities,* e.g.*, "Windows" and "Bing".*

2.2 Finding d-Height Tree Patterns

Now we are ready to define *tree patterns*, i.e., answers for a given keyword query $\mathsf{q} = \{\mathsf{w}_1, \mathsf{w}_2, \ldots, \mathsf{w}_m\}$ in a knowledge graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \tau, \alpha)$. Simply put, a *valid subtree* w.r.t. the query q is a subtree in $\mathcal{G}$ containing all keywords in the text description of its node, node type, or edge type. A *tree pattern* aggregates a set of valid subtrees with the same i) tree structure, ii) entity types and edge types, and iii) positions where keywords are matched.

2.2.1 Valid Subtrees for Keyword Queries

We first formally define a *valid subtree* (T, f) w.r.t. a keyword query q in a knowledge graph $\mathcal{G}$. It satisfies three conditions:

i) (Tree Structure) T is a directed rooted subtree of $\mathcal{G}$, i.e., it has a root r and there is exactly one path from r to each leaf.

ii) (Keyword Mapping) There is a mapping $f : \mathsf{q} \to \mathcal{V}(T) \cup \mathcal{E}(T)$ from words in q to nodes and edges in the subtree T, s.t., each word $\mathsf{w} \in \mathsf{q}$ appears in the text description of a node or node type if $f(\mathsf{w}) \in \mathcal{V}(T)$, and appears in the text description of an edge type if $f(\mathsf{w}) \in \mathcal{E}(T)$.

iii) (Minimality) For any leaf $v \in \mathcal{V}$ with edge $e_v \in \mathcal{E}$ pointing to v, there exists $\mathsf{w} \in \mathsf{q}$ s.t. $f(\mathsf{w}) = v$ or $f(\mathsf{w}) = e_v$.

Condition ii) ensures that all words appear in subtree T and specifies where they appear. Condition iii) ensures that T is *minimal* in the sense that, under the current mapping f (from words to nodes or edges wherever they appear), removing any leaf node from T will make it invalid. We will also refer to a valid subtree (T, f) as T if the mapping f is clear from the context.

EXAMPLE 2.2. (Valid Subtree) *Consider a keyword query q: "database software company revenue" (w_1-w_4). T_1 in Figure 1(d) is a valid subtree w.r.t. q. The associated mapping f from keywords to nodes in T_1 is: $f(\mathsf{w}_1) = v_2$ (appearing in the text description of node), $f(\mathsf{w}_2) = v_1$ (appearing in the node type), $f(\mathsf{w}_3) = v_3$ (appearing in the node type), and $f(\mathsf{w}_4) = (v_3, v_4)$ (appearing in the attribute type). T_1 is minimal and attaching any edge like (v_1, v_6) or (v_3, v_{11}) to T_1 will make it invalid (violating condition iii)). Similarly, T_2 and T_3 are also valid subtrees w.r.t. q.*

2.2.2 Tree Patterns: Aggregations of Subtrees

Consider a valid subtree (T, f) w.r.t. a keyword query q with the mapping $f : \mathsf{q} \to \mathcal{V}(T) \cup \mathcal{E}(T)$. Before defining the *tree pattern* of (T, f) for q, we first define *path patterns*.

Path patterns. For each word $\mathsf{w} \in \mathsf{q}$, if w is matched to some node $v = f(\mathsf{w})$, let $T(\mathsf{w})$ be the path from the root r to the node v: $v_1 e_1 v_2 e_2 \ldots e_{l-1} v_l$, where $v_1 = r$, $v_l = v$, and e_i is the edge from v_i to v_{i+1}. The *path pattern* for w is the concatenation of node/edge types on the path $T(\mathsf{w})$, i.e.,

$$\mathsf{pattern}(T(\mathsf{w})) = \tau(v_1)\alpha(e_1)\tau(v_2)\alpha(e_2)\ldots\alpha(e_{l-1})\tau(v_l),$$

from node v_1 to node v_l. Similarly, if w is matched to some edge $e = f(\mathsf{w})$, then the path pattern

$$\mathsf{pattern}(T(\mathsf{w})) = \tau(v_1)\alpha(e_1)\tau(v_2)\alpha(e_2)\ldots\alpha(e_l)$$

is the concatenation of node/edge types on the path $T(\mathsf{w})$ from node $v_1 = r$ to edge $e_l = e$. The *length* of a path pattern, denoted by $|\mathsf{pattern}(T(\mathsf{w}))|$, is the number of nodes on path $T(\mathsf{w})$.

Tree patterns. The *tree pattern* of a valid subtree T w.r.t. $\mathsf{q} = \{\mathsf{w}_1, \mathsf{w}_2, \ldots, \mathsf{w}_m\}$ is a vector with the i^{th} entry as the path pattern of the root-leaf path containing the i^{th} keyword w_i, denoted as

$$\mathsf{pattern}(T) = (\mathsf{pattern}(T(\mathsf{w}_1)), \ldots, \mathsf{pattern}(T(\mathsf{w}_m))). \quad (1)$$

The *height* of a tree pattern, denoted by $\mathcal{H}(\mathsf{pattern}(T))$, is the max length of the path patterns, i.e., $\max_i |\mathsf{pattern}(T(\mathsf{w}_i))|$.

Valid subtrees can be considered as ordered trees. To check whether patterns of two valid subtrees T_1 and T_2 w.r.t. query q are identical, we only need to check whether the path patterns are identical, $\mathsf{pattern}(T_1(\mathsf{w}_i)) = \mathsf{pattern}(T_2(\mathsf{w}_i))$, for each word $\mathsf{w}_i \in \mathsf{q}$. This can be done in linear time, because even without precomputation, each path pattern can be obtained by retrieving the types of node/edge on the path in order from the root v_1 to a leaf v_l or e_l.

Conceptually, valid subtrees can be grouped by their patterns. For a tree pattern P, let $\mathsf{trees}(P, \mathsf{q})$ be the set of all valid subtrees with the same pattern P w.r.t. a keyword query q, i.e., $\mathsf{trees}(P, \mathsf{q}) = \{T \mid \mathsf{pattern}(T) = P\}$. $\mathsf{trees}(P, \mathsf{q})$ is also written as $\mathsf{trees}(P)$ if the query q is clear from the context.

EXAMPLE 2.3. (Tree Patterns as Answers) *Let's continue with Example 2.2. Tree pattern $P_1 = \mathsf{pattern}(T_1)$ w.r.t. query q is visualized in Figure 2(a). In particular, for w_4 = "Revenue" $\in \mathsf{q}$, we have $T_1(\mathsf{w}_4) = v_1(v_1, v_3)v_3(v_3, v_4)$, and* $\mathsf{pattern}(T_1(\mathsf{w}_4)) =$ (Software) (Developer) (Company) (Revenue). *Similarly, for word w_1, we have* $\mathsf{pattern}(T_1(\mathsf{w}_1)) =$ (Software) (Genre) (Model), *for w_2,* $\mathsf{pattern}(T_1(\mathsf{w}_2)) =$ (Software), *and* $\mathsf{pattern}(T_1(\mathsf{w}_3)) =$ (Software) (Developer) (Company). *Combining them together, we get the tree pattern P_1 in Figure 2(a). It is easy to see that, in Figure 1(d), T_1 and T_2 have the identical tree pattern P_1, and the tree pattern of T_3 is P_2, which is illustrated in Figure 2(b).*

Convert tree patterns into table answers. Once we have the tree pattern P, it is not hard to convert trees in $\mathsf{trees}(P)$ into a *table answer*. For each tree $T \in \mathsf{trees}(P)$, create a row in the following way: for each word $\mathsf{w} \in \mathsf{q}$ and path $T(\mathsf{w}) = v_1e_1v_2e_2 \dots e_{l-1}v_l$, create l columns with values v_1, v_2, ..., v_l and column names $\tau(v_1)$, $\tau(v_1)\alpha(e_1)\tau(v_2)$, ..., and $\tau(v_{l-1})\alpha(e_{l-1})\tau(v_l)$, respectively. From the definition of tree patterns, we know all the rows created in this way have the same set of columns and this can be put and shown in a uniform table scheme. If an edge $e_i = (v_i, v_{i+1})$ appears in more than one root-leaf path (for different words w's), only one column needs to be created with name $\tau(v_i)\alpha(e_i)\tau(v_{i+1})$ and value v_{i+1}. Figure 3 shows the table answer derived from tree pattern P_1 in Figure 2(a). How to name and order columns in the table answers in a more user-friendly way is also an important issue, but it is out of scope of this paper and requires more user study. The rest of this paper will focus on how to find and rank tree patterns as it is the most challenging part of our problem.

2.2.3 Relevance Scores of Tree Patterns

There could be numerous tree patterns w.r.t. a given keyword query q, so we need to define scoring functions to measure their relevance. We will define a general class of scoring functions, *the higher the more relevant*, which can be handled by our algorithms introduced later. First, the relevance score of a tree pattern is an aggregation of relevance scores of valid subtrees that satisfy this pattern, *e.g.*, sum, average, and max of scores, or count of trees. Sum of scores and count of trees prefer tree patterns with more valid subtrees, while average and max prefer tree patterns with highly relevant individual subtrees. There is no global rule on which one is better, and the choice should be made based on extensive user study/feedback, which is out of the scope of this paper. We use sum of scores in the following part, but our approaches can be also extended to other aggregation functions.

$$\mathsf{score}(P, \mathsf{q}) = \sum_{T \in \mathsf{trees}(P)} \mathsf{score}(T, \mathsf{q}). \quad (2)$$

The relevance score $\mathsf{score}(T, \mathsf{q})$ of an individual valid subtree w.r.t. q may depend on several factors: 1) $\mathsf{score}_1(T, \mathsf{q})$: size of T, we prefer small trees that represent compact relationship; 2) $\mathsf{score}_2(T, \mathsf{q})$: importance score of nodes in T, we prefer more important nodes (*e.g.*, with higher PageRank scores) to be included in T; and 3) $\mathsf{score}_3(T, \mathsf{q})$: how well the keywords match the text description in T. Putting them together, we have

$$\mathsf{score}(T, \mathsf{q}) = \mathsf{score}_1(T, \mathsf{q})^{z_1} \cdot \mathsf{score}_2(T, \mathsf{q})^{z_2} \cdot \mathsf{score}_3(T, \mathsf{q})^{z_3}, \quad (3)$$

where z_1, z_2, and z_3 are constants that determine the weights of factors. These constants need to be tuned in practical system through user study. For the completeness, we give examples for scoring functions score_1, score_2, and score_3 below. But note that they can also be replaced by other functions and more can be inserted into (3) if needed – our search algorithms introduced later still work.

To measure the size of T, let $z_1 = -1$ and

$$\mathsf{score}_1(T, \mathsf{q}) = \sum_{\mathsf{w} \in \mathsf{q}} \mathsf{score}_1(T(\mathsf{w}), \mathsf{w}) = \sum_{\mathsf{w} \in \mathsf{q}} |T(\mathsf{w})|, \quad (4)$$

where $|T(\mathsf{w})|$ is the number of nodes on the path $T(\mathsf{w})$.

To measure how significant nodes of T are, let $z_2 = 1$ and

$$\mathsf{score}_2(T, \mathsf{q}) = \sum_{\mathsf{w} \in \mathsf{q}} \mathsf{score}_2(T(\mathsf{w}), \mathsf{w}) = \sum_{\mathsf{w} \in \mathsf{q}} \mathsf{PR}(f(\mathsf{w})), \quad (5)$$

where $\mathsf{PR}(f(\mathsf{w}))$ is the PageRank score of the node that contains word $\mathsf{w} \in \mathsf{q}$ (or, of the node that has an out-going edge contain word w, if $f(\mathsf{w})$ is an edge). The PageRank score $\mathsf{PR}(v)$ of a node v is computed using the iterative method: the initial value of $\mathsf{PR}(v)$ is set to $1/|\mathcal{V}|$ for all $v \in \mathcal{V}$; and in each iteration, $\mathsf{PR}(v)$ is updated

$$\mathsf{PR}(v) \leftarrow \frac{1-a}{|\mathcal{V}|} + a \sum_{(u,v) \in \mathcal{E}} \frac{\mathsf{PR}(u)}{\mathrm{OutDegree}(u)},$$

where $a = 0.85$ is the damping factor. The computation ends when $\mathsf{PR}(v)$ changes less than 10^{-8} during an iteration for all $v \in \mathcal{V}$.

To measure how well the keywords match the text description in T, let $z_3 = 1$ and

$$\mathsf{score}_3(T, \mathsf{q}) = \sum_{\mathsf{w} \in \mathsf{q}} \mathsf{score}_3(T(\mathsf{w}), \mathsf{w}) = \sum_{\mathsf{w} \in \mathsf{q}} \mathsf{sim}(\mathsf{w}, f(\mathsf{w})), \quad (6)$$

where $\mathsf{sim}(\mathsf{w}, f(\mathsf{w}))$ is the Jaccard similarity between w and the text description on the entity (type) or the attribute type of $f(\mathsf{w})$.

EXAMPLE 2.4. (Relevance Score) *Comparing the two tree patterns P_1 and P_2 in Figure 2 w.r.t. the query q in Example 2.2, which one is more relevant to q? First, consider valid subtrees $T_1, T_2 \in \mathsf{trees}(P_1)$ and $T_3 \in \mathsf{trees}(P_2)$ in Figure 1(d), T_3 is smaller than T_1 and T_2 – to measure the sizes,* $\mathsf{score}_1(T_1, \mathsf{q}) = \mathsf{score}_1(T_2, \mathsf{q}) = 2 + 1 + 2 + 3 = 8$, *and* $\mathsf{score}_1(T_3, \mathsf{q}) = 1 + 1 + 2 + 3 = 7$. *Second, assuming every node has the same PageRank score* 1, *we have* $\mathsf{score}_2(T_1, \mathsf{q}) = \mathsf{score}_2(T_2, \mathsf{q}) = \mathsf{score}_2(T_3, \mathsf{q}) = 4$. *Third, considering the similarity between keywords and text description in valid subtrees T_1, T_2, and T_3, we have* $\mathsf{score}_3(T_1, \mathsf{q}) = \mathsf{score}_3(T_2, \mathsf{q}) = \frac{1}{2} + 1 + 1 + 1 = 3.5$ *and* $\mathsf{score}_3(T_3, \mathsf{q}) = \frac{1}{6} + \frac{1}{6} + 1 + 1 = 2.33$. *It can be found that while the scoring function prefers smaller trees, it also prefers tree patterns with more valid subtrees and subtrees matching to keywords in text description with higher similarity. So we have* $\mathsf{score}(P_1, \mathsf{q}) > \mathsf{score}(P_2, \mathsf{q})$ *with $z_1 = -1$ and $z_2 = z_3 = 1$.*

2.2.4 Problem Statement

We now formally define the *d-height tree pattern problem* to be solved in the rest of this paper: given a keyword query q in a knowledge graph $\mathcal{G}$, the *d-height tree pattern problem* is to find all tree

patterns P, with height *at most* d, w.r.t. q. Users are usually interested in the top-k answers, so we focus on generating d-height tree patterns with the top-k highest relevance scores $\mathsf{score}(P, \mathsf{q})$'s.

We introduce the height threshold d of tree patterns for considerations of both search accuracy and efficiency. First, more compact answers (i.e., patterns with lower heights or tables with smaller numbers of columns) are usually more meaningful to users. Second, as keyword search is an online service, bounded height d ensures in-time response. The setting of d is *independent* on the number of keywords in the query, as it bounds the length of path from the root to *each* keyword. Such thresholds also appear in earlier work, *e.g.*, [20] as tree size constraint, and more recent work [25] as radius constraint, for similar considerations. Experimental study about the impact of d will be reported in Section 5.1.

2.3 Enumeration-Aggregation Approach and Hardness Result

An obvious baseline that adapts previous works on finding subtrees in RDB graph using keywords (*e.g.*, [11, 13, 18, 25]) for our problem is called *enumeration-aggregation approach*. First, in the *enumeration step*, individual valid subtrees of height at most d are generated one by one with an adaption of the backward search algorithm in [11]. No ranking or order of the generated subtrees is required, so the adapted algorithm in this step can ensure that, with proper preprocessing, the time needed to generate the i-th individual valid subtree is linear to the size of this tree, which is the best we can expect for an enumeration algorithm. Second, in the *aggregation step*, these valid subtrees are grouped by their tree patterns. Group-by in the second step is the bottleneck of this approach, but as the tree pattern of a subtree can be efficiently computed as discussed in Section 2.2.2, we can optimize this step using an efficient in-memory dictionary from tree patterns to valid subtrees.

Carefully-designed top-k search strategies in [11, 13, 18, 25] does not help for producing top-k tree patterns, because i) no matter in which order the valid subtrees are generated, a highly relevant tree pattern may appear at the end of this order (for example, it is possible that each valid subtree of the tree pattern has low relevance, but the tree pattern has a high aggregate score because there are many such subtrees); and ii) optimization for the top-k incurs additional cost (our baseline described above avoids to do so).

If we know the total number of tree patterns in advance, the enumeration-aggregation approach can early terminate as soon as we collect enough number of tree patterns during the enumeration. However, the hardness result below implies that it is impossible.

THEOREM 1. **(Counting Complexity)** *The problem of counting the number of tree patterns with height at most* d *for a keyword query* q *in a knowledge graph (*COUNTPAT*) is* #P*-Complete.*

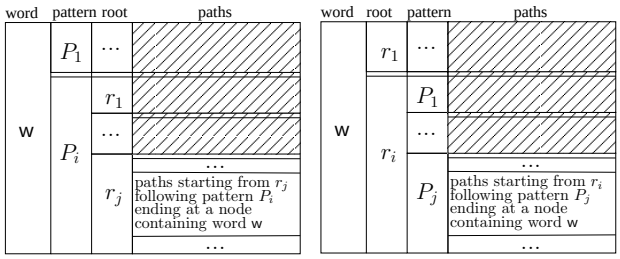
#P-Completeness is an analogue of NP-Completeness for counting problems. Our proof uses a reduction from the #P-Complete problem s-t PATHS [40]. Details are in the full version [1].

The hardness result and the reduction inspire us to precompute and index path patterns, as introduced next in Section 3.

3. INDEXING PATH PATTERNS

We propose a *path-pattern based index*, and it will be used to design efficient search algorithms introduced later in Section 4.

In the index, for each keyword w, we materialize all paths starting from some node (root) r in the knowledge graph $\mathcal{G}$, following certain pattern P, and ending at a node or an edge containing w. Recall that a word w may be contained in the text description of a node or the type of a node/edge. These paths are grouped by root r and pattern P. Only paths with length *at most* d need to be stored if we are considering the d-height tree pattern problem. Depending on the needs of algorithms (introduced in Sections 4.1 and 4.2), these paths are either sorted by patterns first and then roots (*pattern-first path index* in Figure 4(a)), or by roots first and then patterns (*root-first path index* in Figure 4(b)).



(a) Pattern-first path index (b) Root-first path index

Figure 4: Indexing patterns of paths ending at each word w **with length no more than** d

word	pattern	root	path
database	(Software)(Genre)(Model)	v_1	v_1v_2
database	(Software)(Genre)(Model)	v_7	v_7v_9
database	(Software)(Reference)(Book)	v_1	v_1v_{12}
database	(Book)	v_{12}	v_{12}
...	...	...	...

(a) Pattern-first path index for word "database".

word	root	pattern	path
database	v_1	(Software)(Genre)(Model)	v_1v_2
database	v_1	(Software)(Reference)(Book)	v_1v_{12}
database	v_7	(Software)(Genre)(Model)	v_7v_9
database	v_{12}	(Book)	v_{12}
...	...	...	...

(b) Root-first path index for word "database".

Figure 5: Examples of two types of path indexes for the knowledge graph in Figure 1(d)

The *pattern-first path index* (Figure 4(a)) provides the following methods to access the paths:

- $\mathrm{Patterns}(\mathsf{w})$: get all patterns following which some root can reach some node/edge containing w.
- $\mathrm{Roots}(\mathsf{w}, P)$: get all roots which reach some node/edge containing w through some path with pattern P.
- $\mathrm{Paths}(\mathsf{w}, P, r)$: get all paths with pattern P starting at root r and ending at some node/edge containing w.

Similarly, the *root-first path index* (Figure 4(b)) provides the following methods to access the paths:

- $\mathrm{Roots}(\mathsf{w})$: get all root nodes which can reach some node/edge containing w.
- $\mathrm{Patterns}(\mathsf{w}, r)$: get all patterns following which the root r can reach some node/edge containing w.
- $\mathrm{Paths}(\mathsf{w}, r)$: get all paths which start at root r and end at some node/edge containing w.
- $\mathrm{Paths}(\mathsf{w}, r, P)$: get all paths with pattern P starting at root r and ending at some node/edge containing w.

Following is a tiny example of how to access these two different types of indexes.

EXAMPLE 3.1. *For the knowledge graph in Figure 1(d), Figure 5 shows the two types of indexes on word* $\mathsf{w} =$ *"database".*

For the pattern-first path index in Figure 5(a), $\mathrm{Patterns}(\mathsf{w})$ *returns three patterns. Consider the pattern* $P =$ (Software) (Reference) (Book)*,* $\mathrm{Roots}(\mathsf{w}, P)$ *returns one root* $\{v_1\}$.

For the root-first path index in Figure 5(b), $\text{Roots}(\mathsf{w})$ *returns three roots* $\{v_1, v_7, v_{12}\}$. $\text{Patterns}(\mathsf{w}, v_1)$ *returns two patterns. Consider the pattern* $P =$ (Software) (Genre) (Model) *in particular,* $\text{Paths}(\mathsf{w}, v_1, P)$ *returns only one path* $\{v_1 v_2\}$.

Index Construction

To construct the indexes for a (user-specified) height threshold d, for each possible root r, we use DFS to find all paths p starting from r and ending at some node t/edge e with length no more than d. Let $\mathsf{text}(p)$ be the set of words in the text description or type of the node t/edge e, and recall $\mathsf{pattern}(p)$ is the path pattern of p. The index construction process is illustrated in Algorithm 1: each path p, together with its starting node r and pattern P, is inserted into proper positions of the two indexes in lines 5-6 (we use "+" to denote the insertion of an element into a dictionary).

The same set of paths are stored in these two types of indexes, but in different orders. We can use dictionary data structures, such as hash tables, to support the access methods $\text{Roots}()$, $\text{Patterns}()$, and $\text{Paths}()$ (in constant time). But to improve the efficiency of the access methods in practice, we then sort and store paths sequentially in memory: by patterns first and then roots for *pattern-first path index* as in Figure 4(a), or by roots first and then patterns for *root-first path index* as in Figure 4(b). Also, we store pointers pointing to the beginning of a list of paths with the same root r and/or pattern P to support the above access methods,

Note that the terms like $|T(\mathsf{w})|$, $\mathsf{PR}(f(\mathsf{w}))$, and $\mathsf{sim}(\mathsf{w}, f(\mathsf{w}))$ in the relevance-scoring functions (4)-(6) can be precomputed and stored in the path index as well, so that the overall score (2) can be computed efficiently online for a tree pattern.

Input: knowledge graph $\mathcal{G}$ and height threshold d

1: For each node r in the knowledge graph $\mathcal{G}$
2: For each path p starting from r with length $\leq d$
3: Let P be $\mathsf{pattern}(p)$, the path pattern of p.
4: For each word w in $\mathsf{text}(p)$
5: Construct pattern-first path index:
$\text{Patterns}(\mathsf{w}) \leftarrow \text{Patterns}(\mathsf{w}) + P$;
$\text{Roots}(\mathsf{w}, P) \leftarrow \text{Roots}(\mathsf{w}, P) + r$;
$\text{Paths}(\mathsf{w}, P, r) \leftarrow \text{Paths}(\mathsf{w}, P, r) + p$.
6: Construct root-first path index:
$\text{Roots}(\mathsf{w}) \leftarrow \text{Roots}(\mathsf{w}) + r$;
$\text{Patterns}(\mathsf{w}, r) \leftarrow \text{Patterns}(\mathsf{w}, r) + P$;
$\text{Paths}(\mathsf{w}, r, P) \leftarrow \text{Paths}(\mathsf{w}, r, P) + p$.
($\text{Paths}(\mathsf{w}, r)$ is supported by enumerating P and accessing $\text{Paths}(\mathsf{w}, r, P)$ for each P)

Algorithm 1: Constructing the two types of indexes

We can show that the size of our index is bounded by the total number of paths with length at most d and the size of text on entities and attributes. As these paths can be enumerated in linear time, the time to compute our path index is linear in the total number of paths and the size of text, with a logarithmic factor for sorting.

THEOREM 2. **(Index Cost)** *Let* $\mathcal{P}$ *be the set of paths in the index (with length at most* d*). For each* s-t *path* $p \in \mathcal{P}$*, let* $|p|$ *be its length,* $\mathsf{text}(p)$ *be the text on the node* t*, and* $|\mathsf{text}(p)|$ *be the number of words in the text. Then both the root-first and the pattern-first path indexes need space* $\mathrm{O}(\sum_{p\in\mathcal{P}} |p| \cdot |\mathsf{text}(p)|)$*, and can be constructed in linear time* $\mathrm{O}(\log |\mathcal{P}| \sum_{p\in\mathcal{P}} |p| \cdot |\mathsf{text}(p)|)$.

In practice, to handle synonyms, every word has its stemmed version and synonyms in our index pointing to the same path-pattern entry. The size of the index does not increase much.

4. SEARCHING WITH PATH INDEX

Two search algorithms for the d-height tree pattern problem are introduced in Sections 4.1 and 4.2: the first one performs well in practice but has exponential running time in the worst case; and the second one provides provable performance guarantee and can be further speedup using sampling techniques. Both of them utilize the path-pattern based index introduced in Section 3.

4.1 Pattern Enumeration-Join Approach

From the definition of a tree pattern in Equation (1), we can see that it is composed of m path patterns if there are m keywords in the query. Our first algorithm enumerates the combinations of these m path patterns in a tree pattern using the pattern-first path index (Figure 4(a)); for each combination, retrieves paths with these patterns from the index, and joins them at the root to check whether the tree pattern is empty (i.e., whether there is any valid subtree with this pattern). For each nonempty one, the valid subtrees in $\mathsf{trees}(P)$ and its score are then computed using the same index.

The algorithm, named as PATTERNENUM, is described in Algorithm 2. It first enumerates the root type of a tree pattern in line 2. For each root type C, it then enumerates the combinations of path patterns starting from C and ending at keywords w_i's in lines 4-8. Each combination of m path patterns forms a tree pattern P, but it might be empty. So lines 5-6 check whether $\mathsf{trees}(P)$ is empty again using the path index in lines 7-8. For each nonempty tree pattern, its score and the valid subtrees in $\mathsf{trees}(P)$ are computed and inserted into the queue Q in line 8. After every root type is considered, the top-k d-height tree patterns in Q can be output.

Input: knowledge graph $\mathcal{G}$, with pattern-first path index, and keyword query $\mathsf{q} = \{\mathsf{w}_1, \ldots, \mathsf{w}_m\}$

1: Initialize a queue Q of tree patterns, ranked by scores.
2: For each type $C \in \mathcal{C}$
3: Let $\text{Patterns}_C(\mathsf{w}_i)$ be the set of path patterns rooted at the type C in $\text{Patterns}(\mathsf{w}_i)$.
4: For each tree pattern $P = (P_1, \ldots, P_m)$
$\in \text{Patterns}_C(\mathsf{w}_1) \times \ldots \times \text{Patterns}_C(\mathsf{w}_m)$
Check whether $\mathsf{trees}(P)$ is empty:
5: Compute candidate roots $R \leftarrow \bigcap_{i=1}^{m} \text{Roots}(\mathsf{w}_i, P_i)$;
6: If $R \neq \emptyset$ then
7: $\mathsf{trees}(P) \leftarrow \bigcup_{r\in R} \text{Paths}(\mathsf{w}_1, P_1, r) \times \ldots \times \text{Paths}(\mathsf{w}_m, P_m, r)$;
8: Compute $\mathsf{score}(P, \mathsf{q})$ and insert P into queue Q.
(only need to maintain k tree patterns in Q)
9: Return the top-k tree patterns in Q and valid subtrees.

Algorithm 2: PATTERNENUM: finding top-k tree patterns by enumerating all possible tree patterns for a keyword query

EXAMPLE 4.1. *Consider a query "database software company revenue" with four keywords* w_1-w_4 *in the knowledge graph in Figure 1(d). When the root type* $C =$ Software*, we have two path patterns* (Software) (Genre) (Model) *and* (Software) (Reference) (Book) *from* $\text{Patterns}_C(\mathsf{w}_1)$*, as in Figure 5(a). To form the tree pattern in Figure 2(a), in line 4, we pick the first path pattern from* $\text{Patterns}_C(\mathsf{w}_1)$*,* (Software) *from* $\text{Patterns}_C(\mathsf{w}_2)$*,* (Software) (Developer) (Company) *from* $\text{Patterns}_C(\mathsf{w}_3)$*, and* (Software) (Developer) (Company) (Revenue) *from* $\text{Patterns}_C(\mathsf{w}_4)$*. We then find this tree pattern is not empty, and paths in the index with these patterns can be joined at nodes* v_1 *and* v_7*, forming two valid subtrees* T_1 *and* T_2*, respectively, in Figure 1(d).*

In the experiments, we will show that PATTERNENUM is efficient especially for queries which have relatively small numbers of tree patterns and valid subtrees. The advantage of this algorithm is that valid subtrees with the same pattern are generated at one time, so no online aggregation is needed. The path index has materialized aggregations of paths which can be used to check whether a tree pattern is empty and to generate valid subtrees. Also, it keeps at most k tree patterns and the corresponding valid subtrees in memory and thus has very small memory footprint.

However, in the worst case, its running time is still exponential both in the size of index and in the number of valid subtrees, mainly because unnecessary costly set-intersection operators are wasted on empty tree patterns (line 5). Consider such a worst-case example: In a knowledge graph, we have two nodes r_1 and r_2 with the same type C; r_1 points to p nodes $v_1, \ldots, v_p$ of types $C_1, \ldots, C_p$ through edges of types $A_1, \ldots, A_p$; and r_2 points to another p nodes $v_{p+1}, \ldots, v_{2p}$ of types $C_{p+1}, \ldots, C_{2p}$ through edges of types $A_{p+1}, \ldots, A_{2p}$. We have two words w_1 and w_2, w_1 appearing in $v_1, \ldots, v_p$ and w_2 appearing in $v_{p+1}, \ldots, v_{2p}$. To answer the query $\{\mathsf{w}_1, \mathsf{w}_2\}$, algorithm PATTERNENUM enumerates a total of p^2 combined tree patterns (CA_iC_i, CA_jC_j)'s for $i = 1, \ldots, p$ and $j = p+1, \ldots, 2p$, but they are all empty. So its running time is $\Theta(p^2)$ or $\Theta(p^m)$ in general for m keywords, where p is in the same order as the size of the index.

4.2 Linear-Time Enumeration Approach

We now introduce an algorithm to enumerate tree patterns for a given keyword using the root-first path index (Figure 4(b)). This algorithm is *optimal for enumeration* in the sense that its running time is linear in the size of the index and linear in the size of the answers (all valid subtrees). We prove its correctness and complexity. We will also introduce how to extend it for finding the top-k, and how to further speed it up using sampling techniques.

The algorithm, LINEARENUM in Algorithm 3, is based on the following idea: instead of enumerating all the tree patterns directly, we first find all possible roots for valid subtrees, and then assemble the trees from paths with these roots by looking up the path index.

These candidate roots, denoted as R, can be found based on the simple fact that a node in the knowledge graph is the root of some valid subtree if and only if it can reach every keyword at some node. So the set R can be obtained by taking the intersection of $\mathrm{Roots}(\mathsf{w}_1), \ldots, \mathrm{Roots}(\mathsf{w}_m)$ from the root-first path index (line 1).

For each candidate root r, recall that, using the path index, we can retrieve all patterns following which r can reach keyword w_i at some node by calling $\mathrm{Patterns}(\mathsf{w}_i, r)$. So pick any pattern $P_i \in \mathrm{Patterns}(\mathsf{w}_i, r)$ for each w_i, $P = (P_1, \ldots, P_m)$ is a nonempty tree pattern (i.e., $\mathsf{trees}(P) \neq \emptyset$). Line 7 of subroutine EXPANDROOT in Algorithm 3 gets all such patterns. Each P must be nonempty (with at least one valid subtree), because by picking any path p_i from $\mathrm{Paths}(\mathsf{w}_i, r, P_i)$ for each P_i, we can get a valid subtree $(p_1, \ldots, p_m)$ with pattern P, as in line 10. Note that valid subtrees with pattern P may be under different roots, so we need a dictionary, TreeDict in line 11, to maintain and aggregate the valid subtrees along the whole process. Finally, $\mathrm{TreeDict}[P]$ is the set of valid subtrees with pattern P as returned in lines 5-6.

EXAMPLE 4.2. *Consider a query "database software company revenue" with four keywords* w_1-w_4 *in the knowledge graph in Figure 1(d). The candidate roots we get are* $\{v_1, v_7, v_{12}\}$ *(line 1 of Algorithm 3). For* v_1 *and* w_1 *= "database", we can get two path patterns from* $\mathrm{Patterns}(\mathsf{w}_1, v_1)$*:* (Software) (Genre) (Model)*, and* (Software) (Reference) (Book)*. Picking the first one, together with patterns* (Software)*,* (Software) (Developer) (Company)*, and* (Software) (Develop) (Company) (Revenue) *for the other three keywords "software", "company", 'revenue", respectively, we can get the tree pattern in Figure 2(a) (one of* $\mathcal{T}$ *obtained in line 7). This pattern must be nonempty, because we can find a valid subtree under* v_1 *by assembling the four paths* v_1v_2*,* v_1*,* v_1v_3*, and* $v_1v_3v_4$ *into a subtree* T_1 *in Figure 1(d) (line 10).*

Another valid subtree, T_2 *in Figure 1(d), with the same pattern can be found later when candidate root* v_7 *is considered. They are both maintained in the dictionary* TreeDict.

Input: knowledge graph $\mathcal{G}$, root-first path indexes, and keyword query $\mathsf{q} = \{\mathsf{w}_1, \ldots, \mathsf{w}_m\}$

1: Compute candidate roots $R \leftarrow \bigcap_{i=1}^{m} \mathrm{Roots}(\mathsf{w}_i)$.
2: Initialize a dictionary TreeDict[].
3: For each candidate root $r \in R$
4: Call EXPANDROOT(r, TreeDict[]).
5: For each tree pattern P, $\mathsf{trees}(P) \leftarrow \mathrm{TreeDict}[P]$.
6: Return tree patterns and valid subtrees in $\mathsf{trees}(\cdot)$.

Subroutine EXPANDROOT(root r, dictionary TreeDict[])
Pattern Product:
7: $\mathcal{T} \leftarrow \mathrm{Patterns}(\mathsf{w}_1, r) \times \ldots \times \mathrm{Patterns}(\mathsf{w}_m, r)$;
8: For each tree pattern $P = (P_1, \ldots, P_m) \in \mathcal{T}$
Path Product:
9: For each $(p_1, \ldots, p_m) \in \mathrm{Paths}(\mathsf{w}_1, r, P_1) \times \ldots \times \mathrm{Paths}(\mathsf{w}_m, r, P_m)$
10: Construct tree T from the m paths $p_1, \ldots, p_m$;
11: $\mathrm{TreeDict}[P] \leftarrow \mathrm{TreeDict}[P] \bigcup \{T\}$.

Algorithm 3: LINEARENUM: finding tree patterns by enumerating valid subtrees rooted from each candidate root for a keyword query

LINEARENUM is optimal in the worst case because it does not waste time on invalid (empty) tree patterns. Every tree pattern it tries in line 8 has at least one valid subtree. And to generate each valid subtree, the time it needs is linear in its tree size (line 10). We formally present its correctness and complexity as follows.

THEOREM 3. **(Running Time and Correctness)** *For a keyword query* $\{\mathsf{w}_1, \ldots, \mathsf{w}_m\}$ *against a knowledge graph* $\mathcal{G}$*, let* S_i *be the size of the path index for word* w_i*, and let* N *be the total number of valid subtrees.* LINEARENUM *can correctly enumerate all tree patterns and valid subtrees in time* $\mathrm{O}\big(N \cdot d \cdot m + \sum_{i=1}^{m} S_i\big)$.

4.2.1 Partitioning by Types to Find Top-k

Now we introduce how to extend LINEARENUM in Algorithm 3 to find the top-k tree patterns (with the highest scores). A naive method is to compute the score $\mathsf{score}(P, \mathsf{q})$ for every tree pattern after we run LINEARENUM for the given keyword query q on the knowledge graph $\mathcal{G}$. An obvious deficiency of this method is that the dictionary TreeDict[] used in Algorithm 3 could be very large (may not fit in memory and may incur higher random-access cost for lookups and insertions), as it keeps every tree patterns and associated valid subtrees, but we only require the top-k.

A better idea is to apply LINEARENUM for candidate roots with the same type at one time. For each type C, we apply LINEARENUM only for candidate roots with type C (only line 3 of Algorithm 3 needs to be changed); then compute the scores of resulting tree patterns/answers but only keep the top-k tree patterns; and repeat the process for another root type. In this way, the size of the dictionary TreeDict[] is upper-bounded by the number of valid subtrees with roots of the same type, which is usually much smaller than the total number of valid subtrees in the whole knowledge graph.

For example, for the knowledge graph and the keyword query in Figure 1(d), the tree pattern P_1 in Figure 2(a) is found and scored when we apply LINEARENUM for the type "Software", and P_2 in Figure 2(b) is found when "Book" is the root type.

This idea, together with the sampling technique introduced a bit later, will be integrated into LINEARENUM-TOPK in Algorithm 4 for finding the top-k d-height tree patterns.

Input: knowledge graph $\mathcal{G}$, with both path indexes, and keyword query $\mathsf{q} = \{\mathsf{w}_1, \ldots, \mathsf{w}_m\}$
Parameters: sampling threshold Λ and sampling rate ρ
1: Initialize a queue Q of tree patterns, ranked by scores.
2: For each type C among all types $\mathcal{C}$
3: Compute candidate roots of type C: $R = (\bigcap_{i=1}^{m} \mathrm{Roots}(\mathsf{w}_i)) \bigcap C$;
4: Compute the number of valid subtrees rooted in R: $N_R = \sum_{r \in R} \prod_{i=1}^{m} |\mathrm{Paths}(\mathsf{w}_i, r)|$;
5: If $N_R \geq \Lambda$ let $rate = \rho$ else $rate = 1$;
6: Initialize dictionary $\mathrm{TreeDict}[]$;
7: For each candidate root $r \in R$,
8: With probability $rate$, call EXPANDROOT$(r, \mathrm{TreeDict}[])$;
9: For each tree pattern P rooted at C in $\mathrm{TreeDict}$
10: Compute estimated score $\widehat{\mathsf{s}}(P, \mathsf{q})$ ($\approx \mathsf{score}(P, \mathsf{q})$) from sample valid subtrees in $\mathrm{TreeDict}[P]$;
11: For each P with the top-k estimated score $\widehat{\mathsf{s}}$, Compute the exact score $\mathsf{score}(P, \mathsf{q})$ and insert P into the queue Q (with size at most k);
12: Return the top-k tree patterns in Q and valid subtrees.

Algorithm 4: LINEARENUM-TOPK (Λ, ρ): partitioning by types and sampling roots to find the top-k tree patterns

4.2.2 Speedup by Sampling

The two most costly steps in LINEARENUM are in subroutine EXPANDROOT: i) the enumeration of tree patterns in the product of $\mathrm{Patterns}(\mathsf{w}_i, r)$'s (line 7); and ii) the enumeration of valid subtrees in the product of $\mathrm{Paths}(\mathsf{w}_i, r, P_i)$'s (line 9). Too many valid subtrees could be generated and inserted into the dictionary $\mathrm{TreeDict}[]$ which is costly in both time and space. Now we introduce how to use sampling techniques to find the top-k tree patterns more efficiently (but with probabilistic errors).

Estimating scores using samples. Instead of computing the valid subtrees for every root candidate (as EXPANDROOT in Algorithm 3), we do so only for a random subset of candidate roots – each candidate root is selected with probability ρ. Equivalently, for each tree pattern P, only a random subset of valid subtrees in $\mathsf{trees}(P)$ are retrieved (kept in $\mathrm{TreeDict}[P]$), and we can use this random subset to estimate $\mathsf{score}(P, \mathsf{q})$ as $\widehat{\mathsf{s}}(P, \mathsf{q})$. We then only maintain tree patterns with the top-k estimated scores, without keeping the complete set of valid subtrees in $\mathsf{trees}(P)$ for each. Finally, we compute the exact scores and the complete sets of valid subtrees only for the estimated top-k, and re-rank them before outputting.

The detailed algorithm, called LINEARENUM-TOPK, is described in Algorithm 4. In addition to the input knowledge graph and keyword query, we have two more parameters Λ and ρ. We first enumerate the type of roots in a tree pattern in line 2. For each type, similarly as LINEARENUM, candidate roots of this are computed in line 3. We can compute the number of valid subtrees (possibly from different tree patterns) with these roots as N_R in line 4, without really enumerating them. To this end, we only need to get the number of paths starting from each candidate root r and ending at each keyword w_i. Only when the number of valid subtrees is no less than Λ, we apply the root sampling technique in lines 7-8 with $rate = \rho$ (otherwise $rate = 1$): for each candidate root r, with probability $rate$, we compute the valid subtrees under it and insert them into the dictionary $\mathrm{TreeDict}[]$ (subroutine EXPANDROOT in Algorithm 3 is re-used for this purpose). After all candidate roots of a type are considered, in lines 9-10, we can compute the estimated score as $\widehat{\mathsf{s}}(P, \mathsf{q})$ for each tree pattern P in $\mathrm{TreeDict}$. Only for tree patterns with the top-k estimated scores, we compute their valid subtrees with exact scores and insert them into a global queue Q in line 11 to find the global top-k tree patterns.

The running time of LINEARENUM-TOPK can be controlled by parameters Λ and ρ. Sampling threshold Λ specifies for which types of roots, we sample the valid subtrees to estimate the pattern scores. By setting $\Lambda = +\infty$ and $\rho = 1$ (no sampling at all), we can get the exact top-k. When $\Lambda < +\infty$ and $\rho < 1$, the algorithm is speedup but there might be errors in the top-k answers. In the experiments, we will show that even when $\rho = 0.1$ (i.e., use 10% valid subtrees to estimate the pattern scores), we can get reasonably precise top-k tree patterns while the algorithm is speedup roughly 10 times. The theoretical analysis about the running time and precision of LINEARENUM-TOPK are in the following two theorems.

THEOREM 4. **(Running Time)** *For a keyword query* $\{\mathsf{w}_1, \ldots, \mathsf{w}_m\}$ *in a knowledge graph* $\mathcal{G}$*, let* S_i *be the size of the path index for word* w_i*, let* N *be the total number of valid subtrees, and let* $|\mathcal{C}|$ *be the total number of types.* LINEARENUM-TOPK *needs time:* $\mathrm{O}\big(\min(\Lambda \cdot |\mathcal{C}|, N) \cdot d \cdot m + \rho \cdot N \cdot d \cdot m + \sum_{i=1}^{m} S_i + N \cdot \log k\big)$.

(Correctness) *When* $\Lambda = +\infty$ *and* $\rho = 1$ *(no sampling), the algorithm outputs the correct top-*k *tree patterns.*

We establish the *pairwise precision* of LINEARENUM-TOPK: for two tree patterns P_1 and P_2 with exact scores $\mathsf{score}(P_1, \mathsf{q}) > \mathsf{score}(P_2, \mathsf{q})$ in the general form of (2), how likely we would order them incorrectly, $\widehat{\mathsf{s}}(P_1, \mathsf{q}) < \widehat{\mathsf{s}}(P_2, \mathsf{q})$, according to the estimated scores obtained from a random sample of valid subtrees (so that P_1 might be missed from the top-k output by the algorithm).

THEOREM 5. **(Precision)** *For a query* q *and tree patterns* P_1 *and* P_2 *with scores* $\mathsf{s}_1 = \mathsf{score}(P_1, \mathsf{q})$ *and* $\mathsf{s}_2 = \mathsf{score}(P_2, \mathsf{q})$ *s.t.* $\mathsf{s}_1 > \mathsf{s}_2$*, if* LINEARENUM-TOPK *runs with* $\Lambda = 0$ *(always sampling) and sampling rate* $\rho < 1$*, then* $\widehat{\mathsf{s}}(P_1, \mathsf{q}) < \widehat{\mathsf{s}}(P_2, \mathsf{q})$ *(P_1 is incorrectly ranked lower than P_2 in estimation) with probability*

$$\Pr[\textit{error}] \leq \exp\left(-2\left(\frac{\mathsf{s}_1 - \mathsf{s}_2}{\mathsf{s}_1 + \mathsf{s}_2}\right)^2 \cdot \rho^2\right). \tag{7}$$

To prove the above theorem, we note that the score $\mathsf{score}(P_i, \mathsf{q})$ can be decomposed among all candidate roots, i.e., rewritten as

$$\mathsf{s}_i = \mathsf{score}(P_i, \mathsf{q}) = \sum_{T \in \mathsf{trees}(P_i)} \mathsf{score}(T, \mathsf{q}) = \sum_{r \in \mathcal{V}} \sum_{T \in \mathsf{trees}_r(P_i)} \mathsf{score}(T, q),$$

where $\mathsf{trees}_r(P_i)$ is the set of valid subtrees with pattern P_i and rooted at node r. Let $\mathsf{s}_i(r) = \sum_{T \in \mathsf{trees}_r(P_i)} \mathsf{score}(T, q)$ be the sum of relevance scores of all valid subtrees rooted at r for pattern P_i, and thus $\mathsf{s}_i = \sum_r \mathsf{s}_i(r)$. In order to compare s_1 and s_2, we can compare $\sum_{r \in R^+} \mathsf{s}_1(r)$ and $\sum_{r \in R^+} \mathsf{s}_2(r)$ on a random subset R^+ of all candidate roots (sampled in line 8 of LINEARENUM-TOPK with rate ρ). Using Hoeffding's inequality [15], we can bound the probability that we make mistakes by a term that is exponentially small in the sampling rate ρ and the difference between s_1 and s_2. Detailed proof can be found in the full version of this paper [1].

The theorem has two direct implications which are consistent to our intuition: i) the error probability decreases when the (relative)

	$d=2$	$d=3$	$d=4$
Time (s)	43	502	7,011
Size (MB)	229	2,633	34,485

Figure 6: Index construction cost on Wiki for different d

difference between $\mathsf{score}(P_1, \mathsf{q})$ and $\mathsf{score}(P_2, \mathsf{q})$ becomes larger; and ii) the error probability is smaller for higher sampling rate ρ (exponentially in ρ^2). They partly explain why the sampling technique works well in practice, as shown in Section 5.2.

How to set sampling threshold and sampling rate. Intuitively, the sampling threshold Λ determines *when to sample*, i.e., for each entity type, applying the sampling technique when the number of valid subtrees with roots of this type is no less than Λ; and the sampling rate ρ determines the *sample size* for each root type.

A global sampling threshold Λ can be set regardless of the query and the number of valid subtrees w.r.t. it. The rationale is that, when the number of entity types is fixed, if the number of valid subtrees rooted in a type is less than Λ, sampling is not necessary (sampling rate set to 1 in line 5 of Algorithm 4) because computing the exact scores is not expensive anyway. On the other hand, when the number of valid subtrees rooted in a type is at least Λ, we sample a fixed portion (ρ) of them to estimate the scores, and Theorem 5 provides a guarantee of precision w.r.t. ρ. So Λ and ρ can be set regardless of the queries, but they do rely on users' preference (trade-off between the response time of the system and the precision) for fixed scheme of the knowledge graph.

5. EXPERIMENTS

The following approaches for the d-height tree pattern problem are implemented in C#. They are evaluated on a machine with 2.4 GHz Intel CPUs and 96 GB memory, under Windows Server.

Baseline: The baseline approach described in Section 2.3.

PETopK: Our first algorithm, the pattern enumeration-join approach PATTERNENUM described in Section 4.1.

LETopK: Our second algorithm, LINEARENUM-TOPK, described in Section 4.2. Recall that, when the two parameters sampling threshold $\Lambda = +\infty$ and sampling rate $\rho = 1$, it gets exact top-k answers; and otherwise, it gets approximate top-k.

Datasets. We compare the algorithms on two real-life datasets, Wiki [5] and IMDB [7]. The Wiki dataset contains 1.89 million entities. The type of each entity and its attributes are extracted from its infobox block on the top-right of its page. There are a total of 3,424 types. The corresponding knowledge graph contains 34.99 million edges. The IMDB dataset contains 7 types of 6.58 million entities, with 79.42 million directed edges in the knowledge graph.

Queries. We randomly selected 500 queries from Bing's log for experiments on Wiki. The numbers of keywords in the queries vary from 1 to 10, and for each we have 50 queries. For IMDB, we randomly constructed 500 queries from IMDB's vocabulary. Again, the numbers of keywords in the queries are from 1 to 10, and for each there are 50 queries. When we report the running time of an algorithm for a set of queries, we report the min / (geometric) average / max execution time in the form of error bars.

Index size and height threshold d. We build the path indexes described in Section 3 with different height thresholds $d = 2$, 3, and 4 for the Wiki dataset. The time needed to construct them and their sizes are reported in Figure 6. Both the time and the size increase exponentially in d mainly because the number of possible tree patterns increases exponentially. For IMDB, the knowledge graph contains only paths of length at most three, and the size of the indexes is 0.8 GB. All the indexes are stored in memory.

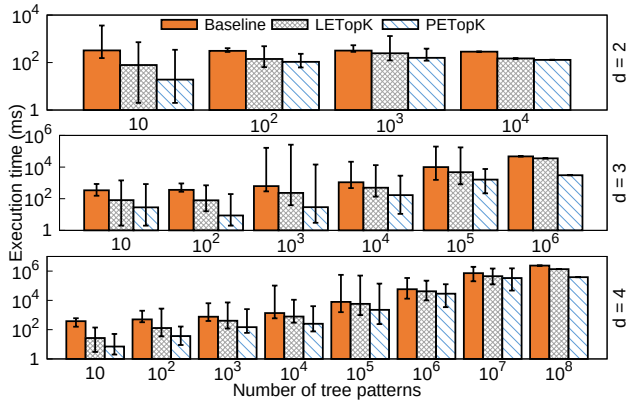


Figure 7: Execution time and number of tree patterns (with height at most d) for different height threshold d on Wiki

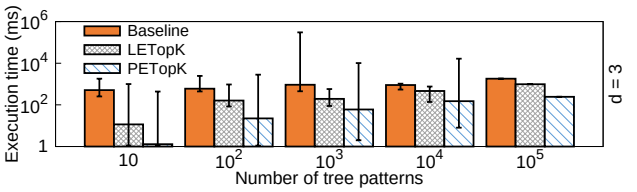


Figure 8: Execution time and number of tree patterns with height at most d on IMDB

5.1 Performance of Exact Algorithms

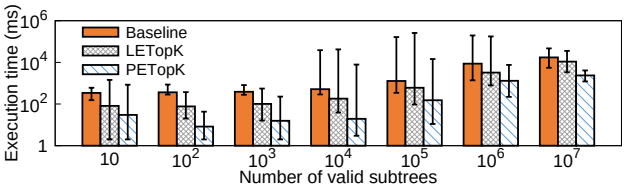
We first compare different approaches when the exact top-k tree patters are desired. No sampling is used in LETopK ($\Lambda = +\infty$ and $\rho = 1$). We use $k = 100$ by default and Exp-IV is about varying k.

Exp-I: Varying height threshold d and number of tree patterns. We first vary the height threshold d for the Wiki dataset. When d increases, the number of paths with length at most d increases significantly, and as a result, for a fixed query, the number of valid subtrees and tree patterns also increase significantly. From Figure 7, we can see that the number of tree patterns increases from $[10, 10^4]$ to $[10, 10^8]$ for $d = 2, 3, 4$. For each d, we study how the number of tree patterns affects the execution time of query processing. The 500 queries on Wiki are partitioned into different groups based on the total number of possible tree patterns that can be found for each query, *e.g.*, group 10^2 contains all queries with $10 - 99$ tree patterns. The results are reported in Figure 7.

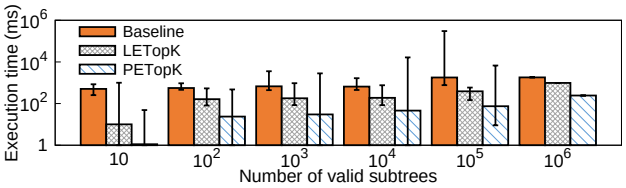
It can be seen that larger d greatly affects the performance of our algorithms, with a larger number of possible tree patterns as the major reason. Overall, LETopK is faster than Baseline, and PETopK is the fastest among the three algorithms. We want to emphasize that the advantage of LETopK in practice mainly relies on the sampling technique. But sampling is disabled for now to compare exact top-k algorithms, and will be discussed in Section 5.2.

In terms of the answer quality, on one hand, d should be large enough to ensure that we explore enough number of interpretations for the query; and on the other hand, if d is too large, some large tree patterns that correspond to loose relationship among keywords may appear among the top answers, which actually deteriorate the answer quality. Similar finding was also made in [25] for ranking individual subtrees. In our case, when $d = 3$, the best interpretations (tree patterns) of the queries on Wiki can be found at an average ranking of 2.797. We will miss some of them for $d = 2$. But for $d = 4$, the (same) best interpretations have an average ranking of 12.514. So we use $d = 3$ in the rest experiments for Wiki.

In IMDB, the max length of directed paths is three, so $d = 3$ suffices (since tree patterns here have heights at most 3). The results are reported in Figure 8 for $d = 3$. The set of answers and



(a) Varying number of valid subtrees on Wiki dataset



(b) Varying number of valid subtrees on IMDB dataset

Figure 9: Execution time for different queries

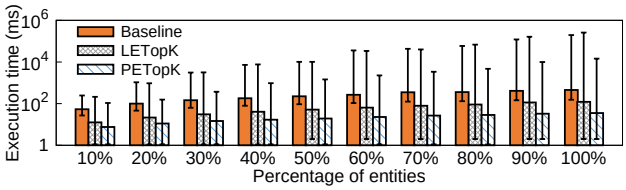


Figure 10: Execution time on Wiki datasets of different sizes

execution time will be exactly the same for $d > 3$. Similar to the results in Wiki, while the number of possible tree patterns affects execution time, PETopK is the fastest one on average.

Exp-II: Varying number of valid subtrees. Besides the number of tree patterns, another important parameter about a keyword query is how many valid subtrees in total can be found in the knowledge graph. This parameter may affect the performance of algorithms a lot. For example, Theorem 3 indicates that the running time of LETopK is linear in this number. So we partition queries into different groups based on how many valid subtrees a query has in total (*e.g.*, group 10^3 contains all queries with 100 – 999 valid subtrees). Figure 9 reports the execution time when varying the number of valid subtrees on both Wiki and IMDB.

Again, LETopK is faster than Baseline, and PETopK is the fastest among the three algorithms. The execution time of Baseline and LETopK is bound by the time on building the dictionary TreeDict. LETopK is faster than Baseline as a result of the "partitioning by types" technique in Section 4.2.1. PETopK is usually the fastest since the pattern-first path index it uses allows it to avoid the time consuming dictionary building and online aggregation.

Exp-III: Varying size of knowledge graph. We study the scalability of different algorithms on the Wiki dataset by varying the number entities and types in the knowledge graph. We randomly select a subset of entities from the Wiki dataset, and construct the induced subgraph of the original knowledge graph w.r.t. the selected subset of entities. The execution time of each algorithm on the induced knowledge graphs for different numbers (10%-100%) of entities is shown in Figure 10. The execution time of each algorithm increases (almost) linearly as the number of entities increases from 10% to 100% of the entities in the Wiki dataset.

Similar results are found for varying numbers of entity types in the knowledge graph. Details are omitted for the space limit.

Exp-IV: Varying parameter k. The value of k has very little impact on the execution time of our algorithms. For each tree pattern, it takes $O(\log k)$ operations to insert it to the priority queue of size k, while the number of operations required to find it is independent of k (which is usually much larger than $O(\log k)$). Thus, the execution time is dominated by the aggregation/enumeration of valid subtrees, and is almost not affected by the value of k.

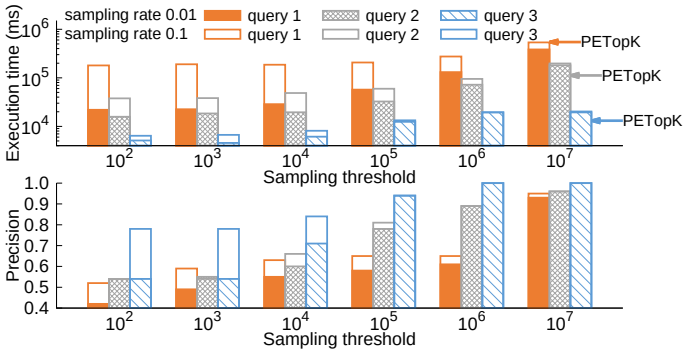
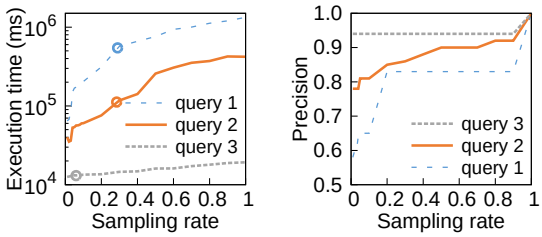


Figure 11: Performance of LETopK with different sampling threshold Λ (for $k = 100$). The execution time of PETopK is marked on the side of histograms.



(a) Execution time (b) Precision

Figure 12: Performance of LETopK with different sampling rate ρ (for $\Lambda = 10^5$, $k = 100$). The circles on curves in Figure 12(a) mark the execution time of PETopK.

5.2 Performance of Sampling Algorithm

Now we study the performance of the sampling technique used in LETopK. Execution time and precision are the two measures that we are interested in. The *precision* here is defined as *the ratio between the number of truely top-k answers found by* LETopK *(with sampling) and* k. We focus on Wiki, since the number of valid subtrees is usually much smaller on IMDB (so sampling is not useful there). We selected three queries with different numbers of valid subtrees. The numbers of valid subtrees / tree patterns for them are (2,479,899 / 314,614), (819,739 / 61,967) and (540,849 / 32,300).

Exp-V: Varying sampling threshold Λ. Recall that the sampling threshold Λ determines *when to sample*: for each entity type, applying the sampling technique when the number of valid subtrees with roots of this type is no less than Λ; and the sampling rate ρ determines the *sample size* for each root type. The performance of LETopK for different sampling threshold is reported in Figure 11 for $\rho = 0.01$ and 0.1. Overall, both the execution time and the precision increase when the sampling threshold increases. We mark the execution time of PETopK in Figure 11. LETopK is slower than PETopK for a very large sampling threshold (e.g., 10^7) but becomes faster when the threshold is less than or equal to 10^5. In the next experiment, we will fix $\Lambda = 10^5$ (as a balance between efficiency and precision), and vary the sampling rate.

Exp-VI: Varying sampling rate ρ. The performance of LETopK for different sampling rate is in Figure 12. The circle on the execution time curve for each query is the execution time of PETopK.

For queries with larger numbers of valid subtrees (query 1 and query 2), LETopK becomes much (5x-20x) faster than PETopK when a smaller sampling rate is used (e.g., 0.2 for query 1 and query 2), while preserving reasonably high precision (above 80%).

For the query with a smaller number of valid subtrees (query 3),

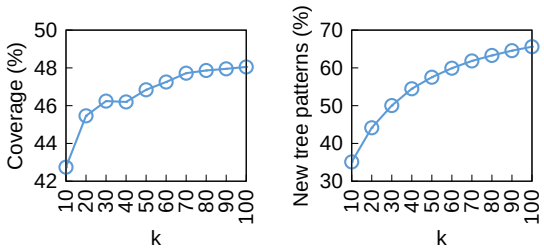


Figure 13: Average coverage of individual relevant trees in top-k tree patterns, and new tree patterns found in top-k

the performance of PETopK is on a par with LETopK. The reason is that, for LETopK, the sampling threshold $\Lambda = 10^5$ is large in comparison to the number of valid subtrees (540,849 for query 3) – so only for a few entity types where the numbers of valid subtrees rooted are larger than 10^5, the sampling technique is enabled. As a result, only when the sampling rate used in LETopK is small enough (≤ 0.05), LETopK is faster than PETopK, but the precision of LETopK is still consistently stable at round 0.95 (because sampling is enabled only for a small number of root types).

5.3 Individual Trees v.s. Tree Patterns

Recall the major motivation of this paper is to search tree patterns when the users want to find table answers (each represented as a set of subtrees with the same tree pattern) using keywords. We are not excluding individual best valid subtrees. But we aim to provide an additional module for the search engine to produce and rank highly relevant tree patterns (table answers). This new module could coexist with the individual-page ranking module or individual-tree ranking module. Which module we want the search engine to direct users to automatically according to the query intention analysis and how to mix individual valid subtrees with tree patterns to provide a universal ranking are both open problems. It will be an interesting future work to address them using extensive user study.

We compute a separate list of individual top-k valid subtrees, based on their tree scores in Equation (3). For the 500 keyword queries on Wiki, we calculate the average coverage of the individual top-k subtrees in top-k tree patterns (each as one row in some aggregated table), and the average percentage of top-k patterns that cannot be found in the individual top-k subtrees. The results are reported in Figure 13, for k varying from 10 to 100. Because of their "singular" patterns (i.e., only a small number of valid subtrees have the same pattern), around half of the individual top-k subtrees are lost in the top-k tree patterns. At the same time, up to 70% of the top-k tree patterns are new to the individual top-k subtrees.

Case study. We consider the query "XBox Game" in the Wiki dataset to compare the individual top valid subtrees and the top tree patterns in Figures 14-15. Both individual subtrees and tree patterns are shown as tables with column names as edge(attribute)/node types and row cells as entities. The top-1 individual valid subtree for "XBox Game" finds the entity "XBox", because of its relatively high PageRank score, with one additional edge/attribute containing the keyword "game". The top-2 finds a bigger subtree with "DVD" as the root and two branches "DVD-usage-XBox" and "DVD-owners-Sony-products-video game", and it ranks high mainly because of the high PageRank score of "DVD". The top-3 finds a singular entity with "XBox" appearing in the entity name and "Game" in the entity type. Of course, when the user's intention is to find "a list of XBox games" by issuing this query, the tree pattern/table answer shown in Figure 15 is better; and when the intention is to find "popular XBox game", the top-1 individual valid subtree in Figure 14 is also a good candidate. Top-2 and top-3 valid subtrees are the cases when a top individual subtree is lost in our

Top-1

information appliance	top **game**
Xbox	Halo 2

Top-2

storage medium	usage	owners/creators	products
DVD	**Xbox**	Sony	video **game**

Top-3

video **game** online service
Xbox Live Arcade

Figure 14: Top individual valid subtrees for "XBox Game"

Top-1

video **game**	platform
Halo 2	**Xbox**
GTA: San Andreas	**Xbox**
Painkiller	**Xbox**
...	...

Figure 15: Top-1 tree pattern for "XBox Game"

top-k tree pattern answers because of the singularity of its pattern (no other valid subtree has the same pattern).

6. RELATED WORK

Searching and ranking tables. As search engines are able to keep more and more tables from the Web, there have been efforts to utilize these tables. On one hand, Web tables can be leveraged and returned directly as answers in response to keyword queries [27, 41, 35, 44]. On the other hand, Web tables can be used to understand keyword queries better through mapping query words to attributes of tables [37] and to provide direct answers to fact lookup queries [45]. Different from the above works, in this paper, we focus on the scenarios when relevant and complete tables are not available for user-given keyword queries, and our goal is to compose tables online as answers to those queries.

Searching subtrees/subgraphs in RDB. Previous studies on keyword search in RDB extend ranking documents/webpages into ranking substructures of joining tuples which together contain all keywords in a query. They model an RDB as a graph, where tuples/tables are nodes and foreign-key links are edges. Each answer to a keyword query in such a graph could be either a subtree ([10], [20], etc.) or a subgraph ([36], [25], etc.) with all the keywords contained. There are two lines of works with the same goal of finding and ranking these answers. The first line materializes the RDB graph and proposes indexes and/or algorithms to enumerate top-k subtrees or subgraphs [11, 21, 22, 13, 18, 17, 25], etc. The second line first enumerates possible join trees/plans (candidate networks) based on the database schema and then evaluates them using SQL to obtain the answers [10, 20, 19, 28, 33, 36, 34], etc. Yu et al. [46] provide a comprehensive survey on these two lines.

Our enumeration-aggregation baseline borrows ideas from the first line of previous works. It essentially first enumerates valid subtrees in our knowledge graph and groups them by their tree patterns. But this method is deficient because the bottleneck now is the grouping step instead of the enumeration step. The second line of works (candidate network enumeration-evaluation) are not applicable in our problem because the schema of a knowledge base is usually much larger than the schema of an RDB, and thus the first step, *candidate network enumeration*, becomes the bottleneck. [23] analyzes the complexity of this subproblem and proposes a novel parameterized algorithm which is interesting in theory.

Keyword search in XML data. Another important line of works are to search LCAs (lowest common ancestors) in XML trees using keywords, [26, 43, 38, 30], etc. The general goal is to find lowest common ancestors of groups of nodes containing the keywords in the query. These LCAs, together with keyword-node matches sometimes, are returned as answers to the keyword query. Various strategies to identify relevant matches by imposing constraints on answers are developed, such as *meaningful LCA* [26], *smallest LCA*

[43, 38], and *MaxMatch* [30]. LCA-based approaches are not applicable in our problem for two reasons: i) our goal is to find and rank tree patterns, each of which aggregates a group of subtrees according to the node/edge types on the paths from the root to each leaf containing a keyword, instead of individual roots/matches as in LCA approaches – if we enumerate all LCA matches and group them by patterns, it would be equivalent to our baseline; and ii) LCA is not well defined in our knowledge graph with cycles.

In addition, XSeek [29] tries to infer users' intention by categorizing keywords in the query into *predicates* and *return nodes*. And [31] defines an equivalence relationship among query results on XML based on the classification of predicates and return nodes of keywords. [32] provides a comprehensive survey on this line.

Keyword search in RDF graphs. Our knowledge graph can be considered as an RDF graph. Previous works on keyword search in RDF graphs extend the two lines of works on keyword search in RDBMS. For example, [39] assume that user-given keyword queries implicitly represent structured triple-pattern queries over RDF. They aim to find the top-k structured queries that are relevant to a keyword query, which essentially extends the candidate network enumeration problem in RDBMS to RDF. [16] and [12] study ranking models and algorithms for the results of those structured queries over RDF. [24] tries to find the top-k entities that are reachable from all the keywords in the query over RDF. [42] proposes a new summarization language which improves result understanding and query refinement. It takes all the answers (subgraphs in RDF) to structured queries as input and output a summarization which is as concise as possible and satisfies certain coverage constraint.

Searching aggregations in multidimensional data. A major motivation of our work is that a meaningful answer to a keyword query may be a collection of tuples/tuple joins, which need to be aggregated before being output. This idea is also explored in multidimensional text data by [14, 47]. With a different data model and application scenarios, an answer there is a "group-by" on a subset of dimensions such that all keywords are contained in the aggregated tuples. In [47], how to enumerate all valid and minimal answers is studied, and in [14], scoring models for those answers and efficient algorithms to find the top-k are proposed.

7. CONCLUSIONS

We introduce the d-height tree pattern problem in a knowledge base for keyword search. Formal models of tree patterns are defined to aggregate subtrees in a knowledge graph which contain all keywords in a query. Such tree patterns can be used to better understand the semantics of keyword queries and to compose table answers for users. We propose path-based indexes and efficient algorithms to find tree patterns for a given keyword query. To further speed up query processing, a sampling-based approach is introduced to provide approximate top-k with higher efficiency. Our approaches are evaluated using real-life datasets.

8. REFERENCES

[1] http://arxiv.org/abs/1409.1292.

[2] http://dbpedia.org/About.

[3] http://office.microsoft.com/en-us/excel/download-microsoft-power-query-for-excel-FX104018616.aspx.

[4] http://research.google.com/tables.

[5] https://www.wikipedia.org/.

[6] http://www.freebase.com/.

[7] http://www.imdb.com/.

[8] http://www.informatik.uni-trier.de/ ley/db/.

[9] http://www.mpi-inf.mpg.de/yago-naga/yago/.

[10] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.

[11] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, 2002.

[12] V. Bicer, T. Tran, and R. Nedkov. Ranking support for keyword search on structured data using relevance models. In *CIKM*, 2011.

[13] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *ICDE*, 2007.

[14] B. Ding, B. Zhao, C. X. Lin, J. Han, C. Zhai, A. N. Srivastava, and N. C. Oza. Efficient keyword-based search for top-k cells in text cube. *IEEE Trans. Knowl. Data Eng.*, 23(12), 2011.

[15] D. P. Dubhashi and A. Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge Univ. Press, 2009.

[16] S. Elbassuoni and R. Blanco. Keyword search over rdf graphs. In *CIKM*, 2011.

[17] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *SIGMOD Conference*, 2008.

[18] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD Conference*, 2007.

[19] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, 2003.

[20] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, 2002.

[21] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, 2005.

[22] B. Kimelfeld and Y. Sagiv. Finding and approximating top-k answers in keyword proximity search. In *PODS*, 2006.

[23] B. Kimelfeld and Y. Sagiv. Finding a minimal tree pattern under neighborhood constraints. In *PODS*, 2011.

[24] F. Li, W. Le, S. Duan, and A. Kementsietsidis. Scalable keyword search on large rdf data. *IEEE Transactions on Knowledge and Data Engineering*, 2014.

[25] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD Conference*, 2008.

[26] Y. Li, C. Yu, and H. V. Jagadish. Schema-free xquery. In *VLDB*, 2004.

[27] G. Limaye, S. Sarawagi, and S. Chakrabarti. Annotating and searching web tables using entities, types and relationships. *PVLDB*, 3(1), 2010.

[28] F. Liu, C. T. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD Conference*, 2006.

[29] Z. Liu and Y. Chen. Identifying meaningful return information for xml keyword search. In *SIGMOD Conference*, 2007.

[30] Z. Liu and Y. Chen. Reasoning and identifying relevant matches for xml keyword search. *PVLDB*, 1(1), 2008.

[31] Z. Liu and Y. Chen. Return specification inference and result clustering for keyword search on xml. *ACM Trans. Database Syst.*, 35(2), 2010.

[32] Z. Liu and Y. Chen. Processing keyword search on xml: a survey. *World Wide Web*, 14(5-6), 2011.

[33] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD Conference*, 2007.

[34] Y. Luo, W. Wang, X. Lin, X. Zhou, J. Wang, and K. Li. Spark2: Top-k keyword query in relational databases. *IEEE Trans. Knowl. Data Eng.*, 23(12), 2011.

[35] R. Pimplikar and S. Sarawagi. Answering table queries on the web using column keywords. *PVLDB*, 5(10), 2012.

[36] L. Qin, J. X. Yu, and L. Chang. Keyword search in databases: the power of rdbms. In *SIGMOD Conference*, 2009.

[37] N. Sarkas, S. Paparizos, and P. Tsaparas. Structured annotations of web queries. In *SIGMOD Conference*, 2010.

[38] C. Sun, C. Y. Chan, and A. K. Goenka. Multiway slca-based keyword search in xml data. In *WWW*, 2007.

[39] T. Tran, H. Wang, S. Rudolph, and P. Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data. In *ICDE*, 2009.

[40] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput*, 8(3), 1979.

[41] P. Venetis, A. Y. Halevy, J. Madhavan, M. Pasca, W. Shen, F. Wu, G. Miao, and C. Wu. Recovering semantics of tables on the web. *PVLDB*, 4(9), 2011.

[42] Y. Wu, S. Yang, M. Srivatsa, A. Iyengar, and X. Yan. Summarizing answer graphs induced by keyword queries. *PVLDB*, 6(14), 2013.

[43] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest lcas in xml databases. In *SIGMOD Conference*, 2005.

[44] M. Yakout, K. Ganjam, K. Chakrabarti, and S. Chaudhuri. Infogather: entity augmentation and attribute discovery by holistic matching with web tables. In *SIGMOD Conference*, 2012.

[45] X. Yin, W. Tan, and C. Liu. Facto: a fact lookup engine based on web tables. In *WWW*, 2011.

[46] J. X. Yu, L. Qin, and L. Chang. *Keyword Search in Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2009.

[47] B. Zhou and J. Pei. Aggregate keyword search on large relational databases. *Knowl. Inf. Syst.*, 30(2), 2012.