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.

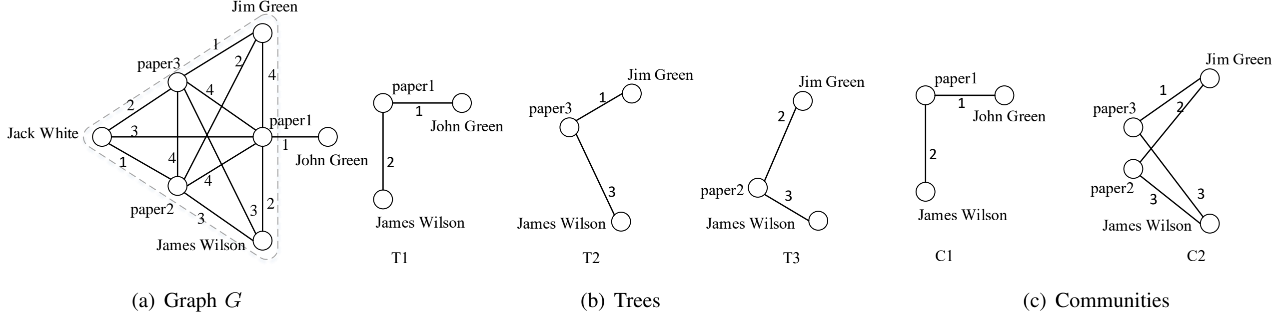
In recent decades, keyword search problem has been extensively studied [1], aiming to find minimal connected trees (Steiner tree [2] and distinct root tree [3]) or subgraphs (r-radius subgraph [4], community [5], and r-clique [6]) containing the keywords. Besides, keyword search can also be considered 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



(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

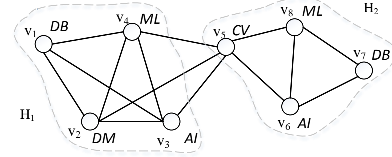


Fig. 2. An Example Graph G

$\triangle(v_2, v_3, v_5)$. 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 (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).

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;
```

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

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.