Keyword Search over Knowledge Graphs via Static and Dynamic Hub Labelings

Yuxuan Shi
National Key Laboratory for Novel Software Technology,
Nanjing University, China
Bosch Center for AI,
Robert Bosch GmbH, Germany
yxshi@smail.nju.edu.cn

Gong Cheng
National Key Laboratory for Novel Software Technology,
Nanjing University, China
gcheng@nju.edu.cn

Evgeny Kharlamov
Bosch Center for AI,
Robert Bosch GmbH, Germany
evgeny.kharlamov@de.bosch.com
Department of Informatics,
University of Oslo, Norway
evgeny.kharlamov@ifi.uio.no

ABSTRACT

Keyword search is a prominent approach to querying Web data. For graph-structured data, a widely accepted semantics for keywords is based on group Steiner trees. For this NP-hard problem, existing algorithms with provable quality guarantees have prohibitive run time on large graphs. In this paper, we propose practical approximation algorithms with a guaranteed quality of computed answers and very low run time. Our algorithms rely on Hub Labeling (HL), a structure that labels each vertex in a graph with a list of vertices reachable from it, which we use to compute distances and shortest paths. We devise two HLs: a conventional static HL that uses a new heuristic to improve pruned landmark labeling, and a novel dynamic HL that inverts and aggregates query-relevant static labels to more efficiently process vertex sets. Our approach allows to compute a reasonably good approximation of answers to keyword queries in milliseconds on million-scale knowledge graphs.

CCS CONCEPTS

• **Mathematics of computing → Graph algorithms**; **Approximation algorithms**; • **Theory of computation** → *Shortest paths.*

KEYWORDS

knowledge graph, keyword search, group Steiner tree, hub labeling

ACM Reference Format:
Yuxuan Shi, Gong Cheng, and Evgeny Kharlamov. 2020. Keyword Search over Knowledge Graphs via Static and Dynamic Hub Labelings. In *Proceedings of The Web Conference 2020 (WWW '20), April 20–24, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3366423.3380110

1 INTRODUCTION

Keyword search allows users to query Web data without a prior knowledge of specialized query languages. A keyword query is a set of words posed by a user that should be matched to the data. Relevant data fragments are then extracted and presented to the user in an appropriate format as answers. The exact way of matching keywords, extracting data, and composing answers depends on the format of the underlying data and the semantics of query answering. Keyword search has been extensively studied for various databases [51] and has recently attracted renewed attention in the context of *knowledge graphs* (KGs) [12, 21, 35, 44].

Problem. In a nutshell, a common type of semantics for keyword queries over graph data is to match each keyword to a vertex of the graph and to extract trees of minimum weight that contain these vertices, known as minimum-weight Steiner trees [49]. More formally, given an edge-weighted data graph and a keyword query, one firstly finds for each keyword the matching set of vertices in the graph, i.e., all the vertices where the keyword can be matched, and then finds a tree in the graph that spans the matching sets, i.e., contains at least one vertex from each matching set, and that minimizes the total edge weight. This optimization problem is the well-known *group Steiner tree* (GST) problem [26]. Note that keywords are also allowed to be matched to edges. Edge matches can be straightforwardly transformed into vertex matches via graph subdivision, and be processed as vertex matches.

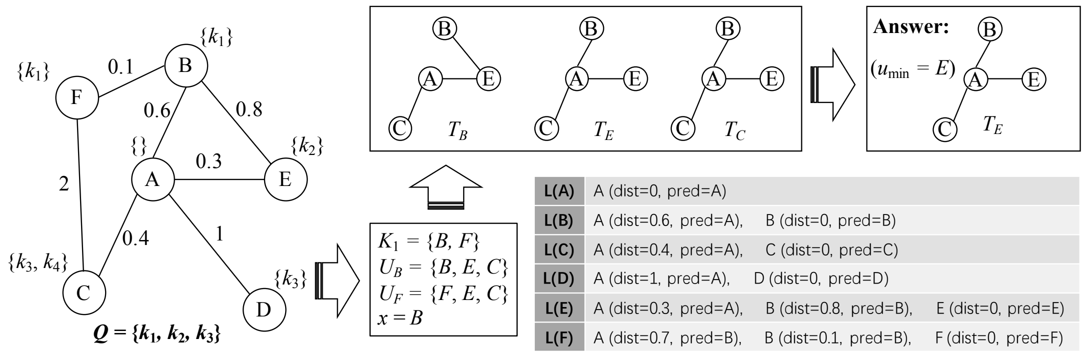
Challenge. Computing answers to keyword queries under the GST semantics is computationally demanding. The GST problem is known to be NP-hard. Moreover, existing approximation algorithms that have provable quality guarantees also have prohibitively high run time for large graphs. As shown in [13], existing methods [9, 15, 27] take thousands of seconds to answer a keyword query on the graph version of IMDb which is not that large—containing 1.6M vertices and 6.1M edges.

This poses a significant challenge for developing efficient keyword search systems over KGs which we see in this paper as collections of interconnected and annotated entities. KGs have become increasingly popular in recent years, and they can be huge. The well-known DBpedia KG [36] contains millions of entities. Google's KG has one billion entities [41]. KGs in industry are also of a huge scale [6, 24, 29–33]. Existing GST-based algorithms for keyword search will not scale for them.

Our Approach. To meet the challenge, we propose practical algorithms that compute approximations of answers to keyword queries under the GST semantics. Our algorithms feature both the quality of approximation and high efficiency. *They compute approximations with guaranteed quality in milliseconds on millions of vertices.* The algorithms rely on *Hub Labeling* (HL) [1], a structure that labels each vertex in a graph with a list of reachable vertices, for efficient computation of distances and shortest paths. We devise two new HLs to realize the significant improvement in efficiency.

WWW '20, April 20–24, 2020, Taipei, Taiwan

ACM ISBN 978-1-4503-7023-3/20/04.
https://doi.org/10.1145/3366423.3380110



L(A)	A (dist=0, pred=A)
L(B)	A (dist=0.6, pred=A), B (dist=0, pred=B)
L(C)	A (dist=0.4, pred=A), C (dist=0, pred=C)
L(D)	A (dist=1, pred=A), D (dist=0, pred=D)
L(E)	A (dist=0.3, pred=A), B (dist=0.8, pred=B), E (dist=0, pred=E)
L(F)	A (dist=0.7, pred=B), B (dist=0.1, pred=B), F (dist=0, pred=F)

Figure 1: A running example of KeyKG.

Our first algorithm KeyKG selects a matching vertex for each keyword and finds a tree that spans these vertices. For g keywords, KeyKG is a $(g-1)$-approximation algorithm, i.e., the total edge weight of a computed GST is at most $(g-1)$ times the minimum weight. This approximation ratio is acceptable because, on the one hand, g is usually a very small integer in practice and, on the other hand, the problem has $O(\ln g)$ inapproximability [26], i.e., there is no polynomial-time algorithm that can approximate the problem with any ratio in $O(\ln g)$. Our approximation ratio is obtained thanks to the selection of matching vertices that are close to each other and the constitution of trees with shortest paths. For efficient online computation of distances and shortest paths, we devise a HL that improves the existing pruned landmark labeling [3] with a new heuristic based on betweenness centrality. This HL by convention is static as it is offline constructed and invariant to queries. On large KGs, KeyKG with a static HL performs at least an order of magnitude faster than the state of the art [27, 38], and computes reasonably good answers with comparable quality.

Our second algorithm KeyKG$^+$ extends KeyKG by using a novel type of HL. This proposed HL is dynamic as it is online constructed when processing a concrete query, by inverting and aggregating certain query-relevant static labels. It helps to reduce repeated operations in KeyKG that are performed in the computation of distances on sets of vertices with a conventional static HL. Despite the extra time for online construction, using dynamic HLs brings about several orders of magnitude improvement in overall efficiency. In particular, on DBpedia, KeyKG$^+$ computes the same answers as KeyKG only in milliseconds, making KeyKG$^+$ a very practical solution.

Contributions. We summarize our contributions as follows.

- For keyword search over KGs under the GST semantics, we design approximation algorithms with provable quality guarantees. These practical algorithms compute reasonably good answers on million-scale KGs only in milliseconds.
- To support efficient online computation of distances and shortest paths in our algorithms, we propose a novel query-relevant dynamic HL which achieves significant improvement in overall performance. We also devise a new static HL using an effective heuristic that outperforms existing HLs.

The remaining sections are organized as follows. Section 2 formulates the problem. Section 3 introduces KeyKG and our static HL. Section 4 describes KeyKG$^+$ and our dynamic HL. Experiments are presented in Section 5. We discuss related work in Section 6, and conclude the paper in Section 7.

2 PROBLEM FORMULATION

We define necessary terms and then formulate the problem to solve.

Knowledge Graph. A knowledge graph (KG) represents a collection of interconnected and annotated entities. For conciseness, we only formulate the essential part of a KG. Formally, it is a simple undirected graph $G = \langle V, E \rangle$ where V is a finite set of n vertices representing entities (i.e., $|V| = n$) and $E \subseteq V \times V$ is a finite set of unordered pairs of vertices as undirected edges representing relations between entities. Self-loops, parallel edges, and edge directions are ignored since they are not essential to our approach. We assume a weighting function $\texttt{wt} : E \mapsto \mathbb{R}^{0+}$ that maps edges into non-negative real numbers. Small weights indicate great importance. The exact weighting function is not our concern. As our running example in this paper, the left of Fig. 1 shows a KG where six vertices are connected by seven edges with weights. For conciseness, vertices and edges are denoted by symbols, but in real KGs they are annotated with meaningful text.

Graph Terminology. The degree of a vertex is its number of incident edges. We define a path in a graph in a standard way. For path p, its length is the sum of the weights of all the edges in p, denoted by $\texttt{len}(p)$. The distance between two vertices $u, v \in V$, denoted by $\texttt{dist}(u, v)$, is the length of a shortest path connecting u and v in G, or ∞ if no such path exists.

Keyword Mapping. Let $\mathbb{K}$ be the set of all keywords. We assume a retrieval function $\texttt{hits} : \mathbb{K} \mapsto 2^V$ that maps keywords to subsets of vertices from V. The exact retrieval function depends on how KGs are implemented and is outside the scope of our research. As an example, in Fig. 1 each vertex is associated with a set of keywords that can be mapped from, based on which we define

$$\texttt{hits}(k_1) = \{B, F\},\ \texttt{hits}(k_2) = \{E\},\ \texttt{hits}(k_3) = \{C, D\},\ \texttt{hits}(k_4) = \{C\}\,.$$

We map keywords to vertices but our approach can be straightforwardly extended to support edge matches via graph subdivision. The subdivision of an edge (u, v) yields a new vertex w and replaces the edge (u, v) by two edges (u, w) and (w, v). Edge matches are then transformed into vertex matches. To simplify the presentation in the paper, we omit edge matches in our problem formulation.

Keyword Query. A keyword query $Q \subseteq \mathbb{K}$ is a finite set of keywords. Given g keywords $Q = \{k_1, \ldots, k_g\}$, for the ease of notation we write $\texttt{hits}(k_i)$ as K_i for $1 \leq i \leq g$ and call them keyword vertices. Given $G = \langle V, E \rangle$, an answer to Q over G is a group Steiner tree (GST) denoted by $T = \langle V_T, E_T \rangle$ such that

(1) $V_T \subseteq V$, $E_T \subseteq E$, and T is a tree,
(2) V_T contains at least one vertex from each K_i for $1 \leq i \leq g$, i.e., $V_T \cap K_i \neq \emptyset$, and
(3) the weight of T, defined as $\texttt{WT}(T) = \sum_{e \in E_T} \texttt{wt}(e)$, is the minimum among all trees satisfying conditions (1) and (2).

Conditions (1) and (2) define a GST. Condition (3) requires it to have the minimum weight. Computing a minimum-weight GST is NP-hard and has $O(\ln g)$ inapproximability [26]. We aim to approximate a minimum-weight GST as an answer to a keyword query over a KG. For example, consider a keyword query $Q = \{k_1, k_2, k_3\}$ in Fig. 1 where $K_1 = \{B, F\}$, $K_2 = \{E\}$, and $K_3 = \{C, D\}$. The tree T_E shown in the top-right corner of the figure is a possible answer since it contains $B \in K_1$, $E \in K_2$, and $C \in K_3$.

3 ALGORITHM KeyKG WITH STATIC HL

As the problem is NP-hard, in this section we present our first approximation algorithm referred to as KeyKG. We describe KeyKG and analyze its approximation ratio in Section 3.1. The algorithm relies on two subroutines: getD for computing distances, and getSP for computing shortest paths. To support efficient computation of getD and getSP, in Section 3.2 we present a new implementation of Hub Labeling, which is an offline constructed index structure. Finally, in Section 3.3 we analyze the run time of KeyKG.

3.1 Algorithm KeyKG

KeyKG is presented in **Algorithm 1**. It finds a GST in a KG that spans g sets of keyword vertices. In a nutshell, KeyKG first greedily selects a set of keyword vertices that are close to each other, denoted by U_x, which contains one vertex from each K_i for $1 \leq i \leq g$ (lines 1–8). Then KeyKG greedily finds a GST to span U_x, denoted by $T_{u_{\min}}$, which is iteratively expanded with shortest paths (lines 9–18).

Specifically, for each vertex $v_1 \in K_1$ (line 1), KeyKG finds a vertex v_i in each other K_i with the minimum distance from v_1 (lines 2–4). Let U_{v_1} be the set of all such vertices v_i (including v_1), and let W_{v_1} be the sum of their distances from v_1 (lines 5–6). Each vertex $v_1 \in K_1$ has a corresponding W_{v_1}. Let $x \in K_1$ be a vertex corresponding to the smallest value of W_{v_1} (line 8). As a result, U_x contains one vertex from each K_i for $1 \leq i \leq g$, and these vertices are selected because *they are relatively close to each other*. Therefore, a GST that spans these vertices may have a small weight.

The remainder of the algorithm constructs a GST T_u starting from each vertex $u \in U_x$, and selects one that has the minimum weight among these $|U_x|$ GSTs. Specifically, each T_u is initialized with a single vertex u (lines 9–10). Then iteratively until T_u spans U_x (line 11), a vertex $s_{\min}$ inside T_u and a vertex $t_{\min} \in U_x$ outside T_u are found such that they have the smallest distance (line 12). A shortest path p between $s_{\min}$ and $t_{\min}$ is found and added to T_u (lines 13–14). This *greedy expansion with shortest paths* may produce a GST having a small weight. Each vertex $u \in U_x$ has a corresponding T_u. Let $u_{\min} \in U_x$ be a vertex corresponding to T_u that has the minimum weight (line 17). Finally, KeyKG returns $T_{u_{\min}}$ (line 18).

```
Input: a KG $G = \langle V, E \rangle$, $g$ sets of keyword vertices
       $K_1, \ldots, K_g$
Output: a GST in $G$ that spans $K_1, \ldots, K_g$
1  foreach $v_1 \in K_1$ do
2      for $i \leftarrow 2$ to $g$ do
3          $v_i \leftarrow \arg\min_{v \in K_i} \texttt{getD}(v_1, v)$;
4      end
5      $U_{v_1} \leftarrow \{v_i : 1 \leq i \leq g\}$;
6      $W_{v_1} \leftarrow \sum_{i=2}^{g} \texttt{dist}(v_1, v_i)$;
7  end
8  $x \leftarrow \arg\min_{v_1 \in K_1} W_{v_1}$;
9  foreach $u \in U_x$ do
10     $T_u = \langle V_{T_u}, E_{T_u} \rangle \leftarrow \langle \{u\}, \emptyset \rangle$;
11     while $U_x \not\subseteq V_{T_u}$ do
12         $\langle s_{\min}, t_{\min} \rangle \leftarrow \arg\min_{\langle s,t \rangle \in V_{T_u} \times (U_x \setminus V_{T_u})} \texttt{getD}(s, t)$;
13         $p \leftarrow \texttt{getSP}(s_{\min}, t_{\min})$;
14         Add the vertices and edges of $p$ to $T_u$;
15     end
16 end
17 $u_{\min} \leftarrow \arg\min_{u \in U_x} \texttt{WT}(T_u)$;
18 return $T_{u_{min}}$;
```

Algorithm 1: KeyKG

KeyKG relies on getD for computing the distance between two vertices (line 3 and line 12), and getSP for computing a shortest path between two vertices (line 13), which we will detail in Section 3.2.

Running Example. In Fig. 1, given $Q = \{k_1, k_2, k_3\}$, assume $K_1 = \{B, F\}$, $K_2 = \{E\}$, and $K_3 = \{C, D\}$. For $B \in K_1$, we select $E \in K_2$ and $C \in K_3$, producing $U_B = \{B, E, C\}$. For $F \in K_1$, we produce $U_F = \{F, E, C\}$. Because $W_B = 1.8$ and $W_F = 2$, we have $W_B < W_F$ and hence $x = B$. Then for $B, E, C \in U_B$, we generate T_B, T_E, T_C, respectively. They satisfy $\texttt{WT}(T_E) = \texttt{WT}(T_C) < \texttt{WT}(T_B)$. Therefore, we have $u_{\min} = E$ (or $u_{\min} = C$), and finally T_E (or T_C) is returned.

Analysis of Approximation Ratio. For a keyword query containing g keywords, Theorem 3.1 shows that KeyKG is a $(g-1)$-approximation algorithm, i.e., the weight of a computed GST is at most $(g-1)$ times the minimum weight. This provable approximation ratio of KeyKG is indeed acceptable since, on the one hand, g is often very small in practice and, on the other hand, the problem cannot be approximated with any ratio in $O(\ln g)$ [26].

THEOREM 3.1. KeyKG *is a $(g-1)$-approximation algorithm.*

PROOF. Let $T_{\text{opt}} = \langle V_{\text{opt}}, E_{\text{opt}} \rangle$ be an optimum solution, i.e., a minimum-weight GST. We show that the following inequality holds:

$$\texttt{WT}(T_{u_{\min}}) \leq (g-1) \cdot \texttt{WT}(T_{\text{opt}}) \,. \tag{1}$$

Observe that for $1 \leq i \leq g$, there is $o_i \in V_{\text{opt}}$ such that $o_i \in K_i$. For $2 \leq i \leq g$, let $p_{1,i}$ be the path between o_1 and o_i in T_{opt}. It is trivial that $\texttt{len}(p_{1,i}) \leq \texttt{WT}(T_{\text{opt}})$, and thus

$$\sum_{i=2}^{g} \texttt{dist}(o_1, o_i) \leq \sum_{i=2}^{g} \texttt{len}(p_{1,i}) \leq (g-1) \cdot \texttt{WT}(T_{\text{opt}}) \,. \tag{2}$$

Now consider the construction of T_u (lines 10–15) for $u = x$. Let $\mathbf{P}$ be the set of all shortest paths added to T_x (line 14). We have

$$\mathtt{WT}(T_{u_{\min}}) \leq \mathtt{WT}(T_x) \leq \sum_{p \in \mathbf{P}} \mathtt{len}(p)\,. \tag{3}$$

For each $p \in \mathbf{P}$ we know $\mathtt{len}(p) = \mathtt{dist}(s_{\min}, t_{\min})$ (line 13), so $\mathtt{len}(p) \leq \mathtt{dist}(x, t_{\min})$ according to the definition of $\langle s_{\min}, t_{\min} \rangle$ (line 12). This leads to the first inequality in the following:

$$\sum_{p \in \mathbf{P}} \mathtt{len}(p) \leq \sum_{v \in U_x \setminus \{x\}} \mathtt{dist}(x, v) = W_x \leq W_{o_1}\,, \tag{4}$$

where the last inequality is by the definition of x (line 8). According to the definitions of W (line 6) and U (line 5 and line 3), we obtain

$$W_{o_1} = \sum_{v \in U_{o_1} \setminus \{o_1\}} \mathtt{dist}(o_1, v) \leq \sum_{i=2}^{g} \mathtt{dist}(o_1, o_i)\,. \tag{5}$$

Finally, combining Eq. (3), Eq. (4), Eq. (5), and Eq. (2) in succession, we obtain Eq. (1) and complete the proof. □

3.2 Static HL

KeyKG relies on getD for computing the distance between two vertices and getSP for computing a shortest path between two vertices. On a large KG, on the one hand, a straightforward online implementation of these subroutines (e.g., using the Dijkstra algorithm) has prohibitively high run time. On the other hand, offline materializing distances and shortest paths between all pairs of vertices has prohibitively large space consumption. To find a practical trade-off between time and space, we consider Hub Labeling (HL) [1], an offline constructed index structure that is dedicated to this purpose. We call it *static* HL as it is offline constructed and invariant to queries, in contrast with *dynamic* HL that is query-relevant and online constructed which we will introduce in Section 4.

Below we review the concept of static HL and the way it is used to implement getD. Then we describe a new method for constructing a static HL where it is extended to also support getSP.

Basic Concepts. A *static HL* is an offline constructed index structure for graphs [1]. For $G = \langle V, E \rangle$, a static HL can be viewed as a function $\mathtt{L} : V \mapsto 2^V$ that maps vertices into sets of vertices (called *hubs*) and satisfies the following condition: $\forall u, v \in V$ that are connected in G, $\exists h \in \mathtt{L}(u) \cap \mathtt{L}(v)$ such that h is on a shortest path between u and v. For $v \in V$, $\mathtt{L}(v)$ is called the *label* of v. In the standard index structure of L, each $\mathtt{L}(v)$ is a list where hubs are sorted by their identifiers. For each hub $h \in \mathtt{L}(v)$, its precomputed distance from v, i.e., $\mathtt{dist}(v, h)$, is also stored. For example, a static HL for the KG in Fig. 1 is shown in the bottom-right corner. At this moment please ignore pred in the figure.

With materialized L, for $u, v \in V$, $\mathtt{getD}(u, v)$ is implemented without accessing the original graph:

$$\mathtt{getD}(u,v) = \begin{cases} \min\limits_{h \in \mathtt{L}(u) \cap \mathtt{L}(v)} \mathtt{dist}(u,h) + \mathtt{dist}(v,h) & \mathtt{L}(u) \cap \mathtt{L}(v) \neq \emptyset\,, \\ \infty & \mathtt{L}(u) \cap \mathtt{L}(v) = \emptyset\,, \end{cases} \tag{6}$$

where $\mathtt{dist}(u, h)$ and $\mathtt{dist}(v, h)$ are stored with h in $\mathtt{L}(u)$ and $\mathtt{L}(v)$, respectively. For example, to compute $\mathtt{getD}(E, F)$ with the static HL in Fig. 1, because $\mathtt{L}(E) \cap \mathtt{L}(F) = \{A, B\}$, we obtain

$$\begin{aligned} \mathtt{getD}(E,F) &= \min\{\mathtt{dist}(E,A) + \mathtt{dist}(F,A),\ \mathtt{dist}(E,B) + \mathtt{dist}(F,B)\} \\ &= \min\{0.3 + 0.7,\ 0.8 + 0.1\} = 0.9\,. \end{aligned}$$

```
Input: a KG $G = \langle V, E \rangle$
Output: a static HL for $G$
1  $\mathtt{L}_0(v) \leftarrow \emptyset$ for all $v \in V$;
2  Sort $V$ in descending order of betweenness centrality;
3  for $i \leftarrow 1$ to $n$ do                                // $n = |V|$
4      $\mathtt{L}_i(v) \leftarrow \mathtt{L}_{i-1}(v)$ for all $v \in V$;
5      $visited[v] \leftarrow 0$ for all $v \in V$;
6      $d[v_i] \leftarrow 0$ and $d[v] \leftarrow \infty$ for all $v \in V \setminus \{v_i\}$;
7      $PQ \leftarrow$ a min-priority queue of vertices initialized
        with $v_i$;
       // The priority of vertex $v$ is set to $d[v]$.
8      while $PQ$ is not empty do
9          $u \leftarrow PQ$.pull();
10         $visited[u] \leftarrow 1$;
11         if $d[u] < \mathtt{getD}(v_i, u)$ then        // using $\mathtt{L}_{i-1}$ for
            getD
12             $\mathtt{L}_i(u) \leftarrow \mathtt{L}_{i-1}(u) \cup \{v_i\}$;  // $\mathtt{dist}(u, v_i) = d[u]$
13             foreach $w \in \mathtt{NBR}(u)$ s.t. $visited[w] = 0$ do
                 // NBR($u$): neighbors of $u$
14                 if $d[u] + \mathtt{wt}((u,w)) < d[w]$ then
15                     $d[w] \leftarrow d[u] + \mathtt{wt}((u,w))$;
16                     $\mathtt{pred}(w, v_i) \leftarrow u$;          // for $\mathtt{L}_i(w)$
17                 end
18                 if $w \notin PQ$ then
19                     $PQ$.insert($w$);
20                 end
21             end
22         end
23     end
24 end
25 return $\mathtt{L}_n$;
```

Algorithm 2: Construction of Static HL

However, the standard index structure of L cannot support efficient implementation of getSP.

Improvement in Construction. According to Eq. (6), the online computation of getD will be faster if vertices have smaller materialized labels. Unfortunately, the problem of minimizing the average size of a label is NP-hard and has logarithmic inapproximability [5]. There have been many and various heuristic methods for constructing reasonably small labels for a given graph [45]. Among others, the pruned landmark labeling (PLL) [3] is a popular implementation, which performs the Dijkstra algorithm and effectively prunes searches to reduce labels. Below we improve PLL to obtain smaller labels and hence compute getD faster.

As shown in **Algorithm 2**, we construct a static HL by improving and extending PLL. Recall that the standard version of PLL basically performs the Dijkstra algorithm n times where n is the number of vertices (lines 3–24), and it iteratively expands vertex labels (line 4 and line 12). We use $\mathtt{L}_i(v)$ to denote v's label after i iterations. In the i-th iteration, the Dijkstra algorithm starts from a distinct vertex $v_i \in V$ (line 7), visits other vertices and calculates their distances from v_i which are stored in d (lines 14–15), and

```
Input: a KG $G = \langle V, E \rangle$ and two vertices $u, v \in V$
Output: a shortest path between $u$ and $v$ in $G$
1  $h_{\min} \leftarrow \arg\min_{h \in \mathsf{L}(u) \cap \mathsf{L}(v)} \mathtt{dist}(u,h) + \mathtt{dist}(v,h)$;
2  $p \leftarrow$ the path consisting of a single vertex $u$;
3  $y \leftarrow u$;
4  while $y \neq h_{min}$ do
5      Find $h_{\min}$ in $\mathsf{L}(y)$;
6      Add the edge between $y$ and $\mathtt{pred}(y, h_{\min})$ to $p$;
7      $y \leftarrow \mathtt{pred}(y, h_{\min})$;
8  end
9  $y \leftarrow v$;
10 while $y \neq h_{min}$ do
11     Find $h_{\min}$ in $\mathsf{L}(y)$;
12     Add the edge between $y$ and $\mathtt{pred}(y, h_{\min})$ to $p$;
13     $y \leftarrow \mathtt{pred}(y, h_{\min})$;
14 end
15 return $p$;
```

Algorithm 3: Implementation of getSP

adds v_i to their labels (line 12). Some vertex u may not be visited and hence its label can be reduced. Such a prune happens when u's distance from v_i can been computed from constructed labels L_{i-1}, i.e., the condition is false in line 11.

Whereas the correctness of the pruning is provable [3], the goal of our improvement is to prune more. We want labels constructed in earlier iterations to support the computation of distances between more pairs of vertices, so that pruning will be more often in later iterations. Intuitively, this can be achieved by choosing *vertices through which many shortest paths pass* as the starting vertex of the Dijkstra algorithm in early iterations. To this end, the original implementation of PLL heuristically sorts starting vertices in descending order of degree because high-degree vertices are likely to appear in shortest paths between many pairs of vertices. We do it differently; we sort in descending order of betweenness centrality (line 2). The *betweenness centrality* of a vertex v is defined as

$$\mathsf{bc}(v) = \sum_{s,t \in V \setminus \{v\}} \frac{\sigma_{st}(v)}{\sigma_{st}}, \tag{7}$$

where σ_{st} is the number of shortest paths between s and t, and $\sigma_{st}(v)$ is the number of the above paths that pass through v. Computing exact betweenness centrality, e.g., using [10], has prohibitively high run time for large graphs. Thus, in our implementation we use a source sampling based approximation algorithm and select the highest-degree vertices as pivots. We refer the reader to [4] for further details of this algorithm as well as the definition of pivot.

Extension of Index Structure. To support efficient implementation of getSP, we need to extend the index structure of L. In **Algorithm 2**, for each hub $v_i \in \mathsf{L}(w)$, we store not only $\mathtt{dist}(w, v_i)$ but also w's predecessor in the search tree rooted at v_i, denoted by $\mathtt{pred}(w, v_i)$ (line 16). Storing pred does not increase the asymptotic space complexity of our static HL.

With such extended labels, getSP is implemented in **Algorithm 3**. To obtain a shortest path p between u and v, we firstly find their common hub $h_{\min}$ on p (line 1), and then repeatedly follow predecessors to construct the u-$h_{\min}$ part of p (lines 2–8) and the v-$h_{\min}$ part of p (lines 9–14). For example, to compute $\mathtt{getSP}(D, F)$ with the extended HL in Fig. 1, we retrieve $h_{\min} = A$ because A is the only common hub in $\mathsf{L}(D)$ and $\mathsf{L}(F)$. The D-A part of p, i.e., the single edge (D, A), is constructed by following $\mathtt{pred}(D, A) = A$ which is stored with A in $\mathsf{L}(D)$. The F-A part of p, i.e., the path consisting of two edges (F, B) and (B, A), is constructed by following $\mathtt{pred}(F, A) = B$ which is stored with A in $\mathsf{L}(F)$ and then following $\mathtt{pred}(B, A) = A$ which is stored with A in $\mathsf{L}(B)$. Finally, the two parts are concatenated into $p = D$-A-B-F.

3.3 Analysis of Run Time

Now we analyze the run time of KeyKG in **Algorithm 1**. Let $t_{\mathtt{getD}}$ and $t_{\mathtt{getSP}}$ be the run time of getD and getSP, respectively, which we will detail later. Recall that g is the number of keywords, and n is the number of vertices of G. Because $|K_i| \leq n$ for $1 \leq i \leq g$, the run time of lines 1–8 is $O(n^2 g t_{\mathtt{getD}})$. For lines 9–18, our implementation uses the following trick. For each vertex in $U_x \setminus V_{T_u}$, we store its smallest distance from the vertices in T_u. When we add a vertex of p to T_u (line 14), we update its stored distance to each vertex in $U_x \setminus V_{T_u}$ by calling getD. We use these stored distances to find $\langle s_{\min}, t_{\min} \rangle$ in line 12, without calling getD there. This trick reduces the total number of calls to getD from $O(ng^3)$ to $O(ng^2)$, although an additional $O(g^3)$ time is needed for finding the minimum distance among stored ones. The total run time of KeyKG is thus

$$O(n^2 g t_{\mathtt{getD}} + n g^2 t_{\mathtt{getD}} + g^3 + g^2 t_{\mathtt{getSP}}) \,. \tag{8}$$

To analyze $t_{\mathtt{getD}}$, recall Eq. (6) and observe that by performing a mergesort-like operation on two sorted lists of hubs, $\mathtt{getD}(u, v)$ is computed in $O(|\mathsf{L}(u)| + |\mathsf{L}(v)|)$ time and thus $t_{\mathtt{getD}}$ is $O(n)$ since $|\mathsf{L}(\cdot)| \leq n$. This is much faster than naively using the Dijkstra algorithm. Besides, since $|\mathsf{L}(\cdot)| \ll n$ in practice, the total size of L is close to $O(n)$ and it is much smaller than that of naively materializing distances between all pairs of vertices. Therefore, static HL provides a good trade-off between time and space.

To analyze $t_{\mathtt{getSP}}$ for **Algorithm 3**, observe that by a mergesort-like operation, $h_{\min}$ is retrieved in $O(n)$ time (line 1). Then, by performing binary search of a label, $h_{\min}$ is located in $O(\log n)$ time (line 5 and line 11). The while loops will run $O(n)$ times. Thus, $t_{\mathtt{getSP}}$ is $O(n \log n)$.

Now we can conclude the analysis. In Eq. (8), by substituting $t_{\mathtt{getD}}$ and $t_{\mathtt{getSP}}$ with $O(n)$ and $O(n \log n)$, respectively, we obtain the following run time of KeyKG:

$$O(n^3 g + n^2 g^2 + g^3 + g^2 n \log n) \,. \tag{9}$$

However, this worst-case complexity will rarely be met in practice because we usually have $|K_i| \ll n$, $|V_{T_u}| \ll n$, and $|\mathsf{L}(\cdot)| \ll n$. Furthermore, g is also often small, so the actual run time of KeyKG is very low, as we will see in the experiments.

4 ALGORITHM KeyKG$^+$ WITH DYNAMIC HL

In this section we present our second approximation algorithm KeyKG$^+$. It extends KeyKG with a novel online constructed HL to realize more efficient computation of distances on sets of vertices. This dynamic HL is described in Section 4.1. Based on that, KeyKG$^+$ is given in Section 4.2. We analyze its run time in Section 4.3.

Table 1: A dynamic HL (M) for our running example, where the number in the subscript of $M_{i,j}$ represents $\texttt{dist}(M_{i,j}, h_j)$.

	A	B	C	D	E	F
K_2	$E_{(0.3)}$	$E_{(0.8)}$	null	null	$E_{(0)}$	null
K_3	$C_{(0.4)}$	null	$C_{(0)}$	$D_{(0)}$	null	null

4.1 Dynamic HL

Recall that in **Algorithm 1**, for each K_i ($2 \le i \le g$), the static labels of all the vertices in K_i will be accessed $O(gn)$ times by $\texttt{getD}$ (line 3) in the same way: by performing a mergesort-like operation to find a hub in the static label of some vertex $v_i \in K_i$ such that the hub also appears in $\mathsf{L}(v_1)$ and it is v_i that minimizes the distance between v_1 and K_i. We optimize the performance of these repeated operations by constructing an inverted index structure called a *dynamic HL*, which aggregates the static labels of the vertices in K_i. Such dynamic HLs are query-relevant and are thus online constructed.

Specifically, a dynamic HL is a $(g-1) \times n$ matrix M. Rows correspond to sets of keyword vertices $K_2, \ldots, K_g$, and columns correspond to hub vertices. The $(i-1)$-th row of M, denoted by M_{i-1}, *inverts and then aggregates* the static labels of the vertices in K_i. The j-th element of M_{i-1}, denoted by $M_{i-1,j}$, is not null if vertex $h_j \in V$ is a hub in the static label of at least one vertex in K_i. Among these vertices in K_i whose static labels contain h_j, $M_{i-1,j}$ denotes the one that minimizes the distance to h_j:

$$M_{i-1,j} = \begin{cases} \underset{u \in K_i \text{ s.t. } h_j \in \mathsf{L}(u)}{\arg\min} \texttt{dist}(u, h_j) & h_j \in \bigcup_{u \in K_i} \mathsf{L}(u)\,, \\ \text{null} & h_j \notin \bigcup_{u \in K_i} \mathsf{L}(u)\,. \end{cases} \tag{10}$$

If h_j is not a hub in the static label of any vertex in K_i, we let $M_{i-1,j}$ be null. We use a two-dimensional array to store M, thereby allowing random access in constant time. For each $M_{i-1,j}$ that is not null, its precomputed distance from h_j is also stored. M_{i-1} can be constructed from K_i and L without accessing the original graph. As we will see in Section 4.2, in the computation of v_i, M_{i-1} can replace the static labels of the vertices in K_i, and show improved efficiency thanks to its compactness and random access capability.

For example, Table 1 presents M for our running example. Take $i = 3$, for instance. Recall that $K_3 = \{C, D\}$ and in Fig. 1 we have

$$\mathsf{L}(C) \cup \mathsf{L}(D) = \{A, C\} \cup \{A, D\} = \{A, C, D\}\,. \tag{11}$$

Therefore, in the second row only these entries are not null. As an example we show how $M_{2,A} = C$ is obtained. Here $h_j = A$, and we observe $A \in \mathsf{L}(C)$ and $A \in \mathsf{L}(D)$. We choose C instead of D as $M_{2,A}$ because $\texttt{dist}(C, A) < \texttt{dist}(D, A)$. Note that $\texttt{dist}(C, A)$ and $\texttt{dist}(D, A)$ are retrieved from the static HL in Fig. 1. Finally, we store $\texttt{dist}(C, A) = 0.4$ with $M_{2,A}$ in the dynamic HL.

4.2 Algorithm KeyKG$^+$

KeyKG$^+$ presented in **Algorithm 4** is an extension of KeyKG. Dynamic HLs are constructed and used in two places to improve the overall efficiency but they will not change the computed results.

In the first place, M is constructed for $K_2, \ldots, K_g$ (lines 1–3). Then M_{i-1} is used to find v_i (line 6) as follows. For each hub $h_j \in \mathsf{L}(v_1)$, we retrieve $\texttt{dist}(v_1, h_j)$ from $\mathsf{L}(v_1)$ and retrieve $M_{i-1,j}$ with

```
Input: a KG $G = \langle V, E \rangle$, $g$ sets of keyword vertices
       $K_1, \ldots, K_g$
Output: a GST in $G$ that spans $K_1, \ldots, K_g$
1  for $i \leftarrow 2$ to $g$ do
2      Construct $M_{i-1}$ for $K_i$;
3  end
4  foreach $v_1 \in K_1$ do
5      for $i \leftarrow 2$ to $g$ do
6          $v_i = \underset{M_{i-1,j} \text{ s.t. } h_j \in \mathsf{L}(v_1) \text{ and } M_{i-1,j} \neq \text{null}}{\arg\min} \texttt{dist}(v_1, h_j) + \texttt{dist}(M_{i-1,j}, h_j)$;
7      end
8      $U_{v_1} \leftarrow \{v_i : 1 \le i \le g\}$;
9      $W_{v_1} \leftarrow \sum_{i=2}^{g} \texttt{dist}(v_1, v_i)$;
10 end
11 $x \leftarrow \underset{v_1 \in K_1}{\arg\min} W_{v_1}$;
12 foreach $u \in U_x$ do
13     $T_u = \langle V_{T_u}, E_{T_u} \rangle \leftarrow \langle \{u\}, \emptyset \rangle$;
14     Construct $M'_u$ for $V_{T_u}$;
15     while $U_x \nsubseteq V_{T_u}$ do
16         foreach $t_i \in (U_x \setminus V_{T_u})$ do
17             $s_i = \underset{M'_{u,j} \text{ s.t. } h_j \in \mathsf{L}(t_i) \text{ and } M'_{u,j} \neq \text{null}}{\arg\min} \texttt{dist}(t_i, h_j) + \texttt{dist}(M'_{u,j}, h_j)$;
18         end
19         $\langle s_{\min}, t_{\min} \rangle \leftarrow \underset{\langle s_i, t_i \rangle}{\arg\min} \texttt{dist}(s_i, t_i)$;
20         $p \leftarrow \texttt{getSP}(s_{\min}, t_{\min})$;
21         Add the vertices and edges of $p$ to $T_u$;
22         Update $M'_u$;
23     end
24 end
25 $u_{\min} \leftarrow \underset{u \in U_x}{\arg\min} \texttt{WT}(T_u)$;
26 return $T_{u_{min}}$;
```

Algorithm 4: KeyKG$^+$

$\texttt{dist}(M_{i-1,j}, h_j)$ from M_{i-1}. If $M_{i-1,j}$ is not null, we calculate

$$\texttt{dist}(v_1, h_j) + \texttt{dist}(M_{i-1,j}, h_j)\,, \tag{12}$$

which represents the smallest distance between v_1 and the vertices in K_i via a particular h_j. Finally, v_i is found over all $h_j \in \mathsf{L}(v_1)$:

$$v_i = \underset{M_{i-1,j} \text{ s.t. } h_j \in \mathsf{L}(v_1) \text{ and } M_{i-1,j} \neq \text{null}}{\arg\min} \texttt{dist}(v_1, h_j) + \texttt{dist}(M_{i-1,j}, h_j)\,. \tag{13}$$

This computation in KeyKG$^+$ (line 6) is equivalent to the computation of v_i in KeyKG (line 3), but is more efficient.

Running Example. In Fig. 1, for $B \in K_1$ as v_1, given $\mathsf{L}(B) = \{A, B\}$, we select $E \in K_2$ as v_2 since $M_{1,A} = M_{1,B} = E$ in Table 1; select $C \in K_3$ as v_3 since $M_{2,A} = C$, $M_{2,B} =$ null. For $F \in K_1$ as v_1, given $\mathsf{L}(F) = \{A, B, F\}$, we select $E \in K_2$ as v_2 since $M_{1,A} = M_{1,B} = E$, $M_{1,F} =$ null; select $C \in K_3$ since $M_{2,A} = C$, $M_{2,B} = M_{2,F} =$ null. This selection in KeyKG$^+$ is the same as that in KeyKG.

Table 2: KGs and keyword queries used in the experiments.

	KG		Keyword Query	
	$\|V\|$	$\|E\|$	Number	g
MONDIAL	37,303	109,577	40	1–4
LinkedMDB	748,593	1,216,325	200	1–10
DBpedia	5,765,042	17,557,947	429	1–10

In the second place, we create M'_u for V_{T_u} (line 14) just as how we create M_{i-1} for K_i. As T_u expands in each iteration (line 21), M'_u is updated with the static labels of the vertices added to T_u (line 22). For each $t_i \in (U_x \setminus V_{T_u})$, M'_u is used to find $s_i \in V_{T_u}$ that has the minimum distance from t_i (lines 16–18), just as how M_{i-1} is used to find v_i. We find $\langle s_{\min}, t_{\min} \rangle$ over all such $\langle s_i, t_i \rangle$ (line 19). This computation of $\langle s_{\min}, t_{\min} \rangle$ in KeyKG$^+$ (lines 16–19) is equivalent to that in KeyKG (line 12), but is more efficient.

Analysis of Approximation Ratio. KeyKG$^+$ computes the same result as KeyKG and hence is also a $(g-1)$-approximation algorithm.

4.3 Analysis of Run Time

Compared with KeyKG in **Algorithm 1** where the run time of lines 1–4 is $O(n^3 g)$, KeyKG$^+$ in **Algorithm 4** constructs M_{i-1} in $O(n^2)$ time and finds v_i in $O(n)$ time since the array-based M supports $O(1)$-time random access, reducing the run time of this part (lines 1–7) to $O(n^2 g)$. Similarly, compared with KeyKG where the run time of lines 9–12 is $O(n^2 g^2 + g^3)$, in KeyKG$^+$ the run time of this part (lines 12–19 and line 22) is $O(n^2 g + ng^3)$. It represents an improvement in practice because we usually have $ng^3 \ll n^2 g$ and thus $O(n^2 g + ng^3)$ is close to $O(n^2 g)$ which is better than $O(n^2 g^2 + g^3)$.

Besides, we reconsider the run time of getSP in KeyKG$^+$ (line 20). In **Algorithm 3**, first, we can reduce the $O(n)$ time for retrieving $h_{\min}$ to zero because in KeyKG$^+$, $h_{\min}$ has been identified (line 17) and can be reused. Second, throughout the construction of T_u, the total number of binary searches of static labels is not $O(gn)$ but is proportional to $|E_{T_u}| \in O(n)$. Therefore, the total run time of getSP is reduced from $O(g^2 n \log n)$ in KeyKG to $O(gn \log n)$ in KeyKG$^+$.

To conclude, we obtain the following run time of KeyKG$^+$:

$$O(n^2 g + n^2 g + ng^3 + gn \log n), \quad \text{i.e.,} \quad O(n^2 g + ng^3). \tag{14}$$

Following our comment on the run time of KeyKG in Section 3.3, this worst-case complexity will rarely be met in practice. The actual run time of KeyKG$^+$ is very low—several orders of magnitude lower than that of KeyKG, as we will see in the experiments

5 EXPERIMENTS

The purpose of our experiments is to empirically investigate the following research hypotheses (RH).

RH1 Our approach shows practicality with low run time on typical KGs and outperforms the state of the art [27, 38], while it computes reasonably good answers with comparable quality.
RH2 Using our proposed dynamic HL brings about significant improvement in overall efficiency.
RH3 Our static HL has a smaller size than existing HLs [3, 14, 39], and hence supports more efficient computation of distances and shortest paths in our approach.

5.1 Experiment Setup

Our experiments were conducted on a 3.5GHz Intel Xeon with 32GB memory for Java programs. Below we detail the setup.

KGs. We used three well-known KGs of different sizes:

- MONDIAL,[1] a small geography KG,
- LinkedMDB,[2] a medium-sized movie KG, and
- DBpedia,[3] a large encyclopedic KG.

Their sizes are presented in Table 2. KGs were stored in memory.

Keyword Queries. MONDIAL had been used in previous evaluation [13] and we reused the 40 keyword queries provided there. For LinkedMDB we randomly sampled 200 questions from WikiMovies [40] and transformed natural language questions into keyword queries by removing stop words and punctuation marks. For DBpedia we reused 429 keyword queries provided by DBpedia-Entity v2 [22]. Table 2 shows the number and size of these queries.

Keyword Mapping. The three KGs are in the Resource Description Framework (RDF) format. Our implementation of the retrieval function hits maps keyword k to the vertices whose human-readable names (rdfs:label) contain k. We found that every keyword in our queries could be matched to at least one vertex in the largest connected component of the KG. Therefore, an answer under the GST semantics was guaranteed to exist.

Edge Weighting. There was no standard way of weighting edges of a KG. We assigned to each edge a random real number that was uniformly sampled from $[0, 1000]$.

5.2 Baselines

Among existing methods for keyword search over KGs under the GST or GST-like semantics, we compared our approach with two representatives:

- **PrunedDP++** [38] is a state-of-the-art exact algorithm for the GST problem based on A* search.
- **BANKS-II** [27] is a state-of-the-art approximation algorithm for the GST problem based on bidirectional search.

We re-implemented them by ourselves.

We did not compare with DPBF [15] and BANKS [9] because PrunedDP++ and BANKS-II are improved versions of them, respectively. We did not include BLINKS [23] because it could not scale to KGs as large as LinkedMDB and DBpedia [13]. We did not consider methods using semantics other than GST because they were not directly comparable with our approach, e.g., STAR [28] based on Steiner trees assuming an one-to-one retrieval function.

Recall that our static HL improves PLL [3] and sorts vertices by their betweenness centrality. We compared it with three existing HLs representing the state of the art:

- **PLL** refers to the original implementation of PLL [3] which sorts vertices by their degrees.
- **RXL** [14] improves PLL by sorting vertices based on shortest-path trees.
- **SHP** [39] is a recent method based on significant paths.

We obtained their implementation from the authors of [39].

[1] www.dbis.informatik.uni-goettingen.de/Mondial/Mondial-RDF/mondial.rdf
[2] www.cs.toronto.edu/~oktie/linkedmdb/linkedmdb-latest-dump.zip
[3] downloads.dbpedia.org/2016-10/core-i18n/en/mappingbased_objects_en.tql.bz2

Table 3: Number of successful queries (SC), timeout exceptions (TO), and mean run time (ms) for a query.

	MONDIAL			LinkedMDB			DBpedia		
	SC	TO	Time	SC	TO	Time	SC	TO	Time
PrunedDP++	40	0	75.65	194	6	357,274.93	413	16	379,250.20
BANKS-II	40	0	418.85	200	0	27,168.20	429	0	647,041.79
KeyKG$^+$	40	0	0.06	200	0	0.16	429	0	82.84
KeyKG$^+$-D	40	0	6.53	200	0	203.35	429	0	29,776.04
KeyKG	40	0	0.08	200	0	0.46	429	0	10,474.65
KeyKG-PLL	40	0	0.09	200	0	0.52	429	0	11,345.97

5.3 Implementation of Our Approach

Below we detail the configuration of our approach. We also implemented several variants of our approach to thoroughly investigate its key components. Our implementation has been open source.[4]

Configuration of Our Approach. We implemented KeyKG$^+$ and KeyKG where static HLs were stored in memory. In KeyKG$^+$, dynamic HLs were online constructed and stored in memory. In the computation of approximate betweenness centrality for constructing our static HL, we selected 200 pivots [4]. The constructed static HLs used affordable 37MB, 183MB, and 7,704MB memory for MONDIAL, LinkedMDB, and DBpedia, respectively.

Variants of Our Approach. We implemented KeyKG$^+$**-D**, a variant of KeyKG$^+$ where static HLs were stored on disk using a MySQL database. It would help to show the performance of our approach in a memory-efficient setting, e.g., on a low-resource machine.

We also implemented KeyKG-**PLL**, a variant of KeyKG where our static HL was replaced by the original implementation of PLL. It would help to evaluate the effectiveness of our improvement with the betweenness centrality based heuristic.

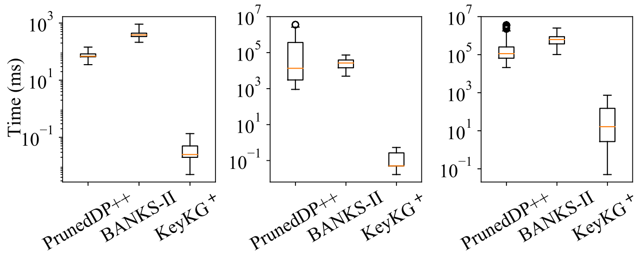
We implemented two variants of our static HL using 10 pivots and 100 pivots, referred to as **SHL-10** and **SHL-100**, respectively. Accordingly, the default version was referred to as **SHL-200**. Increasing the number of pivots could improve the accuracy of approximating betweenness centrality with increased run time. These variants would help to observe the influence of this parameter to our static HL.

5.4 Evaluation Metrics

For keyword search over KGs, our approach and the two baseline methods are all under the GST semantics. Their common goal is to compute or approximate a minimum-weight GST as fast as possible. Therefore, following the standard way of evaluating an optimization algorithm, we measured the *quality of approximation* and the *run time*. For quality assessment, we calculated the approximation ratio of a computed GST by dividing its total edge weight by that of a minimum-weight GST. We did not use IR metrics such as precision/recall/F1 to evaluate keyword search, because the results would be dependent on the concrete weighting scheme for edges, which is outside the scope of our research in this work.

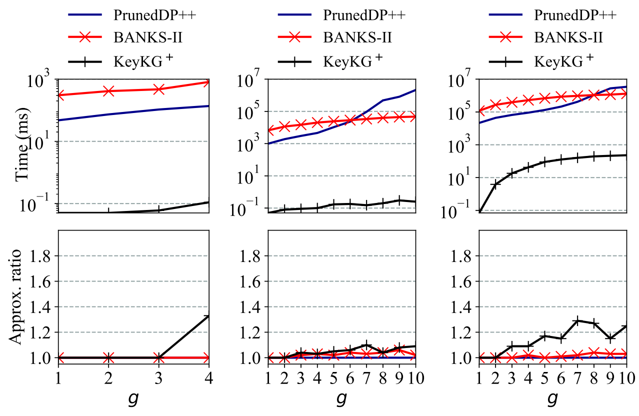
For comparing static HLs, we measured the *average size of a vertex label*, i.e., the mean number of hubs in a label. Small labels would lead to fast computation of distances and shortest paths according to Eq. (6) and Algorithm 3, respectively.

[4]github.com/nju-websoft/KeyKG



(a) MONDIAL (b) LinkedMDB (c) DBpedia

Figure 2: Distribution of run time for all the queries.



(a) MONDIAL (b) LinkedMDB (c) DBpedia

Figure 3: Mean run time and approximation ratio for a query, with a varying number of query keywords (i.e., g).

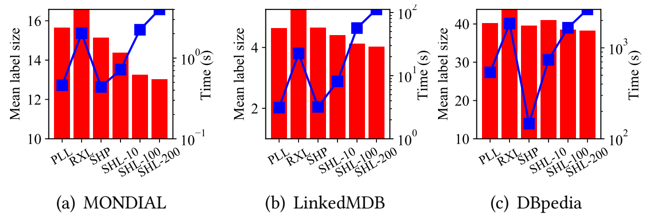
5.5 Results and Analysis

Below we report experiment results and investigate each of the three research hypotheses.

5.5.1 Investigation of RH1. For RH1, to show the effectiveness and efficiency of our approach, we compared KeyKG$^+$ with PrunedDP++ and BANKS-II. Table 3 summarizes the mean run time of each algorithm for a query. Timeout (>1h) was observed only on PrunedDP++. Besides, this algorithm occasionally ran out of memory (>32GB). We configured it to return the best GST it had found till timeout or out of memory.

As shown in Table 3, on the small MONDIAL KG, all the three algorithms answered a query in less than 1s. However, on the larger LinkedMDB and DBpedia KGs, PrunedDP++ and BANKS-II unsatisfactorily spent dozens to hundreds of seconds. By contrast, KeyKG$^+$ used less than 1ms on LinkedMDB and less than 100ms on DBpedia, being 3 orders of magnitude faster than the baselines.

We visualize the distribution of run time for all the queries as box plots in Fig. 2. KeyKG$^+$ never exceeded 1s for any query. Its worst-case performance was at least 1 order of magnitude faster than the best-case performance of the baselines. PrunedDP++ showed many outliers due to its exponential run time.



(a) MONDIAL (b) LinkedMDB (c) DBpedia

Figure 4: Mean number of hubs in the label of a vertex (bars); run time for constructing a static HL (points).

Run time is related to the number of keywords in a query (i.e., g). The mean run time for a query under different values of g is shown in Fig. 3 (top). Generally, more time was spent when g grew. The run time of KeyKG$^+$ increased slowly, showing its good scalability in terms of g. By contrast, the run time of PrunedDP++ should in theory rise at an exponential rate, and indeed it rose sharply.

In Fig. 3 (bottom), we present the mean approximation ratio for a query. For PrunedDP++, as an exact algorithm its approximation ratio was fixed at 1.0. Both BANKS-II and KeyKG$^+$ computed reasonably good answers since their approximation ratios were satisfyingly below 1.2 on LinkedMDB and below 1.4 on the other KGs, while BANKS-II achieved slightly better results than KeyKG$^+$. Their approximation ratios increased very slowly as g increased, being consistent with their provable approximation ratios in $O(g)$.

Conclusion. Whereas PrunedDP++ and BANKS-II generated optimum or near-optimum GSTs, their run time was hardly acceptable on the large DBpedia KG. KeyKG$^+$ achieved a better trade-off between result quality and run time. It never used more than 1s for any query, and the quality of its computed GSTs was close to the optimum. Therefore, the experiment results supported RH1.

Besides, in Table 3, KeyKG$^+$-D with static HLs stored on disk used less than 1s to answer a query over MONDIAL and LinkedMDB, showing the potential to work on a low-resource machine. Even in this memory-efficient setting, KeyKG$^+$-D was at least 1 order of magnitude faster than the baselines on all the three KGs.

5.5.2 Investigation of RH2. For RH2, to show the usefulness of our proposed dynamic HL, we compared KeyKG$^+$ with KeyKG. As shown in Table 3, on MONDIAL and LinkedMDB, both algorithms used less than 1ms for a query and their differences were not important. On the large DBpedia KG, KeyKG$^+$ remained at the millisecond level, whereas KeyKG spent more than 10s. KeyKG was not fast particularly when keywords were matched to large sets of vertices, which in turn, led to quadratically many calls to the `getD` subroutine using a static HL. By contrast, using a dynamic HL in KeyKG$^+$, the run time of this part was in theory reduced by 1 order of magnitude, and indeed the performance improvement was significant in the experiments. Therefore, the experiment results supported RH2.

Besides, in Table 3, even with only static HL, KeyKG achieved at least 1 order of magnitude improvement over the baselines on all the three KGs, showing the technical superiority of our proposed HL-powered algorithm.

5.5.3 Investigation of RH3. For RH3, to show the usefulness of our new static HL (i.e., SHL), we compared it with PLL, RXL, and SHP. The average size of a constructed vertex label is shown in Fig. 4 (bars). SHL-200 constructed the smallest HLs on all the three KGs. Compared with PLL and RXL, SHL-200 reduced the label size by 17%–21% on MONDIAL, 13%–23% on LinkedMDB, and 5%–13% on DBpedia. The results demonstrated the effectiveness of our betweenness centrality based heuristic, relative to the degree based heuristic (PLL) and the shortest-path trees based heuristic (RXL). SHL-200 also outperformed the recent SHP, reducing the label size by 3%–14% on different KGs. Therefore, the experiment results supported RH3.

The effectiveness of SHL was also observed in Table 3. On all the three KGs, KeyKG using SHL was consistently faster than KeyKG-PLL using the original implementation of PLL. On the largest DBpedia KG, the run time was reduced by 8%.

Last but not least, we show the run time for constructing a static HL in Fig. 4 (points). Comparing SHL-10, SHL-100, and SHL-200, smaller HLs were constructed when increasing the number of pivots and thus the run time. Although SHL-200 used more time than other methods to construct a static HL, the construction was performed offline. Indeed, it spent affordabe 4s, 112s, and 2,647s constructing for MONDIAL, LinkedMDB, and DBpedia, respectively.

6 RELATED WORK

In this section we review related work and consider various approaches to keyword search over graph data and to Hub Labeling.

6.1 Keyword Search over Graph Data

Query Interpretation. There is a body of work on keyword search over graph data where the general idea is to first transform a keyword query into a structured query [17, 18, 21, 44, 47, 48, 50, 53], such as a SPARQL query for RDF graph, and then execute the structured query over a graph to retrieve answers. These methods are orthogonal to ours. We follow a different direction. We directly search for an optimum subgraph to answer the keyword query.

GST-based Methods. The direction we follow usually formulates a GST problem or a variant thereof. To solve the problem, BANKS [9] merges paths from keyword vertices to a common root vertex to approximate a minimum-weight GST. BANKS-II [27] performs bidirectional search to improve the performance of BANKS. BLINKS [23] exploits precomputed distances and graph partitioning to make bidirectional search more efficient. In [35], search is pruned according to graph summaries. Apart from these approximation algorithms, DPBF [15] is a dynamic programming solution and finds a minimum-weight GST. PrunedDP++ [38] improves DPBF with A* search, and achieves state-of-the-art performance.

Since the standard GST problem is NP-hard, it is not surprising that exact solutions such as DPBF and PrunedDP++ run in exponential time, which is prohibitive for large graphs. Regarding polynomial-time approximation algorithms with provable quality guarantees, Table 4 compares our approach with other practical solutions. The worst-case approximation ratio and run time of KeyKG$^+$ are generally comparable to the existing results. However, this worst case is rarely met in practice, and our experiments show that KeyKG$^+$ can be orders of magnitude faster than existing algorithms on large KGs. Note that for Table 4 we only select some practical algorithms that have the potential to work on a large KG.

Table 4: Comparison of approximation algorithms for keyword search under the GST semantics. The results of BANKS, BANKS-II, and BLINKS are sourced from [13]. (n: number of vertices; m: number of edges; g: number of keywords.)

Algorithm	Approx. Ratio	Run Time
BANKS [9]	$O(g)$	$O(n^2 \log n + nm)$
BANKS-II [27]	$O(g)$	$O(n^2 \log n + nm)$
BLINKS [23]	$O(g)$	dependent on partitioning
KeyKG$^+$	$O(g)$	$O(n^2 g + ng^3)$

Some other algorithms are not included due to their unacceptable run time. For example, 2-Star [8] has a better approximation ratio of $O(\sqrt{g} \ln g)$ but it cannot scale to large graphs.

Our approach shares some technical features with 1-Star [25]. Both of them rely on the availability of distances and shortest paths, and they have the same provable approximation ratio. However, KeyKG$^+$ may generate empirically better GSTs due to its more thoughtful design. For example, to span a set of selected keyword vertices, 1-Star simply merges the shortest paths from one vertex to the other vertices in the set, whereas in KeyKG$^+$ we construct a small-weight tree in a greedy manner. Thanks to the proposed static and dynamic HLs which realize efficient computation of distances and shortest paths, KeyKG$^+$ can be much faster than 1-Star which has prohibitively high run time for large graphs.

Steiner Tree. The GST problem we address in this paper is a generalized version of the Steiner tree problem. With the original Steiner tree based formulation it is assumed that, in terms of keyword search, each keyword is matched to only one vertex. Algorithms for solving this problem, such as STAR [28] and SketchLS [20], cannot be directly applied to our GST problem.

Retrieval-based Methods. To scale to large graphs, there are methods that precompute and index a large number of size-bounded subgraphs as candidate answers, e.g., r-radius graphs in EASE [37], tuple units in SAINT [16]. Keyword search is then transformed into a problem that is similar to traditional information retrieval—to retrieve and rank subgraphs that are relevant to a keyword query. The scalability and efficiency of these methods are essentially obtained by significantly limiting the search space and thus sacrificing quality. Besides, when the keywords in a query are not very close to each other in the graph, e.g., their distances exceed the predefined size bound of an indexed subgraph, empty answers will be produced. It limits the application and usability of these methods. By contrast, GSTs are not restricted by such structural bounds.

6.2 Hub Labeling

Exact HLs. Hub Labeling (HL) [1] is a popular type of distance oracle [45] for answering exact distances between vertices. The Pruned Landmark Labeling (PLL) [3] is an implementation of HL. It leverages landmarks (i.e., hubs) to prune the Dijkstra algorithm. The size of generated labels is related to the order of vertices where pruned searches start. The original version of PLL [3] sorts vertices by their degrees. The subsequent Robust eXact Labeling (RXL) [14] uses shortest-path trees to establish a better order. Our static HL sorts by approximate betweenness centrality and generates smaller labels than PLL and RXL in the experiments. Our static HL also empirically outperforms the Significant path based Hub Pushing (SHP) [39], which picks significant vertices as landmarks. The computation of significance combines a set of heuristics.

In addition to static HL, we further propose dynamic HL which is online constructed and is relevant to the concrete keyword query. Compared with static HL which indexes hubs for each vertex, our dynamic HL can be viewed as an inverted index that maps hubs to keyword vertices. By aggregating static labels and enabling random access in constant time, it is particularly suitable for computing distances on sets of vertices and brings about orders of magnitude improvement in efficiency—both theoretically and empirically.

Approximate and Other HLs. There are other types of distance oracle [45]. Some are designed to compute approximate distances [7, 11, 43, 46, 52]. However, they are not suitable for our setting because using approximate distances would affect the approximation ratio of our algorithm. More efficient implementations have been developed for road networks [2, 19, 42, 54]. Unfortunately, KGs generally do no have the special spatial properties of a road network and hence these methods do not apply.

7 CONCLUSION

In this paper, we study keyword search over KGs under the GST semantics. We introduced two algorithms: KeyKG that is based on a static HL and KeyKG$^+$ that is based on a dynamic HL, to efficiently approximate GST-based keyword query answers. Our experiments show that KeyKG$^+$ is 3 orders of magnitude faster than previous algorithms and is with comparable approximation loss. In particular, on large KGs such as the well-known DBpedia, reasonably good answers can be computed in milliseconds, showing the practicality of our approach. The achieved satisfying performance is mainly attributed to our novel dynamic HL that inverts and aggregates query-relevant labels, and to our new static HL using a betweenness centrality based heuristic that outperforms existing HLs. Potential applications of these HLs are clearly not limited to our approach.

As for the future work, we note that rare distance oracles support efficient edge deletion, which is a basic operation in the framework of top-k GSTs with polynomial delay [34], so efficient top-k keyword search remains unsolved in our approach. We plan to explore this in our future work. Then, compared to the approximation algorithm BANKS-II, the approximation ratio of KeyKG$^+$ may be further improved. Besides, there exist some large and dense graphs that all existing HLs cannot index using small label sizes. In order to cope with such graphs, one would have to develop other types of efficient distance oracles or to consider alternative techniques.

ACKNOWLEDGMENTS

This work was supported in part by the National Key R&D Program of China under Grant 2018YFB1005100, in part by the NSFC under Grant 61772264, in part by the Six Talent Peaks Program of Jiangsu Province under Grant RJFW-011, and in part by the SIRIUS Centre, Norwegian Research Council project number 237898.

REFERENCES

[1] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato Fonseca F. Werneck. 2011. A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks. In *SEA*. 230–241.

[2] Takuya Akiba, Yoichi Iwata, Ken-ichi Kawarabayashi, and Yuki Kawata. 2014. Fast Shortest-path Distance Queries on Road Networks by Pruned Highway Labeling. In *ALENEX*. 147–154.

[3] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*. 349–360.

[4] Ziyad AlGhamdi, Fuad Jamour, Spiros Skiadopoulos, and Panos Kalnis. 2017. A Benchmark for Betweenness Centrality Approximation Algorithms on Large Graphs. In *SSDBM*.

[5] Haris Angelidakis, Yury Makarychev, and Vsevolod Oparin. 2017. Algorithmic and Hardness Results for the Hub Labeling Problem. In *SODA*. 1442–1461.

[6] Marcelo Arenas, Bernardo Cuenca Grau, Evgeny Kharlamov, Sarunas Marciuska, and Dmitriy Zheleznyakov. 2016. Faceted search over RDF-based knowledge graphs. *J. Web Semant.* 37-38 (2016), 55–74.

[7] Bahman Bahmani and Ashish Goel. 2012. Partitioned multi-indexing: bringing order to social search. In *WWW*. 399–408.

[8] C. Douglass Bateman, Christopher S. Helvig, Gabriel Robins, and Alexander Zelikovsky. 1997. Provably good routing tree construction with multi-port terminals. In *ISPD*. 96–102.

[9] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. 2002. Keyword Searching and Browsing in Databases using BANKS. In *ICDE*. 431–440.

[10] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *J. Math. Soc.* 25, 2 (2001), 163–177.

[11] Wei Chen, Christian Sommer, Shang-Hua Teng, and Yajun Wang. 2009. Compact Routing in Power-Law Graphs. In *DISC (Lecture Notes in Computer Science)*, Vol. 5805. 379–391.

[12] Gong Cheng and Evgeny Kharlamov. 2017. Towards a semantic keyword search over industrial knowledge graphs (extended abstract). In *BigData*. 1698–1700.

[13] Joel Coffman and Alfred C. Weaver. 2014. An Empirical Performance Evaluation of Relational Keyword Search Techniques. *IEEE Trans. Knowl. Data Eng.* 26, 1 (2014), 30–42.

[14] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. 2014. Robust Distance Queries on Massive Networks. In *ESA*. 321–333.

[15] Bolin Ding, Jeffrey Xu Yu, Shan Wang, Lu Qin, Xiao Zhang, and Xuemin Lin. 2007. Finding Top-k Min-Cost Connected Trees in Databases. In *ICDE*. 836–845.

[16] Jianhua Feng, Guoliang Li, and Jianyong Wang. 2011. Finding Top-k Answers in Keyword Search over Relational Databases Using Tuple Units. *IEEE Trans. Knowl. Data Eng.* 23, 12 (2011), 1781–1794.

[17] Haizhou Fu and Kemafor Anyanwu. 2011. Effectively Interpreting Keyword Queries on RDF Databases with a Rear View. In *ISWC*. 193–208.

[18] Grettel García, Yenier Izquierdo, Elisa Menendez, Frederic Dartayre, and Marco A. Casanova. 2017. RDF Keyword-based Query Technology Meets a Real-World Dataset. In *EDBT*. 656–667.

[19] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. 2008. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *WEA*. 319–333.

[20] Andrey Gubichev and Thomas Neumann. 2012. Fast approximation of steiner trees in large graphs. In *CIKM*. 1497–1501.

[21] Shuo Han, Lei Zou, Jeffrey Xu Yu, and Dongyan Zhao. 2017. Keyword Search on RDF Graphs - A Query Graph Assembly Approach. In *CIKM*. 227–236.

[22] Faegheh Hasibi, Fedor Nikolaev, Chenyan Xiong, Krisztian Balog, Svein Erik Bratsberg, Alexander Kotov, and Jamie Callan. 2017. DBpedia-Entity v2: A Test Collection for Entity Search. In *SIGIR*.

[23] Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. 2007. BLINKS: ranked keyword searches on graphs. In *SIGMOD*. 305–316.

[24] Ian Horrocks, Martin Giese, Evgeny Kharlamov, and Arild Waaler. 2016. Using Semantic Technology to Tame the Data Variety Challenge. *IEEE Internet Computing* 20, 6 (2016), 62–66.

[25] Edmund Ihler. 1990. Bounds on the quality of approximate solutions to the Group Steiner Problem. In *WG*. 109–118.

[26] Edmund Ihler. 1991. The Complexity of Approximating the Class Steiner Tree Problem. In *WG*. 85–96.

[27] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. 2005. Bidirectional Expansion For Keyword Search on Graph Databases. In *VLDB*. 505–516.

[28] Gjergji Kasneci, Maya Ramanath, Mauro Sozio, Fabian M. Suchanek, and Gerhard Weikum. 2009. STAR: Steiner-Tree Approximation in Relationship Graphs. In *ICDE*. 868–879.

[29] Evgeny Kharlamov, Dag Hovland, Martin G. Skjæveland, Dimitris Bilidas, Ernesto Jiménez-Ruiz, Guohui Xiao, Ahmet Soylu, Davide Lanti, Martin Rezk, Dmitriy Zheleznyakov, Martin Giese, Hallstein Lie, Yannis E. Ioannidis, Yannis Kotidis, Manolis Koubarakis, and Arild Waaler. 2017. Ontology Based Data Access in Statoil. *J. Web Semant.* 44 (2017), 3–36.

[30] Evgeny Kharlamov, Yannis Kotidis, Theofilos Mailis, Christian Neuenstadt, Charalampos Nikolaou, Özgür L. Özçep, Christoforos Svingos, Dmitriy Zheleznyakov, Yannis E. Ioannidis, Steffen Lamparter, Ralf Möller, and Arild Waaler. 2019. An ontology-mediated analytics-aware approach to support monitoring and diagnostics of static and streaming data. *J. Web Semant.* 56 (2019), 30–55.

[31] Evgeny Kharlamov, Theofilos Mailis, Gulnar Mehdi, Christian Neuenstadt, Özgür L. Özçep, Mikhail Roshchin, Nina Solomakhina, Ahmet Soylu, Christoforos Svingos, Sebastian Brandt, Martin Giese, Yannis E. Ioannidis, Steffen Lamparter, Ralf Möller, Yannis Kotidis, and Arild Waaler. 2017. Semantic access to streaming and static data at Siemens. *J. Web Semant.* 44 (2017), 54–74.

[32] Evgeny Kharlamov, Gulnar Mehdi, Ognjen Savkovic, Guohui Xiao, Elem Güzel Kalayci, and Mikhail Roshchin. 2019. Semantically-enhanced rule-based diagnostics for industrial Internet of Things: The SDRL language and case study for Siemens trains and turbines. *J. Web Semant.* 56 (2019), 11–29.

[33] Evgeny Kharlamov, Martin G. Skjæveland, Dag Hovland, Theofilos Mailis, Ernesto Jiménez-Ruiz, Guohui Xiao, Ahmet Soylu, Ian Horrocks, and Arild Waaler. 2018. Finding Data Should be Easier than Finding Oil. In *BigData*. 1747–1756.

[34] Benny Kimelfeld and Yehoshua Sagiv. 2006. Finding and approximating top-k answers in keyword proximity search. In *PODS*. 173–182.

[35] Wangchao Le, Feifei Li, Anastasios Kementsietsidis, and Songyun Duan. 2014. Scalable Keyword Search on Large RDF Data. *IEEE Trans. Knowl. Data Eng.* 26, 11 (2014), 2774–2788.

[36] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. 2015. DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. *Semant. Web* 6, 2 (2015), 167–195.

[37] Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. 2008. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD*. 903–914.

[38] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. 2016. Efficient and Progressive Group Steiner Tree Search. In *SIGMOD*. 91–106.

[39] Ye Li, Leong Hou U, Man Lung Yiu, and Ngai Meng Kou. 2017. An Experimental Study on Hub Labeling based Shortest Path Algorithms. *PVLDB* 11, 4 (2017), 445–457.

[40] Alexander H. Miller, Adam Fisch, Jesse Dodge, Amir-Hossein Karimi, Antoine Bordes, and Jason Weston. 2016. Key-Value Memory Networks for Directly Reading Documents. In *EMNLP*.

[41] Natalya Fridman Noy, Yuqing Gao, Anshu Jain, Anant Narayanan, Alan Patterson, and Jamie Taylor. 2019. Industry-scale knowledge graphs: lessons and challenges. *Commun. ACM* 62, 8 (2019), 36–43.

[42] Dian Ouyang, Lu Qin, Lijun Chang, Xuemin Lin, Ying Zhang, and Qing Zhu. 2018. When Hierarchy Meets 2-Hop-Labeling: Efficient Shortest Distance Queries on Road Networks. In *SIGMOD*. 709–724.

[43] Atish Das Sarma, Sreenivas Gollapudi, Marc Najork, and Rina Panigrahy. 2010. A sketch-based distance oracle for web-scale graphs. In *WSDM*. 401–410.

[44] Saeedeh Shekarpour, Edgard Marx, Axel-Cyrille Ngonga Ngomo, and Sören Auer. 2015. SINA: Semantic interpretation of user queries for question answering on interlinked data. *J. Web Semant.* 30 (2015).

[45] Christian Sommer. 2014. Shortest-path queries in static networks. *ACM Comput. Surv.* 46, 4 (2014), 45:1–45:31.

[46] Mikkel Thorup and Uri Zwick. 2005. Approximate distance oracles. *J. ACM* 52, 1 (2005), 1–24.

[47] Thanh Tran, Philipp Cimiano, Sebastian Rudolph, and Rudi Studer. 2007. Ontology-Based Interpretation of Keywords for Semantic Search. In *ISWC + ASWC*. 523–536.

[48] Thanh Tran, Haofen Wang, Sebastian Rudolph, and Philipp Cimiano. 2009. Top-k Exploration of Query Candidates for Efficient Keyword Search on Graph-Shaped (RDF) Data. In *ICDE*. 405–416.

[49] Stefan Voß. 1992. Steiner's Problem in Graphs: Heuristic Methods. *Discr. Appl. Math.* 40, 1 (1992), 45–72.

[50] Mohan Yang, Bolin Ding, Surajit Chaudhuri, and Kaushik Chakrabarti. 2014. Finding Patterns in a Knowledge Base using Keywords to Compose Table Answers. *PVLDB* 7, 14 (2014), 1809–1820.

[51] Jeffrey Xu Yu, Lu Qin, and Lijun Chang. 2010. Keyword Search in Relational Databases: A Survey. *IEEE Data Eng. Bull.* 33, 1 (2010), 67–78.

[52] Hongyang Zhang, Huacheng Yu, and Ashish Goel. 2019. Pruning based Distance Sketches with Provable Guarantees on Random Graphs. In *WWW*. 2301–2311.

[53] Qi Zhou, Chong Wang, Miao Xiong, Haofen Wang, and Yong Yu. 2007. SPARK: Adapting Keyword Query to Semantic Search. In *ISWC + ASWC*. 694–707.

[54] Andy Diwen Zhu, Hui Ma, Xiaokui Xiao, Siqiang Luo, Youze Tang, and Shuigeng Zhou. 2013. Shortest path and distance queries on road networks: towards bridging theory and practice. In *SIGMOD*. 857–868.