Efficient Pruned Top-K Subgraph Matching with Topology-Aware Bounds

Linglin Yang
Peking University
Beijing, China
linglinyang@stu.pku.edu.cn

Yuqi Zhou
Peking University
Beijing, China
zhouyuqi@pku.edu.cn

Yue Pang
Peking University
Beijing, China
michelle.py@pku.edu.cn

Lei Zou
Peking University
Beijing, China
zoulei@pku.edu.cn

Abstract

Given a query graph, top-k subgraph matching finds up to k matches in a data graph with the highest scores according to a user-defined scoring function. It has wide applications across many fields, including knowledge graphs and social networks. Due to the enormous search space, existing methods are not efficient enough on large graphs. In this paper, we propose PTAB, an efficient algorithm for top-k subgraph matching. It traverses an efficiently pruned search space by topology-aware sub-space score upper bounds computed from a novel hop index, which stores the range of node properties in a constrained multi-hop neighborhood of each node. Additionally, PTAB integrates a cost-aware root selection strategy, which chooses query nodes leading to a search process that utilizes the pruning power of the hop index as much as possible. Furthermore, we use a novel edge-cut strategy to handle general query graphs with cycles. Experimental results on real and synthetic datasets demonstrate that our method outperforms existing methods.

CCS Concepts

• **Information systems** → **Information retrieval query processing**.

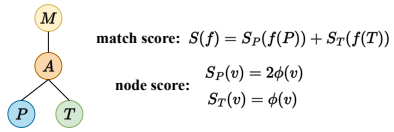
Keywords

Graph DB, Top-k query, Subgraph matching, Query optimization

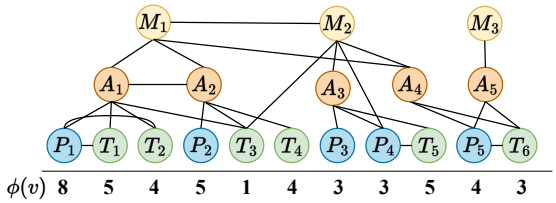
ACM Reference Format:
Linglin Yang, Yuqi Zhou, Yue Pang, and Lei Zou. 2024. Efficient Pruned Top-K Subgraph Matching with Topology-Aware Bounds. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management (CIKM '24), October 21–25, 2024, Boise, ID, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3627673.3679790

1 Introduction

Graph databases have garnered significant attention recently for their capacity to store entity relationships in a graphical format. One of the most important query forms in graph databases is top-k subgraph matching [12, 30]. Similar to top-k queries in relational databases, these queries retrieve representative data subsets characterized by specific property combinations, which translate to high scores computed from node properties in the matching results. Specifically, given a data graph G, a query graph Q, and a score function S, top-k subgraph matching returns the k subgraph matches of Q in G with the highest scores according to S.



(a) Query graph and the score function



(b) Data graph

Figure 1: A top-k subgraph matching example. The numbers below the programmers and the testers represent their competency. Edges are labeled "past successful collaboration" and are omitted for simplicity.

Fig. 1 demonstrates an application of top-k subgraph matching for mission assignment within a company. The data graph in Fig. 1(b) illustrates an organizational network of project managers, architects, programmers, and testers (labeled M, A, P, and T, respectively). In this network, edges indicate previous successful collaborations, and the competency of the individual v, denoted by $\phi(v)$, is recorded for programmers and testers as their node properties.

In scenarios where a project requires one individual from each of these four categories, it is crucial for smooth project execution that these professionals have prior cooperative experience, as represented by the query graph in Fig. 1(a). Moreover, the quality of the project's output is influenced by the competency of the programmer and the tester. In a project that more heavily depends on the programmers' competency, we may calculate the project's output quality as $S(f) = 2\phi(f(P)) + \phi(f(T))$. The top-k subgraph matching outlined in Fig. 1 can be used to identify high-quality candidate personnel assignments based on these criteria. The top-3 subgraph matches of this example are $f_1 = \{(M, M_1), (A, A_1), (P, P_1), (T, T_1)\}$, $f_2 = \{(M, M_1), (A, A_1), (P, P_1), (T, T_2)\}$, $f_3 = \{(M, M_1), (A, A_1), (P, P_2), (T, T_4)\}$, and the corresponding match scores are 21, 20 and 14.

Top-k subgraph matching is challenging mainly due to the following two factors:

CIKM '24, October 21–25, 2024, Boise, ID, USA.

ACM ISBN 979-8-4007-0436-9/24/10
https://doi.org/10.1145/3627673.3679790

- The search space is vast because of the problem's NP-hardness [25]. However, the query parameter k is typically smaller than the total number of subgraph matches. A naive approach, which explores the entire search space to obtain all subgraph matching results and then sorts them to get the top-k, is inefficient as it examines too much unpromising space and generates too many redundant matches.
- Efficient pruning is difficult. Identifying sub-spaces of the search space that can be safely pruned requires knowledge of the overall top-k scores and the score range of the matches that these sub-spaces can produce, which are difficult to obtain without complete exploration. Meanwhile, the pruning methods need to be efficient, incurring as little overhead as possible.

To address the two challenges above, we propose a novel *hop index* that stores precomputed property value bounds for each data node, from which we can efficiently infer topology-aware score bounds of the sub-spaces in the search space spawned from any top-k subgraph matching query. With the help of the hop index, **PTAB**, our top-k subgraph matching algorithm, can identify the promising sub-spaces to explore first and safely prune unpromising sub-spaces. Moreover, we propose a root node selection optimization for acyclic queries and extend our method to general queries by an edge-cut strategy. With these techniques, PTAB is able to excessively prune the search space (Sec. 8.2.2) and achieve significant speedup in terms of end-to-end query time compared with state-of-the-art top-k subgraph matching methods (Sec. 8.2.1).

To sum up, we make the following contributions in this paper:

- **Hop-index-based topology-aware bounding.** We design a novel *hop index*, which stores for each node the property value bounds in its constrained multi-hop neighborhood. Based on the hop index, we propose an efficient sub-space score bound estimation method that requires little exploration. The estimated score bounds are topology-aware due to the hop index's structure, leading to improved tightness.
- **Pruned PTAB exploration.** We propose a search framework, PTAB, which gauges how promising each sub-space of the search space is with the hop-index-based bounds and guides the search accordingly.
- **Cost-based root selection.** We propose a novel cost-based strategy for selecting the root of tree queries to search from, which minimizes the exploration cost.
- **Edge-cut strategy for general query graphs.** We extend our method to general query graphs with cycles by an edge-cut strategy that cuts low-selectivity edges from the query graph to transform it into a tree, largely preserving the query graph structure to expose pruning opportunities.
- **Extensive experiments on real and synthetic datasets.** We conduct experiments on several real-world and synthetic datasets. Results show that our method outperforms existing methods.

2 Related work

2.1 Subgraph Matching

The existing subgraph matching algorithms can be classified into two main categories [21]: search-based methods and join-based methods. The search-based methods conduct backtracking search in the search space consisting of the Cartesian product of candidate data nodes of the query nodes. The first search-based method for subgraph matching is the Ullmann algorithm [26], while the subsequent methods [2, 4, 9, 13, 17, 22, 23] improve the performance by pruning the search space, using a better search node order, reducing redundant computation, etc. The join-based methods [1, 18, 28] convert the subgraph matching problem to a multi-way join problem in which relations correspond to edges in the query graph.

Subgraph matching methods can be used to answer top-k subgraph matching queries in a naive sort-after-match framework, which tends to be inefficient since they need to explore the whole search space to retrieve all the matches, compute the scores, and sort the matches accordingly.

2.2 Top-k Queries in Relational Databases

Many classical algorithms have been proposed for relational top-k queries, including TA [10], NRA [10], and rank-join [15]. Most top-k algorithms for relational databases cannot be applied to the top-k subgraph matching problem due to the differences in the underlying data structures. However, Take2 [24], the state-of-the-art algorithm with theoretical optimal time complexity for top-k full conjunctive queries in relational databases, models the problem as finding a min-cost path using dynamic programming, which is closest to the graph setting. Therefore, we adapted Take2 [24] for top-k subgraph matching and compared our method with it in our experiments.

2.3 Top-k Subgraph Matching

Top-k subgraph matching with tree subgraph patterns has been studied extensively. Tree patterns, which are also called twigs or acyclic patterns in the existing literature, refer to patterns that do not contain cycles. The search space of top-k subgraph matching can be seen as the Cartesian product of the candidate data node sets of the query nodes. Existing methods partition the search space into several sub-spaces according to some criteria and probe the sub-spaces for the top-k matches. Partitioning methods are either *node-centric*, which span a forest of subgraph matches from the candidate data nodes of a selected root query node, where each tree in the forest constitutes a sub-space [7, 8, 11, 27, 29]; or *subgraph-centric*, which divide the data graph into several subgraphs possibly with overlap, ensuring that each match of the query graph is in a single subgraph [31].

Some methods additionally use precomputed information to order the sub-spaces by how promising they are in terms of producing the top-k acyclic subgraph matches. For example, [27] orders the sub-spaces simply by their root candidate data node's score. Other works compute the score upper bounds of each sub-space and then explore the sub-space with a high upper bound first. Specifically, [31] computes the score upper bound of a sub-space as the summation of the maximum scores of each node candidate, which ignores the topological constraints imposed by the subgraph pattern, leading to a loose bound.

In this paper, we propose to tighten the sub-spaces' score upper bounds by accounting for the subgraph pattern's topology using the hop index (Sec. 6). In fact, our upper bound estimation method is independent of the partitioning scheme and search strategy. Based on this novel index-based upper bound estimation, we propose a

top-down search framework called PTAB with node-centric search space partitioning, which significantly outperforms [11] and [7] experimentally.

As for top-k subgraph matching with general subgraph patterns that contain cycles, existing works decompose these patterns into several acyclic subpatterns. They then either rank-join these subpatterns [29], or alternate among subpatterns to expand their results in the descending order of scores to retrieve the top-ranking matches of the original patterns [8]. Instead of decomposing the query graph, we propose an edge-cut strategy (Sec. 5.3), which largely preserves the original query graph structure to expose more pruning opportunities and adopts a lightweight checking mechanism to retrieve matches of the original patterns.

3 Preliminary

We will introduce the problem definitions and necessary notations in this section. Tab. 1 summarizes the notations used in this paper.

For ease of presentation, we focus on undirected labeled graphs in this paper. The extension of our method to directed graphs is straightforward and will be discussed in Sec. 7.2.

Definition 3.1 ((Data) Graph). A (data) graph G is an undirected labeled graph represented as a quadruple (V, E, L, ϕ), where V is a finite set of nodes, $E \subseteq V \times V$ is the edge set, L is a function mapping every node v or edge (u, v) to its label $L(v)$ or $L((u, v))$, and $\phi : V \rightarrow \mathbb{R}$ is a function mapping every node v to its property value.

Without causing ambiguity, the graph is interchangeably denoted as $G(V, E)$ or $G(V, E, L)$. For $v \in V$ and a positive integer h, we use $N_h(v)$ to denote the set of nodes within a distance no greater than h to the node v (including v), i.e., $N_h(v) = \{u | d(u, v) \leq h\}$. We omit the digit 1 for 1-hop neighbors: $N(v) = N_1(v)$. Additionally, we represent the h-hop neighbors with a specific label l of the node v as $N_h(v, l) = \{u | u \in N_h(v) \text{ and } L(u) = l\}$. The label set of node v's neighbors is represented by $L(N_h(v)) = \{L(u) | u \in N_h(v)\}$.

Definition 3.2 (Query Graph). A query graph is an undirected labeled graph $Q(V_Q, E_Q, L_Q)$, where V_Q and E_Q restrict the topological structure of the matched subgraph, and L_Q specifies the labels of query nodes and edges.

Definition 3.3 (Cyclic Query / Tree Query (Acyclic Query)). A query is cyclic if its query graph contains at least one cycle; otherwise, it is a tree query, also called an acyclic query.

Definition 3.4 (Subgraph Matching). Given a data graph $G(V_G, E_G, L_G)$ and a query graph $Q(V_Q, E_Q, L_Q)$, a subgraph matching is a function $f: V_Q \rightarrow V_G$, which satisfies the following conditions:

- Vertex constraint. $\forall u \in V_Q, L_Q(u) = L_G(f(u))$.
- Edge constraint. $\forall e = (u_1, u_2) \in E_Q$, there is also an edge $(f(u_1), f(u_2))$ in E_G with $L_Q((u_1, u_2)) = L_G((f(u_1), f(u_2)))$.

The mapped subgraph in the data graph is called a subgraph match.

We use the semantics of *subgraph homomorphism* in this work. Note that our method can be easily extended to top-k subgraph isomorphism by checking whether two query nodes are mapped to the same data node in each matching.

Score functions reflect the importance or user preference of a subgraph match, defined as follows:

Table 1: Notations

Notation	Description
G, Q	data graph and query graph
$N_h(u, l)$	u's l-labeled h-hop neighbors
$L(N(u))$	label set of u's neighbors
$S(f)$	score of a matching f
$S_u(f(u))$	node score about the query node u of $f(u)$

Definition 3.5 (Score Function). Given a query Q and a subgraph matching f of it on the data graph, a score function S is a linear combination of the matched nodes' property values[1]:

$$S(f) = \sum_{u \in V_Q} S_u(f(u)) = \sum_{u \in V_Q} c_u \times \phi(f(u)), \tag{1}$$

where $\{c_u | u \in V_Q\}$ is the score coefficient set, and $S_u(f(u)) = c_u \times \phi(f(u))$ is the node score function.

Example. In Fig. 1, $\phi(v)$ represents the competency of each company employee v. The matching score function is $S(f) = S_P(f(P)) + S_T(f(T))$, where $S_P(f(P)) = 2\phi(f(P))$ and $S_T(f(T)) = \phi(f(T))$ are the node score functions for the query nodes P and T, and $c_P = 2$ and $c_T = 1$ are the score coefficients.

Notably, our methods can be generalized to cases where each node has multiple properties and the score function is defined by multiple property values (Sec. 7.2).

Now we give the definition of the research problem, top-k subgraph matching, as below:

Definition 3.6 (Top-k Subgraph Matching). Given a data graph G, a query graph Q, and a score function S, a top-k subgraph matching problem (G, Q, S) is to find a set M of at most k subgraph matches ($|M| \leq k$) such that there does not exist a subgraph matching $f' \notin M$ which has a higher score than some matches in M, i.e., $\nexists f \in M, S(f') \geq S(f)$.

4 Overview

As illustrated in Fig. 2, our top-k subgraph matching method consists of two components: (1) offline hop index construction (dashed arrows) and (2) online enumeration (solid arrows).

Given a data graph, a hop index is constructed offline, storing property ranges of each node's neighborhood within a certain number of hops (Sec. 6). Then, in the online phase, PTAB traverses the data graph to retrieve the top-k results (Sec. 5). During the traversal, PTAB leverages the hop index to estimate the score upper bound for each sub-space of the search space (Sec. 6.3), which aids in identifying the promising sub-spaces for exploration and pruning the unpromising ones. Additionally, the root selection strategy (Sec. 7.1) minimizes the exploration cost starting from the selected root. We also introduce a novel heuristic-based edge-cut strategy for extending our methods from tree query graphs to general query graphs (Sec. 5.3).

[1]However, our methods can also fit score functions that are monotone on each node property, such as the monotone ranking function [16]. Formally, a function F is monotone if $F(x_1, \cdots, x_n) \leq F(x'_1, \cdots, x'_n)$ whenever $x_i \leq x'_i$ for every i.

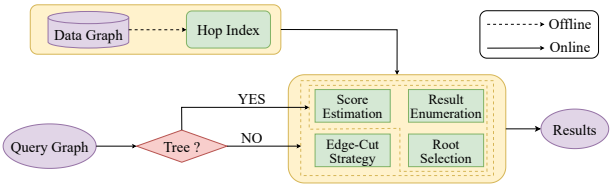


Figure 2: Overview of PTAB.

5 PTAB

In this section, we first introduce our top-k subgraph matching framework, PTAB for tree queries, which is efficient due to pruning with the sub-spaces' upper bounds obtained from the hop index (Sec. 6). Then we extend it to general queries using a novel edge-cut strategy in Sec. 5.3.

5.1 Key Abstraction: Search Trees

In the subsequent sections, we will rely on *search trees* as a key abstraction for illustrating PTAB. Intuitively, a search tree represents the sub-space spanning from a data node that can be matched to the root query node, assuming the query graph is a tree query. (We will introduce how to handle general query graphs in 5.3, which is based on the framework for tree query graphs.)

Relevant notations. Given a tree query graph Q, we denote its root node chosen by the root selection strategy (Sec. 7.1) as u_r. For every query node $u \in V_Q$, we denote the subquery rooted at u as Q_u, and the tree height, i.e., the maximum path length from u to a leaf of Q_u, as $height(Q_u)$. A query node u's candidate node set is denoted as $C(u)$, which can be any set containing each node $v \in V_G$ such that exists a subgraph matching f with $f(u) = v$.

5.1.1 Search Tree Exploration. Given a query Q_u, any top-k subgraph matching method conceptually explores a *search forest* to identify the top-k results, in which each *search tree* is rooted at a data node $v \in C(u)$, denoted as G_v^u. For example, in Fig. 3, the search trees $G_{M_1}^M$, $G_{M_2}^M$, $G_{M_3}^M$ collectively form a search forest of the top-k subgraph matching query in Fig. 1. Each search tree is a sub-space of the entire search space, constructed in a top-down fashion starting from the root data node v, expanding a query edge (u_1, u_2) at each step, adding each $v_2 \in C(u_2)$ as a child node of each $v_1 \in C(u_1)$, where $C(u_2)$ can be pruned based on v_1, v_1's ancestor nodes, u_2's node label, and (u_1, u_2)'s edge label. A subgraph matching that leads to a match in G_v^u is denoted as f_v. Note that the exploration does not involve explicitly materializing the search trees in most cases, and is usually implemented as some variation of depth-first or breadth-first search.

5.1.2 Search Tree Pruning. The search forest of a top-k subgraph matching query can be huge on large data graphs due to enormous candidate node set sizes. It is clearly wasteful to explore the entire search space for answering top-k subgraph matching queries, since the total number of subgraph matches is often much greater than k. There are two ways to prune the search space. Firstly, we can identify more promising search trees to explore first, which are more likely to produce the top-k results. Secondly, we can avoid exploring the search trees that cannot produce the top-k results. An effective tool that enables both pruning approaches is the estimate of a search tree's score upper bound, i.e., an upper bound on the scores of the subgraph matches that this search tree can produce, as shown in the following example.

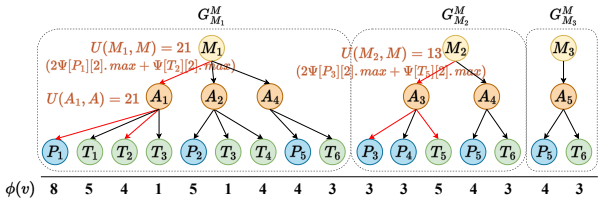


Figure 3: The search forest of the top-k subgraph matching query in Fig. 1. The upper bound estimation method will be introduced in Sec. 6.3.

Example. With the search forest in Fig. 3, suppose $k = 3$ and we know the score upper bounds of the query trees $G_{M_1}^M$ and $G_{M_2}^M$ are 21 and 13, respectively. (The bound for $G_{M_1}^M$ happens to be tight, i.e., equal to the score of the top-1 matching score from it.) We can identify $G_{M_1}^M$ as the more promising search tree due to its higher score upper bound and traverse it first to get results. The third largest score in $G_{M_1}^M$ is 14, of the matching $\{(M, M_1), (A, A_2), (P, P_2), (T, T_4)\}$. Since $k = 3$ and $G_{M_2}^M$'s score upper bound is not greater than 14, we can avoid traversing $G_{M_2}^M$ altogether.

Clearly, the tighter these upper bounds are, the more effective the pruning will be. However, it is challenging to obtain tight score upper bounds of the search trees, since it is necessary to account for both label and topology information, which no existing work has considered simultaneously. In addition, we would also like the upper bound estimation method to incur as low an overhead as possible. We will discuss our solution using the hop-index-based estimation technique, which efficiently produces topology-aware bounds, in Sec. 6.

5.2 PTAB for Tree Queries

Supposing that we have access to any search tree's score upper bound, Alg. 1 describes the PTAB search framework for tree queries that traverses the promising search trees first and avoids the unpromising ones using these bounds.

To implement search space pruning, we use the following two data structures:

- `Pool`: `Pool` is a max-heap maintained for a tree query, containing elements in the form of $(v, f_v, \widetilde{S}(v, u_r))$, where u_r is the root query node, v is a candidate data node in $C(u_r)$, and $\widetilde{S}(v, u_r)$ is a score estimate of the search tree $G_v^{u_r}$, which is either its score upper bound estimated as in Sec. 6.3 or the largest score among its currently remaining subgraph matches. `Pool`, sorted by $\widetilde{S}$, guides the exploration of promising search trees by always popping the search tree with the highest score estimate (line 7). Unpromising search trees are pruned from `Pool` (line 5).
- `Queue`: A `Queue` is a max-heap maintained for each (u, v) pair, denoted as `Queue`$[u][v]$, where u is a query node and $v \in C(u)$. Each `Queue` contains elements in the form of $(f_v, \widetilde{S}(v, u_r))$ and serves as an intermediate data structure for maintaining `Pool`.

The algorithm first selects a node u_r as the root of the query Q and generates a node candidate set $C(u_r)$ for it using LDF and NFL [21] (line 1; details in Sec. 7.1). For each candidate node v of

the root node u_r, PTAB computes the score upper bound of the corresponding search tree $G_v^{u_r}$ (lines 2-3; details in Sec. 6.3). We then construct Pool, which is the key step towards search space pruning (lines 4-5, Sec. 5.2.1 and Sec. 5.2.2). In the subsequent loop, we pop elements from Pool and explore the respective search tree until the top-k results are retrieved (line 6). If the popped matching f_v has a score equal to the score estimate $\widetilde{S}(v, u_r)$, it is the current top-1 among the remaining matches in this search tree, and since this search tree is at the top of Pool, it is also the current top-1 in the remaining search space. Therefore, we put it in the result set M and save the next largest-score matching in this search tree in Pool and the respective Queue for further inspection (lines 9-14). Otherwise, we compute the top-1 match in this search tree and save it (lines 16-23). In both cases, we invoke GetNextMatching, a recursive procedure that gets the matching with the next largest score in a search tree, which can be implemented by the lattice search [11]. Specifically, when this procedure is invoked with an empty matching, it retrieves the top-1 result (line 18).

5.2.1 Identifying Promising Search Trees. PTAB identifies promising search trees as those with large score estimates, since these search trees are likely to produce at least one top-k result. After computing the score upper bounds of each search tree (lines 2-3), they are added into Pool, excluding those that are pruned (line 5). The max-heap Pool always pops the search tree with the largest score estimate for exploration (line 7), which is expected to be the most promising. To continuously reflect how promising each search tree is as the search progresses, each iteration of the loop in Alg. 1 refines the score estimate of the explored search tree in Pool to be the current top-1 score among its remaining matches (lines 14 and 23).

5.2.2 Pruning Unpromising Search Trees. We can avoid exploring an entire search tree when its score upper bound is lower than the lowest score among the top-k matches. Though we cannot know this lowest score before Alg. 1 completes, we can estimate a lower bound during the upper bound computation, assuming $|C(u_r)| \geq k$, which usually holds on large real-world graphs. Since the upper bound computation procedure examines at least $|C(u_r)|$ subgraph matches (Sec. 6.3), the k-th largest matching score among them is a lower bound on the k-th largest matching score across the search space, denoted as L (line 4). Then, if a search tree's score upper bound is lower than L, we avoid adding it into Pool (line 5), thus ensuring that it is pruned from the subsequent exploration. Note that this pruning technique only works for tree queries, as the edge-cut extension to general queries (Sec. 5.3) requires an *any-k* algorithm, which continuously emits top results until explicitly told to stop, while this pruning technique only ensures the top-k results.

5.3 Edge-Cut for General Queries

So far, we have only introduced PTAB for tree query graphs. In this section, we introduce an edge-cut technique to extend PTAB to general cases when the query graph Q has cycles, where $|E_Q| \geq |V_Q|$. The technique temporarily removes $|E_Q| - |V_Q| + 1$ edges from Q so that the remaining query graph becomes a tree, denoted as Q_{tree}, which can thus be processed by Alg. 1.

The extended version of PTAB is shown in Alg. 2. We first cut the selected edges to construct Q_{tree} (lines 1-5) according to Eqn. 2, which we will elaborate on later in this section. Then, we continuously invoke Alg. 1 on Q_{tree} (line 7) and check whether the subgraph matching obtained is also a matching of Q (lines 8-9), only adding it to the result set if so, until we get k results.

Note that we formulated Alg. 1 as a top-k algorithm that takes the required result size k as an input, while in line 7 of Alg. 2, we need an *any-k* algorithm that can produce an indeterminable number of matches in descending order of scores. To this end, we designed Alg. 1 so that its framework does not depend on the input parameter k, so turning it into an *any-k* algorithm only requires modifying the loop's stopping condition (line 6, Alg. 1) and removing the lower-bound-based pruning (setting $L = -\infty$ in line 4, Alg. 1).

Algorithm 1: PTAB for Tree Query Graphs

Input: Data graph G; tree query Q; score function S; result size k.
Output: The top-k subgraph matching results M, where $|M| \leq k$.

1 Select root node u_r; generate candidate node set $C(u_r)$; Queue$\leftarrow \emptyset$
2 **foreach** $v \in C(u_r)$ **do**
3 $\quad f_v \leftarrow$ ComputeUpperBound$(v, u_r, G_v^{u_r}, Q_{u_r}$, Queue$)$
4 $L \leftarrow$ the k-th largest $S(f_v)$ in the above loop // Lower bound
5 Pool$\leftarrow \bigcup_{v \in C(u_r)} \{(v, f_v, U(v, u_r)) \mid U(v, u_r) >= L\}$
6 **while** $|M| \leq k$ *and* Pool *is not empty* **do**
7 $\quad (v, f_v, \widetilde{S}(v, u_r)) \leftarrow$ Pool.pop()
8 $\quad$ Queue$[u_r][v]$.pop()
9 $\quad$ **if** $S(f_v) = \widetilde{S}(v, u_r)$ **then**
10 $\quad\quad M \leftarrow M \cup \{f_v\}$
11 $\quad\quad$ **if** $G_v^{u_r}$ *is not fully searched* **then**
12 $\quad\quad\quad f'_v \leftarrow$GetNextMatching$(G_v^{u_r}, f_v)$
13 $\quad\quad\quad$ Queue$[u_r][v]$.push$((f'_v, S(f'_v)))$
14 $\quad\quad\quad$ Pool.push$((v, f'_v, S(f'_v)))$
15 $\quad$ **else**
16 $\quad\quad$ **foreach** $u_c \in u_r$.children **do**
17 $\quad\quad\quad$ **foreach** $x \in C(u_c)$ **do**
18 $\quad\quad\quad\quad f_x \leftarrow$ GetNextMatching$(G_x^{u_c}, \emptyset)$
19 $\quad\quad\quad\quad$ Queue$[u_c][x]$.push$((f_x, S(f_x))$
20 $\quad\quad\quad f^{u_c} \leftarrow f_{\text{argmax}_{x \in C(u_c)} S(f_x)}$
21 $\quad\quad f'_v \leftarrow \bigcup_{u_c \in u_r.\text{children}} f^{u_c} \cup \{(u, v)\}$ // Top-1 in $G_v^{u_r}$
22 $\quad\quad$ Queue$[u_r][v]$.push$((f'_v, S(f'_v)))$
23 $\quad\quad$ Pool.push$((v, f'_v, S(f'_v)))$
24 **Function** ComputeUpperBound$(v, u, G_v^u, Q_u$, Queue$)$:
25 $\quad$ **if** Queue$[u][v]$ *is not empty* **then**
26 $\quad\quad$ **return**
27 $\quad$ **if** $height(Q_u) \leq H$ **then**
28 $\quad\quad f_v \leftarrow$ Get a match of Q_u
29 $\quad\quad U(v, u) \leftarrow$ EstimateScore(f_v, Ψ) // Eqn. 7
30 $\quad$ **else**
31 $\quad\quad f_v = \{(u, v)\}; U(v, u) \leftarrow S_u(v)$
32 $\quad\quad$ **foreach** $u_c \in u$.children **do**
33 $\quad\quad\quad$ **foreach** $v_c \in C(u_c)$ **do**
34 $\quad\quad\quad\quad$ ComputeUpperBound$(v_c, u_c, G_{v_c}^{u_c}$, Queue$)$
35 $\quad\quad\quad (f_{v_c}, U(v_c, u_c)) \leftarrow$ argmax$_{v_c \in C(u_c)}$get<1>(Queue$[u_c][v_c]$.top())
36 $\quad\quad\quad f_v \leftarrow f_v \cup f_{v_c}$
37 $\quad\quad\quad U(v, u) \leftarrow U(v, u) + U(v_c, u_c)$ // Eqn. 10
38 $\quad$ Queue$[u][v]$.push$((f_v, U(v, u)))$
39 $\quad$ **return** f_v
40 **Function** GetNextMatching(G_v^u, f_v):
41 $\quad$ **if** u *is a leaf query node* **then**
42 $\quad\quad$ Get v' from $C(u)$ s.t. v' has node score only lower than v
43 $\quad\quad$ **return** $\{(u, v')\}$
44 $\quad$ **foreach** $u_c \in u$.children **do**
45 $\quad\quad f_{fv(u_c)} \leftarrow$GetNextMatching$(G_{fv(u_c)}^{u_c}, f_v|_{Q_{u_c}})$
46 $\quad\quad$ Queue$[u_c][f_v(u_c)]$.push$((f_{fv(u_c)}, S(f_{fv(u_c)})))$
47 $\quad$ Get matching f'_v of Q_u by lattice search with scores just below $S(f_v)$
48 $\quad$ **return** f'_v

Algorithm 2: PTAB for Cyclic Query Graphs

Input: Data graph G; cyclic query graph Q; positive integer k.
Output: The top-k subgraph matching results M, where $|M| \leq k$.

```
Initialize an empty set E_cut storing the cut edges
while there is a cycle O in the Q do
    select an edge e from O according to Eqn. 2
    E_cut ← E_cut ∪ {e}
Cut edges E_cut from Q to form an acyclic query Q_tree
while |M| ≤ k and the search space of Q_tree is not completely traversed do
    m ← generate a result for Q_tree in descending order by score
    if m is also a subgraph matching of Q then
        M ← M ∪ {m}
```

Since the matching score is defined on the nodes rather than edges, this edge-cut strategy preserves the rank order of the enumerated results, ensuring that Alg. 2 returns the correct top-k results of the original cyclic query graph.

The only problem left is how to choose which edges to cut. We hope that the cut edges have low selectivity because this makes it more likely that the match of Q_{tree} is also a match of Q, thus reducing the number of redundant calls to Alg. 1 and checks. Therefore, for each cycle O in the query graph, we cut the following edge:

$$e_{cut} = \underset{(u,v)\in O}{\operatorname{argmax}} \frac{M(L(u), L(v))}{M(L(u)) \times M(L(v))}, \tag{2}$$

where $M(L(u), L(v))$ represents the number of edges in the data graph that links a node with label $L(u)$ with a node with label $L(v)$, and $M(L(u))$ represents the number of data nodes which have the same label as u.

Note that the existing decomposition-then-rank-join methods for cyclic queries only consider the features of the query graphs and generate the same decomposition for different data graphs with the same query graph. However, our edge-cut strategy considers features of both the query graph and the data graph as shown in Eqn. 2. Moreover, our edge-cut strategy preserves more of the original query graph structure than existing decomposition methods, such as [29] which decomposes the query graph into stars, to expose pruning opportunities.

6 Hop Index

To efficiently obtain a search tree's score upper bound, we propose the hop index, which accounts for both the label information and the topology of the query pattern. In this section, we first introduce the structure (Sec. 6.1) and construction process (Sec. 6.2) of the hop index. Subsequently, in Sec. 6.3, we explore how to use the hop index to estimate the topology-aware score upper bound of a search tree.

6.1 Structure

Given a data graph $G(V, E, L, \phi)$, the hop index stores the following entry for each node v:

$$\Psi = \{(\Psi[v][i].min, \Psi[v][i].max) \mid i = 1, \cdots, H\}, \tag{3}$$

where $\Psi[v][i].min$ ($\Psi[v][i].max$) is the minimum (maximum) property value of v's $2i$-hop neighbors with the same label, i.e.,

$$\Psi[v][i].min = \min_{u\in N_{2i}(v,\, L(v))} \phi(u), \tag{4}$$

$$\Psi[v][i].max = \max_{u\in N_{2i}(v,\, L(v))} \phi(u), \tag{5}$$

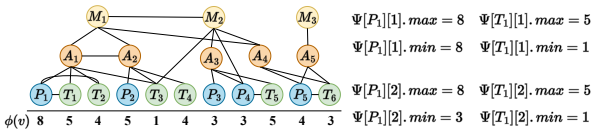


Figure 4: The hop index of the data graph in Fig. 1(b).

where H is an adjustable parameter that controls the range of the hop index built for each node.

Example. Fig. 4 shows the hop index of nodes P_1 and T_1 from the data graph of Fig. 1(b). For example, $\Psi[P_1][2].min = 3$ is derived from P_1's 4-hop neighbor P_4, and $\Psi[T_1][2].max = 5$ because the maximum ϕ value of T_1's 4-hop neighbors is $\phi(T_1)$ itself.

Space Usage. The space usage of the hop index is bounded by $O(H|V|)$. Due to the small-world phenomenon prevalent in real graphs [19], the parameter H, which is always smaller than or equal to the query graph's diameter, is usually much smaller than $|V|$. Thus, the space taken up by the hop index can be treated as $O(|V|)$.

6.2 Construction

The hop index can be constructed via a message-passing mechanism, where each node collects information from its neighbors and aggregates it iteratively until all the necessary information is acquired. Alg. 3 describes this message-passing-based hop index construction process. To streamline calculations, we create an auxiliary data structure called the *pivot hop index* Ψ': $\Psi'[v][l][i]$ stores the maximum and minimum property values of v's i-hop neighbors with label l, which is an extension of the hop index Ψ:

$$\Psi[v][i] = \Psi'[v][L(v)][2i], \text{for } i = 1, 2, \cdots, H. \tag{6}$$

In Alg. 3, lines 1-4 initialize Ψ' for one hop, which simply consists of the maximum and minimum property values among each node and their neighbors. Then the algorithm calculates the remaining $(2H-1)$-hop Ψ' in a message-passing fashion (lines 5-9), as a node's j-hop Ψ' entry can be obtained by comparing the $(j-1)$-hop Ψ' entries of its neighbors and selecting the maximum and minimum among them. Then, the even-order information is retained as the hop index in lines 10-13 (for simplicity, we abbreviate $(\Psi[v][i].max, \Psi[v][i].min)$ as $\Psi[v][i]$ in the pseudocode).

Algorithm 3: Hop Index Construction Algorithm

Input: Data graph G; hop index parameter H.
Output: The hop index Ψ of graph G.

```
1  foreach node v ∈ V_G do
2      foreach label l ∈ L(N(v)) do
3          Ψ'[v][l][1].max = max_{u∈N(v,l)} φ(u)
4          Ψ'[v][l][1].min = min_{u∈N(v,l)} φ(u)
5  for i from 2 to 2H do
6      foreach node v ∈ V_G do
7          foreach label l ∈ L(N_{i-1}(v)) do
8              Ψ'[v][l][i].max = max_{u∈N(v)} Ψ'[u][l][i-1].max
9              Ψ'[v][l][i].min = min_{u∈N(v)} Ψ'[u][l][i-1].min
10 for i from 1 to H do
11     foreach node v ∈ V_G do
12         Ψ[v][i] = Ψ'[v][L(v)][2i]
13         Ψ[v][i] = Ψ'[v][L(v)][2i]
```

Time Complexity. In each iteration, every node gathers the Ψ' entries from its neighbors, which costs $O(|E|)$ for the entire graph.

Hence, the overall time complexity amounts to $O(H|E|)$. As in Sec. 6.1, since $H << |E|$ holds in most cases, the time complexity can be treated as $O(|E|)$.

6.3 Hop-Index-Based Score Upper Bounding

We leverage the hop index to estimate each search tree's score upper bounds, as shown in the `ComputeUpperBound` subroutine in Alg. 1. Such a bounding procedure is efficient because it touches on a very small portion of the search tree, and is sufficiently tight because the hop index carries both label and topology information.

When the search tree's height is within the hop index parameter H, we can estimate its score upper bound by the following lemma, which involves getting an arbitrary matching from the search tree (lines 27-29, Alg. 1):

Lemma 6.1. *Given a tree query Q_u with height no greater than the hop index parameter H and a subgraph matching f_v from the search space G_v^u, then for any matching f'_v from G_v^u, we have $S(f'_v) \leq U(v,u)$, where the upper bound $U(v,u)$ is given by:*

$$U(v,u) = \sum_{w \in V_{Q_u}} B^w_{f_v(w)}(d_u(w)), \quad if\, height(Q_u) \leq H, \tag{7}$$

where $d_u(w)$ is the depth of w in the query tree Q_u, i.e., the length of the path from the root u to w, and $B^w_{f_v(w)}(d_u(w))$ represents the upper score bound of w's matched data node in G_v^u:

$$B^w_{f_v(w)}(d_u(w)) = \begin{cases} c_w \times \Psi[f_v(w)][d_u(w)].max & if\, c_w > 0 \\ c_w \times \Psi[f_v(w)][d_u(w)].min & if\, c_w \leq 0 \end{cases} \tag{8}$$

Proof. For any matching f'_v from G_v^u, suppose that the following equation is true:

$$S_w(f'_v(w)) \leq B^w_{f_v(w)}(d_u(w)), \tag{$*$}$$

then the lemma's correctness is given by

$$S(f'_v) = \sum_{w \in V_{Q_u}} S_w(f'_v(w)) \leq \sum_{w \in V_{Q_u}} B^w_{f_v(w)} = U(v,u). \tag{9}$$

Next, we only need to prove Eqn. ($*$). By the search tree's construction procedure, both $f_v(w)$ and $f'_v(w)$ link data node v by paths with length $d_u(w)$, thus the distance between $f_v(w)$ and $f'_v(w)$ in the data graph is not larger than $2d_u(w)$. Due to the definition of the hop index (Sec. 6.1), the property value of $f'_v(w)$ is thus bounded: $\Psi[f_v(w)][d_u(w)].min \leq \phi(f'_v(w)) \leq \Psi[f_v(w)][d_u(w)].max$. As $S_w(f'_v(w)) = c_w \times \phi(f'_v(w))$, Eqn. ($*$) holds. □

When the tree query Q_u's height exceeds H, the hop index can still help estimate any search tree G_v^u's score upper bound by recursively exploring each of u's child nodes and summing v's score with the maximum estimate among the search trees rooted at each child nodes' candidates (lines 30-37, Alg. 1):

$$U(v,u) = S_u(v) + \sum_{u_c \in u.children} \max_{v_c \in C(u_c)} U(v_c, u_c), \quad \text{if } height(Q_u) > H. \tag{10}$$

Example. In Fig. 1's query, with M chosen as the root, the query tree's height is 2. As in Fig. 4, the hop index is constructed with $H = 2$, so the upper bound of any search tree rooted at a candidate of M can be estimated without exploring its child nodes. Specifically, in the search forest shown in Fig. 1, upon identifying a match $f_{M_1} = \{(M, M_1), (A, A_1), (P, P_1), (T, T_2)\}$ in the search tree $G^M_{M_1}$, we can get a score upper bound for $G^M_{M_1}$: $U(M_1, M) = 2 \times \Psi[P_1][2].max + \Psi[T_2][2].max = 2 \times 8 + 5 = 21$. This upper bound is tight, as it is exactly equal to $S(\{(M, M_1), (A, A_1), (P, P_1), (T, T_1)\})$, the top-1 match in $G^M_{M_1}$.

Our bounding procedure produces score upper bounds that are both label- and topology-aware. Intuitively, the score upper bound is tightest when the minimum or maximum property value recorded in a data node's hop index entry corresponds exactly to its same-label sibling (or itself) with the minimum or maximum property value in the search tree. Such is the case in the above example, where $\Psi[P_1][2] = \phi(P_1)$ (itself) and $\Psi[T_2][2] = \phi(T_1)$ (sibling).

For more efficient bounding, we prevent the same search tree from repeated exploration by checking whether the auxiliary data structure, `Queue`, is empty (lines 25-26, Alg. 1).

7 Optimizations And Extensions

7.1 Root Node Selection

Though every node in a tree query graph can be the root, previous works on top-k subgraph matching with tree patterns assume a fixed root node of the query graph [7, 11, 24]. However, the root node selection affects the tree height, the candidate set size, and the root score coefficient—all of which have a great impact on the query efficiency, detailed as follows:

- The smaller the height of the tree, the fewer nodes need to be fully expanded, since the hop index can provide upper bounds for any tree with a height that is smaller than H (Eqn. 7).
- The smaller the candidate set $C(u_r)$ of the root node, the smaller the number of search trees, thus the fewer the invocations to the score upper bounding procedure.
- The larger the absolute value of the root node's score coefficient $|c_{u_r}|$, the greater the difference in scores matched at the root node, so the more likely it is for the search trees' score upper bounds to vary greatly, increasing the chances for pruning.

Therefore, we heuristically select the root node for the tree query Q as follows:

$$u_r = \operatorname*{argmin}_{u \in V_Q} \frac{H(Q_u) \times |C(u)|}{|c_u| + 1}. \tag{11}$$

We employ the LDF and NLF [21] to generate a candidate set $C(u)$ for each node u to compute $|C(u)|$ and facilitate the subsequent search. Label and degree filtering (LDF) is to find $C(u) = \{v \mid v \in G, L_G(v) = L_Q(u), d_G(v) \geq d_Q(u)\}$. Neighbor label frequency filtering (NLF) is to find $C(u) = \{v \mid v \in G, |N_G(v,l)| \geq |N_Q(u,l)|$ for all $l \in L_Q(N_Q(u))\}$. These two filtering methods can be done in $O(V_G)$ time, ensuring efficiency in the process.

7.2 Extension to Other Graph Models

In the previous sections, we assume that each node v in the graph only has a single property value $\phi(v)$ for the simplicity of explication and convenience of notation. However, our methods are readily adaptable to graphs with multiple properties per node.

Consider that each node v is associated with a property set $\mathcal{A}_v$, where the value of each property $p \in \mathcal{A}_v$ is denoted by $v.p$. Accordingly, the node score function can be defined as a linear combination of these property values:

$$S_u(v) = \sum_{p \in \mathcal{A}_u} c_u^p \times v.p, \tag{12}$$

where $\{c_u^p \mid p \in \mathcal{A}_u\}$ is the coefficient set for the query node u.

To support top-k subgraph matching on graphs with multiple-property nodes, two primary modifications are required as follows:

(1) *Property-specific hop index.* The hop index should be constructed for each property individually. Specifically, we need to maintain $\Psi[v][p][i]$ for each $p \in \mathcal{A}_v$, which stores the extreme values of the property p within v's $2i$-hop neighborhood with the same label as v.

(2) *Multi-property upper score bounds.* The node upper score bound for a node w in query tree Q_u, previously formulated in Eqn. 8, needs to be modified for multiple properties as follows:

$$\begin{aligned} & B^w_{f_v(w)}(d_u(w)) \\ = \; & \textstyle\sum_{p \in \mathcal{A}_w} c_w^p \times \mathbb{1}_{c_w^o > 0} \Psi[f_v(w)][p][d_u(w)].max \\ & + \textstyle\sum_{p \in \mathcal{A}_w} c_w^p \times \mathbb{1}_{c_w^p \le 0} \Psi[f_v(w)][p][d_u(w)].min, \end{aligned} \tag{13}$$

here, $\mathbb{1}$ represents the indicator function, which evaluates to 1 if the condition is true and 0 otherwise.

Our methods are also applicable to the directed graph model. This involves constructing and utilizing the hop index without regard to edge direction, and enumerating matching results following the directions of the query edges.

8 Experiment

8.1 Setup

8.1.1 Algorithms. We compare our PTAB with four other algorithms: DP-B [11], kTPM [7], Take2 [24], and Eager [24]. We selected these methods not only because they are capable of fitting the score function defined in Eqn. 12[2], but also for their state-of-the-art efficiency. DP-B and kTPM are two classic and high-performance algorithms specializing in top-k subgraph matching, while Take2 and Eager, originally designed for ranked join queries in relational databases, can be adapted with some modifications to suit graph-based scenarios. We set the hop index parameter $H = 2$, which is sufficient for the majority of real-world queries, since over 90% of real queries have at most 6 edges [6].

8.1.2 Datasets. We use one synthetic dataset BSBM[3] and two real-world datasets (DBpedia[4] and Yago[5]) for our experiments. BSBM is a SPARQL benchmark widely recognized as a criterion for evaluating graph database engines [5]. DBpedia is an open knowledge graph spanning various domains and extensively employed in knowledge discovery [3]. YAGO2 is a high-quality knowledge graph extracted from Wikipedia [14]. As these datasets are in RDF graph format, we preprocess them as follows: (1) treating the object of the `rdf:type` predicate as the node's label, and (2) retaining only one label for nodes with multiple labels. Statistics of these datasets are provided in Tab. 2.

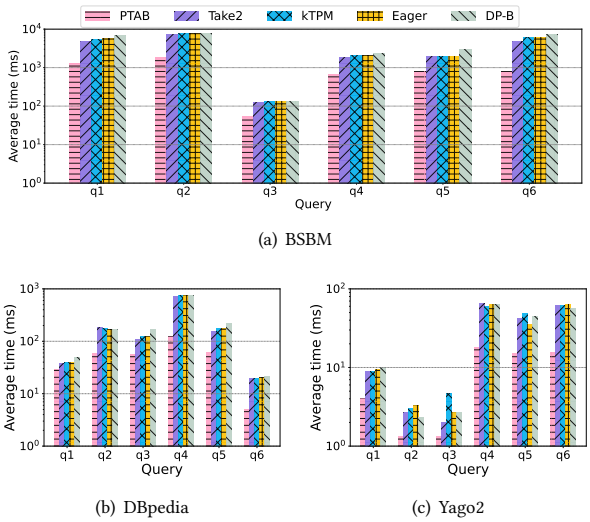
Table 2: Datasets Description

Dataset	Nodes	Edges	Labels
BSBM	15,522,016	94,646,775	1,287
DBpedia	17,768,464	77,490,708	230
Yago2	76,887,746	148,160,386	349,841

8.1.3 Queries. For the BSBM dataset, we use queries from its official website[3]. For the DBpedia dataset, we adopt the Feasible [20] benchmark to generate top-k queries. To the best of our knowledge, there isn't an established top-k benchmark for the Yago2 dataset, so we designed a set of top-k queries that fits the conclusion of the study on real SPARQL query logs [6]. Across each dataset, we selected 6 tree queries, and we set $k = 20$ for all experiments except the experiments studying the performance under various k.

8.1.4 Setup. We run our experiments on a server with an Intel Xeon Gold 6326 2.90GHz CPU and 256GB RAM running Ubuntu 20.04.5 LTS. All algorithms are implemented in C++[6]. Each experiment was repeated three times, and we report the averaged metrics.

8.2 Experiment Results



(a) BSBM

(b) DBpedia (c) Yago2

Figure 5: Total time comparison on three datasets ($k = 20$).

8.2.1 Total time. The total execution times for each method are depicted in Fig. 5, with $k = 20$ and the selected root is given by the method from Sec. 7.1 for this experiment. Across all queries, PTAB demonstrated superior performance, achieving 1.5 ~ 10 × speedup to the other methods.

[2]Some methods assume the node score functions have uniform monotonicity, and some other methods assume the property value of each node is non-negative (e.g. DP-P [11]). However, our methods can handle a wider range of score functions and property values.

[3]http://wbsg.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/

[4]https://www.dbpedia.org/

[5]https://yago-knowledge.org/

[6]The code and queries used in our experiments can be accessed at https://github.com/fyulingi/PTAB

Table 3: Reduced Proportion of Search Space

Datasets	q1	q2	q3	q4	q5	q6
BSBM	0.377	0.589	0.254	0.605	0.497	0.872
DBpedia	0.111	0.116	0.331	0.249	0.563	0.412
Yago2	0.396	0.480	0.185	0.431	0.198	0.282

Most of the subgraph queries in our experiments have enormous result sizes without the top-k constraint. For example, queries on BSBM (except $q3$, which has only 30 results) all have more than 10,000 results, and $q4$ even has more than 22,000,000 results. Our experiments shows that PTAB is more efficient than compared methods in such cases due to less search space exploration (detailed in Sec. 8.2.2).

Additionally, we investigated how matching times varied with different k values. We selected two representative queries, $q1$ from BSBM and $q5$ from DBpedia, for presentation, while similar results are observed for other queries. The results are displayed in Fig. 6. $q1$ query BSBM is a simple tree query degenerated into a chain, and the time consumed by all methods increases slowly. Conversely, the $q5$ query from DBpedia is a tree query with a height of 3. In this case, the matching times of kTPM and Eager notably escalate with the increase of k. This increase can be attributed to the need for extensive search space partitioning during enumeration (kTPM) or due to prolonged time for constructing auxiliary data structures for enumeration (the other methods). Meanwhile, our PTAB exhibits minimal time growth, showcasing consistent efficiency across queries of varying scales.

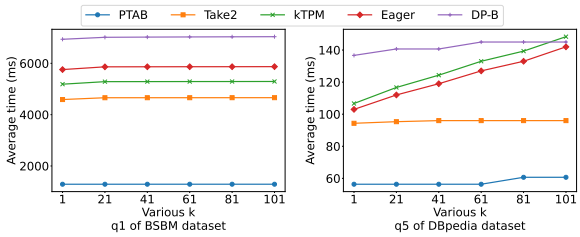
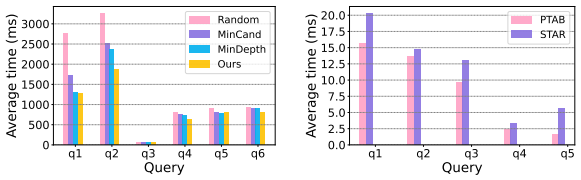


Figure 6: Matching time comparison of various k.

8.2.2 Explored space comparison. A key advantage of PTAB lies in its score bound estimation based on the hop index, effectively avoiding unnecessary exploration. To illustrate it, we show the ratio of the search space explored by PTAB in Tab. 3 ($k = 20$ in this experiment). Across all datasets and queries, PTAB reduces at least 10% and up to nearly 90% search space exploration. This reduction in exploration space aligns with the observed highest acceleration ratio of PTAB compared to other methods in the experiment illustrated in Fig. 5. Furthermore, we also counted the size of the used hop index. The hop index is lightweight, occupying only about 5% of the entire built data graph's size, and can be built in less than 2 minutes for the largest dataset Yago2. This cost-effectiveness makes it highly efficient for acceleration purposes.

8.2.3 Root selection. The compared methods do not mention the root selection, so we conducted a comparative analysis of our root selection strategy outlined in Sec. 7.1 against three alternative strategies ($k = 20$): (1) Random; (2) MinCand; (3) MinDepth to show the effectiveness of our method. The MinCand (MinDepth) strategy selects the query node which has a minimum candidate size (forms the query tree with the minimum height) to be the root node.



(a) Root selection strategy (BSBM) (b) Edge cut strategy (DBpedia)

Figure 7: Experiments of root selection and edge cut strategy

The performance of different root node strategies on the BSBM dataset is illustrated in Fig. 7(a). We can find that both MinCand and MinDepth outperform Random on average, and either of them could perform better than the other. However, our proposed strategy, which considers factors encompassing the height of the search tree, the size of the root node candidate set, and the query node score coefficient, enables better performance compared to other methods.

8.2.4 Edge-cut strategy. We extend PTAB to a general cyclic query graph through the edge-cut strategy introduced in Sec. 5.3. To assess its effectiveness, we constructed another 5 cyclic queries using the DBpedia dataset. Subsequently, we compared the runtime performance between our edge-cut strategy and the existing decomposition-then-rank-join method. We follow the algorithm STAR [29] to decompose the cyclic query graph into several stars and rank-join their results. For a fair comparison, we use PTAB for the decomposed star queries.

Remarkably, our edge-cut strategy reduces matching time compared to the decomposition-then-rank-join method, which enumerates the top-ranking results of each decomposed query pattern regardless of whether these results are connected in the data graph, while our edge-cut adopts a more holistic perspective.

9 Conclusion

In this paper, we propose a new algorithm, PTAB, for the top-k subgraph matching problem, which is enhanced by a hop index for score upper bound estimation to prune the search space, and a cost-aware root node selection strategy that enables PTAB to initiate the search from nodes leading to minimal exploration costs. We also extend PTAB to support both acyclic and cyclic query graphs via an edge-cut strategy. Experimental results on various real and synthetic datasets demonstrate that our method outperforms existing algorithms for top-k subgraph matching.

Acknowledgments

This work was supported by The National Key Research and Development Program of China under grant 2023YFB4502303 and NSFC under grant 61932001 and U20A20174. Lei Zou is the corresponding author of this work.

References

[1] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)* 42, 4 (2017), 1–44.

[2] Junya Arai, Yasuhiro Fujiwara, and Makoto Onizuka. 2023. Gup: Fast subgraph matching by guard-based pruning. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.

[3] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. Dbpedia: A nucleus for a web of open data. In *international semantic web conference*. Springer, 722–735.

[4] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*. 1199–1214.

[5] Christian Bizer and Andreas Schultz. 2009. The berlin sparql benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)* 5, 2 (2009), 1–24.

[6] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. An analytical study of large SPARQL query logs. *The VLDB Journal* 29, 2-3 (2020), 655–679.

[7] Lijun Chang, Xuemin Lin, Wenjie Zhang, Jeffrey Xu Yu, Ying Zhang, and Lu Qin. 2015. Optimal enumeration: efficient top-k tree matching. *Proceedings of the VLDB Endowment* 8, 5 (2015), 533–544.

[8] Jiefeng Cheng, Xianggang Zeng, and Jeffrey Xu Yu. 2013. Top-k graph pattern matching over large graphs. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 1033–1044.

[9] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence* 26, 10 (2004), 1367–1372.

[10] Ronald Fagin, Amnon Lotem, and Moni Naor. 2001. Optimal aggregation algorithms for middleware. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 102–113.

[11] Gang Gou and Rada Chirkova. 2008. Efficient algorithms for exact ranked twig-pattern matching over graphs. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 581–594.

[12] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. 2003. XRANK: Ranked keyword search over XML documents. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 16–27.

[13] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the 2019 International Conference on Management of Data*. 1429–1446.

[14] Johannes Hoffart, Fabian M Suchanek, Klaus Berberich, and Gerhard Weikum. 2013. YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. *Artificial intelligence* 194 (2013), 28–61.

[15] Ihab F Ilyas, Walid G Aref, and Ahmed K Elmagarmid. 2004. Supporting top-k join queries in relational databases. *The VLDB journal* 13 (2004), 207–221.

[16] Ihab F Ilyas, George Beskales, and Mohamed A Soliman. 2008. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)* 40, 4 (2008), 1–58.

[17] Tatiana Jin, Boyang Li, Yichao Li, Qihui Zhou, Qianli Ma, Yunjian Zhao, Hongzhi Chen, and James Cheng. 2023. Circinus: Fast redundancy-reduced subgraph matching. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.

[18] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1692–1704.

[19] M. E. J. Newman. 2000. Models of the Small World. *Journal of Statistical Physics* 101, 3/4 (2000), 819–841. https://doi.org/10.1023/A:1026485807148

[20] Muhammad Saleem, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. 2015. Feasible: A feature-based sparql benchmark generation framework. In *The Semantic Web-ISWC 2015: 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I 14*. Springer, 52–69.

[21] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-depth Study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1083–1098. https://doi.org/10.1145/3318464.3380581

[22] Shixuan Sun and Qiong Luo. 2020. Subgraph matching with effective matching order and indexing. *IEEE Transactions on Knowledge and Data Engineering* 34, 1 (2020), 491–505.

[23] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. RapidMatch: A holistic approach to subgraph query processing. *Proceedings of the VLDB Endowment* 14, 2 (2020), 176–188.

[24] Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald, and Xiaofeng Yang. 2020. Optimal algorithms for ranked enumeration of answers to full conjunctive queries. *Proceedings of the VLDB Endowment* 13, 9 (2020), 1582–1597.

[25] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. 2020. Optimal join algorithms meet top-k. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2659–2665.

[26] J. R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (jan 1976), 31–42. https://doi.org/10.1145/321921.321925

[27] Dong Wang, Lei Zou, and Dongyan Zhao. 2015. Top-k queries on RDF graphs. *Information Sciences* 316 (2015), 201–217. https://doi.org/10.1016/j.ins.2015.04.032 Nature-Inspired Algorithms for Large Scale Global Optimization.

[28] Linglin Yang, Lei Yang, Yue Pang, and Lei Zou. 2022. gCBO: A Cost-based Optimizer for Graph Databases. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management* (Atlanta, GA, USA) *(CIKM '22)*. Association for Computing Machinery, New York, NY, USA, 5054–5058. https://doi.org/10.1145/3511808.3557197

[29] Shengqi Yang, Fangqiu Han, Yinghui Wu, and Xifeng Yan. 2016. Fast top-k search in knowledge graphs. In *2016 IEEE 32nd international conference on data engineering (ICDE)*. IEEE, 990–1001.

[30] Shengqi Yang, Yinghui Wu, Huan Sun, and Xifeng Yan. 2014. Schemaless and structureless graph querying. *Proceedings of the VLDB Endowment* 7, 7 (2014), 565–576.

[31] Lei Zou, Lei Chen, and Yansheng Lu. 2007. Top-k subgraph matching query in a large graph. In *Proceedings of the ACM First Ph.D. Workshop in CIKM* (Lisbon, Portugal) *(PIKM '07)*. Association for Computing Machinery, New York, NY, USA, 139–146. https://doi.org/10.1145/1316874.1316897