Data Science and Engineering (2021) 6:142–162
https://doi.org/10.1007/s41019-021-00154-4

Keyword Search on Large Graphs: A Survey

Jianye Yang[1] · **Wu Yao**[1] · **Wenjie Zhang**[2]

Received: 30 October 2020 / Revised: 11 January 2021 / Accepted: 16 March 2021 / Published online: 31 March 2021

Abstract
With the prevalence of Internet access and online services, various big graphs are generated in many real applications (e.g., online social networks and knowledge graphs). An important task on analyzing and mining these graphs is keyword search. Essentially, given a graph G and query Q associated with a set of keywords, the keyword search aims to find a substructure (e.g., rooted tree or subgraph) S in G such that nodes in S collectively cover part of or all keywords in Q, and in the meanwhile, S is optimal on some user specified semantics. Keyword search on graphs can be applied in many real-life applications, such as point-of-interests recommendation and web search facility. In spite of the great importance of graph keyword search, we, however, notice that the latest survey on this topic is far out of date. Consequently, there is prompt need to conduct a comprehensive survey in this research direction. Motivated by this, in this survey, we systematically review graph keyword search studies by classifying the existing works into different categories according to the specific problem definition. This survey aims to provide the researchers a comprehensive understanding of existing graph keyword search solutions.

Keywords Keyword search · Big graph · Algorithm · Index structure

1 Introduction

With the prevalence of Internet access and online services, various big graphs are generated in many real applications (e.g., online social networks and knowledge graphs). An important task on analyzing and mining these graphs is keyword search, which can be informally described as follows. Given a graph G and query Q associated with a set of keywords, the keyword search aims to find a substructure (e.g., rooted tree or subgraph) S in G such that nodes in S collectively cover part of or all keywords in Q, and in the meanwhile, S is optimal on some user specified semantics.

Let us illustrate graph keyword search by an example. Figure 1a shows a data graph G with 12 vertices, each of which contains a set of keywords. Given a query $Q = \{c, d\}$, we find two candidate result trees T_1 and T_2 in Fig. 1b, which are returned following the tree-based search model. Then if the minimum total edge weights are used to rank the answers, T_1 is considered to be the top-1 result since it has least weight.

Keyword search is a prominent operation for analyzing graph data, which allows users to query the investigated graph data without a prior knowledge of specialized query languages. It is applicable to many real-life applications. Here are some typical applications, to name a few:

- *POIs recommendation*. Many location-based services provide point-of-interest (POI) recommendation for users. Consider a tourist who wants to spend a day exploring a city. She might pose a query containing a set of keywords, e.g., "hotel", "restaurant", "shopping mall", etc. Intuitively, a good recommendation tends to return POIs staying close to each other. This implies that the query result is a compact subgraph in the road network.
- *Web search facility*. Keyword search on the Web are ubiquitous in our daily life. For example, a user may be interested in two actors, e.g., "Jason Statham" and "Dwayne Johnson". A decent keyword search result should present to the user the whole picture about the two actors, such as the movies they co-acted. This schema-free keyword

✉ Jianye Yang
jyyang@hnu.edu.cn

Wu Yao
wyao@hnu.edu.cn

Wenjie Zhang
zhangw@cse.unsw.edu.au

[1] Hunan University, Changsha, China

[2] University of New South Wales, Sydney, NSW, Australia

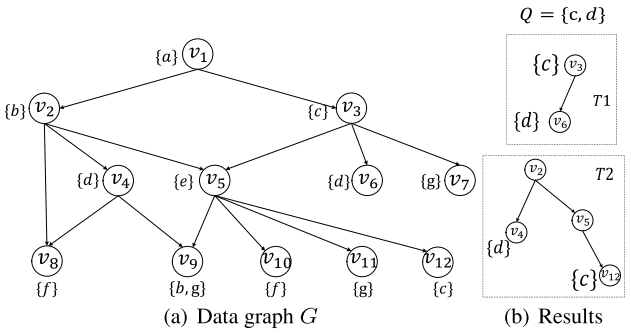


Fig. 1 An example of graph keyword search

search has been witnessed a great necessity for nowadays web search, especially for large-scale knowledge bases.

- *Keyword-aware routing*. A user might want to drive in a city from one place to another and, at the same time, pass through several POIs (e.g., "supermarket", "gas station", "bank", etc). A high-quality route plan is required to present a route passing through all these POIs with least travel time. Clearly, keyword-aware routing in road network is another important application of graph keyword search.

Owing to the great value of real-life application, keyword search over graphs is an important research topic in the past two decades. In spite of this fact, we are surprisingly noticing that the latest surveys [83, 99], to the best of our knowledge, for graph keyword search, were conducted one decade ago. However, in the past decade, we have witnessed rapid development for graph data analysis, including graph keyword search. A lot of works have been proposed in this period of time. Therefore, there is a prompt need to conduct a comprehensive up-to-date survey in this research direction. Besides, the existing surveys [83, 99] mainly focus on keyword search on schema-based graphs, such as XML data or relational graphs, which are not suitable to deal with the present-day schema-free large graphs. For these schema-free graphs, the existing surveys only discuss several pioneer approaches, such as BANKS [5, 49], BLINKS [39], and DPBF [20].

Motivated by the above issues, in this survey, we systematically review recent advances on keyword search on schema-free graphs. In general, this survey focuses on keyword search algorithms over graph data under different ranking models for the desired answer. As shown in Table 1, we provide a systematic classification for studies on graph keyword search. Particularly, we classify these studies according to the answer ranking models, including tree-based keyword search, nearest neighbor-based keyword search, subgraph-based keyword search, and other semantics-based keyword search. Compared to the existing surveys [83, 99], the last three classes of studies are new, which have attracted much research attention in the past decade. For each class of works, we first give the formal definition for the specific research problem and then review the representative studies.

This paper is organized as follows: In Sect. 2, we introduce basic concepts related to keyword search on graph. From Sects. 3 to 6, we comprehensively discuss graph keyword search solutions in each category. We review the related work in Sect. 7. In Sect. 8, we point out a list of future research directions. Finally, we conclude this paper in Sect. 9.

2 Preliminaries

In this section, we introduce important concepts and notations which are used thoughout the paper.

2.1 Data Graph

We consider a weighted and directed graph $G = \langle V, E \rangle$, where $V(G)$ is a finite set of n vertices (i.e., $|V(G)| = n$) and $E(G) \subseteq V(G) \times V(G)$ is a finite set of m edges (i.e., $|E(G)| = m$). For ease of exposition, in this paper, we only consider the weight on the edges.

In particular, given an edge $e \in E(G)$, the weight on e, denoted by $\mathsf{weight}(e)$, is a nonnegative real number, where smaller weights are preferred. For example, in a road

Table 1 Classification of works of graph keyword search ("P" means Problem)

Semantics		Problem and References
Tree-based semantics	Steiner tree-based	P1 [5, 20, 30, 54, 55, 62, 79]
	Distinct root tree-based	P2 [16, 39, 49, 60]
Nearest neighbor-based semantics	Top-k nearest neighbors	P3 [3, 41, 48, 70, 81, 109, 113]
	Top-k relevant neighbors	P4 [1, 66, 73, 111]
Subgraph-based semantics		P5 [61], P6 [71], P7 [51, 52, 107], P8 [7], P9 [114, 115]
Other semantics		P10 [102], P11 [27], P12 [27, 37, 87, 97, 108]

network, the weight on an edge denotes the estimated travel time. A path $p = (v_1, v_2, \ldots, v_l)$ is a sequence of l nodes in $V(G)$ such that for each $v_i (1 \leq i < l)$, $(v_i, v_{i+1}) \in E(G)$. The weight of a path is the total weight of all edges on the path. For two vertices $u \in V(G)$ and $v \in V(G)$, the distance from u to v, denoted by $\mathsf{dist}(u, v)$, is the minimum weight of all paths from u to v in G.

Given a vertex $v \in V(G)$, we use $\mathsf{nbr}_{\mathsf{in}}(v, G)$ (resp. $\mathsf{nbr}_{\mathsf{out}}(v, G)$) to the incoming (resp. outcoming) neighbors of v. The degree of a vertex v is the number of neighbors of v in G. Specifically, we use $\mathsf{deg}_{\mathsf{in}}(v, G)$ and $\mathsf{deg}_{\mathsf{out}}(v, G)$ to denote the in-degree and out-degree of vertex v in G, respectively. That is, $\mathsf{deg}_{\mathsf{in}}(v, G) = |\mathsf{nbr}_{\mathsf{in}}(v, G)|$ and $\mathsf{deg}_{\mathsf{out}}(v, G) = |\mathsf{nbr}_{\mathsf{out}}(v, G)|$. In the following, for presentation simplicity, we omit G in the notations if the context is obvious and refer to a weighted and directed graph simply as a graph.

2.2 Keyword Terminology

In a graph G, each vertex $v \in V$ contains a set of zero or more keywords which is denoted as $\mathsf{doc}(v)$, and the union of keywords for all vertices in G is denoted as $\mathsf{doc}(V)$. According to the specific application, a keyword may denote the label, attribute, or text information of a vertex. Given a keyword w, all vertices in graph G containing w are called (keyword) hitting vertices of w, which are denoted as $\mathsf{hit}(w)$. For a vertex v, if v contains at least one keyword, we call v the keyword vertex.

3 Tree-Based Keyword Search

In this section, we review graph keyword search works that return tree structures as the desired answer, which is referred to as tree-based keyword search. According to the applied cost functions for the returned trees, we further classify the related works into two categories, namely Steiner tree-based semantics and distinct root-based semantics. Next, we discuss them in detail.

3.1 Steiner Tree-Based Semantics

The problem based on Steiner tree semantics is formally described as follows.

Problem 1 Given a weighted graph $G = (V, E)$, a set of query keywords $Q = \{w_1, w_2, \ldots, w_l\}$, and a cost function f, return a tree $T(V_T, E_T)$ of G, such that

1. V_T covers all keywords in Q;
2. $f(T)$ is minimized among all feasible choices for T, where $f(T) = \sum_{e \in E_T} \mathsf{weight}(e)$.

It is easy to verify that Problem 1 aims to return a Steiner tree T as the result. The leaves of T only come from the keyword hitting vertices of G, i.e., $\mathsf{leaves}(T) \subseteq \mathsf{hit}(w_1) \cup \mathsf{hit}(w_2) \cup \cdots \cup \mathsf{hit}(w_l)$, since, otherwise, we can recursively remove all non-hitting leaves of T to get a new tree T' without breaking the condition 1 in Problem 1. Under the Steiner tree-based semantics, Problem 1 is the well-known *group Steiner tree* (GST) which is NP-complete [21] in general. In [44], Ihler showed that the GST problem cannot be approximated within a constant performance ratio by any polynomial algorithm unless $P = NP$. In theory, there can be exponentially many feasible choices for T, i.e., $O(2^m)$ where m is the number of edges in G. This implies that an exact algorithm for solving Problem 1 will take exponential computation cost, and thus, it is impractical for large graphs.

In the literature, many approximate solutions have been proposed to solve the GST problem. In the theoretical computer science community, several LP (linear programming)-based approximation algorithms [9, 29] have been devised, which, however, are very hard to handle medium-sized graphs, since these algorithms need to invoke the expensive LP procedure. Therefore, we focus on the database community and introduce several practical approximation algorithms [5, 20, 30, 54, 55, 62, 79] that are mostly devised for the keyword search application.

• *A backward search algorithm.* Bhalotia et al. [5] propose a backward search algorithm, called BANKS-I, searching backwards from the hitting vertices. Given a query $Q = \{w_1, w_2, \ldots, w_l\}$, we first find the hitting vertices $\mathsf{hit}(w_i)$ for each keyword w_i, which can be facilitated by using an inverted list index to store the hitting vertices. Let $H = \mathsf{hit}(w_1) \cup \mathsf{hit}(w_2) \cup \cdots \cup \mathsf{hit}(w_l)$ be the overall hitting vertices in G relevant to query Q. Then, we create $|H|$ iterators to concurrently execute $|H|$ copies of Dijkstra's single-source shortest path algorithm, one for each keyword vertex v in H with v as the source. At any point during the execution of the algorithm, we maintain l clusters, denoted by $C_1, C_2, \ldots, C_l$, one for each of the l keywords. Cluster C_i denotes the set of vertices that we know can reach query keyword w_i, which is initialized as $\mathsf{hit}(w_i)$. In each search iteration, we choose a previously visited vertices v and select one of its incoming edges backward to the source vertex u. Then, any C_i containing v now expands to include u as well. Once a node is visited, all its incoming edges become known to the search and available for choice in future iterations. The idea of this concurrent backward search is to find a common node from which there exists a shortest path to at least one node in each set $\mathsf{hit}(w_i)$. Such paths will define a rooted directed tree with the common node as the root and the corresponding hitting vertices as the leaves.

The key to facilitate the search efficiency is which visited vertex to expand in each iteration. BANKS-I proposes two strategies as follows.

Equi-distance expansion in each cluster: This strategy decides which node to visit for expanding a keyword. Intuitively, the algorithm expands a cluster by visiting vertices in increasing order of distance from the leaves. In specific, the vertex u to visit next for cluster C_i (by following edge $u \to v$ backward for some $v \in C_i$) is the vertex with the shortest distance (among all vertices not in C_i) to $\mathsf{hit}(w_i)$.

Distance-balanced expansion across clusters: This strategy decides which keyword to expand next. In a high-level point of view, the algorithm attempts to balance the distance between each leaves to its frontier across all clusters. In particular, let (u, C_i) be the vertex-cluster pair such that $u \notin C_i$ and the distance from u to C_i is the shortest possible. Then, the cluster to expand next is C_i.

• *A dynamic programming algorithm*. Although it is NP-complete to find the optimal Steiner tree in general, Ding et al. [20] propose an efficient dynamic programming algorithms, called DPBF, to find the optimal Steiner tree for the cases where the number of keywords l is small. Let $\mathbf{q}$, $\mathbf{q_1}$, $\mathbf{q_2}$ be a non-empty subset of the query $Q = \{w_1, w_2, \ldots, w_l\}$. For presentation simplicity, we use $T(v, \mathbf{q})$ to denote both the tree and its weight with the minimum weight among all the trees rooted at v and containing a set of keywords $\mathbf{q}$. By maintaining trees in a priority queue $\mathcal{T}$, DPBF can find the optimal tree $T(v, \mathbf{q})$ for each $v \in V(G)$ and $\mathbf{q} \subseteq Q$. Initially, for each keyword vertex v and a keyword $w \in \mathsf{doc}(v)$, $T(v, \{w\})$ is a single vertex tree rooted at v with a zero weight, i.e., $T(v, \{w\}) = 0$. For a general case where a tree consists of more than one vertices, $T(v, \{w\})$ can be computed by the following equations.

$$T(v, \mathbf{q}) = \min(T_g(v, \mathbf{q}), T_m(v, \mathbf{q})) \tag{1}$$

$$T_g(v, \mathbf{q}) = \min_{\langle v,u\rangle \in E(G)} \{\langle v, u\rangle \oplus T(u, \mathbf{q})\} \tag{2}$$

$$T_m(v, \mathbf{q_1} \cup \mathbf{q_2}) = \min_{\mathbf{q_1} \cap \mathbf{q_2} = \emptyset} \{T(v, \mathbf{q_1}) \oplus T(v, \mathbf{q_2})\} \tag{3}$$

Here min means to choose the tree with minimum weight, and $\oplus$ is an operation to merge two trees into a new tree. Note that, $T(v, \mathbf{q})$ may not exist for some v and $\mathbf{q}$, which implies that vertex v cannot reach the hitting vertices for some keywords in $\mathbf{q}$. In this case, $T(v, \mathbf{q}) = 0$. In general, Eqs. (2) and (3) reflect two tree expanding cases, namely tree grow and tree merge, respectively. Since all trees are stored in the priority queue by the increasing order of weight of trees, DPBF ensures to find the optimal tree first among all feasible trees covering all the keywords in $Q = \{w_1, w_2, \ldots, w_l\}$. Besides, the time complexity of DPBF is $O(3^l n + 2^l((l + n) \log n + m))$, which is reduced to $O(n \log n + m))$ for small and fixed l.

• *A progressive algorithm*. Although it is shown to be efficient to find the optimal solution in reasonable time when the number of keywords l is very small [13], DPBF still suffers two major limitations. First, due to the exponential time and space complexity, DPBF quickly becomes impractical even for small l (e.g., $l = 8$) in large graphs. Second, it cannot generate a solution until the algorithm has completed its entire execution. Against this background, Li et al. [62] propose an efficient progressive programming algorithm, called PrunedDP, which is devised on top of DPBF.

In PrunedDP, a state, denoted by (v, X), corresponds to a connected tree rooted at v that covers all keywords in $X \subseteq Q$. Let $T(v, X)$ be the minimum-weight connected tree corresponding to state (v, X), and $f_T^*(v, X)$ be the weight of $T(v, X)$. The general idea of PrunedDP is that we construct a feasible solution for an intermediate state (v, X), then keep refining the feasible solution until $X = Q$. To facilitate the search processing, for a keyword $w \in Q$, we create a virtual node $\tilde{v}_w$, and create an undirected edge $(\tilde{v}_w, v)$ with zero weight for each $v \in \mathsf{hit}(w)$. For each state (v, X), let $\bar{X} = Q \setminus X$. For a vertex v and label set $\bar{X}$, we merge all $|\bar{X}|$ pre-computed shortest paths from v to $\tilde{v}_w$ for all $w \in \bar{X}$, resulting in a tree denoted by $T'(v, \bar{X})$. Then, by uniting trees $T(v, X)$ and $T'(v, \bar{X})$, we can obtain a minimum spanning tree of the united result, i.e., $\mathsf{MST}(T(v, X) \cup T'(v, \bar{X}))$, denoted by $\tilde{T}(v, Q)$. Clearly, $\tilde{T}(v, Q)$ is a feasible solution, since it covers all keywords in Q.

Although the above basic version of PrunedDP is more efficient than DPBF, it still needs to search a large number of states to find the optimal solution. That is because the optimal solution is popped later from the priority queue than any computed intermediate state (v, X) due to best-first search strategy. To avoid such expensive computation cost, PrunedDP is further equipped with two advanced techniques, namely optimal-tree decomposition and conditional tree merging.

Optimal-tree decomposition theorem states that, for the optimal tree $T^*(Q)$, there always exists a vertex $u \in T^*(Q)$ such that (i) the tree $T^*(Q)$ rooted at u has $k (k \geq 1)$ subtrees $T_1, T_2, \ldots, T_k$, and (ii) each subtree T_i has a weight smaller than $f^*(Q)/2$ where $f^*(Q)$ is the weight of $T^*(Q)$. This result motivates us to first compute all optimal subtrees that have weights smaller than $f(best)/2$ where $best$ is the best feasible solution seen so far and then obtain the optimal tree via merging the optimal subtrees.

Conditional tree merging theorem states that, to expand a state (v, X) by a tree merging operation in PrunedDP, we can merge two subtrees $T(v, X)$ and $T(v, X')$ for $X' \subset Q \setminus X$ only when the total weight of these two subtrees is no larger

than $2/3 \times f(best)$. By this theorem, we can further reduce a number of states generating in PrunedDP without loss of optimality.

To further speed up the PrunedDP, the authors propose a novel progressive algorithm, called PrunedDP++, based on the A^*-search strategy over the pruned search space. The key of PrunedDP++ is to establish an effective lower bound for each state (v, X) in the search space, which is usually constructed via relaxing the constraints of the optimal subtree $T(v, \bar{X})$. Let $\pi(v, X)$ be the constructed lower bound. PrunedDP++ makes use of $f_T^*(v, X) + \pi(v, X)$ as the priority for each state (v, X) to perform best-first search.

• *An index-based algorithm*. Recently, Shi et al. [79] propose an index-based method, called KeyKG, to deal with keyword search over large knowledge graphs. In a high-level viewpoint, KeyKG finds a GST in two stages. First, it greedily selects a set of keyword vertices that are close to each other, denoted by U_x, which contains one vertex from each $\mathsf{hit}(w_i)$ for $1 \le i \le l$. Then, it greedily finds a GST to span U_x, denoted by T_{min}, which is iteratively expanded with shortest paths.

In specific, for each vertex $v_1 \in \mathsf{hit}(w_1)$, KeyKG finds a vertex v_i in each remaining $\mathsf{hit}(w_i)$ with minimum distance from v_1. 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. Further, let $x \in \mathsf{hit}(w_1)$ be the vertex with the smallest value of W_{v_1}. Clearly, U_x covers the query Q and vertices in it are intuitively close to each other. Therefore, a GST that spans vertices in U_x may have a small weight.

To find a promising GST spanning U_x, KeyKG attempts starting from each vertex $u \in U_x$ and selects the one with the minimum weight among these $|U_x|$ GSTs. Specifically, each T_u is initialized with a single vertex u. Then, we iteratively span the remaining vertices in U_x. In each following round, we find a vertex pair (s, t) with the smallest distance where $s \in T_u$ and $t \in U_x - T_u$. A shortest path p between s and t is found and added to T_u. Following this strategy, we find $|U_x|$ trees, each corresponding to a vertex $u \in U_x$. Finally, KeyKG returns the tree with minimum weight.

Apparently, the performance of KeyKG heavily relies on the computation of shortest distance and path between two vertices. To accelerate the performance, KeyKG utilizes the Hub Labeling (HL) technique [2]. To construct a compact HL, vertices are sorted in descending order of betweenness centrality such that labels constructed in earlier iterations to support the computation of distances between more pairs of vertices. To further boost the performance, the authors devise a dynamic HL which is query-relevant and thus is online-constructed. By using this dynamic HL, one can reduce a number of the merge sort-like operations to find a hub in the static label when computing the shortest distance between vertex pairs.

• *Enumerating with polynomial delay algorithm*. Due to the NP-completeness of Problem 1, the above introduced algorithms cannot guarantee the quality (i.e., approximation ratio) of non-first results or the delay between consecutive results. In [30, 55], the authors aim to enumerate answers in θ-approximate increasing *weight/height* order with polynomial delay.

θ-approximate order. Given an answer tree T, the rank of T, denoted by $\mathsf{rank}(T)$, is the weight or height of T, where smaller is better. Let $T_1, T_2, \ldots, T_s$ be a sequence of all answer trees. Ideally, the trees should be enumerated in an increasing ranked order. However, it is not practical due to the intractable nature of Problem 1. Instead, the authors turn to find θ-approximate increasing ranked order, where θ-approximate order means that if one answer precedes another, then the first is worse than the second by at most a factor of θ. More formally, the answer sequence $T_1, T_2, \ldots, T_s$ is in a θ-approximate order if $\mathsf{rank}(T_i) \le \theta \cdot \mathsf{rank}(T_j)$ for all $1 \le i \le j \le s$.

Polynomial delay. The efficiency of an enumeration algorithm is measured in terms of the delay between printing each pair of consecutive answers. We say that an algorithm enumerates with polynomial delay if there is a polynomial $p(n)$, where n is the size of the input (i.e., G and Q), such that the time needed to produce the next answer is always bounded by $p(n)$.

In [30, 55], the authors apply shortest-path iterators to find the first answer, namely a minimal-rank feasible tree, and then adapt the Lawler's procedure [59] to enumerate the remaining answers without redundancies. In specific, the algorithm uses two types of constraints: *inclusion constraints* and *exclusion constraints*, each of which contains a set of edges. An answer tree T satisfies a set I of inclusion constraints and a set E of exclusion constrains if it includes all the edges of I and none of E. The key in their algorithm is to devise a subroutine QSubtree(G,Q,I,E). By investigating the *partial answer* w.r.t. the query Q, the authors develop a polynomial-time algorithm for QSubtree(G,Q,I,E) with 2-approximation in terms of tree height [30].

3.2 Distinct Root-Based Semantics

Since the problem under Steiner tree-based semantics (i.e., Problem 1) is generally a hard problem, many works resort to easier semantics. In this section, we discuss another problem, which is based on distinct root semantics as below.

Problem 2 Given a weighted graph $G = (V, E)$, a set of query keywords $Q = \{w_1, w_2, \ldots, w_l\}$, and a cost function f, return a tree $T(V_T, E_T)$ of G, such that

1. V_T covers all keywords in Q;

2. $f(T)$ is minimized among all feasible choices for T, where $f(T) = \sum_{i=1}^{l} \mathsf{dist}(root(T), leaf(w_i))$.

Note here that $root(T)$ is the root of T, $leaf(w_i)$ is the leaf node containing keyword w_i in T, and $\mathsf{dist}(root(T), leaf(w_i))$ is the distance from the root to $leaf(w_i)$.

Unlike Problem 1, Problem 2 can be resolved in polynomial time as the number of feasible answer trees is at most n, which is the number of vertices in G. In particular, for each vertex $v \in V(G)$, zero or one potential tree rooted at v can be found by uniting the shortest paths from v to each keyword $w_i \in Q$. The final result is the one with minimum $f(T)$ value. Next, we introduce approaches, which are proposed to deal with very large graphs in general.

• *A bidirectional search algorithm*. BANKS-I can be directly applied to handle Problem 2 as it aims to find a common root vertex in the graph by searching backwards. However, the backward search may lead to poor performance in the following two scenarios. First, the query contains keywords with high frequency. Since BANKS-I creates an iterator for each keyword vertex, the algorithm would generate a large number of iterators in this scenario. Second, an iterator reaches a vertex with many incoming edges, which means the algorithm needs to explore a large number of nodes.

To address the above problems, Kacholiaet al. [49] propose a bidirectional search algorithm, called BANKS-II. The main idea of BANKS-II is as follows. First, all the single-source shortest path iterators from BANKS-I are merged into a single iterator, which is called *incoming iterator*. Second, an *outgoing iterator* runs concurrently, which follows the forward edges starting form all vertices explored by the incoming iterator. Third, *spreading activation* is used to prioritize the search, which chooses incoming iterator or outgoing iterator to be called next. Activation is a kind of "scent" spread from keyword vertices, and edge weights are taken into consideration when spreading the activation.

• *A Bilevel index-based algorithm*. In [39], a bilevel index, called BLINKS, is proposed to speed up BANKS-II, as no index (except the keyword-vertex index) is used in BANKS-II. A naive index that precomputes and indexes all the shortest distances from the vertices to keyword vertices is not feasible, as it will incur very large index size when dealing with large graphs and large number of distinct keywords. To reduce the index size, BLINKS uses a divide-and-conquer approach to create a bilevel index, which can be built by first partitioning the graph and then building intra-block index and block index.

In specific, BLINKS applies vertex-based partitioning methods to partition a graph into blocks. In a vertex-based partitioning of a graph, a vertex separator is called a *portal* vertex (or portal for short). A block consists of all vertices in a partition as well as all portals incident to the partition. A portal is called in-portal if it has at least one incoming edge from another block and at least one outgoing edge in this block. Similarly, a portal is called out-portal if it has at least one outgoing edge to another block and at least one incoming edge from this block.

For each block b, the intra-block index (IB-index) is built, which consists of the following data structures, including *Intra-block keyword-vertex lists*, *Intra-block vertex-keyword map*, *Intra-block portal-vertex lists*, and *Intra-block vertex-portal distance map*. These structures are utilized to efficiently fetch the distance information between keywords, vertices, and portals. Besides the intro-block index, the block index is also built, which is a simple data structure consisting of *Keyword-block lists* and *Portal-block lists*.

In BLINKS, to support backward search, we use a priority queue $\mathcal{Q}_i$ of cursors for each query keyword w_i to simulate Dijkstra's algorithm by utilizing the distance information stored in the IB-index. Initially, for each keyword w_i, we use the keyword-block list to find blocks containing w_i. Then, a cursor is used to scan each intra-block keyword-vertex list for w_i and put in query $\mathcal{Q}_i$. When we reach an in-portal u of the current block, we need to continue backward expansion in all blocks that have u as their out-portal, as a shorter path may cross several blocks.

• *An external memory-based algorithm*. Dalvi et al. [16] study the problem of keyword search on graphs that can not fit into main memory. To efficiently address this problem, they first partition the graph into small components using a clustering algorithm and build a much smaller supernode graph, which is defined as follows:

SuperNode: A component is treated as a *supernode* in the top-level graph. Each supernode thus contains a subset of $V(G)$, and the contained nodes are called *innernodes*.

SuperEdge: An *superedge* is constructed between two supernodes s_1 and s_2 if there is at least one edge from an innernode of s_1 to an innernode of s_2, and the weight of the superedge is the minimum weight over all such edges.

The supernode graph is constructed such that it fits into the available amount of main memory. Each supernode has a fixed number of innernodes and is stored on disk. On top of the supernode graph, a multi-granular graph structure is proposed to exploit information present in lower-level nodes (i.e., innernodes) that are cache-resident at the time a query is executed. A multi-granular graph is a hybrid graph that contains both supernodes and innernodes.

When searching the multi-granular graph, the answers generated may contain supernodes, called supernode answer. If an answer does not contain any supernodes, we call it a pure answer. The final answer returned to users must be pure answer. The *Iterative Expansion* algorithm is a multistage algorithm, which is applicable to multi-granular graphs. Each iteration of Iterative Expansion is broken up into two phases as follows:

Explore phase: Run an in-memory search algorithm on the current state of the multi-granular graph that is entirely in memory. The details of expanded supernodes are stored in cache. When the search reaches an expanded supernode, it searches on the corresponding innernodes in cache.

Expand phase: Expand the supernodes found in top-s ($s > k$) results of the previous phase and add them to input graph to produce an expanded multi-granular graph.

The graph produced at the end of Expand phase of iteration i acts as the graph for iteration $i + 1$. The algorithm stops when all top-k results are pure.

• *A graph summarization-based algorithm*. Le et al. [60] study keyword search on large RDF data. The authors first condense the RDF data into a generic graph by merging the entity vertex together with its associated keyword and type vertices. To speed up the search performance, the authors propose a type-based summarization approach to summarize the graph. The key observation is that neighborhoods in close proximity surrounding vertices of the same type often share similar structures in how they connect to vertices of other types, whereas a similar effort can be seen in [82].

To implement graph summarization, the authors apply a very similar strategy that is used in [16]. That is we first split the graph into multiple, smaller partitions and then define a minimal set of common type-based structures that summarize the partitions. Since the graph partitioning is a well-studied problem in the literature, the authors focus on how to build semantically similar partitions. The summarization algorithm identifies a set of templates from the set of partitions. Such templates serve as a summary for the partitions $\mathcal{P}$. In addition, the summarization algorithm guarantees that every partition in $\mathcal{P}$ is homomorphic to one of the templates in the summary. This property allows the query optimizer to (*i*) efficiently estimate any path length in the backward expansion without frequently accessing the RDF data being queried and (*ii*) efficiently reconstruct the partitions of interest by querying the RDF data without explicitly storing and indexing the partitions.

Based on the summarized graph, the authors present an exact search algorithm, which performs a two-level backward search: one backward search at the summary level and one at the data level. The backward search is only initiated at the data level on the partitions that are found to contain all the distinct keywords at the summary level and whose score could enter the top-k answers. A early termination condition is devised by maintaining the candidate answers in a priority queue.

3.3 Discussion

In this section, we review graph keyword search studies that return tree structures as the desired answer. According to the applied cost functions for the returned trees, we divide them into two groups, where the first group [5, 20, 30, 54, 55, 62, 79] employ Steiner tree-based semantics, while the second group [16, 39, 49, 60] employ distinct root-based semantics.

In particular, the Steiner tree-based semantics uses the total weight of edges in the answer tree as the cost, which makes the problem NP-complete in general. The distinct root-based semantics uses the total weight of paths from root to keyword vertices as the cost, which makes it solvable in polynomial time.

4 Nearest Neighbor-Based Keyword Search

In this section, we review graph keyword search works that return k best vertices as the desired answers, which are usually the nearest or most relevant neighbors to the query vertex. Based on the specific calculation of scoring function, we further divide these works into two groups. One group is top-k nearest neighbor keyword search, which considers the distance only when ranking vertices. The other is top-k relevant neighbor keyword search, which combines both textual relevance and distance. Next, we discuss them in detail.

4.1 Top-k Nearest Neighbor Keyword Search

The problem of top-k nearest neighbor keyword search is formally described as follows.

Problem 3 (k-NK) Given a weighted graph $G = (V, E)$ and a query $Q = (q, w, k)$, where $q \in V$ is a query vertex in G, w is a keyword, and k is a positive integer, return a set of k keyword vertices, denoted by $R = \{v_1, v_2, \ldots, v_k\} \subseteq \mathsf{hit}(w)$, and there does not exist a vertex $u \in \mathsf{hit}(w) \setminus R$ such that $\mathsf{dist}(q, u) < \max_{v \in R} \mathsf{dist}(q, v)$.

A straightforward approach for handling Problem 3 is to use Dijkstra's algorithm to compute the shortest paths from q to all vertices in $\mathsf{hit}(w)$ and return the k vertices with minimum distances to q. The time complexity is $O(|E| + |V| \cdot \log |V|)$. Clearly, this straightforward approach is inefficient when the size of the graph is large. To avoid such cost-prohibitive computation of shortest paths in query phase, many solutions [3, 41, 48, 70, 81, 109, 113] in the literature are proposed to utilize index structures, which are introduced in detail as below.

• *A distance oracle-based algorithm*. Bahmani and Goel [3] propose a distance oracle-based method to answer Problem 3. In general, distance oracle is a technique for estimating the distance of two vertices in a graph [75]. Given a graph $G = \langle V, E \rangle$, a distance oracle is a Voronoi partition of V determined by a set of randomly selected *center vertices*. More specifically, given a number n_c, we randomly select n_c vertices from V as the center vertices to construct a distance

oracle $\mathcal{O}$. Then the partition is constructed by assigning each vertex $v \in V$ to its nearest center vertex c, where the distance from v to its correspondent center node is precomputed. After constructing $\mathcal{O}$, for two vertices u and v in G, if u and v are in the same partition in $\mathcal{O}$ with a center vertex c, then the estimated distance $\overline{\mathsf{dist}}_{\mathcal{O}}(u, v) = \mathsf{dist}_{\mathcal{O}}(u, c) + \mathsf{dist}_{\mathcal{O}}(v, c)$. If u and v are not in the same partition in $\mathcal{O}$, $\overline{\mathsf{dist}}_{\mathcal{O}}(u, v) = \infty$.

To improve the distance estimation accuracy, we usually construct a set of $r = p \times \log |V|$ distance oracles $\{\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_r\}$, where p is a user-specified parameter. Oracle $\mathcal{O}_i$ contains $2^{\lceil i/p \rceil - 1}$ randomly selected center vertices. Given r distance oracles, the overall estimated distance of two vertices u and v in G is then given by $\overline{\mathsf{dist}}(u, v) = \min_{1 \le i \le r} \overline{\mathsf{dist}}_{\mathcal{O}_i}(u, v)$. It is shown in [75] that when $p = \theta(|V|^{1/\log |V|})$, the estimated distance can be bounded by $\mathsf{dist}(u, v) \le \overline{\mathsf{dist}}(u, v) \le (2 \log_2 |V| - 1) \cdot \mathsf{dist}(u, v)$ with a high probability.

The query processing based on distance oracle is as follows. For each keyword in a distance oracle $\mathcal{O}_i$, an inverted list is constructed for each keyword in the partition. Specifically, for a partition with a center vertex c and a keyword w, the inverted list contains all vertices in the partition that contain w ranked in non-decreasing order of their distances to c. Given a query $Q = (q, w, k)$, the algorithm finds the partition that q belongs to in an oracle $\mathcal{O}_i$. The result w.r.t. $\mathcal{O}_i$, denoted by $R_{\mathcal{O}_i}$, is the first k elements in the inverted list for w in the partition. The final result R is computed by merging the vertices in each $R_{\mathcal{O}_i}$ and returning k vertices with the shortest distances to q. The query time complexity is $O(k \cdot \log |V|)$.

• *A shortest path tree-based algorithm*. In practice, the distance error estimated by the distance oracle may be large, which can greatly distort the ranking of the vertices containing the query keyword. To alleviate the issue of distance oracle, Qiao et al. [70] propose to integrate the *tree distance* [80] in estimating the distances. In particular, for a partition of a distance oracle, we construct a shortest path tree rooted at the randomly selected center vertex of the partition. For a distance oracle $\mathcal{O}_i$, let the set of trees constructed in $\mathcal{O}_i$ be T_i, which can be considered as a tree by adding a virtual root and several virtual edges with weight $+\infty$ that connect the new virtual root to every root vertex in T_i, respectively. Let R_T be the result of k-NK query on tree T. Finally, we can solve Problem 3 by merging the result R_{T_i} on all trees $T_i, 1 \le i \le r$.

The key of the shortest path tree-based algorithm on solving Problem 3 is how to compute R_T on a tree T, which is non-trivial. In [70], the authors introduce two algorithms for answering exact k-NK on a tree $T = \langle V, E \rangle$. The first algorithm tree-boundk can only handle bounded k values with query processing time $O(k + \log |\mathsf{hit}(w)|)$ and index size $O(\bar{k} \cdot |\mathsf{doc}(V)|$ for all keywords where $\bar{k}$ is an upper bound value of k, while the second algorithm tree-pivot can handle an arbitrary k with query processing time $k \cdot \log |V|$ and index size $O(|\mathsf{doc}(V)| \cdot \log |V|)$.

In tree-boundk, the authors utilize a data structure, namely Compact Tree CT, which is defined as follows. For a tree T and a keyword w, a compact tree $\mathsf{CT}(w)$ is a tree that keeps only two types of vertices in T: a keyword vertex that contains keyword w, and a vertex that has at least two direct subtrees containing vertices carrying keyword w. The main idea of tree-boundk is to precompute the top-$\bar{k}$ results for every keyword w and every vertex on $\mathsf{CT}(w)$. Since the total size of all compact trees is bounded by $O(|\mathsf{doc}(V)|)$, the total space to store the top-$\bar{k}$ results of vertices on all compact trees is bounded by $O(\bar{k} \cdot |\mathsf{doc}(V)|$. Then, given a k-NK query $Q = (q, w, k)$, if q is on $\mathsf{CT}(w)$, the result can be easily reported by the precomputed answer on $\mathsf{CT}(w)$. The difficult case is when q is not in $\mathsf{CT}(w)$. In high level, this case can be dealt by first finding the *entry vertex* and *entry edge* for q on $\mathsf{CT}(w)$ and then merging the candidate answer on the nearest vertices on $\mathsf{CT}(w)$.

In tree-pivot, a main observation is as follows. For a vertex u containing keyword w and an arbitrary vertex v in a tree T, the path from v to u is unique on T, and can be divided into two segments, namely the segment from v to their lowest common ancestor $\mathsf{LCA}(u, v)$, and the second segment from $\mathsf{LCA}(u, v)$ to u. The basic idea is to compute the first segment online and precompute the results regarding the second segment offline. In the precomputing phase, we propagate the keyword vertex u to all its ancestors in T, such that we can construct a candidate list for each vertex s in T, which records all distance from s to the keyword vertex in the subtree rooted at s. Then, given a k-NK query $Q = (q, w, k)$, we just need to merge all candidate lists of vertices starting from q to the root of T by adding a proper distance. The final answer is the k vertices with shortest distance. A tree balancing technique against T is further devised to optimize both the index space and query processing.

• *A 2-hop labeling-based algorithm*. The above two algorithms for Problem 3 only give approximate answers. In [48], Jiang et al. propose two exact algorithms based on 2-hop labeling techniques [14], namely forward search (FS) and forward backward search (FBS).

Particularly, FS is proposed to deal with the scenario where the keyword is not frequent. For a k-NK query $Q = (q, w, k)$, we compute the distance between the query vertex q and all vertices in $\mathsf{hit}(w)$ by using the 2-hop label index and return the k vertices with minimum distances to q. In specific, let L be the 2-hop label index, which consists of a set of label entry lists, each for a vertex $v \in V$. In the list $L(v)$ for vertex v, there are a set of label entries (u_i, d_i) where $u_i \in V$, and d_i is the distance between u_i and v. FS finds the answers by forward search from $L(q)$ and $L(y)$

where $y \in \mathsf{hit}(w)$. FS becomes inefficient when the keyword w has high frequencies resulting in many candidates in $\mathsf{hit}(w)$ to be checked.

To handle the cases where the keywords have high frequencies, FBS incorporates forward search and backward search on the label index. In addition to the 2-hop label index L, the authors build two more indices, namely 2-hop label backward index LB and keyword-lookup tree index KT. Similar to L, for each vertex v, $LB(v)$ consists of a list of label entries (u, d), which is constructed as follows. For each entry $(v, d) \in L(u)$, we add an entry (u, d) into $LB(v)$. We also sort entries in $LB(v)$ by non-increasing order of the distance value d. For a k-NK query $Q = (q, w, k)$, to avoid scanning all vertices in $\mathsf{hit}(w)$, we now search from each label entry $(x, d) \in L(q)$. Then, we search for label entries in the backward index $LB(x)$ with vertices that contain keyword w, i.e., $(y, d') \in LB(x)$ where y contains w. We can obtain one possible answer y from the path $(q \rightarrow x \rightarrow y, d + d')$, that is, the distance from q to y is $d + d'$.

Since $LB(v)$ may contain vertices which do not contain keyword w, we need a mechanism to look up the entries of vertices that do contain w efficiently. To this end, the authors further propose tree index, called keyword-lookup tree index (KT). Specifically, for each vertex $v \in V$, $KT(v)$ is a forest built by breaking the set of label entries of $LB(v)$ into fragments. Each tree node in $KT(v)$ contains some keyword information of a fragment, so that the entries containing the query keyword in a fragment can be retrieved efficiently.

• *An I/O-efficient algorithm*. The above methods for Problem 3 are all main memory oriented, which assume that the graph as well as the constructed index can fit entirely in memory. To deal with large-scale networks, Zhu et al. [113] propose an I/O-efficient approach, which uses a compact disk index to answer a k-NK query with constant I/Os.

Since the shortest path computation is a key operation in answering a k-NK query, the authors follow the computation paradigm proposed in [70] to speed up the calculation, where a set of spanning trees are used as an approximate representation of a graph and the shortest distance in trees is used as an approximation of the shortest distance in a graph. In particular, they focus on how to lay out the balanced compact tree on disk, such that only a small number of block accesses are needed to answer a k-NK query, and the index size on disk is small. To this end, the authors split a balanced compact tree $\mathsf{BCT}(\lambda)$ into two levels, where the top level is stored as paths and the bottom level is stored as subtrees.

In general, the authors lay out $\mathsf{BCT}(\lambda)$ by paths and subtrees blocking into disks in the bottom-up manner. In the bottom level, for a vertex $v \in \mathsf{BCT}(\lambda)$, the subtree rooted by v, denoted by $\mathsf{ST}(v)$, contains all v's descendants and itself. To make use the best use of the block, we should choose to store a set of maximal subtrees, under the condition that the total size of all candidate lists in $\mathsf{ST}(v)$ can fit into a block. In the top level, we store for the remaining vertices in the form of paths into blocks. The idea of path blocking is, for each leaf vertex v in the remaining vertices, we store the path from v to the root and the associated candidate lists into a block. Moreover, it is clear that all other inner vertices are covered by such paths.

• *A privacy-preserving algorithm*. To reduce the business running cost, more data owners are motivated to outsource their graphs to the cloud for storage, management and retrieval. However, directly outsourcing the graphs may cause serious privacy concerns. For example, keyword search on graphs can threat the privacy of data users, as the graph or the query request may reveal sensitive information such as user's addresses or personal information. Under such circumstances, Teng et al. [81] study Problem 3 in a privacy-preserving manner, which aims to process k-NK query on graphs without privacy leakage.

The authors build a two-level index for the shortest distance trees of the graph which is proposed in [70]. In the index, the first level indexes the paths from the vertices of the tree to the root, and the second indexes the routes from the root to the keyword vertices. To achieve distance computation without privacy breaches, the index is encrypted using secret sharing encryption. To handle the keyword filtering during the search processing, the authors also propose a trapdoor generation method for query keywords leveraging privacy-preserving set-intersection.

To process a k-NK query, we first retrieve the ancestors of the query node in the first level index and then compute the candidate nodes with the query keywords in the second level. The final results are calculated and merged with the candidates.

• *A continuous query-oriented algorithm*. In [109], Zheng et al. investigate a continuous variant of k-NK query on road networks. Formally, given a road network G, and a moving query $Q = (q, w, k)$, the query keeps returning the k-NK results for every new location q of Q. For presentation simplicity, we denote this variant of k-NK query by k-CNK.

The key to deal with k-CNK query efficiently is to reduce the number of recomputation as much as possible for query locations with the same query results. To this end, the authors make use of the so-called dominance interval or region on road network, which share the similar intuition with safe region for processing continuous queries in Euclidean space. To implement the dominance interval with high efficiency, the authors propose an important index structure, namely EP index (short for enclosed path index). An enclosed path, denoted as $\mathcal{EP}(s, ..., e)$, is a path that only the starting and ending vertices are intersection vertices among all vertices it passes. Then, for each $\mathcal{EP}(s, ..., e)$ we construct an inverted list for each keyword w contained by $o \in \mathcal{EP} \setminus \{s, e\}$, which is a list of the vertices that contain w.

Given a moving query $Q = (q, w, k)$, we first identify the enclosed path $\mathcal{EP}_q$ that q locates on. Then, we compute the k -NK results for both vertices s and e of $\mathcal{EP}_q$ by existing algorithms such as [48, 70], denoted by R_s and R_e, respectively. Besides, we also obtain all vertices on $\mathcal{EP}_q$ that contain w by accessing the EP index of $\mathcal{EP}_q$. Finally, we proceed to divide $\mathcal{EP}_q$ into dominance intervals by a window sliding approach. As long as q is on $\mathcal{EP}_q$, the query result is simply the result set with corresponding dominance interval covering q.

In the literature, there are also works on studying k-NK query on other types of graphs, such as temporal graphs [41] where edges in the graph have time constraint.

4.2 Top-*k* Relevant Neighbor Keyword Search

In the literature, many research efforts are devoted to the problem of most relevant neighbor keyword search where the ranking function considers both keyword relevance and network distance. In particular, studies in this category are mainly focusing on spatial keyword queries on road networks [1, 66, 73, 111], which implies that the vertices and edges also have spatial location information besides the graph connecting structure. Since the road network contains extra spatial location information, many spatial-textual index structures, such as [15, 63, 72], can be utilized to accelerate the query performance. Note that there might be spatial objects (points of interest) in the road network apart from the network vertices. To simplify the presentation, we regard these spatial objects as vertices as well, which can be realized by simply splitting the edge it locates.

The formal definition of top-k relevant neighbor keyword search is stated as follows.

Problem 4 Given a road network $G = (V, E)$ and a query $Q = (q, k, \psi)$, where $q \in V$ is a query vertex in G, k is a positive integer, and ψ is a set of query keywords, return a set of k vertices with the largest scores. The score of each vertex v is defined in Eq. (4).

$$\tau(v) = \frac{\mathsf{rel}(\psi, \mathsf{doc}(v))}{\mathsf{dist}(q, v)} \tag{4}$$

here, $\mathsf{rel}(\psi, \mathsf{doc}(v))$ is the textual relevance of vertex v to the query keywords ψ, and $\mathsf{dist}(q, v)$ is the network distance from q to o. A widely used metric for textual relevance is cosine similarity [116].

• *A Dijkstra-like expansion algorithm*. Rocha-Junior et al. [73], for the first time, investigate Problem 4. To deal with Problem 4, the authors employ an expansion strategy similar to Dijkstra's algorithm [19].

Specifically, the k best spatiotextual result vertices are maintained in a heap $\mathcal{R}$ in decreasing order of ranking score shown in Eq. (4). During the expansion, another heap $\mathcal{Q}$ is used to store the adjacent vertices according to increasing order of network distance to q.

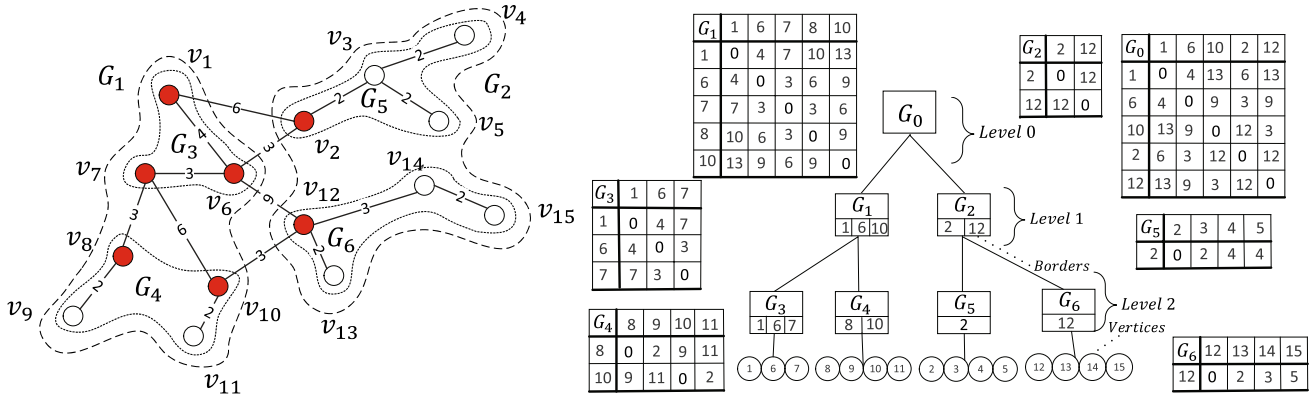
Initially, the algorithm finds the edge (v, v') in which q locates. Then vertices v and v' are inserted into $\mathcal{Q}$ and marked as visited. In each expansion step, we dequeue the top vertex v in $\mathcal{Q}$ and insert all its unvisited neighbor vertices into $\mathcal{Q}$. Note that, whenever we visit a new vertex v, we also update the result heap $\mathcal{R}$ with v by computing the distance between q and v. The algorithm stops when the remaining vertices cannot achieve a better score (i.e., score upper bound) than the score of the k-th vertex already found, or the entire network has been expanded.

In particular, the score upper bound of a vertex v can be estimated by the distance between v and q, and the maximum textual relevance (i.e., $\mathsf{rel}(\psi, \mathsf{doc}(v)) = 1$ in Eq. 4). The correctness of the early termination condition is guaranteed by the fact that the algorithm always expands the vertex v with minimum distance to q.

It is worth noticing that, to facilitate the query performance, many spatial-textual-related index structures are employed in this algorithm. For example, R $*$-tree is used to store road network edges, and IR-tree stores spatial and keyword information of vertices.

• *A* G-tree *-based algorithm*. The above Dijkstra-like expansion algorithm is rather inefficient for the scenario that vertices in the result set are far away from the query location q, since the searching space is in the shape of a circle with q as the center. To improve the searching performance, Zhong et al. [111] propose a G-tree [110]-based method.

Similar to its spatial version R-tree [34] that is used to facilitate the proximity-related search on metric space, G -tree is devised to compute the shortest-path distances on road network with high efficiency and good scalability (i.e., low storage cost). In general, G-tree is a balanced search tree constructed by recursively partitioning the road network into sub-networks where each G-tree node corresponds to a sub-network. It satisfies the following properties. **(i)** Each node represents a subgraph. The root node corresponds to the graph G. **(ii)** Each non-leaf node has $f(\geq 2)$ child nodes. **(iii)** Each leaf node contains at most $\tau(\geq 1)$ vertices. All leaf nodes appear at the same level. **(iv)** Each node maintains its border set and a distance matrix. For non-leaf nodes, the distance matrix maintains the distance between two borders in its child nodes. For leaf nodes, the distance matrix stores the distances between all vertices and the borders in this node. **(v)** For keyword queries, each node also stores additional information, such as inverted list. Note that, given a subgraph G_i of G, a vertex $u \in V(G_i)$ is called a border if $\exists (u, v) \in E(G)$ and $v \notin V(G_i)$.



(a) Graph partition (b) G-tree

Fig. 2 An example for G-tree index structure [110]

Example 1 Figure 2b shows the G-tree of the road network in Fig. 2a. The borders of each node are shown in the rectangle box under the node. The distance matrix of each node is listed around the tree node. For G_1, its children G_3 and G_4 contain five borders $\{v_1, v_6, v_7, v_8, v_{10}\}$; thus, the rows/columns of G_1's distance matrix are the five borders. The set of vertices of each leaf node are shown in the circled numbers. For instance, in G_4's distance matrix, the rows are borders $\{v_8, v_{10}\}$ and the columns are vertices $\{v_8, v_9, v_{10}, v_{11}\}$. Then entry $(v_8, v_{11}) = 11$ since the distance between borders v_8 and v_{11} is 11.

Given a query $Q = (q, k, \psi)$ for Problem 4, we first locate the leaf node of q. Then we create a maximum priority query $\mathcal{Q}$ to store an entry e according to decreasing order of the ranking score (i.e., $\tau(e)$ in Eq. 4), where e is a vertex or a G-tree node. Note that $\tau(e)$ is possible maximum value for a vertex in the corresponding subgraph if e is a G-tree node, which can be computed by the possible maximum textual relevance of vertices in e and the minimum distance between q and borders in e. Next, we start from the leaf node in which q locates, expand upwards in G-tree, and insert the sibling nodes of current node e into $\mathcal{Q}$. In each iteration, we dequeue the top entry in $\mathcal{Q}$ and insert all its unvisited children into $\mathcal{Q}$. The algorithm stops once we have collected k vertices or the tree has been completely expanded.

• *A keyword separated indexing-based algorithm*. The above-discussed methods all use the technique of *keyword aggregation* for road networks, which is used extensively by spatial keyword query techniques in Euclidean space [10, 15, 88, 103]. However, the disadvantage of keyword aggregation is the generation of many false positives. Computing distance in Euclidean space is quick arithmetic operation, but in road networks computing distance is a complex graph operation and far more expensive. Consequently, the computation overhead for incurring false positives in road networks is significantly higher than in Euclidean space. Therefore, keyword aggregation is far less effective for road networks.

Abeywickrama et al. [1] resort to *keyword separated indexing* techniques to delay and avoid the expensive network distance computation. The key of this method is the *on-demand inverted heap* $\mathcal{H}$ for each keyword w. An important property of this heap is as follows. Given the current top vertex v in $\mathcal{H}$ for keyword w and its lower-bound distance $\mathsf{dist}_{\mathsf{LB}}(q, v)$ from query vertex q, any vertex u containing w, not yet extracted from $\mathcal{H}$, has network distance $\mathsf{dist}(q, v) \geq \mathsf{dist}_{\mathsf{LB}}(q, v)$. This property allows our query algorithms to access keyword vertices in $\mathsf{hit}(w)$ in order of their lower-bound network distances from q.

Before illustrating the query algorithm, we clarify that the ranking score used in [1] is inverse of the ranking score shown in Eq. (4). Now, given a query (q, k, ψ), we aim to find the k vertices with the smallest scores. To deal with query efficiently, the algorithm uses a minimum priority queue $\mathcal{Q}$ where the value of an entry is a well-observed lower-bound score. First, we create an on-demand inverted heap $\mathcal{H}_i$ for each keyword $w_i \in \psi$. Then, we insert the top vertex in each $\mathcal{H}_i$ into $\mathcal{Q}$. In each iteration, we extracted a candidate vertex c from $\mathcal{Q}$ that has not already been processed and insert into $\mathcal{Q}$ the next vertex in the heap containing c. After that, we compute the lower-bound score using its actual textual relevance and lower-bound network distance, i.e., $\frac{\mathsf{dist}_{\mathsf{LB}}(q,c)}{\mathsf{rel}(\psi, \mathsf{doc}(c))}$. If this lower-bound score is smaller than the current best kth result R_k, then its actual score is computed using its exact network distance $\mathsf{dist}(q, c)$. If its actual score is smaller than R_k, then the result list $\mathcal{R}$ and R_k are updated accordingly. The algorithm terminates when $\mathcal{Q}$ is empty or the top of $\mathcal{Q}$ is no less than R_k.

A crucial operation in the above algorithm is to create an on-demand inverted heap for each query keyword. A simple approach to insert all vertices in $\mathsf{hit}(w)$ in the heap with their

lower-bound distances. However, this is not feasible as it would be required for every query. In the paper, the authors utilize the network Voronoi diagram (NVD) [56] that allows inverted heaps to be populated lazily. The main idea is that we build an NVD on vertices in $\mathsf{hit}(w)$ for each keyword w offline. With the help of NVDs, we can maintain the inverted heap incrementally by expanding the NVD one layer at a time.

4.3 Discussion

In this section, we review graph keyword search works that return k best vertices as the desired answers, which can be further divided into two groups based on the specific calculation of scoring function. One group [3, 41, 48, 70, 81, 109, 113] is top-k nearest neighbor keyword search, which considers the distance only when ranking vertices. The other group [1, 66, 73, 111] is top-k relevant neighbor keyword search, which combines both textual relevance and distance.

5 Subgraph-Based Keyword Search

Both categories of keyword search reviewed in Sects. 3 and 4 have some limitations. For tree-based methods discussed in Sect. 3), an answer result may only show partial information about how those vertices in the result are connected. For nearest neighbor-based methods discussed in Sect. 4, an answer result is simply a single vertex in the graph. In some application scenarios, a subgraph is more desired for inspecting the whole picture. In this section, we review graph keyword search works that return compact subgraphs as the desired answers. Next, we discuss them in detail according to the specific definition of the desired answer subgraph.

5.1 *r*-Radius Steiner Graph-Based Semantics

Definition 1 (*Centric distance*) Given a graph $G = (V, E)$ and any vertex v in G, the centric distance of v, denoted as $\mathsf{CD}(v)$, is the maximal value among the distances between v and any vertex u in G, i.e., $\mathsf{CD}(v) = \max_{u \in G}\{\mathsf{dist}(v, u)\}$.

Definition 2 (*Radius*) The radius of a graph G, denoted as $\mathsf{R}(G)$, is the minimal value among the centric distances of every vertex in G, i.e., $\mathsf{R}(G) = \min_{v \in G}\{\mathsf{CD}(v)\}$. G is called an r-radius graph if the radius of G is exactly r.

Definition 3 (*r-Radius Steiner graph*) Given an r-radius graph G and a keyword set ψ, vertex s in G is called a Steiner vertex if there exist two keyword vertices u and v regarding ψ, and s is on the path between u and v. The subgraph of G composed of the Steiner nodes and associated edges is called an r-radius Steiner graph. The radius of an r-radius Steiner graph must satisfy that $\mathsf{R}(G) \leq r$.

Problem 5 Given a graph $G = (V, E)$ and a query $Q = (r, k, \psi)$, where r is a positive real value, k is a positive integer, and ψ is a set of query keywords, return k r-radius Steiner graphs in G with the largest relevance score regarding ψ.

Li et al. [61] study the r-radius Steiner graph problem. To facilitate efficient retrieval of r-radius graphs, they construct a novel graph index. The entries of the graph index are keywords contained in the graph, and each entry preserves the r-radius graphs that contain the keyword. For each keyword w_i, we keep the set of all r-radius graphs that contains w_i, denoted as $\mathcal{I}_{w_i}$.

To process a query $Q = (r, k, \psi)$, we first retrieve the set $\mathcal{I}_{w_i}$ of those r-radius graphs which contain w_i based on the graph index and then union every $\mathcal{I}_{w_i}$ to compute $\cup_{i=1}^{m} \mathcal{I}_{w_i}$, which is the set of r-radius graphs that contain all or a portion of the keywords in ψ. Finally, we extract the r-radius Steiner graphs by removing the non-Steiner vertices from the corresponding r-radius graphs, and rank the results to return the top-k answers.

5.2 Multicenter Community-Based Semantics

Definition 4 (*Multicenter community*) Given a graph $G = (V, E)$ and a set of keywords ψ, a community, denoted as $R = (V_R, E_R)$, is a multicenter induced subgraph of G. Here, V_R is a union of three subsets, i.e., $V_R = V_c \cup V_l \cup V_p$. (1) V_l is a set of keyword vertices. Every vertex $v_l \in V_l$ contains at least a keyword in ψ and all keywords in ψ must appear in at least one vertex in V_l. (2) V_c represents a set of vertices called center vertices. For any vertex $v_c \in V_c$, there exists at least a single path such that $\mathsf{dist}(v_c, v_l) \leq \mathsf{R}_{\mathsf{max}}$ between v_c and every $v_l \in V_l$, where $\mathsf{R}_{\mathsf{max}}$ is a user-given radius threshold. (3) V_p represents a set of path vertices, which appear on any path from a vertex $v_c \in V_c$ to a vertex $v_l \in V_l$ if $\mathsf{dist}(v_c, v_l) \leq \mathsf{R}_{\mathsf{max}}$.

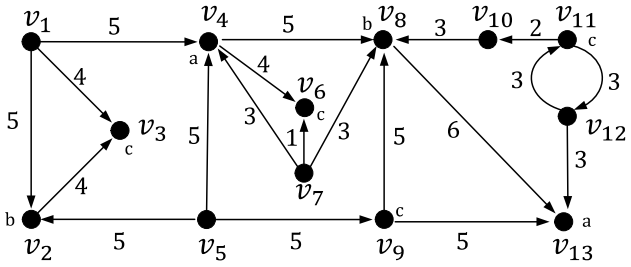


Fig. 3 An graph G for multicenter community query [71]

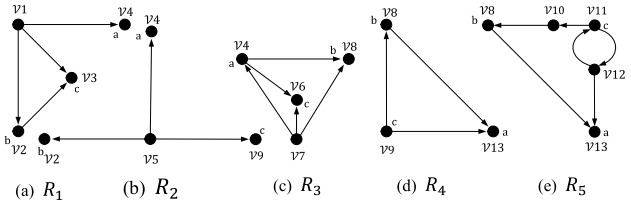


Fig. 4 Five communities [71]

Example 2 Consider the graph in Fig. 3. Let $\mathsf{R}_{\max} = 8$. For a 3-keyword query $\{a, b, c\}$, five communities are shown in Fig. 4. For example, for R_5 (Fig. 4e), the keyword vertices are $V_l = \{v_{13}, v_8, v_{11}\}$, center vertices are $V_c = \{v_{11}, v_{12}\}$, and path vertices are $V_p = \{v_{10}\}$.

For a community R, a cost function can be defined, denoted by $\mathsf{cost}(R)$, as the minimum total distance from a center vertex to every keyword node. For example, consider community R_5 (Fig. 4e). There are two centers, v_{11} and v_{12}. The total edge weight over the shortest paths from v_{11} to the 3 keyword vertices, v_8, v_{11}, and v_{13}, is $11 = (2 + 3) + 0 + (3 + 3)$. Similarly, we can get the total weight for v_{12} is 14. Therefore, $\mathsf{cost}(R_5) = 11$.

Problem 6 Given a graph $G = (V, E)$ and a query $Q = (\mathsf{R}_{\max}, k, \psi)$, return the top-$k$ multicenter communities.

Qin et al. [71] study the problem of finding multicenter communities. A key observation is that a community R is uniquely determined by keyword vertices V_l, which is called the core of the community, denoted by $C = [c_1, c_2, \ldots, c_l]$. To efficiently find the top-k communities with minimum cost, the authors use a Fibonacci heap $\mathcal{H}$ to maintain the generated communities.

Initially, we find the first best core $C = \{c_1, c_2, \ldots, c_l\}$ in the entire space $V_1 \times V_2 \times \cdots \times V_l$, and we ensure the first core found is the core for the top-1 community. Note that the next best core can be found in the next l subspaces: $S_1 : (V_1 - \{c_1\}) \times V_2 \times \cdots \times V_l$, $S_2 : V_1 \times (V_2 - \{c_2\}) \times \cdots \times V_l, \ldots, S_l : V_1 \times V_2 \cdots \times (V_l - \{c_l\})$ It is important to know the following facts. (i) $V_1 \times V_2 \times \cdots V_l = C \cup S_1 \cup S_2 \cup \cdots \cup S_l.(ii) S_i \cap S_j = \emptyset (i \neq j)$.

After that, we enheap C with other information into heap $\mathcal{H}$ and enter a while loop. In the while loop, we first deheap the core C, which is the current best result. Then, we attempt to find the next best core in each of the l subspaces, $S_1, S_2, \ldots S_l$, individually. If we find the best core, C_i, in S_i, for $1 \leq i \leq l$, we enheap C_i to $\mathcal{H}$. With $\mathcal{H}$, the next best core can be selected in the next iteration from all cores kept in $\mathcal{H}$. We repeat this process until k cores are found, which are the top-k results.

5.3 *r*-Clique-Based Semantics

Definition 5 (r-Clique) Given a graph G and a set of query keywords $Q = \{w_1, w_2, \ldots, w_l\}$, an r-clique of G with respect to Q is a set of keyword vertices that together cover all keywords in Q and in which the distance between each pair of keyword vertices is no larger than r.

Definition 6 (*Weight ofr-clique*) For a given r-clique C, suppose that the vertices of C are denoted as $\{v_1, v_2, \ldots, v_l\}$. Then, the weight C is defined as

$$\mathsf{weight}(C) = \sum_{i=1}^{l} \sum_{j=i+1}^{l} \mathsf{dist}(v_i, v_j) \tag{5}$$

In [51, 52], r-cliques with smaller weights are considered to be better. The problem of find r-clique is formally stated as follows.

Problem 7 Given a distance threshold r, a graph G and a set of input keywords, find an r-clique in G with minimum weight.

Kargar et al. [51, 52] propose the r-clique problem and show that the problem is NP-hard. The authors first present a branch and bound algorithm for finding all r-cliques in a graph. The candidate partial r-cliques are store in a list called *rList*. The basic idea of the algorithm is as follows. First, the keyword vertices containing the first keyword are added to *rList*. Then, for the second keyword w_2, we compute the distance between each vertices in $\mathsf{hit}(w_2)$ and each node in *rList*. If the distance is within r, a new candidate that combines the corresponding vertices in $\mathsf{hit}(w_2)$ and *rList* is added to a new candidate list called *newRList*. After all pairs of vertices in $\mathsf{hit}(w_2)$ and *rList* have been checked, the content of *rList* is replaced by the content of *newRList*. The process continues in the same way to consider all of the remaining keywords. The final content of *rList* is the set of all r-cliques.

Because the branch and bound algorithm is slow when the number of keywords is large. Also, it does not rank the generated r-cliques. To speed up the process, the authors propose an approximation algorithm with approximation ratio of 2 for finding r-cliques with polynomial delay.

Zhao et al. [107] extend r-clique to road networks to retrieve the POIs (Points of Interest). Particularly, they advocate the popularity-aware collective keyword (PACK)

query in road networks, which aims to find a group of popular POIs that cover the query's keywords and satisfy the distance requirements, such that the sum of rating scores over these vertices for the query keywords is maximized. The authors show that the PACK query is NP-hard. Exact and heuristic solutions on small and large road networks are then developed.

5.4 Strongly Connected Subgraph-Based Semantics

The above-mentioned studies all utilize the shortest path distance to evaluate the compactness of the answers. In [7], Bryson et al. argue that this method may lack robustness since it may not reflect the overall structure of the answer subgraph. Therefore, they propose a random walk-based approach to measure the distance between vertices containing the query keywords.

Definition 7 (*Candidate answer*) Given a graph G and a query $Q = \{w_1, w_2, \ldots, w_l\}$, a candidate answer C is a subgraph of G whose vertices cover all keywords in Q.

Definition 8 (*Connection score of a candidate answer*) Given a candidate answer C for query $Q = \{w_1, w_2, \ldots, w_l\}$, suppose that the vertex in C containing w_i is v_i for $1 \le i \le l$. Then, the connection score of C is defined as

$$\mathsf{ConSc}(C) = \sum_{i=1}^{l} \sum_{j=i+1}^{l} \mathsf{sc}(v_i, v_j) \tag{6}$$

Here $\mathsf{sc}(v_i, v_j)$ is the connection score between vertices v_i and v_j defined by random walk. The higher the score $\mathsf{sc}(v_i, v_j)$, the stronger the relationship between vertices v_i and v_j in graph G.

Problem 8 Given a graph G and a query Q, return a candidate answer C for Q with a maximal connection score $\mathsf{ConSc}(C)$.

The authors prove that Problem 8 is NP-hard. Therefore, they propose an heuristic method to solve it in polynomial time. The idea is as follows.

For a given query Q, take each vertex containing the rarest keyword and form a subgraph (candidate answer) around that vertex. The answer's score is initialized to 0. Then, in each iteration, we include one of the uncovered keywords. In order to do that, we run the RWR and set the restart vertices to the current vertices of the subgraph.

At the beginning, the only vertex in the subgraph is the one with the rarest keyword. When the RWR process is finished, the vertex with the highest score that also covers the current required keyword is selected as the best vertex of the subgraph. Among all the candidate subgraphs that are formed around the vertex containing the rarest keywords, the one with the highest sum of RWR scores is selected as the best subgraph.

5.5 Cohesive Subgraph-Based Semantics

In [114, 115], Zhu et al. advocate the problem of querying cohesive subgraphs by keywords. Particularly, the authors employ k-truss [42] to model the cohesiveness of a subgraph, which is formally defined as follows.

Definition 9 (*Connected k-truss*) Given a graph G and an integer k, a connected k-truss is a connected subgraph $S \subseteq G$, such that $\forall e \in E(S), \mathsf{sup}_S(e) \le k-2$. Here $\mathsf{sup}_S(e)$ is the support of an edge $e = (u, v)$ in G, which is the number of triangles in which e appears.

Problem 9 (*Minimal dense truss search by keywords*) Given a graph $G = (V, E)$ and a set of query keywords $Q = \{w_1, w_2, \ldots, w_l\}$, return a subgraph S of G, such that

1. $V(S)$ covers all keywords in Q;
2. S is a connected truss in G that maximizes the trussness;
3. Any subgraph of S cannot satisfy Conditions 1 and 2 at the same time.

Here the trussness of a subgraph $S \subseteq G$ is the minimum support of all edges in S plus 2.

Example 3 Consider the example in Fig. 5. Suppose $Q = \{DB, ML\}$. H_1 and H_2 are 4-truss and 3-truss containing Q. Clearly, H_1 is a dense truss over Q. We also have another 4-truss induced by $\{v_1, v_2, v_3, v_4, v_5\}$ containing Q, but it is not minimal. Thus H_1 is the minimal dense truss for the query Q.

Zhu et al. [114, 115] propose a keyword-truss index (KT-Index)-based algorithm. In particular, KT-Index is designed to include two parts: truss index and keyword index. Truss index is a multilayer structure, where we index

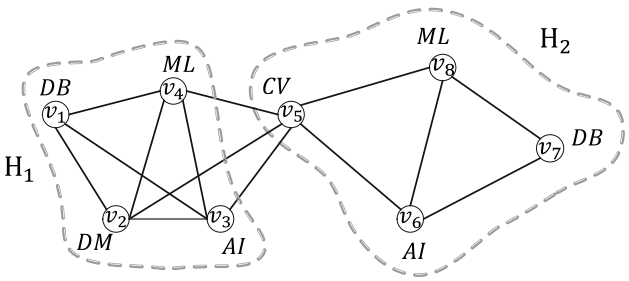


Fig. 5 A graph G for cohesive subgraph keyword query [114, 115]

the information of all the connected k-truss in the k-th layer. In each layer, there are a set of connected components. In the keyword index, we first store a inverted keyword list to keep the vertex IDs that contain each keyword. Meanwhile, we record the upper bound of trussness $\tau'(w_i)$ for each keyword. Moreover, for each keyword, we record IDs of the component CID_k it occurs in the k-th layer.

Given a query Q, the search algorithm checks each layer of truss index by a binary search to avoid the worst case of checking all the value of k_{max}. In the k-th layer, we obtain the set of components IDs CC that contains all the keywords. If CC is empty, we will search layers with truss value smaller than current k; otherwise, we will search layers with truss value larger than current k. After we find the set of component IDs CC that containing all the keywords, we select the component with the minimum size as dense truss G_{den}. Then, we extract the minimal dense truss S by using the anti-monotonic property of k-truss.

5.6 Discussion

In this section, we review graph keyword search studies that return compact subgraphs as the desired answers. In particular, five different types of subgraph-based semantics are introduced [7, 51, 61, 71, 114, 115].

There are also some other more complicated query intention-based studies, such as target-aware query result-based keyword search [76], keyword search on public–private networks [47], and parallel keyword search for large knowledge bases [95, 96].

6 Other Graph-Based Keyword Search

In this section, we review other graph keyword search studies, including a generic ontology-based indexing framework, spatial keyword query on road networks, and keyword routing on road networks.

6.1 A Generic Ontology-Based Indexing Framework

Jiang et al. [46] propose a generic ontology-based indexing framework for keyword search, called Bisimulation of Generalized Graph Index (BiG-index), to enhance the search performance. The novelties of BiG-index reside in using an ontology graph G_{ont} to summarize and index a data graph G iteratively, to form a hierarchical index structure $\mathcal{G}$. BiG-index is generic since it only requires keyword search algorithms to generate query answers from summary graphs having two simple properties.

To process a query Q, we transform Q into $\mathcal{Q}$ according to G_{ont} in runtime. The transformed query is searched on the summary graphs in $\mathcal{G}$. The efficiency is guaranteed due to the small sizes of the summary graphs and the early pruning of semantically irrelevant subgraphs. They authors also show that the existing popular keyword search algorithms (i.e., BLINKS [39] and r-clique [51]) can be easily implemented on top of BiG-index.

6.2 Spatial Keyword Query on Road Networks

• *Diversified spatial keyword search*. Zhang et al. [102] study the problem of diversified spatial keyword search on road networks.

Definition 10 (*SK Query*) Given a road network G and a query $Q = (q, \psi, \delta_{max})$, where q is the query location, $\psi = \{w_1, w_2, \ldots, w_l\}$ is a set of query keywords, and δ_{max} is the network distance threshold, a spatial keyword query retrieves vertices each of which contains all query keywords in ψ and is within network distance δ_{max} from q.

Definition 11 (*Bi-criteria objective function*) Given a set S of vertices with $|S| = k$, the bi-criteria objective function, denoted by f, is defined as

$$f(S) = \lambda \times Rel(S) + (1 - \lambda) \times Div(S) \tag{7}$$

Here, $Rel(S)$ is the relevance of S measured by the network distances of vertices in S to the query location q, $Div(S)$ is the diversity of S captured by their pairwise network distances, and $\lambda (0 \leq \lambda \leq 1)$ is a parameter specifying the trade-off between the relevance and the diversity.

Problem 10 Given a road network G and a query $Q = (q, k, \psi, \delta_{max})$, return a set S of vertices in G such that

1. $|S| = k$;
2. $S \subseteq SK(q, \psi, \delta_{max})$;
3. $f(S)$ is maximized.

To efficiently deal with Problem 10, an efficient signature-based inverted indexing technique is proposed in [102]. Besides, an efficient incremental network expansion algorithm is proposed as well such that the spatial keyword pruning and diversity pruning techniques can be seamlessly integrated and hence significantly reduce the overall cost.

• *Collective spatial keyword search*. Gao et al. [27] study the problem of collective spatial keyword search on road networks, which is formally defined as below.

Problem 11 Given a road network G and a query $Q = (q, \psi)$ where q is a query location and $\psi = \{w_1, w_2, \ldots, w_l\}$ is a set of keywords, a collective spatial keyword query on road networks aims to find a set S of vertices, such that

1. vertices in S collectively cover all keywords in ψ;
2. $f(S)$ is minimized among all possible choices of S.

Here the cost function $f(S)$ is defined as follows

$$f(S) = \alpha \times \max_{v \in S} \mathsf{dist}(q, v) + (1 - \alpha) \times \max_{v_1, v_2 \in S} \mathsf{dist}(v_1, v_2) \quad (8)$$

Gao et al. [27] prove that Problem 11 is NP-complete. In light of this, the authors propose two approximate algorithms with guaranteed approximation errors. The first is network expansion-based algorithm, denoted by *NEB*. The main idea of *NEB* is to find a set of POIs that are close to the given query location q and cover the query keywords in ψ. A min-priority queue $\mathcal{Q}$ is utilized to keep tracks of the edges that have been visited, and such edges are sorted in ascending order of their distances to q. Whenever a POI o having some uncovered keywords, o, is added to the result set S. The expansion proceeds until all keywords in ψ are covered. The authors show that *NEB* can achieve a 3-factor approximate guarantee.

Note that *NEB* does not consider the proximity of vertices in S and thus has a loose approximation. Based on this, the authors further propose an iterative *NEB*-based algorithm, namely *IEB*. They show that this new approximate algorithm *IEB* can achieve a 2-factor approximate guarantee.

In the literature, many other spatial keyword queries on road networks are investigated, such as reverse spatial keyword query [26, 105], why-not questions query [104], and diversified geo-social keyword query [106].

6.3 Keyword Routing on Road Networks

In a road network, a route is a path such that it goes through a sequence of vertices following the relevant edges in the road network. In [8], an optimal route is defined based on two attributes on each edge (v_i, v_j), namely (*i*) the *objective value* of this edge, which is denoted by $o(v_i, v_j)$ (e.g., the popularity), and (*ii*) the *budget value* of this edge, which is denoted by $b(v_i, v_j)$ (e.g., the travel time).

Definition 12 (*Objective Score and Budget Score*) Given a route $R = \langle v_0, v_1, \ldots, v_n \rangle$, the objective score of R is defined as the sum of the objective values of all edges in R:

$$\mathsf{OS}(R) = \sum_{i=1}^{n} o(v_{i-1}, v_i) \quad (9)$$

and the budget score is defined as the sum of the budget values of all the edges in R:

$$\mathsf{BS}(R) = \sum_{i=1}^{n} b(v_{i-1}, v_i) \quad (10)$$

Problem 12 (KOR) Given a road network G, the keyword-aware optimal route query $Q = (v_s, v_t, \psi, \eta)$, where v_s and v_t are the source and target locations, respectively, $\psi = \{w_1, w_2, \ldots, w_l\}$ is a set of keywords, and η specifies the budget limit, aims to find the route R starting from v_s and ending at v_t such that

1. vertices in R collectively cover all keywords in ψ;
2. $\mathsf{BS}(R) \leq \eta$;
3. $R = \arg\min_R \mathsf{OS}(R)$.

Cao et al. [8] show that Problem 12 is NP-hard. Therefore, the authors resort to approximate solutions. In specific, an approximation algorithm called OSScaling is proposed. In OSScaling, we first scale the objective value of every edge to an integer by a parameter ϵ to obtain a scaled graph denoted by G_ϵ.

Specifically, in the scaled graph G_ϵ, each partial route is represented by a "label", which records the query keywords already covered by the partial route, the scaled objective score, the original objective score, and the budget score of the route. At each node, we maintain a list of "useful" labels corresponding to the routes that go to that node. Starting from the source node, we keep creating new partial routes by extending the current "best" partial route to generate new labels, until all the potentially useful labels on the target node are generated. Finally, the route represented by the label with the best objective score at the target node is returned.

The authors prove that OSScaling returns routes with objective scores no worse than $\frac{1}{1-\epsilon}$ times of that of the optimal route. To improve the performance of OSScaling, the authors further propose an approximate algorithm, namely BucketBound, which not only returns better approximate guarantees, but is more efficient than OSScaling.

In the literature, a string of other keyword-aware routing problems are studied [37, 38, 50, 87, 97, 108]. All these works aim at finding optimal routes by considering keywords and other application-specified constraints at the same time.

7 Related Work

In this section, we briefly review related studies, including query interpretation-oriented keyword search, community search on attributed graphs, team formation in social networks, spatial keyword search, and keyword-based similarity query.

7.1 Query Interpretation-Oriented Keyword Search

Query interpretation-oriented keyword search is an orthogonal task to the studies reviewed in previous sections. In general, the aim of query interpretation is to first transform the keyword search into a structured query pattern and then execute the query using underlying engine of the graph to retrieve answers. Methods based on query interpretation usually consider the query keyword sequence entered by the user as a query intent. Besides, most of the studies focus on the RDF data or knowledge base as graph data. Some representative works [25, 28, 36, 77, 78, 82, 94, 112] can be found in the literature.

7.2 Community Search on Attributed Graphs

Recently, community search on attributed graphs has attracted a lot of research efforts [22, 43, 64, 65]. In an attributed graph, nodes may contain a set of attributes which capture theirs properties. Take the collaboration network for example, node attributes (e.g., DB, ML) represent authors' topics of expertise. Community search on such attributed graphs often tend to be more complicated than keyword search, since it takes more factors into consideration, such as participation of query nodes, cohesiveness of candidate subgraph, attribute coverage and correlation and communication cost.

7.3 Team Formation in Social Networks

Another related problem to graph keyword search is team formation in expert networks, introduced by [58]. Each expert possesses a set of skills, and experts are connected to each other based on their past experience. Given a network of experts, and a set of required skills to complete a project, the goal is to find a subgraph of this network in which members of the subgraph collectively cover all the required skills. To rank a subgraph, objective functions that favor connectedness and minimize communication cost are utilized. The original team formation problem is similar to the graph keyword search problem, meaning that the solutions to the latter problem can be directly applied to the former problem. However, due to the nature of expert networks and special circumstances, a variety of methods were proposed over the last decade to address different requirements [53, 68, 85, 98].

7.4 Spatial Keyword Search

A parallel problem to graph keyword search is spatial keyword search, which have been extensively studied in the past two decades. Given a query q with a set of keywords and a location, the related spatial keyword queries can be roughly divided into two categories, namely soft cover and hard cover. For soft cover, we do not require that the answer covers all query keywords. A hybrid way, which uses a parameter α to trade off the balance between text relevancy and location proximity, is adopted to evaluate the quality of a candidate result. Representative studies in this category include [15, 74, 88, 103]. For hard cover, an answer must cover all query keywords. Representative studies in this category include [12, 23, 33, 93, 100, 101]. For a detailed survey of spatial keyword search, please refer to the recent survey paper [11].

7.5 Keyword-Based Similarity Join

Given a collection of records, each of which consists a set of keywords, keyword-based similarity join aims to retrieve the similar records. This problem can be divided into two groups, namely set similarity join and set containment join, according to the specific similarity definition. In the literature, both types of problem have been extensively studied. In particular, set similarity join [4, 17, 18, 24, 69, 84, 86, 89, 90] aims to find the record pair with a similarity score (i.e., Jaccard similarity) no smaller than a user-given threshold, while set containment join [6, 57, 67, 91, 92] retrieves record pairs such that keywords in one record are all contained in the other.

8 Future Research

There are many remaining challenges in the area of keyword search on graphs. In this section, we point out a list of promising future research directions as follows.

8.1 Other Types of Graphs

In recent years, many novel graph models have been proposed and the representative ones are as follows:

- *Knowledge graph* [46, 79]. Given a knowledge graph G, it is usually accompanied with an ontology graph $\mathcal{G}$, which encodes the ontology information, such as information of properties, classes, and their super classes. These ontology information can substantially improve the performance of keyword search algorithm on these knowledge graphs, in terms of both efficiency and effectiveness.
- *Public–private network* [47, 81]. In a public–private network, there is a public graph G, containing a set of vertices and a set of edges that are visible to all users. Besides, each vertex u has its private network G_u, which

is only known to u. Keyword search on such public–private networks must combine both public and private networks.

- *Uncertain graph* [40]. In many real applications (e.g., biology), the graph data are often noisy and inaccurate. An common way is to model them as uncertain graphs, where each edge is associated with a value denoting its existence probability. Thus, keyword search on such uncertain graph needs to consider existence probability of an answer.

8.2 Utilizing Graph Embedding

In recent years, graph embedding (also known as network representation learning) [32, 35] is one of the most successful techniques in the area of machine learning and data mining. Given a graph, graph embedding aims to map each node as a dense vector embedding in a low-dimensional Euclidean space. These node embeddings can then be fed to downstream machine learning systems and aid in tasks such as node classification, clustering, and link prediction.

Although graph embedding has proved extremely useful for a wide variety of prediction and graph analysis tasks, we do not notice any existing works on using it to solve keyword search problem. This is because graph embedding is generally to encode graph structure information into node embeddings, and they can be used as node features to aid learning tasks. However, keyword search usually targets to find a set of nodes in the graph to match the keywords, which is more a search task. Thus, it is not natural to direct use the node embeddings in keyword search.

A possible way to combine graph embedding and keyword search is to develop novel problem semantics. For example, apart from the keywords, we also consider the similarity of nodes in the query answer. That is the nodes in an answer should be similar to each other in terms of their embeddings, rather than the network distances.

8.3 Real Big Graphs

Most existing graph keyword search studies assume that the graphs can fit the main memory of a single machine, only a few of them consider external memory oriented [16] or I/O-efficient techniques [113]. However, in many real applications (e.g., Facebook), the graphs might contain billions of vertices and edges. As a result, how to efficiently perform online keyword search on such big graphs is a challenging task. To deal with such big graphs, a possible research direction is to utilize distributed computation platforms (e.g., GraphX [31]) or GPU systems [45].

9 Conclusion

In this paper, we conduct a comprehensive survey on the topic of keyword search over large graphs. We systematically review about 30 research articles, which covers all representative studies in the field of keyword search over graphs. Particularly, we classify these studies according to the answer ranking models, including tree-based keyword search, nearest neighbor-based keyword search, subgraph-based keyword search, and other semantics-based keyword search. For each class of works, we first give the formal definition for the research problem and then review the representative studies. In summary, our survey provides an overview of the state-of-the-art research advances on the topic of graph keyword search.

Funding Funding was provided by National Natural Science Foundation of China (Grant No. 62002108).

References

1. Abeywickrama T, Cheema MA, Khan A (2020) K-spin: Efficiently processing spatial keyword queries on road networks. Trans Knowl Data Eng 32:983–997
2. Abraham I, Delling D, Goldberg AV, Werneck RFF (2011) A hub-based labeling algorithm for shortest paths in road networks. In: SEA, pp 230–241
3. Bahmani B, Goel A (2012) Partitioned multi-indexing: bringing order to social search. In: WWW, pp 399–408
4. Bayardo RJ, Ma Y, Srikant R (2007) Scaling up all pairs similarity search. In: WWW, pp 131–140
5. Bhalotia G, Hulgeri A, Nakhe C, Chakrabarti S, Sudarshan S (2002) Keyword searching and browsing in databases using banks. In: ICDE, pp 431–440
6. Bouros P, Mamoulis N, Ge S, Terrovitis M (2015) Set containment join revisited. Knowl Inf Syst 49:375–402

7. Bryson S, Davoudi H, Golab L, Kargar M, Lytvyn Y, Mierzejewski P, Szlichta J, Zihaya M (2020) Robust keyword search in large attributed graphs. Inf Retriev J 23:502–524
8. Cao X, Chen L, Cong G, Xiao X (2012) Keyword-aware optimal route search. VLDB 5:1136–1147
9. Charikar M, Chekuri C, Goel A, Guha S (1998) Rounding via trees: deterministic approximation algorithms for group Steiner trees and k-median. In: STOC
10. Chen L, Cong G, Jensen CS, Wu D (2013) Spatial keyword query processing: an experimental evaluation. VLDB 6:217–228
11. Chen L, Shang S, Yang C, Li J (2020) Spatial keyword search: a survey. Geoinformatica 24:85–106
12. Choi DW, Pei J, Lin X (2016) Finding the minimum spatial keyword cover. In: ICDE, pp 685–696
13. Coffman J, Weaver AC (2014) An empirical performance evaluation of relational keyword search techniques. TKDE 26:30–42
14. Cohen E, Halperin E, Kaplan H, Zwick U (2003) Reachability and distance queries via 2-hop labels. SIAM J Comput 32:1338–1355
15. Cong G, Jensen CS, Wu D (2009) Efficient retrieval of the top-k most relevant spatial web objects. VLDB 2(337):348
16. Dalvi BB, Kshirsagar M, Sudarshan S (2008) Keyword search on external memory data graphs. VLDB. https://doi.org/10.14778/1453856.1453982
17. Deng D, Li G, Wen H, Feng J (2015) An efficient partition based method for exact set similarity joints. VLDB 10(14778/2856318):2856330
18. Deng D, Tao Y, Li G (2018) Overlap set similarity joins with theoretical guarantees. In: SIGMOD, pp 905–920
19. Dijkstra EW (1959) A note on two problems in connexion with graphs. Numer Math 7:48–50
20. Ding B, Yu JX, Wang S, Qin L, Zhang X, Lin X (2007) Finding top-k min-cost connected trees in databases. In: ICDE, pp 836–845
21. Dreyfus SE, Wagner RA (1971) The Steiner problem in graphs. Networks 132:185–207
22. Fang Y, Cheng R, Luo S, Hu J (2016) Effective community search for large attributed graphs. VLDB. https://doi.org/10.14778/2994509.2994538
23. Felipe ID, Hristidis V, Rishe N (2008) Keyword search on spatial databases. In: ICDE, pp 656–665
24. Fier F, Augsten N, Bouros P, Leser U, Freytag JC (2018) Set similarity joins on mapreduce: an experimental survey. PVLDB 11(10):1110–1122
25. Fu H, Anyanwu K (2011) Effectively interpreting keyword queries on RDF databases with a rear view. In: International semantic web conference, pp 193–208
26. Gao Y, Qin X, Zheng B, Chen G (2015) Efficient reverse top-k boolean spatial keyword queries on road networks. Trans Knowl Data Eng 27:1205–1218
27. Gao Y, Zhao J, Zheng B, Chen G (2016) Efficient collective spatial keyword query processing on road networks. TITS 17:469–480
28. Garca G, Izquierdo Y, Menendez E, Dartayre F, Casanova MA (2017) RDF keyword-based query technology meets a real-world dataset. In: EDBT, pp 656–667
29. Garg N, Konjevod G, Ravi R (2000) A polylogarithmic approximation algorithm for the group Steiner tree problem. J Algorithms 37:66–84
30. Golenberg K, Kimelfeld B, Sagiv Y (2008) Keyword proximity search in complex data graphs. In: SIGMOD, pp 927–940
31. Gonzalez J, Xin R, Dave A, Crankshaw D, Franklin M, Stoica I (2014) Graphx: graph processing in a distributed dataflow framework. In: OSDI, pp 599–613
32. Grover A, Leskovec J (2016) node2vec: Scalable feature learning for networks. In: KDD
33. Guo T, Cao X, Cong G (2015) Efficient algorithms for answering the m-closest keywords query. In: SIGMOD, pp 405–418
34. Guttman A (1984) R-trees: a dynamic index structure for spatial searching. In: SIGMOD, pp 47–57
35. Hamilton WL, Ying R, Leskovec J (2017) Inductive representation learning on large graphs. In: NIPS
36. Han S, Zou L, Yu JX, Zhao D (2017) Keyword search on RDF graph—a query graph assembly approach. In: CIKM, pp 227–236
37. Hao J, Niu B, Qin X (2019) A keyword-aware optimal route query algorithm on large-scale road networks. In: 2019 20th IEEE international conference on mobile data management (MDM), pp 587–592
38. Haryanto AA, Islam MS, Taniar D, Cheema MA (2019) Ig-tree: an efficient spatial keyword index for planning best path queries on road networks. In: WWW, pp 1359–1399
39. He H, Wang H, Yang J, Yu PS (2007) Blinks: ranked keyword searches on graphs. In: SIGMOD, pp 305–316
40. Hu J, Cheng R, Huang Z, Fang Y, Luo S (2017) On embedding uncertain graphs. In: CIKM, pp 157–166
41. Huang W, Dai G, Ge Y, Liu Y (2019) Top-k nearest keyword search in public transportation networks. In: 2019 15th international conference on semantics, knowledge and grids, pp 67–74
42. Huang X, Cheng H, Qin L, Tian W, Yu JX (2014) Querying k-truss community in large and dynamic graphs. In: SIGMOD, pp 1311–1322
43. Huang X, Lakshmanan LVS (2017) Attribute-driven community search. VLDB 10:949–960
44. Ihler E (1991) The complexity of approximating the class Steiner tree problem. In: 17th international workshop, WG
45. Jia Z, Kwon Y, Shipman G, McCormick P, Erez M, Aiken A (2018) A distributed multi-GPU system for fast graph processing. In: SIGMOD, pp 297–310
46. Jiang J, Choi B, Xu J, Bhowmick SS (2019) A generic ontology framework for indexing keyword search on massive graphs. Trans Knowl Data Eng. https://doi.org/10.1109/TKDE.2019.2956535
47. Jiang J, Huang X, Choi B, Xu J, Bhowmick SS, Xu L (2020) Ppkws: An efficient framework for keyword search on public-private networks. In: ICDE, pp 457–468
48. Jiang M, Fu AW, Wong RC (2015) Exact top-k nearest keyword search in large networks. In: SIGMOD, pp 393–404
49. Kacholia V, Pandit S, Chakrabarti S, Sudarshan S, Desai R, Karambelkar H (2005) Bidirectional expansion for keyword search on graph databases. In: VLDB '05: Proceedings of the 31st international conference on very large data bases, pp 505–516
50. Kaffes V, Belesiotis A, Skoutas D, Skiadopoulos S (2018) Finding shortest keyword covering routes in road networks. In: SSDBM
51. Kargar M, An A (2011) Keyword search in graphs: finding r-cliques. VLDB 10(14778/2021017):2021025
52. Kargar M, An A (2012) Efficient top-k keyword search in graphs with polynomial delay. In: ICDE, pp 1269–1272
53. Kargar M, Zihayat M, An A (2013) Finding affordable and collaborative teams from a network of experts. In: Proceedings of the 2013 SIAM international conference on data mining, pp 587–595
54. Kasneci G, Ramanath M, Sozio M, Suchanek FM, Weikum G (2009) Star: Steiner-tree approximation in relationship graphs. In: ICDE
55. Kimelfeld B, Sagiv Y(2006) Finding and approximating top-k answers in keyword proximity search. In: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pp 173–182

56. Kolahdouzan M, Shahabi C (2004) Voronoi-based k nearest neighbor search for spatial network databases. VLDB, pp 840–851
57. Kunkel A, Rheinländer A, Schiefer C, Helmer S, Bouros+3 P, Leser U (2016) Piejoin: Towards parallel set containment joins. In: SSDBM, p 11
58. Lappas T, Liu K, Terzi E (2009) Finding a team of experts in social networks. In: KDD, pp 467–476
59. Lawler EL (1972) A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. Manag Sci. https://doi.org/10.1287/mnsc.18.7.401
60. Le W, Li F, Kementsietsidie A, Duan S (2014) Scalable keyword search on large RDF data. TKDE 26:2774–2788
61. Li G, Ooi BC, Feng J, Wang J, Zhou L (2008) Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In: SIGMOD, pp 903–914
62. Li RH, Qin L, Yu JX, Mao R (2016) Efficient and progressive group Steiner tree search. In: SIGMOD, pp 91–106
63. Li Z, Lee KC, Zheng B, Lee WC, Lee D, Wang X (2010) Ir-tree: an efficient index for geographic document search. TKDE
64. Liu Q, Zhu Y, Zhao M, Huang X, Xu J, Gao Y (2020) Vac: Vertex-centric attributed community search. In: ICDE, pp 937–948
65. Luo J, Cao X, Xie X, Qu Q, Xu Z, Jensen CS (2020) Efficient attribute-constrained co-located community search. In: ICDE, pp 1201–1212
66. Luo S, Luo Y, Zhou S, Cong G, Guan J (2012) Disks: A system for distributed spatial group keyword search on road networks. VLDB 5:1966–1969
67. Luo Y, Fletcher GH, Hidders J, De Bra P (2015) Efficient and scalable trie-based algorithms for computing set containment relations. In: ICDE, pp 303–314
68. Majumder A, Datta S, Naidu K (2012) Capacitated team formation problem on social networks. In: KDD, pp 1005–1013
69. Mann W, Augsten N, Bouros P (2016) An empirical evaluation of set similarity join techniques. PVLDB 9(9):636–647
70. Qiao M, Qin L, Cheng H, Yu JX, Tian W (2013) Top-k nearest keyword search on large graphs. VLDB, pp 901–912
71. Qin L, Yu JX, Chang L, Tao Y (2009) Querying communities in relational databases. In: ICDE, pp 724–735
72. Rocha-Junior JB, Gkorgkas O, Jonassen S, Norvag K (2011) Efficient processing of top-k spatial keyword queries. In: SSTD
73. Rocha-Junior JB, Norvag K (2012) Top-k spatial keyword queries on road networks. In: EDBT, pp 168–179
74. Rocha-Junior JB, Vlachou A, Doulkeridis C, Nørvåg K (2010) Efficient processing of top-k spatial preference queries. VLDB 4:93–104
75. Sarma AD, Gollapudi S, Najork M, Panigrahy R (2010) A sketch-based distance oracle for web-scale graphs. In: WSDM, pp 401–410
76. Shan Y, Li M, Chen Y (2017) Constructing target-aware results for keyword search on knowledge graphs. DKE, pp 1–23
77. Shekarpour S, Marx E, Ngomo ACN, Auer S (2015) Sina: Semantic interpretation of user queries for question answering on interlinked data. J Web Seman 30:39–51
78. Shi J, Wu D, Mamoulis N (2016) Top-k relevant semantic place retrieval on spatial RDF data. SIGMOD 29:893–917
79. Shi Y, Cheng G, Kharlamov E (2020) Keyword search over knowledge graphs via static and dynamic hub labelings. In: WWW, pp 235–245
80. Tao Y, Papadopoulos S, Sheng C, Stefanidis K (2011) Nearest keyword search in xml documents. In: SIGMOD, pp 589–600
81. Teng Y, Cheng X, Su S, Bi R (2016) Privacy-preserving top-k nearest keyword search on outsourced graphs. In: 2016 IEEE Trustcom/BigDataSE/ISPA, pp 815–822
82. Tran T, Wang H, Rudolph S, Cimiano P (2009) Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. In: ICDE, pp 405–416
83. Wang H, Aggarwal CC (2009) A survey of algorithms for keyword search on graph data. In: Managing and mining graph data, pp 249–273
84. Wang J, Li G, Feng J (2012) Can we beat the prefix filtering?: An adaptive framework for similarity join and search. In: SIGMOD
85. Wang W, He Z, Shi P, Wu W, Jiang Y, An B, Hao Z, Chen B (2018) Strategic social team crowdsourcing: forming a team of truthful workers for crowdsourcing in social networks. IEEE Trans Mobile Comput 18:1419–1432
86. Wang X, Qin L, Lin X, Zhang Y, Chang L (2017) Leveraging set relations in exact set similarity join. PVLDB 10(9):925–936
87. Wen YT, Yeo J, Peng WC, Hwang SW (2017) Efficient keyword-aware representative travel route recommendation. IEEE Trans Knowl Data Eng 29:1639–1652
88. Wu D, Cong G, Jensen CS (2012) A framework for efficient spatial web object retrieval. VLDB 21:797–822
89. Xiao C, Wang W, Lin X, Yu JX, Wang G (2011) Efficient similarity joins for near-duplicate detection. ACM TODS. https://doi.org/10.1145/2000824.2000825
90. Yang J, Zhang W, Wang X, Zhang Y, Lin X (2020) Distributed streaming set similarity join. In: ICDE
91. Yang J, Zhang W, Yang S, Zhang Y, Lin X (2017) Tt-join: Efficient set containment join. In: ICDE
92. Yang J, Zhang W, Yang S, Zhang Y, Lin X, Yuan L (2018) Efficient set containment join. VLDB J 27:471–495
93. Yang J, Zhang W, Zhang Y, Wang X, Lin X (2017) Categorical top-k spatial influence query. WWWJ. https://doi.org/10.1007/s11280-016-0383-3
94. Yang M, Ding B, Chaudhuri S, Chakrabarti K (2014) Finding patterns in a knowledge base using keywords to compose table answers. VLDB. https://doi.org/10.14778/2733085.2733088
95. Yang Y, Agrawal D, Jagadish H, Tung AKH, Wu S (2019) An efficient parallel keyword search engine on knowledge graphs. In: ICDE, pp 338–349
96. Yang Y, Tung AKH (2020) Efficient radial pattern keyword search on knowledge graphs in parallel. arXiv:Databases
97. Yao B, Tang M, Li F (2011) Multi-approximate-keyword routing in GIS data. In: Proceedings of the 19th ACM SIGSPATIAL international conference on advances in geographic information systems, pp 201–210
98. Yin X, Qu C, Wang Q, Wu F, Liu B, Chen F, Chen X, Fang D (2018) Social connection aware team formation for participatory tasks. IEEE Access 6:20309–20319
99. Yu JX, Qin L, Chang L (2010) Keyword search in relational databases: a survey. IEEE BULLETIN, pp 67–78
100. Zhang C, Zhang Y, Zhang W, Lin X (2013) Inverted linear quadtree: efficient top k spatial keyword search. In: ICDE, pp 1706–1721
101. Zhang C, Zhang Y, Zhang W, Lin X (2016) Inverted linear quadtree: efficient top k spatial keyword search. IEEE Trans Knowl Data Eng 28(7):1706–1721
102. Zhang C, Zhang Y, Zhang W, Lin X, Cheema MA, Wang X (2013) Diversified spatial keyword search on road networks. In: EDBT, pp 367–378
103. Zhang D, Chan CY, Tan KL (2014) Processing spatial keyword query as a top-k aggregation query. In: SIGIR, pp 355–364
104. Zhao J, Gao Y, Chen G, Chen R (2018) Why-not questions on top-k geo-social keyword queries in road networks. In: ICDE, pp 965–976
105. Zha, J, Gao Y, Chen G, Jensen CS, Chen R, Cai D (2017) Reverse top-k geo-social keyword queries in road networks. In: ICDE, pp 387–398

106. Zhao J, Gao Y, Ma C, Jin P, Wen S (2020) On efficiently diversified top-k geo-social keyword query processing in road networks. Inf Sci 512:813–829
107. Zhao S, Cheng X, Su S, Shuang K (2017) Popularity-aware collective keyword queries in road networks. GeoInformatica 21:485–518
108. Zhao S, Zhao L, Su S, Cheng X, Xiong L (2018) Group-based keyword-aware route querying in road networks. Inf Sci. https://doi.org/10.1016/j.ins.2018.03.058
109. Zheng B, Zheng K, Xiao X, Su H, Yin H, Zhou X, Li G (2016) Keyword-aware continuous KNN query on road networks. In: ICDE, pp 871–882
110. Zhong R, Li G, Tan KL, Zhou L (2013) G-tree: an efficient index for KNN search on road networks. In: CIKM, pp 39–48
111. Zhong R, Li G, Tan KL, Zhou L, Gong Z (2015) G-tree: an efficient and scalable index for spatial search on road networks. IEEE Trans Knowl Data Eng 27:2175–2189
112. Zhou Q, Wang C, Xiong M, Wang H, Yu Y (2007) Spark: adapting keyword query to semantic search. In: International semantic web conference, pp 694–707
113. Zhu Q, Cheng H, Huang X (2017) I/O-efficient algorithms for top-k nearest keyword search in massive graphs. VLDBJ 26:563–583
114. Zhu Y, Zhang Q, Qin L, Chang L, Yu JX (2018) Querying cohesive subgraphs by keywords. In: ICDE, pp 1324–1327
115. Zhu Y, Zhang Q, Qin L, Chang L, Yu JX (2020) Cohesive subgraph search using keywords in large networks. IEEE Trans Knowl Data Eng. https://doi.org/10.1109/TKDE.2020.2975793
116. Zobel J, Moffat A (2006) Inverted files for text search engines. ACM Comput Surv. https://doi.org/10.1145/1132956.1132959