Cohesive Subgraph Search Using Keywords in Large Networks

Journal:	*Transactions on Knowledge and Data Engineering*
Manuscript ID	TKDE-2019-08-0814
Manuscript Type:	Regular
Keywords:	Cohesive subgraph, Keyword search, Large networks, Subgraph search

Cohesive Subgraph Search Using Keywords in Large Networks

Yuanyuan Zhu, Qian Zhang, Lu Qin, Lijun Chang, Jeffrey Xu Yu

Abstract—Keyword search problem has been widely studied to retrieve relevant substructures from graphs for a given set of keywords. However, existing well-studied approaches aim at finding compact trees/subgraphs containing the keywords, and ignore a critical measure, density, to represent how strongly and stablely the keyword nodes are connected in the substructure. In this paper, given a set of keywords $Q = \{w_1, w_2, \ldots, w_l\}$, we study the problem of finding a cohesive subgraph containing Q with high density and compactness from a graph G. We model the cohesive subgraph based on a carefully chosen k-truss model, and formulate the problem of finding cohesive subgraphs for keyword queries as *minimal dense truss* search problem, i.e., finding minimal subgraph that maximizes the trussness covering Q. However, unlike k-truss based community search that can be efficiently done based on local search from a given set of nodes, *minimal dense truss* search for keyword quires is a nontrivial task as the subset of keyword nodes to be included in the retrieved substructure is previously unknown. To tackle this problem, we first design a novel hybrid KT-Index to keep the keyword and truss information in a compact manner, and propose an efficient algorithm which carries search on KT-Index directly to find the dense truss with the maximum trussness G_{den} without repeated accesses to the original graph. Then, we develop a novel refinement approach to extract minimal dense truss from the dense truss G_{den}, by checking each node at most once based on the anti-monotonicity property derived from k-truss, together with several optimization strategies including batch based deletion, early-stop based deletion, and local exploration. Moreover, we also extend the proposed method to deal with top-r search. Extensive experimental studies on real-world networks validated the effectiveness and efficiency of our approaches.

Index Terms—Cohesive subgraph, Subgraph Search, Keyword search, Graph Database.

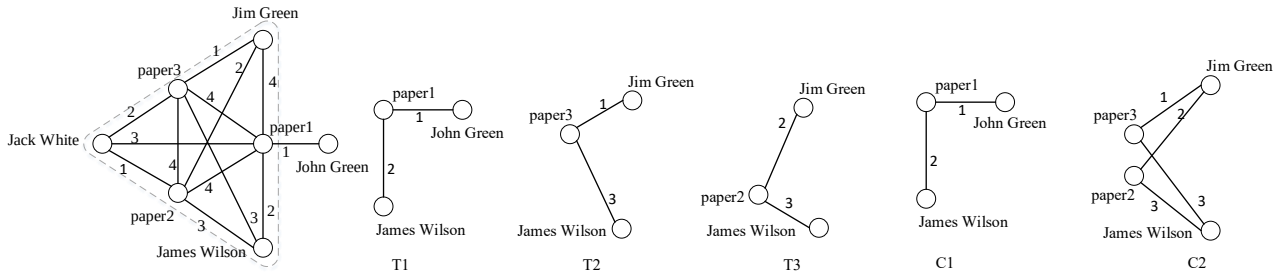
1 Introduction

Keyword search, as a user-friendly query scheme, has been widely used to retrieve useful information in graph data, such as knowledge graphs, information networks, social networks, etc. Given a query consisting of a number of keywords, the target of keyword search over a graph is to find substructures in the graph relevant to the query keywords.

In recent decades, keyword search has been extensively studied in the literature [1]. Most of the earlier works aim to find minimal connected trees containing the keywords based on a weight function [2] [3] [4]. These trees returned may be compact, but each of them only gives a partial view of relationships between the keywords. Thus, connected subgraphs covering the keywords were subsequently proposed, such as r-radius subgraph [5], community [6], and r-clique [7]. Besides, keyword search can also be considered as a special case of partial topology query [8] [9] where label propagation are utilized to find matched components. However, these methods only focus on the compactness of retrieved substructure by evaluating the distance between (keyword) nodes in a connected tree/subgraph, and fail to explore how densely these keywords are connected. In many applications, density is a critical measure to reflect the stability of the relationships between keywords, e.g., forming a team such that the members are stablely close with each other so that the whole team can cooperate well [10]. This means that there are multiple communication paths between two members so that they cannot be disconnected easily, which implies the high density. There are also some recent works on diversified keyword search [11] [12] [13], but they also neglect the density of retrieved substructures. Two recent works considered density in the keyword based community search, but their target is to maximum the keyword cohesiveness [14] or contextual density that combines the keyword cohesiveness and structural cohesiveness [15], which are inherently different from finding dense subgraph covering the keywords in this paper.

In this paper, for the first time, we study the problem of finding cohesive subgraphs that are highly dense and compact for keyword queries. Various cohesive subgraph models have been proposed in the literature, such as k-core [16] [17] [18] [19] [20], k-truss [21] [22] [23], k-edge connected component [24] [25], to name a few. We choose k-truss, within which each edge is contained in at least $(k-2)$ triangles, to model the substructures for keyword queries due to its good properties as follows

- *Y. Zhu and Q. Zhang are with the School of Computer Science, Wuhan University, China.*
  E-mail: yyzhu@whu.edu.cn.
- *L. Qin is with the Center for OCIS, University of Technology, Sydney, Australia.*
  E-mail: lu.qin@uts.edu.au.
- *L. Chang is with School of Computer Science, The University of Sydney, Australia .*
  E-mail: llijun.chang@sydney.edu.au.
- *J. Yu is with Systems Engineering and Engineering Management, Chinese University of Hong Kong, Hong Kong, China.*
  E-mail: yu@se.cuhk.edu.hk.



(a) Graph G (b) Trees (c) Communities

Fig. 1: A Motivating Example

[26]. (1) *Stable relationship*. k-truss is defined based on a higher order graph motif, triangle, rather than primitive vertices/edges, which is a fundamental building block of networks and can ensure us to find strong and stable relationship between keyword nodes in an answer. (2) *Bounded diameter*. The diameter of a connected k-truss with n vertices is bounded by $\lfloor \frac{2n-2}{k} \rfloor$. This ensures the compactness of the retrieved subgraphs. (3) *High density*. A connected k-truss is also a $(k-1)$-core and a $(k-1)$-edge connected subgraph, i.e., all nodes have degree at least $k-1$ and remain connected whenever fewer than $k-1$ edges are removed. This property ensures that the keyword nodes in the retrieved subgraphs are densely connected. We illustrate the differences between k-truss and existing keyword search approaches (e.g., Steiner tree [2] [3], community [6], and r-clique [7]) by the following example.

Fig. 1(a) shows a co-authorship and citation graph G with 4 authors and 3 papers. The weight of an edge between an author and a paper is the rank of the author in the paper, and the weight of an edge between two papers is the citation frequency. For a query $Q = \{James, Green\}$, the top-3 connected trees T_1, T_2, and T_3 with weight 3, 4, and 5 are identified respectively by [2] [3] as shown in Fig. 1(b). Fig. 1(c) shows the communities identified by [6], which are multi-centered subgraphs with the distance between a center node and each keyword node no larger than a given threshold (e.g., 3). They are ranked based on the minimum total edge weight from a center node to each keyword node on the corresponding shortest path. The score of community C_1 with center node paper1 is $1+2=3$. The score of community C_2 is 4 as it has two center nodes paper2 and paper3 with total weights $2+3=5$ and $1+3=4$, respectively. In the r-clique model with diameter no larger than r (e.g., $r=3$) [7], T_1 and T_2 are returned, as only Steiner trees of qualified r-cliques are finally extracted. All these approaches return substructures containing James Wilson and John Green as the top-1 answer. However, James Wilson and Jim Green coauthored more papers together with Jack White, which implies a more stable and closer relationships. Based on our truss model, they can be properly discovered in the form of 4-truss (the dashed line area in Fig. 1(a)).

1.1 Challenges

To attain highly dense and compact substructure for a keyword query Q, a natural way is to find the subgraph with maximum trussness and minimum size containing Q, which is called as *minimum dense truss*. However, as we will discuss in Section 2, finding minimum dense truss containing the query keywords is NP-hard. Moreover, this problem is also APX-hard, which means we cannot find a polynomial-time algorithm that approximates the minimum dense truss search problem within any constant ratio unless P = NP. Thus, in this paper, we study a relaxed version, called *minimal dense truss search*, i.e., find the subgraph with maximum trussness containing Q such that it does not contain any subgraph with the same trussness containing Q. Note that our model is different from the closest truss model [22] with maximum trussness and minimum diameter, as the diameter of a k-truss with n nodes is bounded by $\lfloor \frac{2n-2}{k} \rfloor$ while a k-truss with minimum diameter may have an arbitrary large number of nodes. Moreover, closest truss search is NP-hard [22], while minimal dense truss search can be done in polynomial time.

Despite rich studies on related problems such as community detection [26] [27] [28] [29] and community search [21] [23], finding minimal dense truss for a keyword query is a nontrivial task. The main reason is that minimal dense truss search for query keywords is inherently different from these two problems. Community detection is query independent aiming to detect maximal k-truss communities for each k, which can be done by truss decomposition in $O(|E|^{1.5})$ time [27]. Community search aims to find maximal communities that maximize the trussness and contain a given set of query nodes, which can be done by local search with proper indexes in $O(|\mathcal{A}|)$ time ($\mathcal{A}$ is the answer) [21] [23]. One recent work [30] studied attributed community search to rank communities based on attribute score, but it also require the input of a subset of nodes. The main difficulty of minimal dense truss search for a keyword query is that, unlike community search where the query nodes are given, the subset of nodes containing all the keywords to be included in the dense truss to be retrieved is unknown in advance, and therefore we do not know from which nodes to start if we adopt the local search approach in [21] [23]. One possible solution is that, for a

keyword query $Q = \{w_1, w_2, \ldots, w_l\}$, we explore all the combinations of keyword nodes in $\mathcal{S} = V_1 \times V_2 \times \cdots \times V_l$ to find the subgraph with the maximum trussness, where V_i is the node set containing w_i. Such search space is inexhaustible for very large graphs even when we use the local search method with essentially optimal time $O(|\mathcal{A}|)$ in [21] [23], due to the huge number of combinations. Moreover, suppose we have already obtained a truss with the maximum trussness containing Q, verifying the minimality of such truss is also time consuming, because we need to check all of its subgraphs containing Q to make sure that there is no subgraph with the same trussness.

1.2 Contributions

In this paper, we tackle the minimal dense truss search problem for a keyword query Q by dividing it into two subtasks. The first is finding the dense truss G_{den} with the maximum trussness containing Q. The second is refining G_{den} to obtain a minimal dense truss H containing Q. For this problem, we can either solve it in a bottom-up manner or top-down manner, which differ in the solution for the first subtask. Due to the limit of space, we only give a basic solution of the top-down manner framework in the preliminary version based on KT-Index [31] and did not give the through analysis and optimization techniques. In this version, we will first give the hardness analysis of the problem studied in this paper. Then we discuss and thoroughly compare the bottom-up framework and top-down framework. Furthermore, we discuss the extension of proposed method to top-r search. Moreover, although we only need to check each node at most once based on the anti-monotonicity property of k-truss in the preliminary version [31], the refinement process is still time consuming as we have to do the k-truss verification for each deletion. Thus, we further propose several optimization strategies to accelerate the refinement process, including batch based deletion, early-stop based deletion, and local exploration.We also perform extensive experimental studies on more real-word datasets and parameters to show that our newly proposed method is faster than our previously proposed method [31] by one order of magnitude. We summarize the substantial improvements as follows.

- We study the problem of finding cohesive subgraph for keyword queries, and formulate it as a carefully chosen k-truss model. To the best of our knowledge, this problem has not been studied in the literature.
- We analyze the hardness of the studied problem, and propose two different basic frameworks, namely bottom-up framework and top-down framework with through comparisons.
- We design a novel hybrid index KT-Index to keep the keyword information and truss information, which is space and time efficient, and propose a novel top-down algorithm based on KT-Index, which can efficiently find the dense truss with the maximum trussness for a keyword query without repeated accesses to the original graph G.
- We develop an efficient refinement algorithm to extract minimal dense truss containing Q from dense truss, with several optimization strategies including batch based deletion, early-stop based deletion, and local exploration to further accelerate the refinement process.
- We conducted extensive experimental studies on multiple real-world networks and validate the effectiveness and efficiency of our approach on discovering cohesive substructures for keyword queries.

The rest of the paper is organized as follows. Section 2 gives the problem statement and hardness analysis. Section 3 presents two basic algorithmic frameworks. Section 4 illustrates the improved top-down algorithms based on KT-Index, and a novel algorithm for refining a dense truss to extract the minimal dense truss for a keyword query. Our experimental results are shown in Section 5. Section 6 discusses the related works and Section 7 concludes this paper.

2 Problem Statement

In this section, we will first describe related notations and definitions, and then analyze the hardness of the minimum dense truss search problem for keywords.

2.1 Notations and Definitions

Given a set of labels Σ, a simple undirected vertex labeled graph is represented as $G = (V, E, L)$, where V is the set of vertices, $E \subseteq V \times V$ is the set of edges, and L is a labeling function which assigns each node a set of labels $L(v) \subset \Sigma$. We use $V(G)$ and $E(G)$ to denote the set of vertices and the set of edges of graph G respectively, and use $|V(G)|$ and $|E(G)|$ to denote the number of vertices and number of edges in G respectively. For a vertex $v \in V$, we denote the set of its neighboring vertices by $N(v) = \{u \in V | (u, v) \in E\}$ and its degree by $d(v) = |N(v)|$. A *triangle* $\triangle(u, v, w)$ in G is a substructure such that $(u, v), (v, w), (u, w) \in E$.

Definition 2.1 (Edge Support). The support of an edge $e = (u, v)$ in graph G is the number of triangles in which e occurs, defined as $sup_G(e) = |\{\triangle(u, v, w) | w \in V(G)\}|$.

In the following, we use $sup(e)$ and $sup_G(e)$ interchangeably if the context is clear.

Definition 2.2 (Connected k-Truss). Given a graph G and an integer k, a connected k-truss is a connected subgraph $H \subseteq G$, such that $\forall e \in E(H)$, $sup_H(e) \geq k - 2$.

This means that in a connected k-truss, the end vertices of each edge have at least $k-2$ common neighboring vertices in this truss. Thus, the degree of each vertex in a connected k-truss is at least $k - 1$, and a connected k-truss is a also a $(k - 1)$-core.

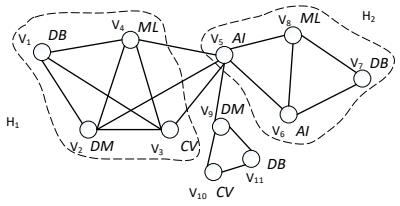


Fig. 2: An Example Graph G

The trussness of a subgraph $H \subseteq G$ is the minimum support of all the edges in H plus 2, defined as $\tau(H) = \min_{e \in E(H)} sup_H(e)+2$. The trussness of an edge $e \in E(G)$ is the maximum trussness of subgraphs containing e, i.e., $\tau(e) = \max_{H \subseteq G \wedge e \in E(H)} \tau(H)$. The trussness of a vertex $v \in V(G)$ equals to the maximum trussness of its adjacent edges, i.e., $\tau(v) = \max_{u \in N(v)} \tau(u, v)$.

For example, in Fig. 2, the edge support of (v_2, v_3) is 3 as it is contained in 3 triangles $\triangle(v_1, v_2, v_3)$, $\triangle(v_2, v_3, v_4)$ and $\triangle(v_2, v_3, v_5)$. Let H_1 denote the subgraph induced by vertices $\{v_1, v_2, v_3, v_4\}$. $\tau(H_1) = 4$ since the minimum support of edges in H_1 is 2. The trussness of edge (v_2, v_3) is 4 because there is no other subgraph with higher trussness containing (v_2, v_3). $\tau(v_2) = 4$ because the maximum trussness of its adjacent edges (v_2, v_1), (v_2, v_3), (v_2, v_4) and (v_2, v_5) is 4.

Definition 2.3 (Dense Truss Over Keywords). Given a graph G and a keyword set Q, a dense truss over Q is a connected truss $G_{den} \subseteq G$ that maximizes the trussness and contains Q.

Definition 2.4 (Minimum Dense Truss Over Keywords). Given a graph G and a keyword set Q, the minimum dense truss over keywords Q is a dense truss $G_{den} \subseteq G$ containing Q with minimum number of nodes.

Definition 2.5 (Minmal Dense Truss Over Keywords). Given a graph G and a keyword set Q, the minimal dense truss over Q is a dense truss $G_{den} \subseteq G$ containing Q such that any subgraph of G_{den} is not a dense truss containing Q.

For example, consider a query $Q = \{DB, ML\}$. H_1 and H_2 in Fig. 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 Q.

Problem 1 *(Minimum Dense Truss Search by Keywords)*. Given a graph G and a keyword set $Q = \{w_1, w_2, \cdots, w_l\}$, find the minimum dense truss containing Q. We refer to it as *minimum dense truss search* problem when the context is clear.

2.2 Hardness of Minimum Dense Truss Search

We show the NP hardness of minimum dense truss search problem by the reduction of maximum clique problem as in [22]. The decision version of minimum dense truss search problem can be defined as follows.

(k, h)-Truss Problem. Given a graph G, a keyword set Q, integers k and $h \geq k$, test whether G contains a connected k-truss with h nodes covering all the keywords in Q.

Theorem 2.1. *(k, h)-truss problem is NP-hard.*

Proof. We prove this by reducing the well known NP-hard problem, maximum clique decision problem, to (k, h)-truss problem. Given a graph G and a number k, the maximum clique decision problem is to check whether G contains a clique of size k. Now we construct an instance of (k, h)-truss problem, consisting of a graph G, parameters k and $h = k$, and query $Q = \emptyset$. Next, we show that the instance of the maximum clique decision problem is a YES-instance iff the instance of (k, h)-truss problem is a YES-instance. Clearly, any clique with k nodes is a connected k-truss with size $h = k$. On the other hand, given a solution H for (k, h)-truss problem. H must contain k nodes since H is a k-truss and $h = k$, which implies H is a clique. □

The above theorem implies the hardness of the minimum dense truss search problem. Next, we further explore whether it can be approximated. For a given constant $c \geq 1$ and any instance G, an algorithm can achieve an c-approximation to the minimum dense truss search problem if it outputs a connected k-truss subgraph H such that $\tau(H) = \tau(H^*)$ and $|V(H)| \leq c \times |V(H^*)|$ where H^* is the optimal solution. In the following, we prove that the problem does not admit a polynomial-time algorithm that can achieve any constant approximation ratio unless P = NP. We obtain this result also based on the reduction of the maximum clique decision problem.

Theorem 2.2. *Unless P=NP, there does not exist any polynomial-time algorithm that approximates the minimum dense truss search problem within any constant ratio.*

Proof. We prove this by contradiction. Assume that there exists a polynomial time algorithm $\mathcal{A}$ for minimum dense truss search problem with a given k that provides a solution H with an approximation ratio c of the optimal solution H^*. Now we consider the case when $Q = \emptyset$. Clearly, based on the assumption we have a subgraph H such that $\tau(H) = \tau(H^*)$ and $|V(H)| \leq c \times |V(H^*)|$. Next, we use this approximate solution to solve maximum clique decision problem, i.e., run algorithm $\mathcal{A}$ on a given instance G of maximum clique decision problem, with parameters k, $h = k$, and query $Q = \emptyset$. We claim that G contains a clique of size k iff $\mathcal{A}$ output a solution H with $\tau(H) = k$ and $|V(H)| = k$. To see this, suppose that $|V(H)| = k$. Then the optimal solution has $|V(H^*)| \leq |V(H)| = k$, which shows H^* is a clique of size k. On the other hand, suppose that $|V(H)| \geq k + 1$. Then we have $c \times |V(H^*)| \geq |V(H)| \geq k + 1$ for any $c > 1$. Thus, we have $|V(H^*)| \geq k + 1$. In this case, G cannot contain a clique of size k, because if it did, that clique would be the optimal solution to the minimum dense truss search problem with parameter k and size $h = k$, which contradicts the optimality of H^*. Thus using algorithm

Algorithm 1: Bottom-up Framework

Input : A graph G, and a keyword query Q.
Output: A minimal dense truss.

1 compute edge and node trussness by truss decomposition;
2 **for** *each keyword* $w_i \in Q$ **do**
3 $\quad V_i \leftarrow$ set of nodes containing w_i;
4 $\mathcal{S} \leftarrow V_1 \times V_2 \times \cdots \times V_l$;
5 **for** *each* $S \in \mathcal{S}$ **do**
6 $\quad \tau'(S) \leftarrow \min_{v \in S} \tau(v)$;
7 sort $\mathcal{S}$ in descending order of $\tau'(S)$;
8 $k_{max} \leftarrow 0$; $j \leftarrow 0$;
9 **while** $j++ \leq |\mathcal{S}| \wedge k_{max} < \tau'(S_j)$ **do**
10 $\quad G_j \leftarrow$ FindDenseTruss(G, S_j);
11 $\quad$ **if** $k_{max} < \tau(G_j)$ **then**
12 $\quad\quad k_{max} \leftarrow \tau(G_j)$;
13 $\quad\quad G_{den} \leftarrow G_j$;
14 $H \leftarrow$ FindMinDenseTruss(G_{den}, Q);
15 return H;

Algorithm 2: Top-Down Framework

Input : A graph G, and a keyword query Q.
Output: A minimal dense truss.

1 compute edge and node trussness by truss decomposition;
2 **for** *each keyword* $w_i \in Q$ **do**
3 $\quad V_i \leftarrow$ set of nodes containing w_i;
4 $\quad \tau'(w_i) \leftarrow \max_{v \in V_i}(v)$;
5 $G_{den} \leftarrow \emptyset$;
6 $k_{max} \leftarrow \min_{1 \le i \le l} \tau'(w_i)$;
7 **while** $G_{den} = \emptyset$ **do**
8 $\quad$ extract k_{max}-truss $G_{k_{max}} = \{e \in G | \tau(e) \ge k_{max})\}$;
9 $\quad$ **for** *each connected component* C_i *in* $G_{k_{max}}$ **do**
10 $\quad\quad$ **if** C_i *contains all keywords in* Q **then**
11 $\quad\quad\quad$ **if** $|V(C_i)| \le |V(G_{den})|$ **then**
12 $\quad\quad\quad\quad G_{den} \leftarrow C_i$;
13 $\quad k_{max} \leftarrow k_{max} - 1$;
14 $H \leftarrow$ FindMinDenseTruss(G_{den}, Q);
15 return H;

$\mathcal{A}$ we can distinguish between the YES and NO instances of the maximum clique decision problem. □

The above theorems show that it is not only intractable to to obtain a minimum dense truss for a keyword query, but also hard to get its approximate solution in polynomial time. Hence, in the following, we focus on finding the minimal dense truss for keyword query, which can be solved in polynomial time.

Problem 2 *(Minimal Dense Truss Search by Keywords)*. Given a graph G and a keyword set $Q = \{w_1, w_2, \ldots, w_l\}$, find the minimal dense truss containing keywords Q. We refer to it as *minimal dense truss search* problem when the context is clear.

Note that some applications may require to find top-r minimal dense trusses for keywords, which are ranked by the trussness. For simplicity, we will first consider the top-1 minimal dense truss search for keywords and discuss how to extend it to top-r version later.

3 Basic Algorithmic Frameworks

As stated before, finding minimal dense truss for keyword query Q can be naturally divided into two subproblems. The first is finding the dense truss G_{den} that maximum the trussness containing Q. The second is refining G_{den} to obtain a minimal dense truss containing Q. In this section, we propose two basic algorithmic frameworks, which mainly differ in solving the first subproblem. The details of tackling the second subproblem will be introduced later in Section 4.

3.1 Basic Bottom-up Framework

To obtain the dense truss containing keyword query Q, one naive method is to explore all the combinations of keyword nodes in $\mathcal{S} = V_1 \times V_2 \times \cdots \times V_l$. Specifically, for each keyword node set $S_j \in \mathcal{S}$, we obtain a dense truss G_j containing S_j that maximum the trussness. Then, among all the dense trusses discovered, we return a truss G_{den} with the largest trussness. Finally, we refine G_{den} to extract the minimal dense truss H containing Q. Such process is quite time consuming because we need to find the dense truss for $|V_1| \times |V_2| \times \cdots \times |V_2|$ sets of nodes.

To find the dense truss G_{den} as early as possible, we will utilize the upper bound for the trussness of a node subset S. For a node subset $S \subseteq V(G)$, the upper bound of its trussness is defined as $\tau'(S) = \min_{v \in S} \tau(v)$.

Lemma 3.1. *Given a graph G and a subset of nodes $S \subseteq V(G)$, for any truss H containing S, we have $\tau(H) \le \tau'(S)$.*

This lemma can be easily derived from the definition of trussness of a subgraph ($\tau(H) = \min_{v \in V(H)} \tau(v)$), the upper bound of the trussness for a node set S ($\tau'(S) = \min_{v \in S} \tau(v)$), and $S \subseteq V(H)$.

The detailed bottom-up framework is shown in Algorithm 1. Lines 1-8 initialize the variables. From line 9 to line 13, we sequentially check each $S_j \in \mathcal{S}$ in descending order of $\tau'(S_j)$ to find the dense truss G_j containing S_j. This process stops when the upper bound for current node subset is no larger than the largest trussness k_{max} found so far. After finding the dense truss G_{den} containing Q, we will refine G_{den} to find the minimal dense truss H containing Q by procedure FindMinDenseTruss, which will be discussed later in Section 4.

Function FindDenseTruss in line 10 is to find a subgraph with the largest trussness containing Q. It can be achieved by truss decomposition, which needs $O(|E(G_j)|^{1.5})$ time [27]. If a proper index is adopted [22], it can be done in $O(|E(G_j)|)$ time, which is optimal.

Theorem 3.1. *The time complexity of finding the dense truss G_{den} containing Q in the* Bottom-up Framework *in Algorithm 1 is* $O(\max\{|E|^{1.5}, |V_1| \times |V_2| \times \cdots \times |V_l| \times |E|\})$.

Proof. First, the trussness of edges and nodes can be computed by truss decomposition in $O(|E|^{1.5})$ time [27]. For each S_j, the process FindDenseTruss can be completed in $O(|E(G_j)|)$ time if the index based method in [22] is adopted. Thus the whole loop needs $O(|V_1| \times |V_2| \times \cdots \times |V_l| \times |E|\}$ time. Without the consideration of the cost of finding the minimal dense truss (line 14), the time complexity of finding dense truss G_{den} is $O(\max\{|E|^{1.5}, |V_1| \times |V_2| \times \cdots \times |V_l| \times |E|\})$.

Such complexity implies that Bottom-up Framework is impractical for real large graphs even for a small l. Consider the case that $l = 3$ and $|V_i| \approx 10^3$. The complexity is even as large as $O(10^9 \times |E|)$.

Note that we can also sort the elements in $\mathcal{S}$ by other measures like trussness of Steiner tree induced by each $S \in \mathcal{S}$ in [22]. But such heuristics does not change the stop condition and thus cannot reduce the worst-case complexity. Meanwhile, it also needs extra $O(|E| + |V|log|V|)$ time [32] to approximately compute the Steiner tree for each $S \in \mathcal{S}$.

3.2 Basic Top-down Framework

To avoid enumerating all the combinations of keyword nodes, we propose a top-down framework by starting the search over the truss with the largest trussness k_{max} in graph G. If it does not contain a connected k_{max}-truss covering Q, we will gradually decrease k_{max} until we find one. Such process can be accelerated by utilizing the property of trussness for keywords as follows. Let V_i be the set of nodes containing keyword w_i. The upper bound of the trussness for w_i is defined as the maximum trussness of nodes in V_i, i.e., $\tau'(w_i) = \max_{v \in V_i} \tau(v)$.

Property 3.1. Given a graph G and a keyword set $Q = \{w_1, w_2, \ldots, w_l\}$, for any truss H containing Q, we have $\tau(H) \leq \min_{1 \leq i \leq l} \tau'(w_i)$.

Proof. For a truss H containing Q, there must exist a node v_i' containing w_i for each $w_i \in Q$. Obviously, we have $\tau(H) = \min_{v \in V(H)} \tau(v) \leq \min_{1 \leq i \leq l} \tau(v_i')$. Moreover, since $\tau(v) \leq \tau'(w_i)$ for any node $v \in V_i$, we have $\tau(H) \leq \min_{1 \leq i \leq l} \tau(v_i') \leq \min_{1 \leq i \leq l} \tau'(w_i)$. □

Th detailed top-down framework is shown in Algorithm 2. First, we obtain the trussness of all the edges and nodes by truss decomposition [27] (line 1). Then, for each keyword $w_i \in Q$, we compute the node set V_i, and obtain the upper bound of trussness for this keyword by $\tau'(w_i) = \max_{v \in V_i} \tau(v)$ (lines 2-4). Based above property, we start searching from k_{max}-truss where $k_{max} = \min_{1 \leq i \leq l} \tau'(w_i)$. Specifically, we extract $G_{k_{max}} = \{e \in G | \tau(e) \geq k_{max}\}$ from G and check each connected component C_i in $G_{k_{max}}$ contains all the keywords (lines 6-13). If yes, we return the component containing Q with the smallest size; otherwise, we search $(k_{max}-1)$-truss and stop when we find a connected truss G_{den} containing Q. Finally, we refine G_{den} to obtain a minimal dense truss H (line 14) as Algorithm 1 does.

Theorem 3.2. *The time complexity of finding the dense truss G_{den} containing Q in the* Top-Down Framework *in Algorithm 1 is* $O(|E|^{1.5})$.

Proof. The complexity of lines 1-4 is bounded by truss decomposition, which is $O(|E|^{1.5})$ [27]. In lines 6-13, for a k_{max}-truss, we need $O(|E(G_{k_{max}})|)$ time to compute the connected components and $O(|V(G_{k_{max}})| \times l)$ time to check whether each component contains all the l keywords. Since l is usually very small, such process can be done in $O(|E(G_{k_{max}}|)$ time. In the worst case, we need to check all the possible values of k_{max} from $\min_{1 \leq i \leq l} \tau'(w_i)$ to 2. Since the maximum trussness of nodes in graph G is no larger than $\sqrt{|E|}$ [27], the complexity of lines 6-13 is $O(\sqrt{|E|} \times |E|)$, and the overall time complexity of finding G_{den} in Algorithm 2 is $O(|E|^{1.5})$. □

4 Improved Top-down Search Algorithms

In this section, we design a novel Keyword-Truss Index (KT-Index) to keep the keywords and trussness information, and propose a highly efficient algorithm to process minimal dense truss search for keyword queries.

4.1 KT-Index Design and Construction

In the basic top-down search framework, trussness computation for each edge is primitive. Since it is independent with keyword queries, we can complete such computation by truss decomposition [27] offline before any query comes. Then, we build a hash table to keep all the edges and their trussness.

Another time consuming part of basic top-down algorithm is that we need to test many values of k to find a k-truss containing Q with the largest k, with time complexity $O(|E|^{1.5})$. To speed up the computation of this part, we design a KT-Index including two parts: truss index and keyword index.

Truss Index. Truss index is a multi-layer structure, where all the connected k-truss are indexed in the k-th layer. Suppose that there are p_k connected components $C_1, C_2, \ldots, C_{p_k}$ in the k-th layer. We sort all the components in the descending order of their sizes (number of nodes) and assign each component an ID. For each component C_i, we only store the node set $V(C_i)$. Thus, we store the k-th layer in the form of list $(1, V(C_1)), \ldots, (i, V(C_i)), \ldots, (p_k, V(C_{p_k}))$.

Keyword Index. In the keyword index, we first store an inverted keyword list to keep the node IDs that contain each keyword, i.e., for each w_i, we store the keyword node set V_i containing w_i. Meanwhile we record the upper bound of trussness $\tau'(w_i)$ for each keyword. Moreover, for each keyword, we record the set of IDs of the component CID_k it occurs in the k-th layer, in the form of (k, CID_k).

Algorithm 3: BuildKTIndex

Input : A graph G, and a keyword query Q.
Output: KTIndex.

1 compute $\tau(e)$ of each edge e by truss decomposition;
2 $k_{max} = \max_{e \in E} \tau(e)$;
3 **for** $k = 2$ *to* k_{max} **do**
4 $\quad E_k \leftarrow \emptyset$;
5 **for** *each edge* $e \in E$ **do**
6 $\quad$ **if** $\tau(e) = k$ **then**
7 $\quad\quad E_k \leftarrow E_k \cup \{e\}$;
8 build the keyword inverted list;
9 **for** $k = 3$ *to* k_{max} **do**
10 $\quad$ obtain G_k by deleting E_{k-1} from G_{k-1};
11 $\quad$ assign each connected component in G_k a CID by the increasing order of component size;
12 $\quad$ **for** *each connected component* C_i *in* G_k **do**
13 $\quad\quad$ output $(i, V(C_i))$ to build the truss index;
14 $\quad\quad$ compute the set of keywords W_i contained in C_i; **for** *each* $w \in W_i$ **do**
15 $\quad\quad\quad$ add i to CID_k for w;

The process of constructing KT-Index is shown in Algorithm 3. First, we apply truss decomposition to obtain the trussness of each edge and store it in a hash table (line 1). Then we divide the edges into a number of groups $E_3, E_3, \ldots, E_{k_{max}}$ according to their edge trussness (lines 2-7). Next, we build the inverted keyword list by scanning the graph G (line 8). Then for each k from 3 to k_{max}, we obtain the truss G_k in k-th level by deleting edge set E_{k-1}. Then we sort the component in G_k in ascending order of component size and assign each component an ID (line 11). For each connected component C_i in G_k, we output $(i, V(C_i))$ to build the truss index (line 13). We also compute the set of keywords W_i occurs in C_i and add the component id to CID_k for each keyword $w \in W_i$ (lines 14-15).

Theorem 4.1. *KT-Index can be constructed in $O(|E|^{1.5})$ time and $O(m)$ space by Algroithm 3. The index size is $O(m)$.*

Proof. First, we analyze the time complexity. Obviously, the complexity of lines 1-8 is $O(|E|^{1.5})$, dominated by the complexity of truss decomposition [27]. Since in each loop of lines 10-15, we need $O(|E(G_k)|)$ time, the complexity of all the loops is $O(\sum_{1 \le k \le k_{max}} |E(G_k)|)$. Since each edge e occurs in at most $\tau(e)$ layers, we have $\sum_{1 \le k \le k_{max}} |E(G_k)| \le \sum_{e \in G} \tau(e) \le \sum_{e \in G} \mathrm{sup}(e) = 3 \times N_{tri}$, where N_{tri} is the number of triangles in G. As proved in [33], N_{tri} is bounded by $O(\alpha \times |E|)$ where $\alpha \le \sqrt{|E|}$ is the arboricity of graph G. Thus the overall time complexity of Algorithm 3 is $O(|E|^{1.5})$.

Next, we analyze the space complexity. Since lines 1-8 need $O(|E|)$ space and lines 10-15 need $O(|E(G_k)|)$ space, the computational space of Algorithm 3 is bounded by $O(|E|)$. Then, we consider the index space. Obviously, hash table for edge trussness needs $O(|E|)$ space. In truss index, for the k-th layer, we need to record the nodes in G_k. Thus the nodes stored in all layers is $\sum_{1 \le k \le k_{max}} |V(G_k)|$. Since each node v occurs in at most $\tau(v)$ layers and $\tau(v) \le d(v)$, we have $\sum_{1 \le k \le k_{max}} |V(G_k)| \le \sum_{v \in V} \tau(v) \le \sum_{v \in V} d(v) = 2 \times |E|$. Thus truss index needs $O(|E|)$ space. In keyword index, we need to store the inverted keyword list, which is bounded by $O(|V| \times n_{avg})$, where n_{avg} is the average number of keywords associated with each node. Since n_{avg} is usually a small constant, the index size of inverted keyword list is bound by $O(|V|)$. For each keyword, we also keep the set of component IDs CID_k containing this keyword in the k-th layer. The needed space can be easily bounded by $O(|E|)$, since CID_k is a usually very small number at most hundreds for real-world graphs (far away from the number of nodes and edges). Thus, the index size of KT-Index is $O(|E|)$. □

TABLE 1: An example for Truss Index

Truss value	Component List
$k = 4$	(1, {v_1, v_2, v_3, v_4})
$k = 3$	(1, {v_9, v_{10}, v_{11}}),(2, {v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8})
$k = 2$	(1,{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}})

TABLE 2: An example Keyword Index

Keyword w_i	$\tau'(w_i)$	Inverted List	(k, CID_k)
AI	4	v_5,v_6	(3,2)
CV	3	v_3,v_{10}	(4,1),(3,1),(3,2)
DB	4	v_1,v_7,v_{11}	(4,1),(3,1),(3,2)
DM	4	v_2,v_9	(4,1),(3,1),(3,2)
ML	4	v_4,v_8	(4,1),(3,2)

Consider the graph G as shown in Fig. 2. Based on Algorithm 3, we fist obtain the trussness of each edge. Since all edges are connected, all the nodes are in one component with trussness 2 as shown in Table 1. We then remove the edges with trussness 2 and obtain two components with trussness 3. The CID of component v_9, v_{10}, v_{11} is assigned to 1 because it is smaller than the other one. The keyword index is shown in Table 2. For keyword AI, we record the upper bound of its trussness 4, and nodes it occurs in v_5 and v_6, and the second component of layer 3.

4.2 The Improved Top-down Search Algorithm

After KT-Index is constructed, we can directly carry out the minimal dense subgraph search on this index.

The Top-Down-KT Search **Algorithm**. The search process is shown in Algorithm 4. To avoid the worst case of checking all the value of k_{max}, we check each layer of truss index by a binary search, which can be completed in $\log(k_{max})$ iterations. In the k-th layer, we obtain the set of component IDs CC that contains all the keywords (lines 4-6). If CC is empty, we will search layers with trusness smaller than current k; otherwise, we will search layers with trussness larger than current k. After finding

Algorithm 4: Top-Down-KT Search

Input : A graph G, and a keyword query Q.
Output: A minimal dense truss.

1 $k_{max} \leftarrow \min_{1 \le i \le l} \tau'(w_i)$; $k_{min} \leftarrow 3$;
2 **while** $k_{max} > k_{min}$ **do**
3 $\quad k \leftarrow \lfloor \frac{k_{max}+k_{min}}{2} \rfloor$;
4 $\quad$ **for** *each keyword* w_i *in* Q **do**
5 $\quad\quad SC_i \leftarrow CID_k$ of w_i;
6 $\quad CC \leftarrow \cap_{1 \le i \le l} SC_i$;
7 $\quad$ **if** $CC \neq \emptyset$ **then**
8 $\quad\quad k_{min} \leftarrow k+1$;
9 $\quad$ **else**
10 $\quad\quad k_{max} \leftarrow k-1$;
11 $id \leftarrow \min_{cid \in CC} cid$;
12 $G_{den} \leftarrow$ component C_{id} at the k-th layer;
13 $H \leftarrow$ FindMinDenseTruss(G_{den}, Q);
14 return H;

the set of component IDs CC that containing all the keywords, we select the smallest component as dense truss G_{den}. Then we extract the minimal dense truss H containing Q from G_{den} by function FindMinDenseTruss, which will be introduced in details later.

Theorem 4.2. *Top-down search based on KT-Index in Algorithm 4 needs $O(log\sqrt{|E|} \times nc_{max})$ time to find the dense truss G_{den} for a keyword query, where nc_{max} is the maximum number of components among all the layers in KT-Index.*

Proof. As we analyzed above, the number of iterations based on binary search (lines 2-10) is bounded by $O(logk_{max})$. In each iteration, we only need to compute the component ID which contains all the keywords, this can be done in $O(p_{max})$ time, where p_{max} is the maximum number of components in each layer. Since the maximum truss ness in graph G is no larger than $\sqrt{|E|}$ [27]. The overall complexity to find the dense truss G_{den} containing Q is $O(log\sqrt{|E|} \times p_{max})$. □

Note that in practice, the number of connected components in each layer is far smaller than the number of nodes, which is usually at most hundreds for real-world graphs. Thus our top-down search algorithm based on KT-Index can identify the dense truss G_{den} efficiently.

Extension to Top-r Search. As stated before, some applications may prefer to find top-r dense subgraphs ranked based on their trussness. From the definition of k-truss, we know that the subgraph contained in a component of k-truss is also contained in a component of $(k-1)$-truss. Thus, to avoid repeatedly returning dense truss containing the same set of keyword nodes, we require that the minimal dense trusses identified are not overlapped. Therefore, we marked a node if it has already been identified. Then, the Top-Down-KT Search in Algorithm 4 can be easily revised to return the top-r minimal dense trusses. Instead of only returning one component ID with the smallest size in line 11 of Algorithm 4, we return all the IDs in CC. If $|CC| \ge r$, we will extract the minimal truss from the r smallest components in CC and return them as the answer. If $|CC| < r$, we will extract the minimal truss from all the components in CC and add them to the answer list, update r by $r - |CC|$, and then check the $(k-1)$-th layer repeatedly until will found r minimal dense trusses.

Algorithm 5: FindMinDenseTruss

Input : A dense truss G_{den}, and a keyword query Q.
Output: A minimal dense truss.

1 $k \leftarrow \tau(G_{den})$; $S_{vis} \leftarrow \emptyset$;
2 **while** $V(G_{den}) \setminus S_{vis} \neq \emptyset$ **do**
3 $\quad$ select a node v from $V(G_{den}) \setminus S_{vis}$;
4 $\quad H \leftarrow$ FindkTruss(G_{den}, Q, k, v);
5 $\quad$ **if** $H \neq empty$ **then**
6 $\quad\quad G_{den} \leftarrow H$;
7 $\quad S_{vis} \leftarrow S_{vis} \cup \{v\}$;
8 return G_{den};
9 **Procedure** FindkTruss(G, Q, k, S)
10 $E_{del} \leftarrow \emptyset$;
11 **for** $v \in S$ **do**
12 $\quad$ **for** $(u,v) \in E$ **do**
13 $\quad\quad E_{del} \leftarrow E_{del} \cup \{(u,v)\}$;
14 **for** $(u,v) \in E_{del}$ **do**
15 $\quad E_{del} \leftarrow E_{del} \setminus \{(u,v)\}$;
16 $\quad$ Remove (u,v) from G;
17 $\quad$ **for** $w \in N(v) \cap N(u)$ **do**
18 $\quad\quad sup(v,w) \leftarrow sup(v,w) - 1$;
19 $\quad\quad sup(u,w) \leftarrow sup(u,w) - 1$;
20 $\quad\quad$ **if** $sup(v,w) < k-2 \wedge (v,w) \notin E_{del}$ **then**
21 $\quad\quad\quad E_{del} \leftarrow E_{del} \cup \{(v,w)\}$;
22 $\quad\quad$ **if** $sup(u,w) < k-2 \wedge (u,w) \notin E_{del}$ **then**
23 $\quad\quad\quad E_{del} \leftarrow E_{del} \cup \{(u,w)\}$;
24 remove isolated vertices from G;
25 **if** $\exists$ *connected component* $H \subseteq G$ *containing* Q **then**
26 $\quad$ return H;
27 return $\emptyset$;

4.3 Minimal Dense Truss Extraction

Now, we move to the subtask of extracting minimal dense truss from G_{den}. Before getting into the details of function FindMinDenseTruss(G_{den}, Q) in Algorithms 2 and 4, we first discuss the anti-monotonic property of k-truss, to provide essential guidelines for refinement.

Property 4.1. Given a connected k-truss H, a node $v \in V(H)$ and set of its adjacent edges $E_v = \{(u,v) \in E(H)\}$, if graph $G_{\neg v} = (V(H) \setminus \{v\}, E(H) \setminus E_v)$ does not contain a connected k-truss, there does not exist a subgraph $H' \subseteq$

H such that $G'_{\neg v} = (V(H') \setminus \{v\}, E(H') \setminus E_v)$ contains a connected k-truss.

Proof. We prove this by contradiction. Assume that $\exists H' \subseteq H$ such that $G'_{\neg v} = (V(H') \setminus \{v\}, E(H') \setminus E_v)$ contains a connected k-truss H^*. Clearly, $H^* \subseteq G'_{\neg v}$. Since $V(H') \setminus \{v\} \subseteq V(H) \setminus \{v\}$ and $E(H') \setminus E_v \subseteq E(H) \setminus E_v$, we have $G'_{\neg v} \subseteq G_{\neg v}$. Thus we have $H^* \subseteq G_{\neg v}$ which contradicts with the assumption that $G_{\neg v}$ does not contain a connected k-truss. □

This property shows that when we refine G_{den} by deleting nodes, each node v in G_{den} only needs to be checked once. If deleting v will result in a subgraph that contains no connected k-truss containing Q, we will keep v and will not check it again in the following deletions.

Algorithm FindMinDenseTruss. Based on above property, we give the process of FindMinDenseTruss in Algorithm 5. The main idea is every time we randomly pick one node v in graph G_{den} to check whether deleting node v and its adjacent edges will still lead to a connected k-truss G' containing Q (lines 3-4). If yes, we update G_{den} by G' (lines 5-6); otherwise, we check the next node. We use the set S_{vis} to keep the set of nodes that have been checked to avoid repeated examinations. Function FindkTruss(G_{den}, Q, k, S) is used to check the existence of a connected k-truss containing Q after deleting node subset S and their adjacent edges from G_{den}. First, we use E_{del} to maintain the set of edges to be deleted from the graph. Then we gradually delete each edge $(u, v) \in E_{del}$ and check whether it will result in new edges that violate the edge support constraint for a k-truss (lines 17-23). If yes, we will continually add these edges into E_{del}. Such process stops when $E_{del} = \emptyset$. Then we remove isolated nodes from G (line 24). If there is a connected component G' containing Q, we will return G' as a k-truss; otherwise, we return $\emptyset$.

Theorem 4.3. *The time complexity of Algorithm 5 is $O(t \times (\alpha - k) \times |E(G_{den})|)$ where $t \leq |V(H)|$ is the number of iterations, k is the trussness of G_{den}, and $\alpha \leq \sqrt{|E(G_{den})|}$) is the arboricity of G_{den} (minimum number of spanning forests needed to cover all the edges in G_{den}).*

Proof. First we analyze the complexity of FindkTruss. It is mainly determined by how many times lines 18-23 are executed. The worst case is that if v cannot be delete, then the support of all the edges in G_{den} will be updated to $k-3$ and deleted. Thus the times of lines 18-23 being executed is bounded by $O(\sum_{e \in E(G_{den})}(sup(e) - (k-3)))$. Since $\sum_{e \in E(G_{den})} = 3 \times N_{tri}$ where N_{tri} is the number of triangles bounded $O(\alpha \times |E(G_{den})|)$ (α is the arboricity of G_{den}), we have $O(t \times (\alpha - k) \times |E(G_{den})|)$ where t is the number of iterations of lines 18-23. □

We say a node is deletable if deleting it still leads to a connected k-truss containing Q; otherwise, we say it is un-deletable. From the above theorem, we can see that the time complexity of FindMinDenseTruss mainly depends on the number of iterations t and the size of G_{den} in each iteration as $\alpha - k$ is a constant for a given graph. To reduce the complexity, we need to reduce the number of iterations and delete the deletable nodes as early as possible to reduce the size of the dense truss quickly. In the following, we will introduce some optimization strategies to accelerate such process.

TABLE 3: Keyword queries for DBLP

KWF	Keywords
.0003	parallelism, preprocessing, greedy, hadoop
.0006	benchmark, crowdsourcing, tolerance, spatially
.0009	topological, answer, multiprocessor, datadriven, evolve, stable
.0012	encryption, prototype, configuration, asynchronous
.0015	frame, attribute, iterative, label

Optimization I: Batch Based Deletion. To reduce the number of iterations, one possible way is to delete the nodes in batch instead of one by one. In fact, if there is a subset $S \subseteq V(G_{den})$ which can be deleted from G_{den} and the remaining part also contains a connected k-truss covering Q, we can delete all the nodes in S immediately. However, always deleting a large number of nodes each time might result in no k-truss containing Q. Therefore, we increase the deletion size step by step, which means after one successful deletion, we will increase the number of nodes to be deleted. When it meets an unsuccessful removal, we reset the size to 1. By such setting, we can always delete the nodes one by one in the last steps of deletion and make sure that the minimal dense truss is returned.

Optimization II: Eearly-stop Based Deletion. In addition to above optimization strategy, we can also accelerate the computation by only find a approximate result of minimal dense truss, i.e., we stop the search if the number of consecutive unsuccessful deletions exceeds a given threshold. In fact, with random deletion, we can output an c-approximation of a minimal dense truss with probability at least $1 - \delta$, if the threshold is set to be $\log \frac{1}{\delta} / \log c$. We omit the proof as it is similar to the proof in [25] of finding minimal Steiner maximum-connected subgraph for a set of query nodes.

Optimization III: Local Exploration. In some cases, G_{den} can be very large, which may need a large number of deletions to obtain the minimal dense truss. Thus, we can extract a small subgraph G'_{den} from G_{den} containing the keywords by local exploration. Specifically, we will first construct a Steiner tree T in G_{den} to connect nodes containing all the keywords. Then we expand T to a subgraph G_T in a BFS manner. At the beginning, G_T is initialized as T. We iteratively add the adjacent vertices with the largest trussness until $|V(G_T)|$ exceeds a threshold t. Then we add all the adjacent edges for each node in G_T and check whether G_T contains a k-truss G'_{den} that covers all the keywords. If yes, we consider G'_{den} as the new densest subgraph and apply above refinement process to G'_{den}; otherwise, we expand G_T until $|V(G_T)|$ exceeds $2t$. We repeat above process until a k-truss G'_{den} can be extracted.

TABLE 4: Keywords queries for DBpedia and YAGO

Query	Keywords
Q_1	*jaguar* place
Q_2	*united states* politician award
Q_3	album music genre *american music awards*
Q_4	fish bird mammal protected area *north american*
Q_5	player club manager league city country
Q_6	actor film award company *hollywood*

TABLE 5: Parameters

Parameter	Range	Default
KWF	.0003, .0006, .0009, .0012, .0015	.0009
l	2, 3, 4, 5, 6	4
k	1, 5, 10, 15, 20	10

5 PERFORMANCE STUDIES

In this section, we will first introduce the setup of the experiments and then discuss the experimental results.

5.1 Experimental Setup

To our best knowledge, there is no existing work on cohesive subgraph search based on k-truss for keyword queries. Thus, we implemented the following versions of our methods to thoroughly evaluate the efficiency and effectiveness of KT-Index, search algorithms, and optimization strategies.

1) basic. Basic Top-Down Framework in Alg. 1 with FindMinDenseTruss implemented as Alg. 5.
2) KT. Top-Down-KT Search in Alg. 4 with FindMinDenseTruss implemented as Alg. 5.
3) KTb. Alg. 4 + Alg. 5 + batch based deletion.
4) KTs. Alg. 4 + Alg. 5 + early-stop based deletion.
5) KTl. Alg. 4 + Alg. 5 + local exploration.
6) KTbs. Alg. 4 + Alg. 5 + batch based deletion + early-stop based deletion.
7) KTbsl. Alg. 4 + Alg. 5 + batch based deletion + early-stop based deletion + local exploration.

All the algorithms were implemented in C++, and all the experiments were conducted on a Linux server with Intel Xeon CPU 2.60GHz and 128GB memory.

Datasets. We evaluate the performance of the algorithms on three real-world datasets that are widely used in previous works on keyword search and attributed community search [6] [13]. (1) DBLP[1], a bibliographic dataset with 2 million nodes and 9.9 million edges. In the dataset, a node denotes an author and an edge denotes the co-authorship between these two authors. (2) DBpedia[2], a knowledge graph including 5.90 million nodes and 17.6 million edges. Each node represents an entity with a type (*e.g.*,'animal', 'architectures', 'famous places') from in total 272 types, with a set of attributes (*e.g.*, 'jaguar', 'Ford'). (3)YAGO[3] is also a knowledge graph with 2.64 million nodes and 5.23 million edges, but it is much sparser than DBLP and DBpedia. Note that we did not test another widely used dataset IMDB since it is a tripartite graph where the maximum truss value of subgraphs is at most 3.

1. http://dblp.uni-trier.de/xml/
2. http://dbpedia.com/
3. https://www.mpi-inf.mpg.de/yago

TABLE 6: Construction time and size of index

Graph	Graph size (MB)	Index size (MB)	Construction time (Sec)
DBLP	133.0	192	348.9
DBpedia	280.2	386	593.7
YAGO	76.4	131	394.9

Keyword Queries. (1) For DBLP, the sets of keyword queries used in the evaluation are the same as those in [6], which is shown in Table 3, with their associated keyword frequency KWF. (2) For DBpedia and YAGO, we use 6 query templates consisting of *type* keywords and *value* keywords designed in [13], as shown in Table 4. Since each *value* keyword only associates with one node representing an entity, to generalize the query, we modify the query templates by replacing the *value* keyword (*e.g.*, *american music awards*) with one of its corresponding *type* that contains the most number of entities (*e.g.*, TelevisionShow).

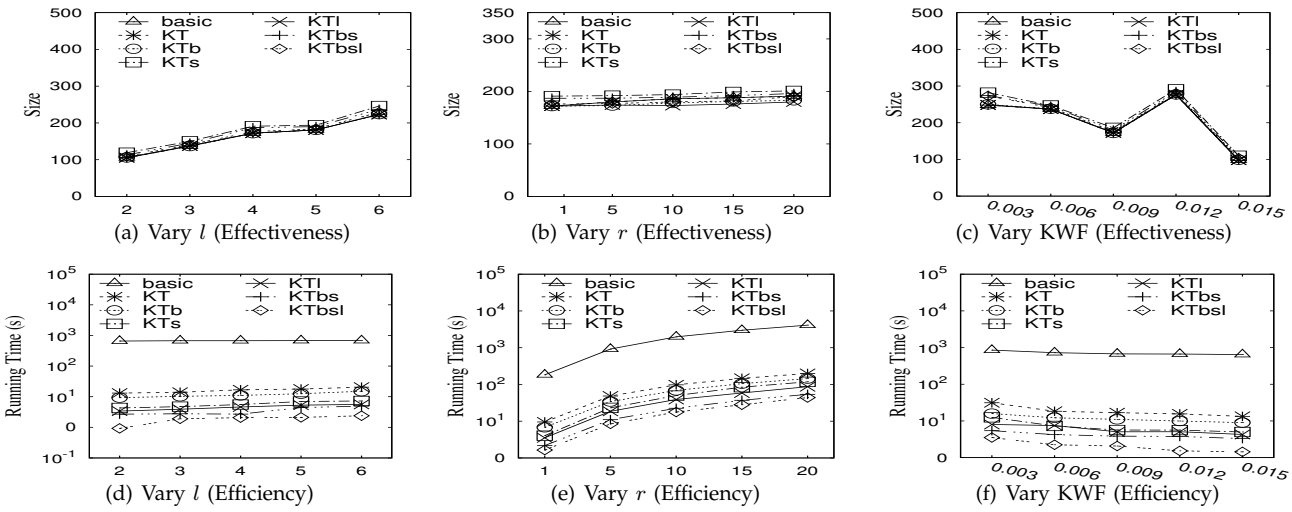
Parameters. We test the performance of all the algorithms by varying three parameters, including the number of keywords l, the r value of top-r search, and the keyword frequency KWF. The ranges and default values of these parameters are shown in Table 5.

5.2 Experiment Results

Exp-1: Index Construction. We start the experiments with the construction of indexes by BuildKTIndex in Alg. 3. This process is typically performed offline before keyword search is carried out. Once the indexes are built, they will reside in main memory to efficiently support keyword search in large graphs. We report the space consumed for the overall index structures in memory, and the time spent for index construction in Table 6. It shows that our KT-Index can be constructed within hundreds of seconds efficiently for all the datasets. The size of KT-Index is no more than 2x of the original graph size, which is consistent with the space complexity $O(|E|)$ in our previous theoretical analysis.

Exp-2: Effectiveness and Efficiency on DBLP. We test the performance of all the algorithms on DBLP by varying parameters l, r, and KWF respectively.

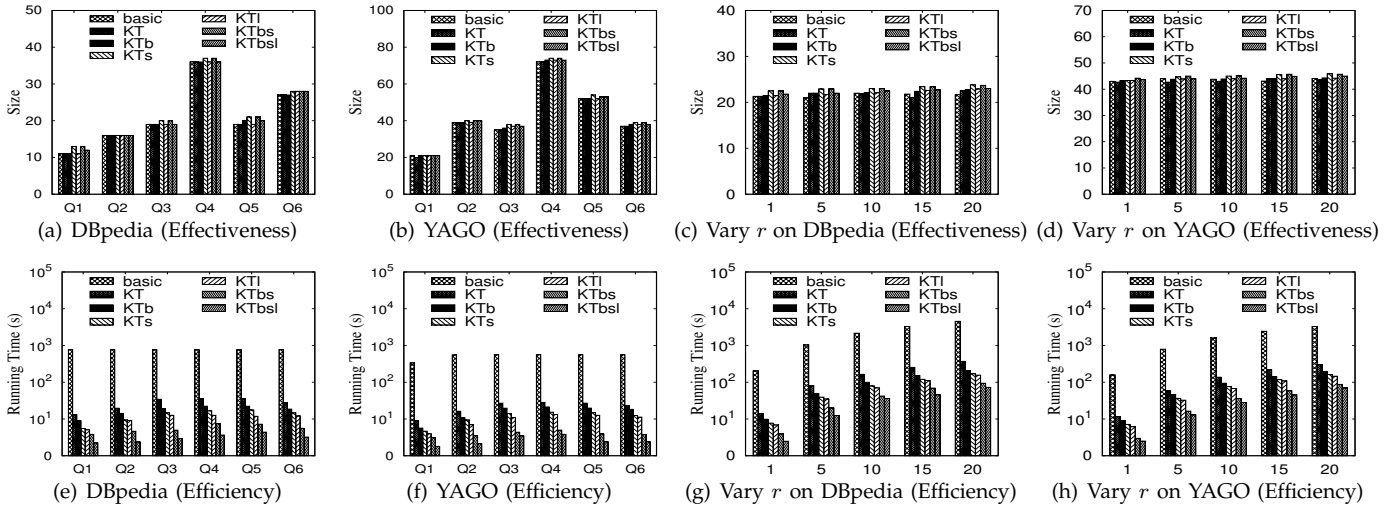
The average size of minimal dense trusses is reported in Fig. 3(a)-3(c). From Fig. 3(a), we can see that the size increases as the number of keywords increases. This is because a larger substructure is returned to include more keywords. The results of all the evaluated algorithms have similar size, where KT achieves the minimum size among all these algorithms. Fig. 3(b) shows that the average size of the top-r results remain stable as r increases. It is because the trussness is stable for a small value of r as there are multiple components contains the keywords at each truss layer. Fig. 3(c) shows the size



(a) Vary l (Effectiveness) (b) Vary r (Effectiveness) (c) Vary KWF (Effectiveness)

(d) Vary l (Efficiency) (e) Vary r (Efficiency) (f) Vary KWF (Efficiency)

Fig. 3: Effectiveness and efficiency on DBLP



(a) DBpedia (Effectiveness) (b) YAGO (Effectiveness) (c) Vary r on DBpedia (Effectiveness) (d) Vary r on YAGO (Effectiveness)

(e) DBpedia (Efficiency) (f) YAGO (Efficiency) (g) Vary r on DBpedia (Efficiency) (h) Vary r on YAGO (Efficiency)

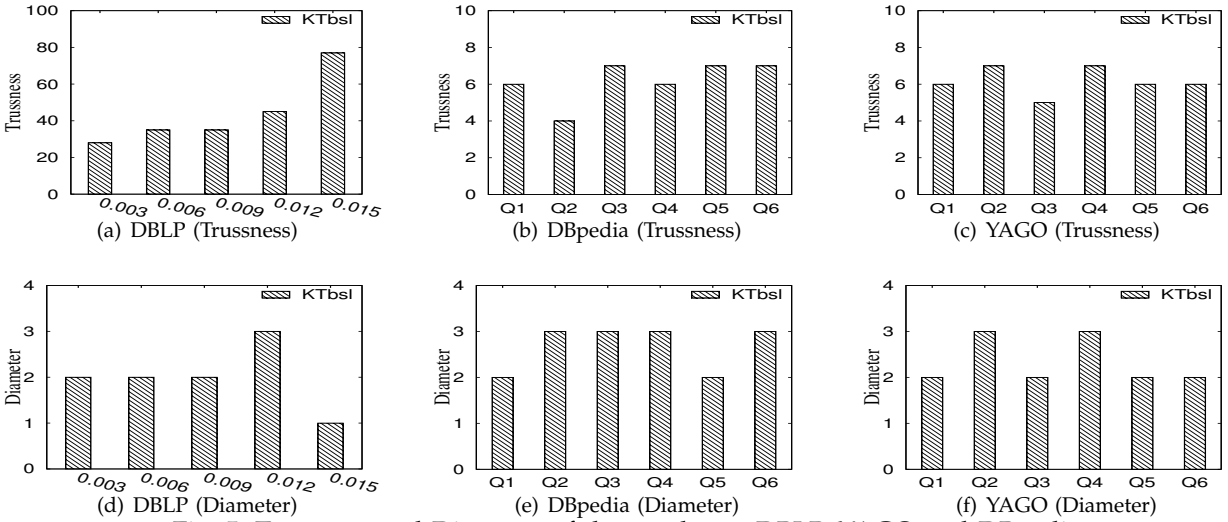
Fig. 4: Effectiveness and efficiency on YAGO and DBpedia

of returned minimal dense trusses for different keyword frequency. When query keywords are not very frequent (e.g., KWF = 0.003), the keyword nodes will distributed sparsely in the graph and we need a large subgraph to include all the keywords. When the keywords frequently occur in the graph (e.g., KWF = 0.015), we may only need a very small subgraph to include all the keywords. However, due to different distributions for each keyword query, it does not always decrease for all the cases.

Fig. 3(d)-3(f) shows the running time of all the algorithms when varying different parameters. Fig. 3(d) shows that as l increases, the running times for all the algorithms increase slowly. This is because the time complexity is mainly determined by sizes of graph G and dense truss G_{den}, and checking more keywords will not cause too much overhead. KT-Index based top-down search algorithm KT is faster than basic without indexes by almost 2 orders of magnitude. Each of the three optimization techniques can also accelerate the computation by 2-5 times. KTbsl based on their combinations is even 10x faster than KT. Fig. 3(e) shows that as r increases, all the methods need more time to return more results. The increasing becomes slow when r is larger, because less time is needed to refine truss with smaller trussness. Fig. 3(f) shows that the running time decreases slowly as the keyword frequency increases, because a smaller truss G_{den} can cover all the keywords which will lead to less refinement time.

Exp-3: Effectiveness and Efficiency on DBPedia and YAGO. We evaluate the performance on DBpedia and YAGO by the queries in Table 4. Note that we cannot vary l and KWF as keyword query sets are fixed.

The average size of retrieved sugraphs is reported in Fig. 4(a)-4(d). Fig. 4(a) shows that basic and KT always obtain the results with smaller size than other algorithms, as they examine every possible deletable word one by one, while other algorithms based on optimization techniques also do not loose too much quality. The size of the returned subgraphs on DBpedia is not as large as that in DBpedia because DBpedia is sparser than DBLP. Similarly, the retrieved subgraphs for DBpedia are also smaller than that of YAGO in Fig.



(a) DBLP (Trussness) (b) DBpedia (Trussness) (c) YAGO (Trussness)

(d) DBLP (Diameter) (e) DBpedia (Diameter) (f) YAGO (Diameter)

Fig. 5: Trussness and Diameter of the results on DBLP, YAGO and DBpedia

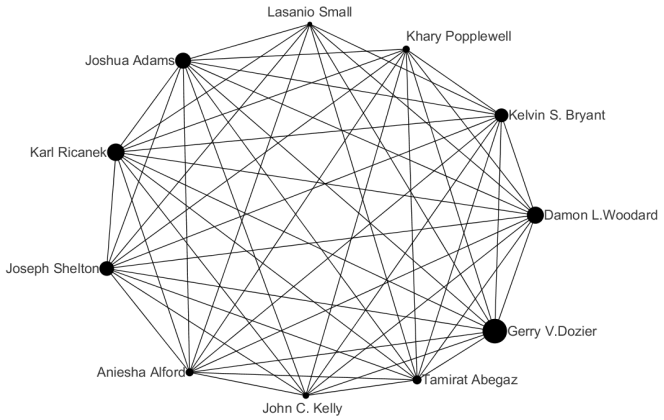


Fig. 6: A case study (Q = {"biometric", "recognition", "face","periocular", "kohonen"})

4(b) because of the same reason. Fig. 4(c)-4(d) show the size of top-r query on DBpedia and YAGO, which also remains stable as r increases, because their trussness will not change too much for a small r as there are multiple components contains the keywords at each k-truss layer.

The running time is reported in Fig. 4(e)-4(h). Fig. 4(e) shows that basic always needs more time than other algorithms, and KT is faster than basic by 1-2 orders of magnitude. KTbsl always needs the least time, which is about 10x faster than KT. Fig. 4(f) has similar trend as that in 4(e), where KTbsl still performs the best and basic performs the worst, but all the algorithms needs less time because YAGO is smaller than DBpedia. Fig. 4(g)-4(h) show the running time on DBpedia and YAGO when we vary top-r, which also increases along with the increase of r. KTbsl achieves the best performance among all the evaluated algorithms.

Exp-4: Statistics of Trussness and Diameters. We also report the trussness and diameter of results for queries given in Tables 3 and 4. We only report most efficient algorithm KTbsl as other algorithms obtain similar results. As shown in Fig. 5, DBLP obtain higher trussness than YAGO and DBPedia because of its high density. Moreover, the trussness increases as query keyword frequency increases, which implies that frequent keywords are more probably to be included in a denser truss. All the diameters of returned results for all the datasets are not larger than 3, which confirms our previous analysis on the tight bound of diameter of the k-truss.

Exp-5: A Case Study on DBLP. We also performed a case study on DBLP to evaluate the effectiveness of our methods. To find cohesive subgraph with stable coauthor relationship, we only consider the edges between authors with at least two coauthored papers. Suppose that an AI company needs to build a team for the task of biometric recognition based on kohonen neural networks, espcially for face and periocular facial recognition. For a possible form of the query Q = {"biometric", "recognition", "face","periocular", "kohonen"}, a minimal dense truss covering Q with trussness 9 is shown in Fig. 6, where larger nodes represent authors with more publications. In this subgraph, Gerry V. Dozier, Karl Ricanek, and Damon L. Woodard are experts (with more than 50 publications) on biometric recognition, face recognition and periocular biometric recognition, respectively. Meanwhile, each two of them has coauthored more than 10 papers. Joshua Adams and Lasanio Small, the students of Gerry V. Dozier, have published papers on face recognition based on kohonen neural networks, and also coauthored several papers with Damon L. Woodard. Other persons are students or colleges of Gerry V. Dozier in the biometric recognition research area, and they also coauthored some papers with other researchers in this subgraph. From above analysis, we can see that it is a densely and stably connected subgraph that can meet the query requirement.

6 Related Work

The related work to our study includes keyword search over graphs and community search based on k-truss.

Keyword Search Over Graphs. Keyword search has been extensively studied in the literature, with query results usually modeled as individual minimal connected trees/graphs containing query keywords [1]. Tree-based

methods popularly use Q-SUBTREE to describe a keyword query answer, where the ranking function is usually defined based on Steiner tree-based semantics or distinct root-based semantics. Since finding optimal Steiner tree (top-1 Q-SUBTREE under the Steiner tree-based semantics) is NP-complete, a heuristic algorithm BANKs [2] was proposed to find approximate solution based on backward search. To find optimal Steiner trees, a parameterized DP algorithm DPBF [34] was proposed where the parameter is determined by the number of Steiner trees. Then, an algorithm producing Steiner trees with polynomial delay was developed [35]. Another method STAR [36] can achieve an $O(log(n))$-approximation of the optimal Steiner tree in pseudo-polynomial run time. Recently, an improved DP algorithm PrunedDP was proposed [37], based on optimal-tree decomposition and conditional tree merging. Another two approaches BANKS-II [3] and BLINKS [4] were also proposed to find Q-SUBTREE under distinct root semantics, where the tree weight is the sum of the shortest distance from the root to each keyword node. BANKS-II is a forward search method that starts from the promising root nodes, and BLINKS is an improved algorithm based on a bi-level index through partitioning graph.

To overcome the drawback that each connected tree only gives a portion of the relationships between query keywords, connected subgraphs such as r-radius subgraph [5], community [6], and r-clique [7] were proposed subsequently. EASE is proposed [5] to find subgraph containing query keywords with radius no larger than r, based on a ranking function of both structural compactness and textual relevancy. [6] presents a polynomial delay algorithm to generate ranked communities which is a multi-centered subgraph such that the distance between center node and each keyword node is no larger than a threshold. [7] prosed a polynomial delay algorithm to approximately find r-cliques with distances between keyword nodes no larger than r. In addition, some other works [11] [12] [13] studied diversified keyword search based on connected trees/subgraphs. Besides, keyword search can also be considered as a special case of partial topology query [8] [9] where label propagation are utilized to find the top-k matched components under a certain ranking score.

However, above methods only focus on evaluating the distance between (querying) nodes to ensure the compactness of query results, but none of them consider density. Recently, two approaches consider the density for keyword search, where one is to maximum the contextual density that combine the contextual cohesiveness and structural cohesiveness [15], and the other is to maximum the contextual cohesiveness of the k-core for a specific k [14], which are inherently different from the problem studied in this work.

Community Search based on k-truss. Community search based on k-truss model aims to find communities that maximize the truss value and contain a given set of query nodes. [21] constructs TCP-Index to support efficient search of all the k-truss communities containing a given query node. [23] proposed a more compact index EquiTruss to accelerate the computation of k-truss communities for a query node. To avoid the free rider effect, [22] studied the problem of finding truss with maximum truss value and minimum diameter, and provided an approximate solution. Two recent works [20] [30] studied attributed community search for given nodes and attributes, and proposed different ranking functions regarding the attributes. All these algorithms require a given set of query nodes, which cannot be directly adopted in this paper since the subset of nodes containing keywords to be included in the dense truss is previously unknown.

Besides community search, there are also some related works on query independent community detection, which aim to detect maximal k-truss for each k. More information can be referred to [27] [28] [29].

7 Conclusion

In this paper, we study the problem of finding cohesive subgraph that are highly dense and compact for given set of keywords. We model the cohesive subgraph based on k-truss model, and formulate the problem of finding cohesive subgraph as *minimal dense truss* search problem for keyword queries. We tackle the minimal dense truss search problem for keyword queries by dividing it into two subtasks. One is finding the dense truss that maximize the truss value containing keywords, and the other is refining the dense truss to obtain a minimal dense truss containing keywords. To deal with large networks efficiently, we design a novel hybrid graph indexing scheme KT-Index to keep the keyword information and truss information in a compact manner efficiently, and propose an efficient algorithm which carries search on KT-Index directly to find the dense truss without repeated accesses to original graph. To extract minimal dense truss, we also develop a novel refinement approach by checking each node at most once based on the anti-monotonicity property of k-truss, and further optimize the refinement by batch based deletion, early-stop based deletion, and local exploration. Extensive experimental studies on real-world graphs show the effectiveness and efficiency of our approaches.

References

[1] J. X. Yu, L. Qin, and L. Chang, "Keyword search in relational databases: A survey," *IEEE Data Eng. Bull.*, vol. 33, no. 1, pp. 67–78, 2010.

[2] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, "Keyword searching and browsing in databases using BANKS," in *ICDE*, 2002, pp. 431–440.

[3] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar, "Bidirectional expansion for keyword search on graph databases," in *VLDB*, 2005, pp. 505–516.

[4] H. He, H. Wang, J. Yang, and P. S. Yu, "BLINKS: ranked keyword searches on graphs," in *SIGMOD*, 2007, pp. 305–316.

[5] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou, "EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data," in *SIGMOD*, 2008, pp. 903–914.
[6] L. Qin, J. X. Yu, L. Chang, and Y. Tao, "Querying communities in relational databases," in *ICDE*, 2009, pp. 724–735.
[7] M. Kargar and A. An, "Keyword search in graphs: Finding r-cliques," *PVLDB*, vol. 4, no. 10, pp. 681–692, 2011.
[8] M. Xie, S. S. Bhowmick, G. Cong, and Q. Wang, "PANDA: toward partial topology-based search on large networks in a single machine," *VLDB J.*, vol. 26, no. 2, pp. 203–228, 2017. [Online]. Available: https://doi.org/10.1007/s00778-016-0447-0
[9] S. Yang, Y. Wu, H. Sun, and X. Yan, "Schemaless and structureless graph querying," *PVLDB*, vol. 7, no. 7, pp. 565–576, 2014. [Online]. Available: http://www.vldb.org/pvldb/vol7/p565-yang.pdf
[10] T. Lappas, K. Liu, and E. Terzi, "Finding a team of experts in social networks," in *SIGKDD*, 2009, pp. 467–476.
[11] F. Zhao, X. Zhang, A. K. H. Tung, and G. Chen, "BROAD: diversified keyword search in databases," *PVLDB*, vol. 4, no. 12, pp. 1355–1358, 2011.
[12] M. Kargar, A. An, and X. Yu, "Efficient duplication free and minimal keyword search in graphs," *TKDE*, vol. 26, no. 7, pp. 1657–1669, 2014.
[13] Y. Wu, S. Yang, M. Srivatsa, A. Iyengar, and X. Yan, "Summarizing answer graphs induced by keyword queries," *PVLDB*, vol. 6, no. 14, pp. 1774–1785, 2013.
[14] Z. Zhang, X. Huang, J. Xu, B. Choi, and Z. Shang, "Keyword-centric community search," in *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, 2019, pp. 422–433.
[15] L. Chen, C. Liu, K. Liao, J. Li, and R. Zhou, "Contextual community search over large social networks," in *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, 2019, pp. 88–99.
[16] W. Cui, Y. Xiao, H. Wang, and W. Wang, "Local search of communities in large graphs," in *SIGMOD*, 2014, pp. 991–1002.
[17] M. Sozio and A. Gionis, "The community-search problem and how to plan a successful cocktail party," in *SIGKDD*, 2010, pp. 939–948.
[18] R. Li, L. Qin, J. X. Yu, and R. Mao, "Influential community search in large networks," *PVLDB*, vol. 8, no. 5, pp. 509–520, 2015.
[19] N. Barbieri, F. Bonchi, E. Galimberti, and F. Gullo, "Efficient and effective community search," *DMKD*, vol. 29, no. 5, pp. 1406–1433, 2015.
[20] Y. Fang, R. Cheng, S. Luo, and J. Hu, "Effective community search for large attributed graphs," *PVLDB*, vol. 9, no. 12, pp. 1233–1244, 2016.
[21] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, "Querying k-truss community in large and dynamic graphs," in *SIGMOD*, 2014, pp. 1311–1322.
[22] X. Huang, L. V. S. Lakshmanan, J. X. Yu, and H. Cheng, "Approximate closest community search in networks," *PVLDB*, vol. 9, no. 4, pp. 276–287, 2015.
[23] E. Akbas and P. Zhao, "Truss-based community search: a truss-equivalence based indexing approach," *PVLDB*, vol. 10, no. 11, pp. 1298–1309, 2017.
[24] L. Chang, X. Lin, L. Qin, J. X. Yu, and W. Zhang, "Index-based optimal algorithms for computing steiner components with maximum connectivity," in *SIGMOD*, 2015, pp. 459–474.
[25] J. Hu, X. Wu, R. Cheng, S. Luo, and Y. Fang, "Querying minimal steiner maximum-connected subgraphs in large graphs," in *CIKM*, 2016, pp. 1241–1250.
[26] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," *National Security Agency Technical Report*, vol. 16, 2008.
[27] J. Wang and J. Cheng, "Truss decomposition in massive networks," *PVLDB*, vol. 5, no. 9, pp. 812–823, 2012.
[28] Y. Zhang and S. Parthasarathy, "Extracting analyzing and visualizing triangle k-core motifs within networks," in *ICDE*, 2012, pp. 1049–1060.
[29] A. E. Sariyüce and A. Pinar, "Fast hierarchy construction for dense subgraphs," *PVLDB*, vol. 10, no. 3, pp. 97–108, 2016.
[30] X. Huang and L. V. S. Lakshmanan, "Attribute-driven community search," *PVLDB*, vol. 10, no. 9, pp. 949–960, 2017.
[31] Y. Zhu, Q. Zhang, L. Qin, L. Chang, and J. X. Yu, "Querying cohesive subgraphs by keywords," in *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, 2018, pp. 1324–1327.
[32] K. Mehlhorn, "A faster approximation algorithm for the steiner problem in graphs," *Inf. Process. Lett.*, vol. 27, no. 3, pp. 125–128, 1988.
[33] N. Chiba and T. Nishizeki, "Arboricity and subgraph listing algorithms," *SIAM J. Comput.*, vol. 14, no. 1, pp. 210–223, 1985.
[34] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin, "Finding top-k min-cost connected trees in databases," in *ICDE*, 2007, pp. 836–845.
[35] K. Golenberg, B. Kimelfeld, and Y. Sagiv, "Keyword proximity search in complex data graphs," in *SIGMOD*, 2008, pp. 927–940.
[36] G. Kasneci, M. Ramanath, M. Sozio, F. M. Suchanek, and G. Weikum, "STAR: steiner-tree approximation in relationship graphs," in *ICDE*, 2009, pp. 868–879.
[37] R. Li, L. Qin, J. X. Yu, and R. Mao, "Efficient and progressive group steiner tree search," in *SIGMOD*, 2016, pp. 91–106.

Yuanyuan Zhu received the BS and MS degrees in computer science from Harbin Institute of Technology (HIT) in 2007 and 2009, respectively, and PhD degree in computer science from the Chinese University of Hong Kong in 2013. She is an associate professor at the State Key Laboratory of Software Engineering, Computer School, Wuhan University, China. Her research interests include graph database querying, graph mining, and big graph processing.

Qian Zhang received her bachelor degree in computer science from SiChuan University in 2016, and the master degree from Wuhan University in 2019. Her research interests include graph database and graph mining, espcially keyword search and community search in large graphs.t

Lu Qin received his bachelor degree from department of Computer Science and Technology in Renmin University of China in 2006, and PhD degree from department of Systems Engineering and Engineering Management in the Chinese University of Hong Kong in 2010. He is now a postdoc research fellow in the Centre of Quantum Computation and Intelligent Systems (QCIS) in University of Technology, Sydney (UTS). His research interests include parallel big graph processing, I/O efficient algorithms on massive graphs, and keyword search in relational database.

Lijun Chang received the BEng degree in computer science and technology from the Renmin University of China, in 2007, and PhD degree in systems engineering and engineering management from the Chinese University of Hong Kong, in 2011. He is currently a lecturer with the School of Information Technologies, University of Sydney. Prior to joining the University of Sydney, he was an ARC DECRA (Discovery Early Career Researcher Award) fellow with the University of New South Wales. His research interests include the fields of big graph (network) analytics, with a focus on devising practical algorithms and theoretical foundations for massive graph analysis.

Jeffery Xu Yu received the B.E., M.E., and Ph.D. degrees in computer science from the University Tsukuba, Japan, in 1985, 1987, and 1990, respectively. He has held teaching positions at the Institute of Information Sciences and Electronics, University of Tsukuba, and at the Department of Computer Science, Australian National University, Australia. Currently, he is a Professor in the Department of Systems Engineering and Engineering Management, the Chinese University of Hong Kong, Hong Kong. His current research interests include graph database, graph mining, keyword search in relational databases, and social network analysis.

2018 IEEE 34th International Conference on Data Engineering

Querying Cohesive Subgraphs by Keywords

Yuanyuan Zhu*, Qian Zhang*, Lu Qin†, Lijun Chang‡, Jeffrey Xu Yu§

*Computer School, Wuhan University, China
†Centre for Quantum Computation and Intelligent Systems, University of Technology Sydney
‡School of Information Technologies, The University of Sydney
§The Chinese University of Hong Kong, Hong Kong, China
*yyzhu@whu.edu.cn, †lu.qin@uts.edu.au, ‡lijun.chang@sydney.edu.au, §yu@se.cuhk.edu.hk

***Abstract*—Keyword search problem has been widely studied to retrieve related substructures from graphs for a keyword set. However, existing well-studied approaches aim at finding compact trees/subgraphs containing the keywords, and ignore a critical measure, density, to reflect how strongly and stablely the keyword nodes are connected in the substructure. In this paper, we study the problem of finding a cohesive subgraph containing the query keywords based on the k-truss model, and formulate it as *minimal dense truss* search problem, i.e., finding minimal subgraph with maximum trussness covering the keywords. We first propose an efficient algorithm to find the dense truss with the maximum trussness containing keywords based on a novel hybrid KT-Index (Keyword-Truss Index). Then, we develop a novel refinement approach to extract the minimal dense truss based on the anti-monotonicity property of k-truss. Experimental studies on real datasets show the outperformance of our method.**

I. Introduction

Keyword search, as a user-friendly query scheme, has been widely used to retrieve useful information in various graph data, such as knowledge graphs, information networks, social networks, etc. Given a query consisting of a number of keywords, the target of keyword search over a graph is to find substructures in the graph related to the query keywords.

In recent decades, keyword search problem has been extensively studied [1], aiming to find minimal connected trees (Steiner tree [2] and distinct root tree [3]) or subgraphs (r-radius subgraph [4], community [5], and r-clique [6]) containing the keywords. Besides, keyword search can also be considered as a special case of partial topology query [7] [8] where label propagation are utilized to find matched components. However, these methods only focus on the compactness of retrieved substructure, and fail to explore how densely these keywords are connected, which is critical to reflect the stability of the relationships between keywords in many applications, e.g., forming a team such that team members are stably close with each other so that the whole team can cooperate well.

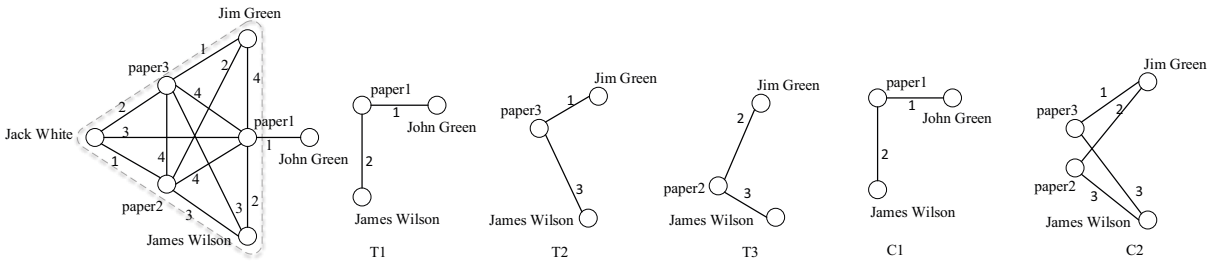
In this paper, for the first time, we study the problem of finding cohesive subgraphs that are highly dense and compact for keyword queries based on k-truss model in which each edge is contained in at least $(k-2)$ triangles. We illustrate the differences between k-truss and existing keyword search approaches by the following example.

Fig. 1(a) shows a co-authorship and citation graph G, where the weight between an author and a paper is the author rank, and the weight between two papers is the citation frequency. For a query $Q = \{James, Green\}$, the top-3 connected trees with weight 3, 4, and 5 respectively are identified by [2] [3] as shown in Fig. 1(b). Fig. 1(c) shows the communities identified by [5], which are multi-centered subgraphs with the distance between a center node and each keyword node no larger than a given threshold (e.g., 3). They are ranked based on the minimum total edge weight from a center node to each keyword node on the corresponding shortest path. The score of community C_1 with center node paper1 is $1+2=3$. The score of community C_2 is 4 as it has two center nodes paper2 and paper3 with total weights $2+3=5$ and $1+3=4$, respectively. In the r-clique model with diameter no larger than r (e.g., $r=3$) [6], T_1 and T_2 are returned, since only Steiner trees of qualified r-cliques are finally extracted. All these approaches output the substructure containing James Wilson and John Green as the top-1 answer. However, James Wilson and Jim Green coauthored more papers together with Jack White, which implies a more stable and closer relationships. Based on the truss model, they can be properly discovered in the form of 4-truss (the dashed line area in Fig. 1(a)).

To attain highly dense and compact substructure for a keyword query Q, a natural way is to find the subgraph with maximum trussness and minimum size containing Q. However, such problem is NP-hard and APX-hard, which can be proved through the reduction of maximum clique problem in a similar manner as the proof in [9]. Thus, in this paper we study a relaxed version, called *minimal dense truss search*, i.e., find the subgraph with maximum trussness containing Q such that it does not contain any subgraph with the same trussness containing Q. Note that our model is different with the closest truss model [9] with maximum trussness and minimum diameter, as the diameter of a k-truss with n nodes is bounded by $\lfloor \frac{2n-2}{k} \rfloor$ while a k-truss with minimum diameter may have an arbitrary large number of nodes. Moreover, closest truss search is NP-hard [9], while *minimal dense truss search* can be done in polynomial time.

Despite rich studies on community search, finding minimal dense truss for keyword queries is nontrivial due to its inherent difference from community search. Community search aims to find maximal communities that maximize the truss value and contain a set of query nodes, which can be done by local search with proper indexes in $O(|\mathcal{A}|)$ time ($\mathcal{A}$ is the answer) [10] [11]. The main difficulty of minimal dense truss search for keyword queries is that, unlike community search where the query nodes are given, the subset of nodes containing all the keywords to be included in the dense truss is unknown in

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00141



(a) Graph G (b) Trees (c) Communities

Fig. 1. A Motivating Example

advance, and therefore we do not know from which nodes to start in the local search [10] [11]. One possible solution is that, for a keyword query $Q = \{w_1, w_2, \ldots, w_l\}$, we explore all the combinations of keyword nodes in $\mathcal{S} = V_1 \times V_2 \times \cdots \times V_l$ to find the subgraph with maximum trussness, where V_i is the node set containing w_i. Such search space will be inexhaustible for large real-world graphs due to the huge number of combinations. Another difficulty is verifying the minimality of a truss, which is also costly as we need to check whether it contains any subgraph with the same trussness.

We tackle these difficulties by dividing the minimal dense search problem into two subproblems. For the first subproblem, we propose a top-down framework based on novel hybrid graph indexing scheme KT-index to find the dense truss G_{den} efficiently. For the second subproblem, we develop a novel approach to extract minimal dense truss H covering Q from G_{den} based on the anti-monotonic property of k-truss.

II. Problem Statement

Given a set of labels Σ, a simple undirected vertex labeled graph is represented as $G = (V, E, L)$, where V is the set of vertices, $E \subseteq V \times V$ is the set of edges, and L is a labeling function which assign each node a set of labels $L(v) \subset \Sigma$. We use $V(G)$ and $E(G)$ to denote the set of vertices and the set of edges of graph G respectively. For a vertex $v \in V$, we denote the set of its neighboring vertices by $N(v) = \{u \in V | (u, v) \in E\}$ and its degree by $d(v) = |N(v)|$. A *triangle* $\triangle(u, v, w)$ in G is a substructure such that $(u, v), (v, w), (u, w) \in E$.

Definition 2.1 (Edge Support). The support of an edge $e = (u, v)$ in graph G is the number of triangles in which e occurs, defined as $sup_G(e) = |\{\triangle(u, v, w) | w \in V(G)\}|$.

Definition 2.2 (Connected k-Truss). Given a graph G and an integer k, a connected k-truss is a connected subgraph $H \subseteq G$, such that $\forall e \in E(H)$, $sup_H(e) \geq k - 2$.

The trussness of a subgraph $H \subseteq G$ is the minimum support of all edges in H plus 2, defined as $\tau(H) = \min_{e \in E(H)} sup_H(e) + 2$. The trussness of an edge $e \in E(G)$ is the maximum trussness of subgraphs containing e, i.e., $\tau(e) = \max_{H \subseteq G \wedge e \in E(H)} \tau(H)$. The trussness of a vertex $v \in V(G)$ equals to the maximum trussness of its adjacent edges, i.e., $\tau(v) = \max_{u \in N(v)} \tau(u, v)$.

For example, in Fig. 2, the edge support of (v_2, v_3) is 3 as it is contained in 3 triangles $\triangle(v_1, v_2, v_3)$, $\triangle(v_2, v_3, v_4)$ and $\triangle(v_2, v_3, v_5)$. Let H_1 denote the subgraph induced by vertices $\{v_1, v_2, v_3, v_4\}$. $\tau(H_1) = 4$ since the minimum support of edges in H_1 is 2. The trussness of edge (v_2, v_3) is 4 because there is no other subgraph with higher trussness containing (v_2, v_3). $\tau(v_2) = 4$ because the maximum trussness of its adjacent edges (v_2, v_1), (v_2, v_3), (v_2, v_4) and (v_2, v_5) is 4.

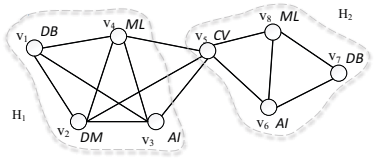


Fig. 2. An Example Graph G

Definition 2.3 (Dense Truss Over Keywords). Given a graph G and a keyword set Q, a dense truss over Q is a connected truss $G_{den} \subseteq G$ that maximizes the trussness and contains Q.

Definition 2.4 (Minmal Dense Truss Over Keywords). Given a graph G and a keyword set Q, the minimal dense truss over Q is a dense truss $G_{den} \subseteq G$ containing Q such that any subgraph of G_{den} is not a dense truss containing Q.

For example, consider a query $Q = \{DB, ML\}$. H_1 and H_2 in Fig. 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.

Problem (*Minimal Dense Truss Search by Keywords*). Given a graph G and a keyword set $Q = \{w_1, w_2, \ldots, w_l\}$, find the minimal dense truss containing Q.

For simplicity, we consider the top-1 minimal dense truss search for keywords and our approaches can be extended to top-r version where the rank is based on the trussness.

III. Our Approaches

In this section, we first propose a basic top-down algorithm in Section III-A, then introduce the KT-index and the improved algorithm in Section III-B, and finally introduce the details of the refinement process in Section III-C.

A. Basic Top-down Search Framework

To avoid enumerating all the combinations of keyword nodes in $\mathcal{S} = V_1 \times V_2 \times \cdots \times V_l$, we propose a top-down search framework by starting the search over the truss with the largest trussness k_{max} in graph G. If it does not contain a

connected k_{max}-truss covering Q, we will gradually decrease k_{max} until we find one. Such process can be accelerated by utilizing the property of trussness for keywords as follows. For a keyword w_i, let V_i be the set of nodes containing w_i. The upper bound of the trussness for w_i is defined as the maximum trussness of nodes in V_i, i.e., $\tau'(w_i) = \max_{v \in V_i} \tau(v)$.

Property 3.1. Given a graph G and a keyword set $Q = \{w_1, w_2, \ldots, w_l\}$, for any truss H containing Q, we have $\tau(H) \leq \min_{1 \leq i \leq l} \tau'(w_i)$.

The main steps of top-down search algorithm are as follows. First, we obtain the trussness of all edges and nodes by truss decomposition [12]. Second, for each keyword w_i, we compute the node set V_i, and obtain the upper bound of trussness by $\tau'(w_i) = \max_{v \in V_i} \tau(v)$. Third, based on above property, we start searching from k_{max}-truss where $k_{max} = \min_{1 \leq i \leq l} \tau'(w_i)$. Specifically, we extract $G_{k_{max}} = \{e \in G | \tau(e) \geq k_{max}\}$ from G and check whether each connected component C_i in $G_{k_{max}}$ contains all the keywords. If yes, we return the component containing Q with smallest size; otherwise, we search $(k_{max}-1)$-truss and stop when we find a connected truss G_{den} containing Q. Finally, we refine G_{den} to obtain a minimal dense truss H.

The complexity of truss decomposition is $O(|E|^{1.5})$ [12]. For a specific value of k_{max}, the process of computing the connected components $G_{k_{max}}$ covering the keywords can be done $O(|E(G_{k_{max}})|)$ time. In the worst case, we need to check all the possible values of k_{max} from $\min_{1 \leq i \leq l} \tau'(w_i)$ to 2. Due to the fact that $\tau(v) \leq \sqrt{|E|}$ for any $v \in V$ [12], the overall complexity of finding G_{den} is $O(|E|^{1.5})$.

B. Improved Algorithm on KT-Index

1) Keyword-Truss Index (KT-Index): In the basic top-down search framework, trussness computation for each edge is primitive. Since it is independent with keyword queries, we can complete such computation by truss decomposition [12] offline before any query comes. Then, we build a hash table to keep all the edges and their trussness.

Another time consuming part of basic top-down algorithm is that we need to test many values of k s to find a k-truss containing Q with the largest k, with time complexity $O(\sqrt{|E|} \times |E|)$. To speed up the computation of this part, we design KT-Index including two parts: truss index and keyword index.

Truss Index. Truss index is a multi-layer structure, where we index the information of all the connected k-truss in the k-th layer. Suppose there are n_k connected components $C_1, C_2, \ldots, C_{n_k}$ in the k-th layer. We sort all the components in the descending order of their size (number of nodes) and assign each component an ID. For each component C_i, we only store the node set $V(C_i)$. Thus, we store the k-th layer in the form of list $(1, V(C_1)), \ldots, (i, V(C_i)), \ldots, (n_k, V(C_{n_k}))$.

Keyword Index. In the keyword index, we first store a inverted keyword list to keep the node IDs that contain each keyword, i.e., for each w_i, we store the keyword node set V_i containing w_i. Meanwhile we record the upper bound of trussness $\tau'(w_i)$ for each keyword. Moreover, for each keyword, we record the IDs of the component CID_k it occurs in the k-th layer, in the form of (k, CID_k).

Obviously, the index size of KT-Index is $O(|E|)$ and it can be constructed in $O(|E|^{1.5})$ time.

Algorithm 1: Improved-KT Search

Input : A graph G, and a keyword query Q.
Output: A minimal dense subgraph.

```
1  k_max ← min_{1≤i≤l} τ'(w_i);
2  k_min ← 3;
3  while k_max > k_min do
4      k ← ⌊(k_max + k_min)/2⌋;
5      for each keyword w_i in Q do
6          SC_i ← CID_k of w_i;
7      CC ← ∩_{1≤i≤l} SC_i;
8      if CC ≠ ∅ then
9          k_min ← k + 1;
10     else
11         k_max ← k − 1;
12 id ← min_{cid∈CC} cid;
13 G_den ← extract compoent C_id at the k-th layer from G;
14 H ← FindMinDenseTruss(G_den, Q);
15 return H;
```

2) The Improved Algorithm: The search algorithm is shown in Algorithm 1. To avoid the worst case of checking all the value of k_{max}, we check each layer of truss index by a binary search, which can be completed in $\log(k_{max})$ iterations. In the k-th layer, we obtain the set of component IDs CC that contains all the keywords (lines 5-7). 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 H containing Q from G_{den} by function FindMinDenseTruss, which will be introduced in detail later.

Algorithm 1 needs $O(\log \sqrt{|E|} \times nc_{max})$ time to find G_{den}, where nc_{max} is the maximum number of components among all the layers in KT-Index. Note that the number of connected components in each layer is far smaller than the node number, which is usually at most hundreds for real-world graphs. Thus our improved algorithm can identify G_{den} very efficiently.

C. Minimal Dense Truss Extraction

Now, we move to the subproblem of extracting minimal dense truss from G_{den}. Before going into the details of function FindMinDenseTruss(G_{den}, Q) aforementioned in Algorithm 1, we will first give the anti-monotonic property of k-truss, to provide essential guidelines for refinement.

Property 3.2. Given a connected k-truss H, a node $v \in V(H)$ and set of its adjacent edges $E_v = \{(u, v) \in E(H)\}$, if graph $G_v = (V(H) \setminus \{v\}, E(H) \setminus E_v)$ does not contain a connected k-truss, there does not exists a subgraph $H' \subseteq H$ such that $G'_v = (V(H') \setminus \{v\}, E(H') \setminus E_v)$ contains a connected k-truss.

Algorithm 2: FindMinDenseTruss

Input : A dense truss G_{den}, and a keyword query Q.
Output: A minimal dense subgraph.

```
1 k ← τ(G_den);
2 S_vis ← ∅;
3 while V(G_den) \ S_vis ≠ ∅ do
4     select a node v from V(G_den) \ S_vis;
5     G' ← FindKTruss(G_den, Q, k, v);
6     if G' ≠ empty then
7         G_den ← G';
8     S_vis ← S_vis ∪ {v};
9 return G_den;
```

This property show that when we refine G_{den} by deleting nodes, each node v in G_{den} only needs to be checked once. If deleting v will result in a subgraph that contains no connected k-truss containing Q, we will keep v and will not check it again in the following deletions.

Algorithm FindMinDenseTruss. Based on above property, we give the process of FindMinDenseTruss in Algorithm 2. The main idea is every time we randomly pick one node v in graph G_{den} to check whether deleting node v and its adjacent edges will still lead to a connected k-truss G' containing Q (lines 4-5). If yes, we update G_{den} by G' (lines 6-7); otherwise, we check the next node. We use the set S_{vis} to keep the set of nodes having been checked to avoid repeated examinations.

Function FindKTruss(G_{den}, Q, k, S) is used to check the existence of a connected k-truss containing Q after deleting a set of nodes S and their adjacent edges from G_{den}. First, we use E_{del} to maintain the set of edges to be deleted from the graph. Then we gradually delete each edge $(u, v) \in E_{del}$ and check whether it will result in new edges that violate the edge support constraint for a k-truss. If yes, we will continually add these edges into E_{del}. Such process stops when $E_{del} = \emptyset$. Then we remove isolated nodes from G_{den}. If there is a connected component G' containing Q, we will return G' as a k-truss; otherwise, we return $\emptyset$.

The time complexity of Algorithm 2 is $O(t \times (\alpha - k) \times |E(G_{den})|)$ where $t \leq |V(H)|$ is the number of iterations, k is the trussness of G_{den}, and $\alpha \leq \sqrt{|E(G_{den})|}$ is the arboricity of G_{den} (minimum number of spanning forests needed to cover all the edges in G_{den}).

IV. Performance Studies

We implemented the following two algorithms for comparison: Basic (Basic top-down search framework), and KT (Improved algorithm Alg. 1). Function FindMinDenseTruss in these two algorithms is implemented as Alg. 2. All the algorithms are implemented in C++, and run on a PC with 3.60GHz CPU and 8GB memory.

Datasets and Queries. We use two real datasets popularly used in previous keyword search works [13]: DBpedia[1] and YAGO[2]. We use 6 query templates designed in [13], consisting of type keywords and value keywords. Since each value keyword is associated with one node representing an entity, to generalize the query, we replace the value keywords (e.g., AmericanMusicAwards) with its type (e.g., TelevisionShow). Refer to [13] for details of the datasets and queries.

[1]http://dbpedia.com/
[2]https://www.mpi-inf.mpg.de/yago

TABLE I
Running time comparison (sec)

Dataset	Alg.	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6
DBpedia	Basic	2594.7	1099.8	2845.1	2488.7	2015.4	1963.5
	KT	4.8	0.5	5.9	9.5	7.0	3.5
YAGO	Basic	1559.0	1005.7	2963.5	2845.2	2777.6	1359.8
	KT	1.8	0.3	6.9	2.5	5.5	1.3

Experimental Results. Table I shows the running time for all the queries. We can see that KT can answer the query in a few seconds while Basic needs thousands of seconds, which validates the efficiency of our improved search algorithm.

V. Conclusion

In this paper, we study the minimal dense truss search problem for keyword queries based on the k-truss model. We develop an efficient approach based on hybrid index KT-index and a novel refinment scheme to solve this problem.

Acknowledgment. This work was supported by grants NSFC 61502349, ARC DP160101513, ARC DE150100563, ARC DP160101513, and Research Grants Council of the Hong Kong SAR, China No. 14221716.

References

[1] J. X. Yu, L. Qin, and L. Chang, "Keyword search in relational databases: A survey," *IEEE Data Eng. Bull.*, vol. 33, no. 1, pp. 67–78, 2010.
[2] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, "Keyword searching and browsing in databases using BANKS," in *ICDE*, 2002, pp. 431–440.
[3] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar, "Bidirectional expansion for keyword search on graph databases," in *VLDB*, 2005, pp. 505–516.
[4] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou, "EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data," in *SIGMOD*, 2008, pp. 903–914.
[5] L. Qin, J. X. Yu, L. Chang, and Y. Tao, "Querying communities in relational databases," in *ICDE*, 2009, pp. 724–735.
[6] M. Kargar and A. An, "Keyword search in graphs: Finding r-cliques," *PVLDB*, vol. 4, no. 10, pp. 681–692, 2011.
[7] M. Xie, S. S. Bhowmick, G. Cong, and Q. Wang, "PANDA: toward partial topology-based search on large networks in a single machine," *VLDB J.*, vol. 26, no. 2, pp. 203–228, 2017.
[8] S. Yang, Y. Wu, H. Sun, and X. Yan, "Schemaless and structureless graph querying," *PVLDB*, vol. 7, no. 7, pp. 565–576, 2014.
[9] X. Huang, L. V. S. Lakshmanan, J. X. Yu, and H. Cheng, "Approximate closest community search in networks," *PVLDB*, vol. 9, no. 4, pp. 276–287, 2015.
[10] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, "Querying k-truss community in large and dynamic graphs," in *SIGMOD*, 2014, pp. 1311–1322.
[11] E. Akbas and P. Zhao, "Truss-based community search: a truss-equivalence based indexing approach," *PVLDB*, vol. 10, no. 11, pp. 1298–1309, 2017.
[12] J. Wang and J. Cheng, "Truss decomposition in massive networks," *PVLDB*, vol. 5, no. 9, pp. 812–823, 2012.
[13] Y. Wu, S. Yang, M. Srivatsa, A. Iyengar, and X. Yan, "Summarizing answer graphs induced by keyword queries," *PVLDB*, vol. 6, no. 14, pp. 1774–1785, 2013.

Cover Letter

Cohesive Subgraph Search Using Keywords in Large Networks

Yuanyuan Zhu, Lu Qin, Lijun Chang, Jeffrey Xu Yu

Dear Editor-in-Chief, Prof Lin,

We are writing for the submission of our paper entitled “Cohesive Subgraph Search Using Keywords in Large Networks” to the prestigious journal, TKDE, for possible publication.

This submission is a substantial extension of our preliminary ICDE'18 short paper “Querying Cohesive Subgraphs by Keywords”.

In the conference version, we study the problem of finding cohesive subgraph containing the query keywords based on the k-truss model, and formulate it as minimal dense truss search problem, i.e., finding minimal subgraph with maximum trussness covering the keywords. We divided the problem into two sub-problems. The first is to find the dense truss G_{den} with the maximum trussness containing Q. The second is to refine G_{den} to obtain a minimal dense truss H containing Q. We first proposed a top-down search framework to find the dense truss with the maximum trussness containing keywords based on our newly proposed KT-Index, and then developed a refinement approach to extract the minimal dense truss based on the anti-monotonicity property of k-truss.

However, the refinement process is still time consuming, as we have to do k-truss verification for each single node deletion. Thus, in this journal submission, we further proposed several optimization techniques to accelerate the refinement process. Besides, we also provided more theoretical analysis for the studied problem, and extended the method proposed in the conference version to answer the top-r search problem. We also conducted more experimental studies on more real-world graphs to thoroughly evaluate the efficiency and effectiveness of our approaches. We summarize the major new materials in this submission as follows:

1. We provided a thorough theoretical analysis on the hardness of the problem studied in this paper (Section 2.2).

2. We discussed another possible choice to find the dense truss G_{den} in the bottom-up manner, and provide the complexity analysis to show why the top-down framework is preferred (Section 3.1).

3. We extended the method proposed for top-1 search in the conference version to answer top-r search (Section 4.2).

4.We proposed several optimization strategies to accelerate the refinement process, including batched based deletion, early-stop based deletion, and local exploration (Section 4.3).

5. We conducted more experiments by evaluating all the proposed algorithms on more real-world datasets, and showed that our newly proposed method is faster than our previously proposed method by one order of magnitude (Section 5).

We rewrite the whole paper by providing more theatrical analysis, algorithm details, illustrating examples, and experimental results, and extended it with new materials in about 10 pages in this submission. We attached our ICDE paper for reference.

Best Regards,

Yuanyuan Zhu, Lu Qin, Lijun Chang, Jeffrey Xu Yu