

Querying Web-Scale Information Networks Through Bounding Matching Scores

Jiahui Jin^{†‡}, Samamon Khemmarat[‡], Lixin Gao[‡], Junzhou Luo[†]

[†]School of Computer Science and Engineering, Southeast University, China

[‡]Department of Electrical and Computer Engineering, University of Massachusetts Amherst, USA

{jhjin, jluo}@seu.edu.cn; {khemmarat, lgao}@ecs.umass.edu

ABSTRACT

Web-scale information networks containing billions of entities are common nowadays. Querying these networks can be modeled as a subgraph matching problem. Since information networks are incomplete and noisy in nature, it is important to discover answers that match exactly as well as answers that are similar to queries. Existing graph matching algorithms usually use graph indices to improve the efficiency of query processing. For web-scale information networks, it may not be feasible to build the graph indices due to the amount of work and the memory/storage required.

In this paper, we propose an efficient algorithm for finding the best k answers for a given query without precomputing graph indices. The quality of an answer is measured by a matching score that is computed online. To speed up query processing, we propose a novel technique for bounding the matching scores during the computation. By using bounds, we can efficiently prune the answers that have low qualities without having to evaluate all possible answers. The bounding technique can be implemented in a distributed environment, allowing our approach to efficiently answer the queries on web-scale information networks. We demonstrate the effectiveness and the efficiency of our approach through a series of experiments on real-world information networks. The result shows that our bounding technique can reduce the running time up to two orders of magnitude comparing to an approach that does not use bounds.

1 INTRODUCTION

Information networks such as Freebase have become massive in recent years. Freebase [11], a collaborative knowledge graph, contains more than 43.9 million entities, interconnected by 2.4 billion facts. The Linked Open Data project [4] connects RDF datasets on the web, resulting in an information network containing 1.77 billion RDF triples. It is important to be able to extract information from these networks. However, performing query operations on such web-scale information networks can be challenging.

In this paper, we study the problem of identifying a set of unknown entities in an information network given a set of related entities and facts/relations. For example, a query

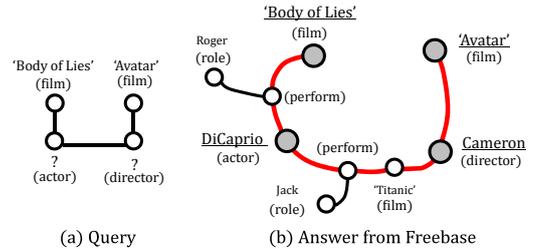


Figure 1. An Example of Query on Information Network

on Freebase can be “find an actor who collaborated with the director of the movie ‘Avatar’ and also performed in the movie ‘Body of Lies’”. We can model an information network or a query as a graph that consists of entities and relationships (or facts) between entities. Answering queries on information networks then becomes a subgraph matching problem. While it is common to seek answers that match exactly to queries, it is equally important to discover an answer that is similar to the queries. Information networks are typically crawled from the web or a large collection of databases. Therefore, they can be incomplete and noisy. Further, a user might not know the schema of the information network well enough to specify a query. It is likely that a user describes a vague query that is similar in structure to the desired answer. Fig. 1(a) shows the query graph for the aforementioned example query. An answer to the query from Freebase, that the actor is DiCaprio and the director is Cameron, is shown in Fig. 1(b). Clearly, the answer is not an exact match to the query. In particular, actor DiCaprio does not connect to director Cameron or the movie ‘Body of Lies’ directly in Freebase. Nevertheless, this is a good answer to the query. Therefore, it is crucial to be able to identify both *exact and similar matches* to a query for information networks.

The problem of subgraph matching has been studied extensively in the past decades [15, 16, 25, 27, 30]. However, most of them utilize graph indices that require precomputation and storage size of super-linear to information network size [24]. It may be infeasible to build the graph indices for web-scale networks due to the amount of work and the memory/storage required. To address this problem, we propose an *index-free*¹ approach for answering information queries. Our approach is able to identify all answers that have the same structure as the query as well as answers that are sim-

¹By index-free, we mean that the algorithm does not require additional indices other than the index for representing an adjacency list of the information networks.

ilar to the query. The quality of an answer is measured by a matching score evaluated by comparing the structural signatures of the answer with that of the query. In contrast to the existing approaches, we compute matching scores online instead of precomputing the scores.

To speed up query processing, we propose **GraB** (Graph Matching using **B**ound), a novel technique for bounding the matching scores during the computation. By using the bounds, we can efficiently prune the low quality answers without evaluating all the possible answers. While we present the bounding technique in the context of a specific matching score function, it can be broadly applied to other functions as well. Furthermore, we implement the bounding technique in a distributed environment. Therefore, our approach can scale to answer the queries on web-scale information networks.

The key contributions of this paper are summarized as follows:

- We propose an algorithm for finding the k best answers. The algorithm can be applied to identify both exact and similar matches. The algorithm uses a novel heuristic to select only the promising candidates of the unknown entities, using the known entities as guides.
- In order to scale the algorithm to billions of nodes, we propose an index-free algorithm that computes matching scores online. Our index-free algorithm does not make assumptions about the extent of similarity between a query and the answer. Therefore, it is able to identify the top- k answers for every query. A novel technique for bounding the matching scores is used to efficiently find the top- k best answers.
- We implement our algorithm in a distributed environment. To evaluate the scalability, we deploy the distributed system on an Amazon EC2 cluster that contains hundreds of machines. The results show that our system can support querying on graphs with billions of nodes and scale to large queries.
- Our evaluation using real-world datasets shows that our algorithm can identify the top- k answers for every query and the answers are accurate. Additionally, our bounding technique can reduce the running time up to two orders of magnitude comparing to an approach that does not use bounds.

The rest of this paper is organized as follows. Section 2 describes the definition of the graph similarity matching problem. Section 3 provides the framework of our algorithm. In Section 4 an index-free algorithm is proposed. Section 5 discusses our distributed solution. In Section 6, we present the evaluation of our algorithm. We discuss related work in Section 7 and conclude the paper in Section 8.

2 PROBLEM DEFINITION

2.1 Preliminaries

We model an information network as a typed graph. A typed graph (V, E, T) is an undirected graph, where V is a set of nodes, E is a set of edges, and T is a set of node types. Each node v has a *node type* and a *name* that is a string. Generally, there are many nodes having the same node type, but a node name is usually unique or shared by only a few nodes. For simplicity, we assume the node names are unique and there are no types on the edges. However, our approach could be extended for graphs with shared names and typed

edges. We represent an information network by a data graph $G = (V_G, E_G, T)$ and refer to its nodes as *data nodes*.

A query on an information graph aims to discover a set of *unknown entities* by providing related entities, or *known entities*. We model the query as a typed graph, $Q = (V_Q, E_Q, T)$. In the query graph, there are two types of nodes: (1) A *specific node* corresponds a known entity. Both the type and the name of the specific node are known. (2) A *query node* corresponds to an unknown entity. Only its type is known. In the query graph in Fig. 1, ‘Avatar’ and ‘Body of Lies’ are the specific nodes, and the two nodes with type *actor* and type *director* are the query nodes. We denote the set of specific nodes by V_Q^S and the set of queries nodes by V_Q^U .

To answer a query, we need to map every query node to a data node. The mapping is referred to as an *embedding*.

Definition 1 (Embedding). *Given a data graph G and a query graph Q , an embedding of Q in G is an injective function $f : V_Q \rightarrow V_G$ where (1) $\forall q \in V_Q$, q and $f(q)$ have the same type; (2) $\forall q^s \in V_Q^S$, q^s and $f(q^s)$ have the same name.*

The embeddings that result in the subgraphs with the same structure as the query graph are *exact matches*. Formally, we have the following definition.

Definition 2 (Exact Match). *An embedding f is an exact match of a query graph Q iff for each edge $(q_i, q_j) \in E_Q$, there is an edge $(f(q_i), f(q_j)) \in E_G$.*

Considering the noisiness and incompleteness of the information networks, to answer a given query we are interested in finding both the exact matches and similar matches. Therefore, we consider the top- k graph similarity matching problem, which returns the best k embeddings according to a similarity measure, as described in the next section.

2.2 Similarity Measure

In this section, we propose a similarity measure to quantify the structural similarity between a query graph and an embedding.

First, we introduce a closeness vector, which is used to represent a structural signature of a node’s neighborhood in a graph. The closeness vector captures the graph structure around a node using the closeness between the node and the other nodes. For a node q_i in a query graph, its closeness vector specifies how close it is to each node in the query graph, defined formally as follows. Let the nodes in a query graph Q be q_1, \dots, q_m . The *closeness vector* of q_i , denoted by $R_Q(q_i)$, is defined as

$$R_Q(q_i) = [\varphi_Q(q_i, q_1), \dots, \varphi_Q(q_i, q_m)],$$

where $\varphi_Q(q_i, q_j)$ is a closeness score between q_i and q_j .

We design our closeness score function, which quantifies the closeness between two nodes, by aiming for two properties: (1) The closer the nodes are in terms of the shortest path distance, the higher the score. (2) When two pairs of nodes have the same shortest path distance, the pair having more shortest paths has a higher score. Based on these properties, the closeness between node u and v in a graph G , denoted by $\varphi_G(u, v)$, is defined by:

$$\varphi_G(u, v) = \begin{cases} 1, & u = v \\ \min\{n_{u,v}\alpha^{l_{u,v}}, N\alpha^{l_{u,v}}\}, & \text{otherwise} \end{cases} \quad (1)$$

where $l_{u,v}$ and $n_{u,v}$ are the length and the number of shortest

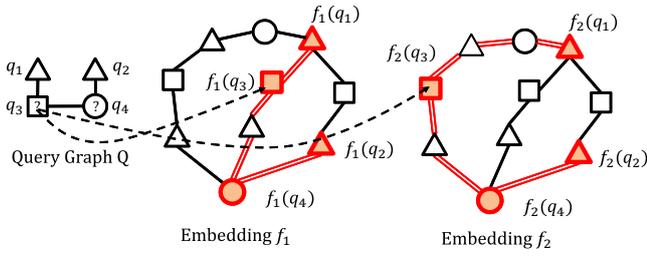


Figure 2. Query Graph and Two embeddings

paths between u and v , α is a constant between 0 and 1, and N is a constant smaller than $\frac{1}{\alpha}$. When $n_{u,v} > N$, the closeness score of u and v is bounded to $N\alpha^{l_{u,v}}$, which guarantees Property (1) since $N\alpha^{l_{u,v}} < \alpha^{l_{u,v}-1}$.

To quantify the similarity between a query graph and an embedding, we also represent a match of a query node in an embedding with a closeness vector. The closeness vector of a match specifies how close it is to the other matches in the data graph. Formally, the *closeness vector* $R_G(q_i, f)$ of a match of q_i in embedding f is defined as

$$R_G(q_i, f) = [\varphi_G(f(q_i), f(q_1)), \dots, \varphi_G(f(q_i), f(q_m))].$$

Based on the closeness vectors, for a given embedding we quantify the cost of matching a node to a query node, or *node match cost*, using the difference between their closeness vectors. For a match of q_i in embedding f , the *node match cost* is computed as follows:

$$C(f, q_i) = \sum_{q_j \in V_Q} \Theta(\varphi_Q(q_i, q_j), \varphi_G(f(q_i), f(q_j))), \quad (2)$$

where Θ is a positive-difference function defined as

$$\Theta(a, b) = \begin{cases} a - b, & a > b \\ 0, & \text{otherwise.} \end{cases} \quad (3)$$

The node match cost is the sum of the difference in all the dimensions of the closeness vectors. The positive-difference function is used for two purposes: (1) To avoid penalizing the case where two matches are closer than their corresponding query nodes. The closer matches mean they are closely related. Intuitively, this should not degrade the quality of the answer. (2) To avoid the negative difference in one dimension canceling out the difference in the other dimensions of the vector, which can introduce false-positive answers.

We illustrate how the node match cost can capture the quality of a match as follows.

Example 1. We compare two embeddings, f_1 and f_2 , in Fig. 2. Intuitively, f_1 is better than f_2 because $f_1(q_3)$ is a better match of q_3 . The reason is that there is a 1-hop path between $f_1(q_3)$ and $f_1(q_1)$, while there is a 3-hop path between $f_2(q_3)$ and $f_2(q_1)$. We show how our node match cost can capture this intuition. The closeness vectors of $f_1(q_3)$ and $f_2(q_3)$ are $[\alpha, 2\alpha^3, 1, \alpha^2]$ and $[\alpha^3, \alpha^3, 1, \alpha^2]$, respectively. Computing the difference with the closeness vector of q_3 , i.e., $[\alpha, \alpha^2, 1, \alpha]$, $f_1(q_3)$ has lower match cost, indicating that it is a better match. This demonstrates that the node match cost agrees with the intuition.

Based on the node match cost, we formulate a similarity measure for an embedding by combining the node match cost of all the nodes in the embedding as follows.

Definition 3 (Similarity Measure). *Given a data graph G ,*

Notation	Description
V_Q^U	A set of query nodes
V_Q^S	A set of specific nodes
q^s	A specific node
$\phi(q^s)$	An anchor node that is a match of specific node q^s (defined in Section 3)
$\varphi_G(u, v)$	The closeness score between u and v in data graph
$\varphi_Q(q_i, q_j)$	The closeness score between q_i and q_j in query graph
$\delta_{q_i, q_j}(u, v)$	A shortened form of $\Theta(\varphi_Q(q_i, q_j), \varphi_G(u, v))$
$\mathcal{C}(f)$	The embedding match cost of embedding f
$C^K(v, q)$	The known match cost of v as a candidate of q (defined in Section 3)
\mathcal{M}^q	The candidate match set of q (defined in Section 3)
\mathcal{F}	The candidate embedding set (defined in Section 3)

Table 1. Frequently Used Notations

a query graph Q , and an embedding f , the match cost of f is defined as:

$$\mathcal{C}(f) = \sum_{q_i \in V_Q} C(f, q_i) \quad (4)$$

The match cost of an embedding takes into account the structural difference between the query graph and the embedding. The more similar an embedding to the query graph, the lower its match cost. Our similarity measure has following properties:

Property 1. *If an embedding f is an exact match of the query graph Q , then $\mathcal{C}(f) = 0$.*

Proof. Since f is an exact match of Q , for any edge (q_i, q_j) in Q , there is a corresponding edge $(f(q_i), f(q_j))$ in the data graph. Therefore, given any pair of query node q_i and q_j , $\varphi_Q(q_i, q_j) \leq \varphi_G(f(q_i), f(q_j))$. By Equation (4), $\mathcal{C}(f) = 0$. \square

Additionally, we can guarantee that the match cost of any inexact match is greater than 0.

Property 2. *For any inexact match f , $\mathcal{C}(f) > 0$*

Proof. Since f is not an exact match, there is an edge (q_i, q_j) in Q such that $(f(q_i), f(q_j))$ is not in the data graph. Since $N < \frac{1}{\alpha}$, we have $\varphi_Q(q_i, q_j) > \varphi_G(f(q_i), f(q_j))$. Furthermore, $C(q_i, f) > 0$ and $C(q_j, f) > 0$. Therefore, $\mathcal{C}(f) > 0$. \square

Our embedding match cost can be extended to support graphs with typed edges. When two nodes in a query graph are connected by a typed edge, their matches should be connected by the edge having the same type. Otherwise, the cost of matching the edge types should be added to the embedding matching costs.

Based on the defined match costs, our top-k graph similarity matching problem is defined as follows.

The top-k similarity matching problem. Given a data graph G and a query graph Q , identify k embeddings that have the smallest match costs.

To simplify the notation, in the rest of the paper, we let $\delta_{q_i, q_j}(u, v)$ denote $\Theta(\varphi_Q(q_i, q_j), \varphi_G(u, v))$. We list the frequently used notations in Table 1.

3 ALGORITHM FRAMEWORK

A naive approach for finding the top-k embeddings is to enumerate and rank all possible embeddings. From the definition, the candidate matches for each query node q include all the data nodes with the same type as q 's. Thus, the number of all possible embeddings can be very large, especially in massive networks, and this naive approach can be

very time-consuming. To speed up the computation, our approach reduces the search space by lessening the number of candidate matches for each query node while ensuring that the high quality (low matching cost) embeddings are still in the reduced search space.

In our approach, the candidate set for each query node is populated by selecting a small number of data nodes that are likely to produce high quality embeddings. For this, we need a measure that can evaluate the quality of the matches of each query node independently. To derive this measure, we decompose the embedding match cost as follows.

$$\mathcal{C}(f) = \underbrace{\sum_{q_i \in V_Q} \sum_{q_j \in V_Q^S} \delta_{q_i, q_j}(f(q_i), f(q_j))}_{\text{Part I}} + \underbrace{\sum_{q_i \in V_Q} \sum_{q_j \in V_Q^U} \delta_{q_i, q_j}(f(q_i), f(q_j))}_{\text{Part II}} \quad (5)$$

In the decomposition, **Part I** is the cost based on each match's closeness with the matches of specific nodes. **Part II** is the cost based on each match's closeness with the matches of query nodes. It can be seen that in **Part I**, the cost associated with each query node depends on only the matches of the specific nodes, independent of the other query nodes. Precisely, the cost of matching a data node v to a query node q in **Part I**, denoted by $C^K(v, q)$, is as follows:

$$C^K(v, q) = \sum_{q^s \in V_Q^S} \delta_{q^s, q}(\phi(q^s), v), \quad (6)$$

where $\phi(q^s)$ is an *anchor node*, the match of specific node q^s . We refer to $C^K(v, q)$ as *known match cost*.

We build a candidate set for each query node by selecting data nodes with the lowest known match cost. Because the selected candidates minimize a significant part of the total cost, the top embeddings are likely to be among the embeddings produced from these candidate sets. We show later through our experiments that this heuristic efficiently prunes search space but still provides accurate answers.

The overview of our approach, **GrAB**, is shown in Algorithm 1. The approach consists of two phases. In Phase 1, a candidate match set for each query node is created by selecting k^* *best* candidate matches, where k^* is a predefined constant. In Phase 2, the approach searches for k embeddings with the lowest match costs among the $k^*|V_Q^U|$ candidate embeddings.

The value of k^* determines the accuracy of the algorithm. When k^* is larger, more candidate matches are selected for each query node; therefore, it is more likely that the real top- k answers are among the $k^*|V_Q^U|$ candidate embeddings. However, with larger k^* the algorithm will need longer time to enumerate the candidate embeddings in Phase 2. Intuitively, if each query node has more than k candidates, the accurate top- k answers are likely to be found. Therefore, we recommend to set k^* between k to $2k$. We discuss the selection of k^* further in the experiment. Note that in order to find all the embeddings that exactly match the query graph, we should select all the candidate matches whose known match costs are zero. In this case, the size of the candidate set may be larger than k^* .

4 ALGORITHM DESCRIPTION

From the algorithm framework, one solution that allows the results to be returned quickly is to create an index that stores a precomputed pairwise closeness scores and use the index in computing the known match cost and the embed-

Algorithm 1: GrAB

Input: data graph G , query graph Q and k
Output: top- k embeddings
 // Phase 1: Identifying the top- k^* candidate matches
 for each query node
 1 $\mathcal{M} \leftarrow \text{IdentifyMatches}(G, Q, k^*)$
 // Phase 2: Identifying the top- k embeddings
 2 $\mathcal{F}_k \leftarrow \text{IdentifyEmbs}(G, Q, \mathcal{M}, k)$
 3 **return** \mathcal{F}_k

Algorithm 2: (Phase 1) IdentifyMatches(G, Q, k^*)

Input: data Graph G , query graph Q, k^*
Output: top- k^* candidate set for each query node \mathcal{M}
 1 $Src \leftarrow$ all the anchor nodes
 2 **foreach** $q \in V_Q^U$ **do**
 3 **repeat**
 4 $\text{BoundRefinement}(Src, \varphi_G^+, \varphi_G^-, G)$ // Sec. 4.4
 5 **until** $\text{TermPhase1}(\varphi_G^+, \varphi_G^-, G, q)$ is true // Sec. 4.3.2
 6 $\mathcal{M}_{k^*}^q \leftarrow$ the top- k^* candidate match set of q
 $\mathcal{M} \leftarrow \mathcal{M} \cup \{q, \mathcal{M}_{k^*}^q\}$
 7 **return** \mathcal{M}

ding match cost. For fast access, the indexed closeness scores have to be stored in memory. However, the number of pairwise closeness scores is $|V|^2$, where $|V|$ is the number of data nodes; for a web-scale information network that contains billions of nodes, storing the pairwise closeness scores in memory is usually infeasible. In this section, we describe our index-free algorithm, which does not require the precomputed closeness scores but still can answer queries efficiently.

4.1 Algorithm Overview

As the algorithm described in Section 3, the index-free algorithm consists of two phases, *i.e.*, identifying the top- k^* candidate matches for each query node and identifying the top- k embeddings. While we can compute all the required closeness scores in both phases online, the computation is time-consuming. To obtain the results faster, our algorithm computes the bounds of the closeness scores and use the bounds to derive the top- k embeddings.

In this algorithm, the bounds of the closeness scores are computed by performing multiple Breadth-First Searches (BFSes). Each BFS iteratively refines the closeness score bounds (denoted by φ_G^+ and φ_G^-) between one data node (source node) and the other data nodes. As the BFSes are progressed, the bounds approach the exact scores. We refer to this process as *bound refinement*. In Phase 1, we need the closeness scores between each of the anchor node and the candidate matches of a query node q to compute the known match costs. Therefore, we perform $|V_Q^S|$ BFSes, each having an anchor node as a source node (line 4 of Algorithm 2). In Phase 2, the closeness scores among the candidate matches are required. Therefore, we perform the BFSes with each match as a source node (line 3 of Algorithm 3).

In both phases, while the bounds are being refined, the algorithm keeps checking whether the targeted results for that phase can be obtained with the current bounds. For Phase 1, for each query node, the algorithm checks whether the top- k^* candidate matches are found (line 5 of Algorithm 2). Once the top- k^* matches are found for all the query nodes, Phase 2 can be started. Similarly, for Phase 2, the algorithm keeps checking whether the top- k embeddings can be identified with the current bounds and returns the results

Algorithm 3: (Phase 2) IdentifyEmbs(G, Q, \mathcal{M}, k)

Input: data graph G , query graph Q , candidate match set \mathcal{M}, k
Output: top- k embedding set \mathcal{F}_k

- 1 $Src \leftarrow$ all the candidate matches
- 2 **repeat**
- 3 BoundRefinement($Src, \varphi_G^+, \varphi_G^-, G$) // Sec. 4.4
- 4 **until** TermPhase2($\varphi_G^+, \varphi_G^-, \mathcal{M}$) is true // Sec. 4.3.3
- 5 $\mathcal{F}_k \leftarrow$ the top- k embeddings
- 6 **return** \mathcal{F}_k

as soon as they are found (line 4 of Algorithm 3). We refer to the process of checking whether the top- k^* candidate matches (or the top- k embeddings) can be identified with the bounds as *termination check*.

In the following sections, we describe the derivation of the closeness score bounds, the termination check process, and the bound refinement process in detail.

4.2 Deriving Closeness Score Bounds

As stated in the overview, we use a BFS to refine the closeness score bounds between one data node and the other data nodes. Here we describe how we compute the closeness score bounds in each BFS.

Given a source node, the BFS iteratively visits the data nodes in the source node's neighborhood. At iteration t , all the data nodes in the source node's t -hop neighborhood are visited by the search.

For the data nodes visited by the BFS in iteration t , we can compute its exact closeness scores. Suppose the source node is s . Let V_s^t denote the set of the data nodes that are visited in iteration t , *i.e.*, the nodes that are t hops away from s . Initially, we have $V_s^0 = \{s\}$ and $\varphi_G(s, s) = 1$. For the later iterations, the closeness score of a data node v in V_s^t is computed by aggregating the closeness scores of v 's neighbors that are $t-1$ hops away from s , as shown in the following equation.

$$\varphi_G(s, v) = \min\{N\alpha^t, \sum_{u \in V_s^{t-1} \cap \mathbb{N}(v)} \varphi_G(s, u) \cdot \alpha\} \quad (7)$$

where $\mathbb{N}(v)$ is v 's adjacent neighbor set.

For the unvisited data nodes in iteration t , we know that they are at least $t+1$ hops away from the source node; therefore, their closeness scores are in the range of $[0, N\alpha^{t+1}]$. Thus, at iteration t of the BFS starting from the source node s , we have the following closeness scores bounds for a data node v .

$$\begin{aligned} \varphi_G^+(s, v) &= \begin{cases} \varphi_G(s, v), & v \text{ is visited} \\ N\alpha^{t+1}, & \text{otherwise} \end{cases} \\ \varphi_G^-(s, v) &= \begin{cases} \varphi_G(s, v), & v \text{ is visited} \\ 0, & \text{otherwise} \end{cases} \end{aligned} \quad (8)$$

4.3 Termination Check

While the closeness score bounds are being refined by the BFSes, **GraB** periodically performs the termination check that tests whether the top- k^* candidate matches or the top- k embeddings are found. In this subsection, we describe how to use the bounds to identify the top- k^* candidate matches and the top- k embeddings.

4.3.1 Termination Check Using Bounds

First, we describe a strategy for finding the top- k embeddings from the candidate embedding set by using the upper

Algorithm 4: TermPhase1($\varphi_G^+, \varphi_G^-, G, q$)

Input: closeness score ub φ_G^+ , closeness score lb φ_G^- , data graph G , query node q
Output: whether the top- k^* matches of q are found

- 1 $\mathcal{M}^q \leftarrow$ all the candidate matches of q
- 2 $\mathcal{M}_{k^*}^{q+} \leftarrow$ the k^* matches in \mathcal{M}^q with the smallest $C^{K^+}(v, q)$.
- 3 $C^{K^+}(v_{k^*}, q) \leftarrow$ the k^{*th} smallest known match cost ub
- 4 **foreach** $u \in \mathcal{M}^q - \mathcal{M}_{k^*}^{q+}$ **do**
- 5 **if** $C^{K^-}(u, q) < C^{K^+}(v_{k^*}, q)$ **then**
- 6 **return** false
- // $\mathcal{M}_{k^*}^{q+}$ is the top- k^* candidate match set of q .
- 7 **return** true

bounds, C^+ , and lower bounds, C^- , of the embedding match costs. The same strategy can be used to find the top- k^* candidates for each query node by using the upper bounds, C^{K^+} , and lower bounds, C^{K^-} , of the known match costs.

The top- k embedding identification consists of the following two steps.

1. Find k embeddings with the smallest cost upper bounds, f_1, \dots, f_k .
2. Determine whether f_1, \dots, f_k are the top- k embeddings. Assuming that $C^+(f_1) \leq C^+(f_2) \dots \leq C^+(f_k)$, we check the following condition:

$$C^+(f_k) < C^-(f) \text{ for all } f \notin \{f_1, \dots, f_k\} \quad (9)$$

Eq. (9) uses the cost lower bounds to ensure that every f that is not in $\{f_1, \dots, f_k\}$ cannot have the cost lower than those of the tentative top- k items. If Eq. (9) is satisfied, it is guaranteed that f_1, \dots, f_k are the real top- k embeddings. Next, we discuss how to apply this strategy in each phase of the algorithm.

4.3.2 Termination Check for Top- k^* Candidates

In Phase 1, for each query node q , our task is to find k^* matches with the lowest known match costs. For each candidate, we derive its lower and upper bounds of known match costs.

Bounds of known match cost. From Eq. (6), we derive the bounds of known match costs in terms of the bounds of closeness scores. For a candidate v of a query node q , the bounds of known match cost can be computed by the following equations.

$$\begin{aligned} C^{K^+}(v, q) &= \sum_{q^s \in V_Q^S} \Theta(\varphi_Q(q^s, q), \varphi_G^-(\phi(q^s), v)) \\ C^{K^-}(v, q) &= \sum_{q^s \in V_Q^S} \Theta(\varphi_Q(q^s, q), \varphi_G^+(\phi(q^s), v)) \end{aligned} \quad (10)$$

Performing termination check. The termination check for Phase 1 is performed as shown in Algorithm 4. In the first step (line 1-3), we initialize a set \mathcal{M}^q that contains all the data nodes having the same type as q . Then, the tentative top- k^* candidates, which have the smallest upper bound costs, are identified from \mathcal{M}^q . In the second step (line 4-7), we test whether the k^* candidates we found are the real top- k^* candidates by checking the lower bound costs.

4.3.3 Termination Check for Top- k embeddings

In Phase 2, we find the top k embeddings with the lowest embedding match costs. The details are given as follows.

Bounds of embedding match cost. We derive the lower and upper bounds of embedding match costs, *i.e.*, $C^-(f)$

Algorithm 5: TermPhase2($\varphi_G^+, \varphi_G^-, \mathcal{M}$)

Input: closeness score ub φ_G^+ , closeness score lb φ_G^- , candidate match set \mathcal{M}

Output: whether the top- k embeddings are found

- 1 $\mathcal{F}_k^+ \leftarrow \text{EmbGetUBTopK}(\varphi_G^-, k)$
 - 2 **if** EmbCheckLB($\varphi_G^+, \mathcal{C}^+(f_k)$) **is false** **then**
 - 3 **return** false
 - // \mathcal{F}_k^+ is the top- k embedding set.
 - 4 **return** true
-

Algorithm 6: BoundRefinement($S, \varphi_G^+, \varphi_G^-, G$)

Input: source node set S , closeness score ub φ_G^+ , closeness score lb φ_G^- and data graph G

Output: refined closeness score bounds φ_G^{*+} and φ_G^{*-}

- 1 $s \leftarrow \text{GetBestSrc}(S)$ // By Eq. (14) or Eq. (13)
 - 2 $\text{BFS}(s).\text{NextIteration}()$ // By Eq. (7)
 - 3 $(\varphi_G^{*+}, \varphi_G^{*-}) \leftarrow \text{UpdateBounds}(\varphi_G^+, \varphi_G^-)$ // By Eq. (8)
 - 4 **return** φ_G^{*+} and φ_G^{*-}
-

and $\mathcal{C}^+(f)$, in terms of the bounds of closeness scores, as shown in the following equations.

$$\begin{aligned} \mathcal{C}^+(f) &= \sum_{q_i \in V_Q} \sum_{q_j \in V_Q} \Theta(\varphi_Q(q_i, q_j), \varphi_G^-(f(q_i), f(q_j))) \\ \mathcal{C}^-(f) &= \sum_{q_i \in V_Q} \sum_{q_j \in V_Q} \Theta(\varphi_Q(q_i, q_j), \varphi_G^+(f(q_i), f(q_j))) \end{aligned} \quad (11)$$

where $\varphi_G^+(\cdot)$ and $\varphi_G^-(\cdot)$ are the upper bound and the lower bound of the closeness scores.

Performing termination check. The outline of the termination check is given in Algorithm 5. For the first step (line 1), we find the top- k embeddings with the smallest match cost upper bounds, computed from the closeness score lower bounds. In the second step (line 2), our task is to check whether there are no more than k embeddings f having $\mathcal{C}^-(f) < \mathcal{C}^+(f_k)$. However, in both of these steps, to enumerate the entire embedding set from the candidate match sets and compute their match cost can be very time-consuming. In Section 4.5, we show how to accelerate the embedding enumeration with a branch-and-bound method.

4.4 Bound Refinement

In the two phases of our algorithm, multiple BFSes are performed to obtain all the required closeness score bounds. We progress the BFSes concurrently. The BFSes are selected one at a time, where each time the selected BFS is executed for one iteration, as shown in Algorithm 6.

How the BFSes are selected for execution is important for the efficiency of our algorithm. Naively performing the BFSes in a round-robin manner *i.e.*, one iteration in turn, can be inefficient. We illustrate the importance of choosing a proper order of the BFSes using Fig. 3. To identify the top-2 matches for query node q_3 , we perform two BFSes starting from v_1 and v_2 . We denote the two BFSes by BFS(v_1) and BFS(v_2), respectively. Consider two orders of execution: (1) perform BFS(v_2) two iterations and then BFS(v_1) one iteration and (2) perform BFS(v_1) and BFS(v_2) alternately for two iterations. The first order is more efficient since it only requires a total of three iterations to find the top-2 matches, as opposed to four iterations in the latter case.

To improve the efficiency, we dynamically assign priorities

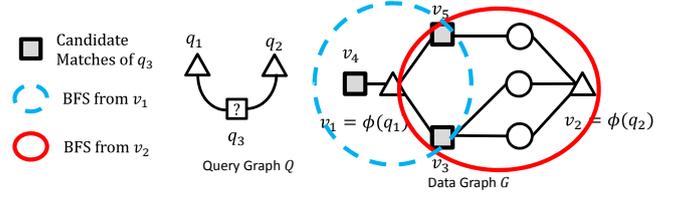


Figure 3. Identifying Top-2 Matches for A Query Node

to the BFSs. We use a heuristic to quantify the benefit of performing each BFS. For Phase 1, intuitively, if the bounds of the known match costs are closer to the exact costs, we are more likely to find the top- k^* matches. Therefore, we should select the BFSs that are the most helpful in reducing the distance between the known match cost bounds and the exact costs of the candidate matches. Since the exact costs are not known, we use the difference between the upper bound and the lower bound, or the *bound gap*, to approximate the distance from the exact costs. For a candidate match v , the gap of the known match cost bounds is computed as follows.

$$C^{K+}(v) - C^{K-}(v) = \sum_{q^s \in V_Q^S} \{\delta^+(q^s, v) - \delta^-(q^s, v)\} \quad (12)$$

From the equation, it can be seen that the BFS having q^s as a source node contributes the amount of $\delta^+(q^s, v) - \delta^-(q^s, v)$ to the bound gap. In order to approach the exact known match costs as quickly as possible, we select the BFS whose associated closeness scores contribute the most to the bound gaps of the candidate matches. Therefore, the priority of a BFS starting at an anchor node $\phi(q^s)$ is assigned by computing its total contribution to the bound gaps of all the candidate matches as follows.

$$\mathcal{P}_1(\phi(q^s)) = \sum_{v \in \mathcal{M}^q} \delta_{q^s, q}^+(\phi(q^s), v) - \delta_{q^s, q}^-(\phi(q^s), v) \quad (13)$$

where $\widetilde{\mathcal{M}}^q$ is the set of q 's candidates having the lower bound costs smaller than $C^{K+}(v_{k^*}, q)$. The nodes in $\widetilde{\mathcal{M}}^q$ are those eligible to be the top- k^* candidates of q in the future.

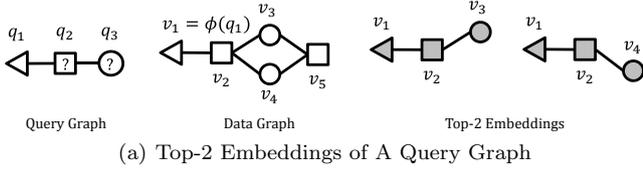
The priority of the BFSs in Phase 2 is assigned with a similar idea, but in Phase 2, we are interested in reducing the distance between the embedding match cost bounds and the exact costs of the candidate embeddings. The embedding match cost bound gaps of an embedding f is computed as follows.

$$C^+(f) - C^-(f) = \sum_{q_i \in V_Q} \sum_{q_j \in V_Q} \delta_{q_i, q_j}^+(f(q_i), f(q_j)) - \delta_{q_i, q_j}^-(f(q_i), f(q_j))$$

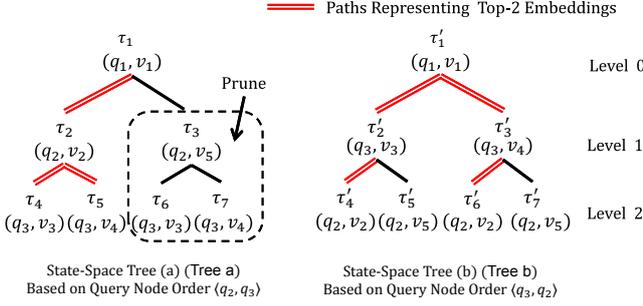
Here the BFS starting from $f(q_i)$ contributes $\delta_{q_i, q_j}^+(f(q_i), f(q_j)) - \delta_{q_i, q_j}^-(f(q_i), f(q_j))$ to the bound gap. Taking into account all the candidate embeddings, the priority $\mathcal{P}(u)$ of the BFS starting from u is computed as in the following equation.

$$\mathcal{P}_2(u) = \sum_{f \in \widetilde{\mathcal{F}}} \sum_{\substack{q_i \in V_Q \\ f(q_i) = u}} \sum_{q_j \in V_Q} \delta_{q_i, q_j}^+(u, f(q_j)) - \delta_{q_i, q_j}^-(u, f(q_j)) \quad (14)$$

where $\widetilde{\mathcal{F}}$ is the set of candidate embeddings whose lower bounds are smaller than $C^+(f_k)$. A high priority indicates that the BFS can efficiently reduce the distance between the bounds and the exact match costs of the candidate embeddings. Evaluating Eq. (14) requires enumerating the entire embedding set, which is time-costly. We next show a branch-and-bound method to optimize the enumeration.



(a) Top-2 Embeddings of A Query Graph



(b) Search Trees for Identifying Top-2 Embeddings

Figure 4. An Example of Search Tree

4.5 Optimizing Embedding Enumeration

The termination check and the BFS prioritization in Phase 2 require the embeddings to be enumerated from the candidate match sets of the query nodes. Since each query node has k^* candidates, the size of the candidate embedding set is $k^*|V_Q^U|$. When k^* or $|V_Q^U|$ is large, this process can be time-costly. In this section, we explain how to apply a *branch-and-bound* technique to accelerate the computation.

4.5.1 Modeling Search Space by Search Tree

With the branch-and-bound technique, the candidate embedding enumeration is modeled as a *search tree*. The search tree has $|V_Q^U| + 1$ levels. The top level of the tree (level 0) contains a *root* node that represents the mappings from specific nodes to anchor nodes. In the other levels, each tree node represents a mapping that maps a query node to one of its candidate matches. The path from the root node to a tree node τ represents a mapping function f_τ . If τ is a leaf node, f_τ is a *complete embedding*. Otherwise, f_τ is a *partial embedding*. The search for the top- k embeddings starts from the root node and progressively generates the other tree nodes. All the candidate embeddings are evaluated when all the leaf nodes are generated.

Example 2. We illustrate the search tree's structure by Figure 4. Figure 4(a) shows a query graph and its top-2 embeddings. These embeddings can be identified by **Tree A** or **Tree B** that are shown in Figure 4(b). Both of the trees have 3 levels and 7 tree nodes, but they are constructed based on different query node orders. Since each tree level is associated with a query node, the query node order determines the way of generating a tree.

From Figure 4, we observe that **Tree A** has higher pruning efficiency than **Tree B**. In **Tree A**, the sub-tree rooted at τ_3 can be pruned since the leaf nodes corresponding to the top-2 embeddings are rooted at τ_2 . However, in **Tree B**, no tree node can be pruned. We next describe how to prune the branch of **Tree A** and how to generate a search tree that has higher pruning efficiency.

4.5.2 Pruning Search Tree

We describe how the tree branches can be pruned during generating the tree. The following definitions are used throughout this sub-section. Suppose f_τ is a partial embedding. (1) If f_τ defines the match of query node q , q is *instantiated* by f_τ . (2) If a leaf node τ' is a descendant of τ , $f_{\tau'}$ is a *descendant embedding* of f_τ . For example, in Figure 4(b), tree node τ_2 of **Tree A** indicates a partial embedding f_{τ_2} that instantiates q_1 and q_2 . f_{τ_2} has two descendant embeddings f_{τ_4} and f_{τ_5} .

Having these definitions, the pruning process is shown as follows. We maintain a set of the top- k embeddings seen so far. The match cost of the k^{th} best embedding in the set, denoted by λ , guides us to perform pruning. To determine whether the sub-tree rooted at tree node τ can be pruned, we evaluate the minimum match cost of f_τ 's descendant embeddings. If the lower bound of the match costs is larger than λ , these descendant embeddings cannot be the top- k embeddings, so this branch can be pruned. For example, in **Tree A**, if τ_4 and τ_5 are generated first, then λ is 0. The sub-tree rooted at τ_3 is pruned since $\mathcal{C}(f_{\tau_6})$ and $\mathcal{C}(f_{\tau_7})$ are greater than 0.

Now we show how to evaluate the lower bound of the match costs of f_τ 's descendant embeddings, denoted by $\check{\mathcal{C}}(f_\tau)$. Suppose $f_{\tau'}$ is a descendant embedding of f_τ . Let the set of the query nodes instantiated in $f_{\tau'}$ be V_Q^τ . For all $q \in V_Q^\tau$, we know that $f_{\tau'}(q) = f_\tau(q)$. Let $\tilde{V}_Q^\tau = V_Q - V_Q^\tau$, which is the set of the non-instantiated nodes. We have following lower bounds of difference scores.

$$\delta_{q_i, q_j}(f_{\tau'}(q_i), f_{\tau'}(q_j)) \geq \min_{u \in \mathcal{M}^{q_j}} \delta_{q_i, q_j}(f_\tau(q_i), u) \quad (15)$$

where q_i is instantiated but q_j is not instantiated, and

$$\delta_{q_i, q_j}(f_{\tau'}(q_i), f_{\tau'}(q_j)) \geq \min_{\substack{u \in \mathcal{M}^{q_i} \\ v \in \mathcal{M}^{q_j}}} \delta_{q_i, q_j}(u, v) \quad (16)$$

where neither q_i nor q_j is instantiated. Base on the definition of the embedding matching cost (see Equation (4)), we can obtain $\check{\mathcal{C}}(f_\tau)$ as follows.

$$\begin{aligned} \check{\mathcal{C}}(f_\tau) &= \sum_{q_i \in V_Q^\tau} \sum_{q_j \in V_Q^\tau} \delta_{q_i, q_j}(f_\tau(q_i), f_\tau(q_j)) \\ &+ 2 \sum_{q_i \in V_Q^\tau} \sum_{q_j \in \tilde{V}_Q^\tau} \min_{u \in \mathcal{M}^{q_j}} \delta_{q_i, q_j}(f_\tau(q_i), u) \\ &+ \sum_{q_i \in \tilde{V}_Q^\tau} \sum_{q_j \in \tilde{V}_Q^\tau} \min_{\substack{u \in \mathcal{M}^{q_i} \\ v \in \mathcal{M}^{q_j}}} \delta_{q_i, q_j}(u, v) \end{aligned} \quad (17)$$

Pruning Condition. Given a tree node τ , if $\check{\mathcal{C}}(f_\tau) \geq \lambda$, then the sub-tree rooted at τ can be pruned.

4.5.3 Generating Search Tree

The way of generating the state-space tree affects the pruning efficiency since it affects the value of $\check{\mathcal{C}}(f_\tau)$ and λ . First, the value of $\check{\mathcal{C}}(f_\tau)$ affected by the *query node order* that maps the query nodes to the tree levels. An example of this effect has been shown in Figure 4(b). Second, the value of λ is affected by how to expand the tree nodes. If the leaf nodes that correspond to the top- k embeddings are visited earlier, then we may have lower λ and more branches can be pruned.

Mapping query nodes to tree levels. Since each non-root level corresponds to a query node, the query node order

determines how to map a query node to a tree level. We propose a heuristic method for determining the query node order.

To start with, we explain the reason for the higher pruning efficiency of Tree A. In Tree A level 1 determines q_2 's matches. Comparing v_2 and v_5 , v_2 is closer to v_1 , so $\check{C}(f_{\tau_2})$ is smaller than $\check{C}(f_{\tau_3})$ such that both the top-2 embeddings are rooted at τ_2 . However, in Tree B level 1 determines q_3 's matches. Since v_3 is as good as v_4 , no tree nodes can be pruned due to $\check{C}(f'_{\tau_2}) = \check{C}(f'_{\tau_3}) = 0$. This observation motivates us to assign high ranks to the query nodes whose candidate matches have various *qualities*.

Our heuristic for measuring the quality of a candidate match is to approximate the candidate match's node match cost. A match's quality consists of the match's known match cost and the average unknown match cost. Let v be a candidate match of query node q_i and $C'(v, q_i)$ denote the quality of v . We have

$$C'(v, q_i) = C^K(v, q_i) + \sum_{q_j \in V_Q^U} \frac{\sum_{u \in \mathcal{M}^{q_j}} \delta_{q_i, q_j}(v, u)}{|\mathcal{M}^{q_j}|} \quad (18)$$

where \mathcal{M}^{q_j} is the candidate match set of query node q_j . We measure the variance of the candidates' qualities for each query node. The query nodes are sorted by the variance in descending order. Then, the tree is constructed from top to bottom according to the query node order.

Expanding search tree. The order of expanding the tree nodes can also affect the pruning efficiency. For example, in Tree A, if τ_3 is expanded earlier than τ_2 , then no branch can be pruned.

Our approach is based on the A^* class of search algorithm. The idea is to maintain a priority queue of the frontier of the state-space tree. The tree node τ with smaller $\check{C}(f_\tau)$ has higher priority. When expanding the next tree node, our approach selects the best tree node from the top of the priority queue. Then, the non-leaf children of the tree node are pushed into the priority queue. If the children are leaf nodes, we evaluate the match costs of the corresponding embeddings and update the threshold λ . The tree nodes in the frontier which satisfy $\check{C}(f_\tau) \geq \lambda$ are pruned. The expansion is terminated when the priority queue is empty.

4.5.4 Applying to Our Algorithm

We outline how to apply the technique in each step of our algorithm. For the termination check in Phase 2, in the first step, we need to identify $C^+(f_k)$, the k^{th} smallest match cost upper bound. During the search, a tree branch is pruned if the lower bound of $C^+(\cdot)$ of its descendant embeddings is larger than the k^{th} smallest $C^+(f_k)$ seen so far. In the second step, we need to check whether there are more than k embeddings whose match cost lower bounds, $C^-(\cdot)$, are lower than $C^+(f_k)$. In this step, a tree branch is pruned if the lower bound of $C^-(\cdot)$ of its descendant embeddings is larger than $C^+(f_k)$. The search can stop as soon as more than k embeddings with the match cost lower bounds smaller than $C^+(f_k)$ are found.

For the BFS prioritization, evaluating Eq. (14) requires fully enumerating the candidate embedding set. To reduce the computation, in our implementation, we only consider the candidate embeddings with small lower bounds. We use the search tree to find P embeddings with the match cost lower bounds smaller than $C^+(f_k)$ and use these P embeddings to compute the priorities.

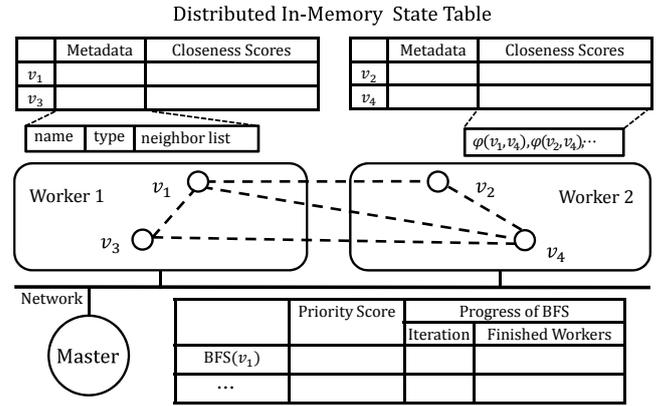


Figure 5. System Architecture

5 DISTRIBUTED IMPLEMENTATION

Since a single machine has limited memory and computation resources, we design a distributed system to store the web-scale information networks and execute our algorithm.

5.1 System Overview

Our system is implemented based on a distributed graph computation framework, Piccolo [21], which consists of one master process and multiple worker processes. Fig. 5 shows the architecture of the system. In our system, the master coordinates the workers to perform the bound refinement and the termination check in the two phases of the algorithm.

Distributed in-memory state table. For efficiency, the data graph is stored in an *in-memory state table* that is distributed across the workers. Each data node corresponds to one row in the table. The row associated with a node v , $\text{row}(v)$, stores the metadata of node v including the node name, the node type, and the neighbors' keys. Additionally, $\text{row}(v)$ contains a hash map that stores the closeness scores associated with node v . Each worker stores a partition of the state table. In our implementation, we assign data nodes to the workers by applying a hash function to the node keys, but other partition strategies can also be applied. With the distributed in-memory state table, we next show how to perform the distributed bound refinement and the distributed termination check.

5.2 Distributed Bound Refinement

For the bound refinement, a total of $|V_Q^S| + k^*|V_Q^U|$ BFSes are performed simultaneously (from Phase 1 and Phase 2). To efficiently coordinate multiple workers to perform the BFSes is non-trivial. We explain our key designs as follows.

Computing priorities of BFSes. To schedule the BFSes, the master maintains the priority score of each BFS. The priority scores, as in Eq. (13) and Eq. (14), are computed distributedly. The computation is similar to the distributed termination check, which is explained in Section 5.3.

Expanding one BFS. The workers expand the BFS with the highest priority each time. Each iteration of a BFS finishes when all workers have finished processing for that iteration. The master keeps track of the workers that have finished the iteration and notifies the workers to expand the next BFS when the iteration is completed.

Aggressive expansion for multiple BFSes. When expanding a BFS, an iteration is not completed until all the workers have finished. This synchronization can degrade

the efficiency, especially when the workloads are imbalanced among the workers. To solve this problem, we propose aggressive expansion. When a worker is idle, we let the worker aggressively expand the next BFS in the priority queue without waiting for the previous BFS’s completion.

Multiple message queues. With aggressive expansion, an iteration may take longer time since multiple BFSes compete for the computation resources. We give an example as follows. Suppose there are two BFSes, $BFS(v)$ and $BFS(u)$. $BFS(v)$ has the highest priority. When a worker, w , is working on $BFS(v)$, it may receive the messages for $BFS(u)$ from the workers that have finished processing $BFS(v)$. Because w have to process the messages of both $BFS(u)$ and $BFS(v)$, the iteration of $BFS(v)$ could be slowed down. Our solution is to implement multiple message queues in each worker, one message queue for each BFS. When processing messages, the message queues with higher priorities are processed earlier. With this design, the processing of the BFS with the highest priority is not interfered by the other BFSes.

5.3 Distributed Termination Check

We now discuss how to perform termination checks when the closeness scores are distributed across the workers.

Distributed termination check for Phase 1. We distribute the termination check workload among the workers. Each worker first finds its local top- k^* candidates (according to the upper bound costs) and send them to the master. The master finds the global top- k^* candidates among the local top- k^* candidates. Additionally, each worker is responsible for checking the top- k condition in Eq. (9) on its local nodes. Using the results from the workers, the master determines whether the real top- k^* candidates are found.

Distributed termination check for Phase 2. In Phase 2, the termination check is performed by the master. The master aggregates the closeness score bounds of the candidate pairs from the workers and performs the termination check by generating the search trees, as in Section 4.5. The tree generation is accelerated by using multiple threads. Each thread maintains a partition of the tree’s frontier and expands the subtrees rooted at its assigned tree nodes.

6 EVALUATION

6.1 Experiment Settings

System. We performed our experiments on two clusters: a local cluster and an Amazon EC2 cluster. The local cluster consisted of four machines connected by a 1Gb Ethernet switch. Each machine had 16GB RAM and one 1.86GHz Intel Xeon E5502 CPU with four cores. The Amazon EC2 cluster consisted of 100 m3.large instances, each of which had 2 vCPUs and 7.5GB RAM. All the experiments were implemented using C++.

Datasets. Two real-life networks, DBLP and Freebase, and several synthetic graphs were used in our experiments. (1) The DBLP graph [2] is a bibliography network containing three types of nodes: author, paper, and venue. There are 3.5 million nodes and 9.5 million edges in the DBLP graph. (2) A subset of Freebase containing film information was used in our experiments. The film dataset has five types of nodes: film, director, actor, perform, and role. There are 1.4 million nodes and 1.8 million edges. (3) Synthetic graphs were generated by duplicating the DBLP dataset. Each node in DBLP has several copies in the synthetic graphs. We

	Pattern	Content
Q1	Chain	Find an author of CIDR11 and an author of ISMB08, who have co-authored.
Q2	Star	Find an author who published papers in SIGMETRICS, IMC, and ICML between the years 2010 and 2012.
Q3	Circuit	Find two films that are directed by the director of ‘A Beautiful Mind’ and performed by an actor who performed in ‘The Mask’.

Table 2. Real-life Queries

varied the duplication factor from 30 to 300 to produce the synthetic graphs with different sizes ranging from 100 million nodes to 1 billion nodes.

Query graphs. Real-life and synthetic queries were used in the experiments. We selected the real-life queries that are representative of different patterns of query graphs, including a chain, a star, and a circuit. The real-life queries are listed in Table 2. Q1, Q2 are for DBLP. Q3 is for Freebase. The corresponding query graphs are shown in Fig. 6(a).

The synthetic queries were composed based on a subgraph of a data graph. The size of the synthetic query is specified by two parameters: the number of specific nodes ($|V_Q^S|$) and the number of query nodes ($|V_Q^U|$), denoted in the form ($|V_Q^S|, |V_Q^U|$). To generate a query, we randomly selected a connected subgraph of specified size from a data graph and added noise to the query by inserting and deleting edges.

Parameter setting. We evaluated our index-free algorithm, GraB, and an index-based algorithm, GraB-Index. In our algorithms, α was set to 0.01. GraB-Index follows the algorithm framework shown in Section 3. In contrast to GraB, GraB-Index indexes the pairwise distances of nodes using the pruned labeling algorithm [5]. Since only the lengths of the shortest paths were indexed, N was set to 1 in GraB-Index. However, GraB considered the shortest path numbers and N was set to 99.

6.2 Answering Real-Life Queries

Here we show the effectiveness of our algorithm in answering real-life queries. The best embeddings obtained with our algorithm for the three queries in Table 2 are shown in Fig. 6(b). For Q1, there are multiple pairs of authors that can satisfy the query. Our algorithm found all of the authors and returned the exact matches of the query graph. For Q2, in DBLP no author published in all of the three conferences between 2010 and 2012, so there are no exact matches. However, our algorithm could still provide answers to the query with similar matches. As shown in Fig. 6(b), the algorithm found Ling Huang as the best answer because his coauthor published in SIGMETRICS 2010. Since they have coauthored ten papers, there are ten 3-hop paths between them. For Q3, the query does not conform with Freebase’s schema. However, our algorithm found two films, ‘Presidential Reunion’ and ‘How the Grinch Stole Christmas!’, as the best answer, which corresponds to the fact in real life. These results illustrate that our algorithm can provide answers to real-life queries effectively with both exact matches and similar matches.

6.3 Comparison with Existing Graph Matching Algorithms

We compared GraB-Index and GraB with four state-of-the-art graph matching algorithms, RDF-3X [20], Graph-Exploration-based subgraph matching (GE) [24], Ness [15], and NeMa [16]. RDF-3X is an index-based search engine that identifies exact matches on RDF documents. The RDF documents of

is sufficient for finding the real top-10 embeddings. The accuracy of the top- k answers was measured as follows. Given the real top-10 embeddings, let C_{real}^k be the matching cost of the 10th embedding. From the top-10 embeddings found by GraB, let N_{acc} be the number of embeddings that have match costs less than or equal to C_{real}^k . These embeddings are considered to be the accurate answers. The accuracy of the top-10 embeddings is computed as $\frac{N_{acc}}{10}$.

We generated 4 groups of queries for different query sizes. Each group contained 10 queries. We increased k^* from 5 to 30. For each setting of k^* , the average accuracy of each query group is shown in Fig. 7. It can be seen that 1) the accuracy increased significantly when k^* was changed from 5 to 10; 2) when $k^* = 10$, in most query groups, over 80% of the answers were accurate; 3) when $k^* = 30$, all the real top-10 answers were found. However, when k^* is larger, the running time of query processing is longer. Fig. 8 shows the average running time of each query group. When the query size is larger, the impact of k^* on the running time is more significant. For example, it requires 12.01 seconds to answer queries in group (5,10) in DBLP when k^* is 10, but when k^* is 20 and 30, it takes 18.10 and 34.31 seconds, respectively. Therefore, taking into account the accuracy and the running time, we recommend setting k^* to be between k and $2k$. In the rest of this section, we let k^* equal to k .

6.5 Performance

We evaluated the performance of our algorithm by measuring the running time. The experiments were performed on the local cluster. For each query size, we generated 10 queries and measured the average running time.

Effects of bounding technique. We set k to 10 and evaluated the efficiency of the bounding technique by comparing our algorithm with the naive algorithm that computes the exact matching costs of the embeddings. The running time of the two algorithms is shown in Fig. 9. It can be seen that our algorithm is 100 to 400 times faster than the naive algorithm. This result shows that using bounds can effectively reduce the running time. Additionally, the result demonstrates that our index-free algorithm can answer the top-10 queries within a few seconds.

Effects of priority scheduling. We show how the BFS priority scheduling policy introduced in Section 4.4 affects the running time. We set k to 10 and compared the priority scheduling policy with a round-robin scheduling policy. The running time of the two policies is shown in Fig. 10. When the query graph is larger, the priority scheduling is more efficient. We observe that DBLP benefits more from using the priority scheduling. Because the DBLP network is denser than Freebase, the priority scheduling greatly helps to reduce the expensive cost of performing BFSes on DBLP.

Effects of varying k . We studied how the value of k affects the running time. Fig. 11 shows the running time of finding the top-5, the top-10, and the top-20 answers for each query group. Because the running time is mainly determined by k^* , and we set k^* to be equal to k , when k becomes larger, more embeddings have to be evaluated in Phase 2. Therefore, the running time is longer when k is larger.

6.6 Scalability

Scalability with worker number. We evaluated how our algorithm scales with the worker number using the Amazon EC2 cluster. For this experiment, the 100-million-node syn-

thetic graph was used and k was set to 10. The worker number was varied from 20 to 100. We measured the average running time for each query group. Fig. 12(a) shows the running time for different numbers of workers. For the small queries, *i.e.*, group (4,2) and group (6,3), our algorithm could answer the queries within about 3 seconds. It can be seen that when the worker number increases, the running time of answering small queries is not significantly reduced. This is because using more workers introduce more communication overhead. However, for the large queries, *i.e.*, group (8,4) and group (10,5), the running time decreases as the worker number becomes larger. This is because the computation for answering larger queries is more intensive; using more workers could efficiently reduce the running time.

Scalability with graph size. To evaluate the scalability with graph size, we utilized 100 workers in the Amazon cluster. The graph size was varied from 200 million to 1 billion nodes. We let k be 10 and measured the average running time for each query group. Fig. 12(b) shows the running time for answering the queries on different sizes of data graph. The running time increases slowly with the data graph size. This result demonstrates the ability of our distributed system to scale for billion-node graphs.

7 RELATED WORK

Graph matching. Graph matching has been extensively studied over the past decades [23]. Some recent works [7, 9, 10, 12, 15, 16, 30] studied how to perform subgraph matching on large graphs. However, the proposed approaches are not suitable for billion-node graphs due to expensive indexing costs. The graph-exploration-based subgraph matching algorithm by Sun et al. [24] finds exact matches in billion-node graphs. In contrast, our algorithm finds both exact matches and similar matches. Distributed graph pattern matching [18] finds a one-to-many mapping between query nodes and data nodes, but our problem is to find a one-to-one mapping, which is more common in real life.

Top- k query. Top- k queries request the best- k answers for a given query [14]. The Threshold Algorithm and Fagin’s Algorithm are commonly used to answer top- k queries [8]. However, they require precomputing sorted lists to derive the upper bounds and the lower bounds. For graph databases, the top- k graph matching problem has been studied in recent years [7, 10, 12, 15, 16], but as discussed earlier most existing works require building indices.

RDF query. RDF queries are important for discovering information on information networks. Some RDF query systems [1, 6, 20] have been developed. These systems build sophisticated indices on a single machine for query processing. In recent years, distributed RDF query systems [13, 26, 28] are proposed to support web-scale RDF graphs. However, the systems focus on finding the answers that exactly match queries. The flexible graph pattern queries have been studied in [22], which finds the answers similar to the RDF queries. The approach stores RDF triples on HDFS [3] and answers the queries by repeatedly joining intermediate results. In contrast, our approach does not produce a huge amount of intermediate join results.

Distributed graph computation. Since billion-node graphs are increasingly common, distributed graph computation frameworks [17, 19, 21, 29] have been developed to support graph operations, such as computing PageRank scores and computing shortest paths. Although our system is imple-

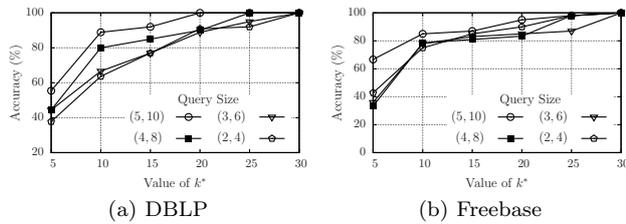


Figure 7. The Accuracy When Varying k^*

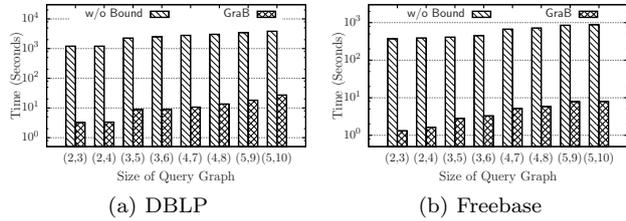


Figure 9. Efficiency of Bounding Technique

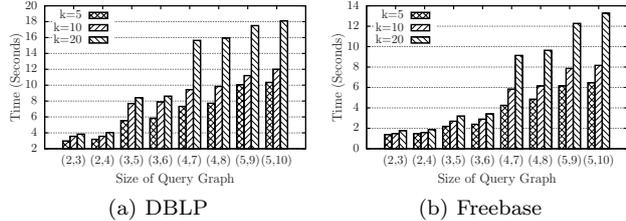


Figure 11. The Running Time When Varying K

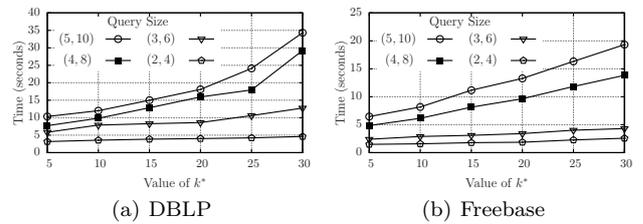


Figure 8. The Running Time When Varying k^*

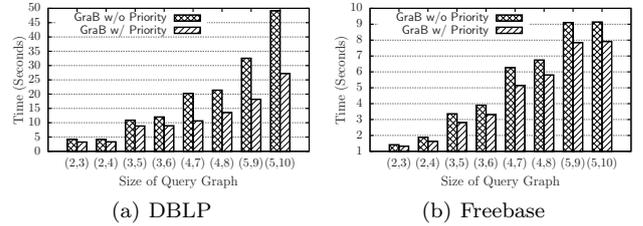


Figure 10. Efficiency of Priority Scheduling

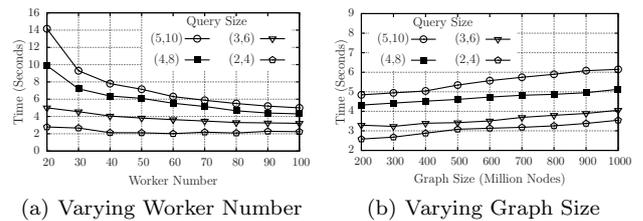


Figure 12. Scalability

mented based on Piccolo [21], it can also be implemented on any other frameworks that can support breadth-first search.

8 CONCLUSION

In this paper, we propose GraB, an index-free graph matching algorithm for answering top- k queries on web-scale information networks. Our approach provides meaningful answers via exact and similar matching. To obtain the answers quickly without indices, GraB utilizes a novel heuristic for selecting the top candidates of query nodes and computes the bounds of matching scores to derive the top- k answers instead of computing the exact matching scores. A distributed system for the algorithm is proposed to allow scalability. Our experiments demonstrate that the proposed approach can efficiently answer queries on billion-node information networks.

References

- [1] Apache Jena. <http://http://jena.apache.org/>.
- [2] DBLP. <http://www.informatik.uni-trier.de/~ley/db/>.
- [3] Hadoop Distributed File System. <http://hadoop.apache.org/>.
- [4] Linked Open Data. http://www.w3.org/wiki/SweoIG/Task_Forces/Community_Projects/LinkingOpenData.
- [5] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*, pages 349–360, 2013.
- [6] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: a scalable lightweight join query processor for rdf data. In *WWW*, pages 41–50, 2010.
- [7] J. Cheng, X. Zeng, and J. X. Yu. Top- k graph pattern

matching over large graphs. In *ICDE*, pages 1033–1044, 2013.

- [8] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, pages 614–656, 2003.
- [9] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractable to polynomial time. *PVLDB*, pages 264–275, 2010.
- [10] W. Fan, X. Wang, and Y. Wu. Diversified top- k graph pattern matching. *PVLDB*, pages 1510–1521, 2013.
- [11] Google. "freebase data dumps". <https://developers.google.com/freebase/data>.
- [12] M. Gupta, J. Gao, X. Yan, H. Cam, and J. Han. Top- k interesting subgraph discovery in information networks. In *ICDE*, pages 820–831, 2014.
- [13] J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs., 2011.
- [14] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top- k query processing techniques in relational database systems. *ACM Comput. Surv.*, 2008.
- [15] A. Khan, N. Li, X. Yan, Z. Guan, S. Chakraborty, and S. Tao. Neighborhood based fast graph search in large networks. In *SIGMOD*, pages 901–912, 2011.
- [16] A. Khan, Y. Wu, C. C. Aggarwal, and X. Yan. Nema: Fast graph search with label similarity. *PVLDB*, pages 181–192, 2013.
- [17] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, pages 716–727, 2012.

- [18] S. Ma, Y. Cao, J. Huai, and T. Wo. Distributed graph pattern matching. In *WWW*, pages 949–958, 2012.
- [19] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.
- [20] T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. *PVLDB*, pages 647–659, 2008.
- [21] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, pages 293–306, 2010.
- [22] P. Ravindra and K. Anyanwu. Scalable processing of flexible graph pattern queries on the cloud. In *WWW (Companion Volume)*, pages 169–170, 2013.
- [23] D. Shasha, J. T.-L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, pages 39–52, 2002.
- [24] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, pages 788–799, 2012.
- [25] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In *ICDE*, pages 963–972, 2008.
- [26] R. Verborgh, M. V. Sande, P. Colpaert, S. Coppens, E. Mannens, and R. V. de Walle. Web-scale querying through linked data fragments. In *LDOW*, 2014.
- [27] X. Yan, F. Zhu, P. S. Yu, and J. Han. Feature-based similarity search in graph structures. *ACM Trans. Database Syst.*, pages 1418–1453, 2006.
- [28] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. *PVLDB*, pages 265–276, 2013.
- [29] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Priter: A distributed framework for prioritizing iterative computations. *IEEE Trans. Parallel Distrib. Syst.*, pages 1884–1893, 2013.
- [30] L. Zou, L. Chen, and M. T. Özsu. Distancejoin: Pattern match query in a large graph database. *PVLDB*, pages 886–897, 2009.