Querying Communities in Relational Databases

Lu Qin, Jeffrey Xu Yu, Lijun Chang, Yufei Tao

The Chinese University of Hong Kong, Hong Kong, China
{lqin,yu,ljchang}@se.cuhk.edu.hk, taoyf@cse.cuhk.edu.hk

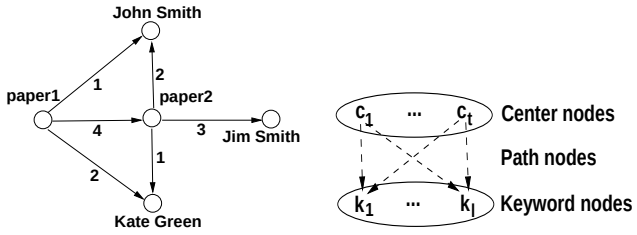
***Abstract*— Keyword search on relational databases provides users with insights that they can not easily observe using the traditional RDBMS techniques. Here, an l-keyword query is specified by a set of l keywords, $\{k_1, k_2, \cdots, k_l\}$. It finds how the tuples that contain the keywords are connected in a relational database via the possible foreign key references. Conceptually, it is to find some structural information in a database graph, where nodes are tuples and edges are foreign key references. The existing work studied how to find connected trees for an l-keyword query. However, a tree may only show partial information about how those tuples that contain the keywords are connected. In this paper, we focus on finding communities for an l-keyword query. A community is an induced subgraph that contains all the l-keywords within a given distance. We propose new efficient algorithms to find all/top-k communities which consume small memory, for an l-keyword query. For top-k l-keyword queries, our algorithm allows users to interactively enlarge k at run time. We conducted extensive performance studies using two large real datasets to confirm the efficiency of our algorithms.**

I. Introduction

Keyword search on relational databases has been widely studied in recent years. It takes a relational database as a database graph G_D by considering the tuples as nodes and the foreign key references as edges between nodes, and searches the hidden connections between those tuples that contain keywords specified in a user-given l-keyword query $(k_1, k_2, \cdots, k_l)$. Almost all existing work aim at finding the minimal connected trees that contain all the l-keywords in a database graph or in the underneath relational database [1], [2], [3], [4], [5], [6], [7], where some focused on finding all the minimal connected trees and some focused on finding the top-k minimal connected trees.

In this paper, we explore two key issues. The first is whether it is the best of users' interest to find minimal connected trees on a database graph G_D, and the second is how to efficiently find subgraphs (instead of trees) for user-given l-keyword queries, if it is highly desirable. We discuss the first issue in the introduction, and focus on the efficiency issue in the rest of the paper.

Consider a small graph, G, shown in Fig. 1(a). The graph G shows the co-authorship and the citation between two papers. There are 5 nodes (3 authors and 2 papers). The three authors are: John Smith, Jim Smith, and Kate Green, and two papers are: paper1 and paper2. There are 6 edges. The paper, paper1, was co-authored between John Smith and Kate Green, and the paper, paper2, was co-authored among Kate Green, John Smith, and Jim Smith. In addition, paper1 cited paper2. The edges are weighted. The edge from paper1 to John Smith and Kate Green are 1 and 2, respectively, because John Smith was the first author, and Kate Green was the second author. In a similar fashion, the author order for the three authors who wrote paper2 was indicated in the weights associated with the corresponding edges. We assume that the weight on the citation edge from paper1 to paper2 is 4.

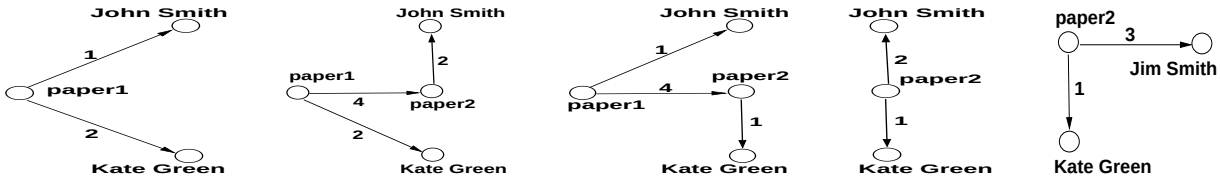


(a) Graph (b) Community

Fig. 1. Graph and Community

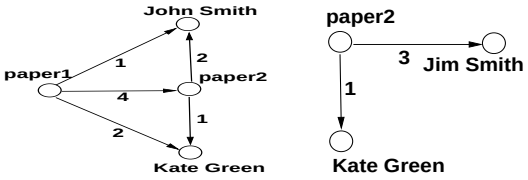
Next, consider a 2-keyword query, Kate and Smith, against the small graph. All the 5 trees, T_i, for $1 \leq i \leq 5$, are listed in Fig. 2. T_1 (Fig. 2(a)) shows that John Smith and Kate Green wrote a paper paper1. T_2 (Fig. 2(b)) indicates that John Smith wrote paper2, which was cited by paper1 written by Kate Green. Each of the first 4 trees, T_i, for $1 \leq i \leq 4$, in Fig. 2 gives some pieces of information between John Smith and Kate Green. But, none of the 4 trees give a better whole picture of the relationships between these two authors. There are two problems. One is that a user may find some information, but at the same time may miss some information he/she is really interested in when the user is browsing the resulting trees. For example, a user may want to find how many papers John Smith and Kate Green co-authored. T_1 only shows that they co-authored one paper. The other problem is that the number of resulting trees can be large for an l-keyword query, and it makes difficult for users to find all information he/she needs.

We propose to find communities (multi-center graphs) for an l-keyword query. Fig. 1(b) illustrates a community. It is an induced subgraph over a set of nodes, namely, keyword nodes, center nodes, and additional path nodes. (a) For a given l-keyword query, in a community, there are up to l keyword nodes. All the user-given l keywords appear in the keyword nodes. (b) There are center nodes, where each center connects to every keyword node within a threshold called radius. In other words, the distance along the shortest path between a center node and a keyword node is less than or equal to the radius. (c) The additional nodes, called path nodes, are the nodes that appear on any path from a center node, v_c, to a keyword node, v_l, if the distance along such a path between



(a) T_1 (b) T_2 (c) T_3 (d) T_4 (e) T_5

Fig. 2. Five Trees



(a) R_1 (2-Center) (b) R_2 (1-Center)

Fig. 3. Two Communities

v_c and v_l is less than or equal to the radius. Fig. 3 shows two such communities for the same 2-keyword query with a radius 6. Consider the community (R_1) in Fig. 3(a). There are 2 keyword nodes that contain the two keywords, Kate and Smith, respectively. There are 2 center nodes indicated by paper1 and paper2. Because the given radius is 6, there is a path from paper1 to Kate Green via paper2, with the total weight 5 (≤ 6), the edge from paper1 to paper2 is also included in the community. The community, R_1 (Fig. 3(a)) includes all the information represented by the 4 trees, T_i, for $1 \leq i \leq 4$ in Fig. 2. The semantics of such communities have been studied. The similar concepts are also used in co-citation analysis, authority/hub [8]. To our best knowledge, it is the first time that the authors study finding multi-center communities in relational databases. In addition, we study finding communities using a radius, which is the minimum total weight along a path from a center node to a keyword node. It is different from other reported studies that find the core of web communities as bipartite graphs [8].

Contributions of this paper: (1) We propose a general concept called community as a multi-center directed graph in a relational database when the relational database is considered as a database graph, G_D. (2) We propose an algorithm to enumerate all communities in polynomial delay under *query-and-data* complexity [9]. We show that the communities found are complete and duplication-free. By complete, we mean that we find all communities. We introduce a weak duplication-free concept under which we design a polynomial delay algorithm with time complexity of $O(l \cdot (n \cdot \log n + m))$ and space complexity of $O(l \cdot n + m)$, where n and m are the number of nodes and the number of edges in G_D, and l is the number of user-given keywords. The polynomial delay enumeration algorithms are considered as the best algorithm when enumerating results [9], [10], [4]. (3) We propose a polynomial delay algorithm with the same time complexity of $O(l \cdot (n \cdot \log n + m))$ and space complexity of $O(l^2 \cdot k + l \cdot n + m)$, to find the exact top-$k$ communities under a ranking order. One main advantage of our algorithm is that we allow a user to interactively reset the value of k for finding the top-k communities during run-time without overhead. (4) We propose an efficient indexing method to index the database graph, G_D. With such an index, we show that we can project a small database graph for an l-keyword query, and find the same set of communities. The search space can be significantly reduced. (5) We conduct extensive performance studies to confirm the efficiency of our algorithms using real datasets.

Organizations: The remainder of the paper is organized as follows. Section II gives our problem statement. Section III discusses several possible solutions, and highlights the main ideas of our polynomial delay algorithms after introducing several categories of enumeration algorithms. In Section IV, we discuss our algorithm to find all communities, and in Section V, we discuss our algorithm to find top-k communities. We also introduce our indexing and give an algorithm to project a subgraph of G_D for an l-keyword query to be evaluated. Experimental studies are given in Section VII followed by discussions on related work in Section VIII. Finally, Section IX concludes the paper.

II. Problem Statement

Following the notations used in [2], [3], [4], [7], we model a relational database *RDB* as a weighted directed graph, $G_D = (V, E)$, where V is the set of nodes (tuples) in *RDB*, and E is the set of edges between nodes based on the foreign key references between the corresponding tuples in *RDB*. Here, a node, $v \in V$, may contain keywords, a directed edge, $(u, v) \in E$, is associated with a weight, denoted $w_e((u, v))$. Given two nodes u and v, we use $\mathsf{dist}(u, v)$ to denote the total weight along the shortest path between u and v. In the following discussions, we focus on a general directed graph. Our approach can be easily applied to undirected or bi-directed graphs [2], [3], [7]. We use $V(G)$ and $E(G)$ to denote the set of nodes and the set of edges for a given graph G. We also denote the number of nodes and the number of edges in the database graph, G_D, using $n = |V(G_D)|$ and $m = |E(G_D)|$.

Fig. 4 shows an example of a database graph, G_D, which consists of 13 nodes, v_i, for $1 \leq i \leq 13$. Some nodes contain keywords: v_4 and v_{13} contain a keyword a, v_2 and v_8 contain a keyword b, and v_3, v_6, v_9 and v_{11} contain a keyword c. All the edges are weighted, for example, $w_e((v_1, v_2)) = 5$.

Given a database graph G_D, an l-keyword query consists of a set of l keywords, $\{k_1, k_2, \cdots, k_l\}$. An l-keyword query is to find a set of communities, $\mathcal{R} = \{R_1(V, E), R_2(V, E), \cdots\}$, which we define below.

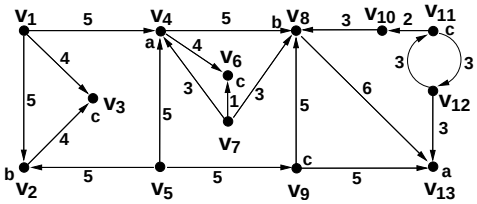
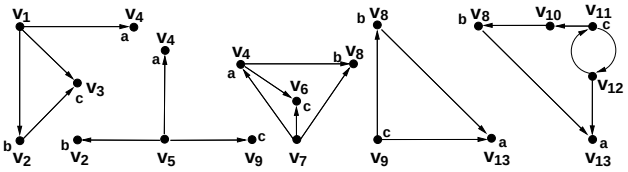


Fig. 4. A Simple Database Graph, G_D



(a) R_1 (b) R_2 (c) R_3 (d) R_4 (e) R_5

Fig. 5. Five communities

Definition 2.1: (**Community**) A community, $R_i(V, E)$, is a multi-center induced subgraph of the database graph G_D. Here, V is a union of three subsets, $V = V_c \cup V_l \cup V_p$. (1) V_l represents a set of nodes called *knode*s (keyword nodes). Every *knode* $v_l \in V_l$ contains at least a keyword and all l keywords must appear in at least one *knode* in V_l. (2) V_c represents a set of nodes called *cnode*s (center nodes). For any *cnode* $v_c \in V_c$, there exists at least a single path such that $\mathsf{dist}(v_c, v_l) \leq \mathsf{Rmax}$ (radius) between v_c and every $v_l \in V_l$. (3) V_p represents the nodes, called *pnode*s, which appear on any path from a *cnode* $v_c \in V_c$ to a *knode* $v_l \in V_l$ if $\mathsf{dist}(v_c, v_l) \leq \mathsf{Rmax}$. Note that a *cnode* or a *pnode* may also contain some keywords, and a node can be *knode* and *cnode* at the same time. $E(R_i)$ is the set of edges, $(u, v) \in E(G_D)$, for every pair of u and v that appear in $V(R_i)$. □

It is worth noting that a community, R_i, is uniquely determined by *knode*s, V_l, which we call the core of the community, and denote it as $\mathsf{core}(R_i)$. For simplicity, we use C to represent a core as a list of l nodes, $C = [c_1, c_2, \cdots, c_l]$, and may use $C[i]$ to denote $c_i \in C$, where c_i contains the keyword k_i.

Example 2.1: Consider the database graph G_D (Fig. 4). Let $\mathsf{Rmax} = 8$. For a 3-keyword query $\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$, 5 communities, R_i, for $1 \leq i \leq 5$, are shown in Fig. 5. For example, for R_5 (Fig. 5 (e)), *knode*s ($\mathsf{core}(R_5)$) are $V_l = \{v_{13}, v_8, v_{11}\}$, *cnode*s are $V_c = \{v_{11}, v_{12}\}$, and *pnode*s are $V_p = \{v_{10}\}$. □

For a community, R_i, a cost function can be defined, denoted $\mathsf{cost}(R_i)$, using edge weights $w_e()$.[1] We define $\mathsf{cost}(R_i)$ as the minimum total edge weight from a *cnode* to every $knode$ on the corresponding shortest path, in this paper. For example, consider the community R_5 (Fig. 5 (e)). There are two centers, v_{11} and v_{12}. The total edge weight over the shortest paths from v_{11} to the 3 *knode*s, v_8, v_{11}, and v_{13}, is $11 = (2+3) + 0 + (3+3)$. The total edge weight over the shortest paths from v_{12} to the 3 *knode*s, v_8, v_{11}, and v_{13}, is $14 = (3+2+3) + 3 + 3$. Therefore, $\mathsf{cost}(R_5) = 11$. Given two communities, R_i and R_j, R_i is ranked higher than R_j, denoted $R_i \succeq R_j$, if $\mathsf{cost}(R_i) \leq \mathsf{cost}(R_j)$. The highest rank is number 1. The ranking for the 5 communities given in Fig. 5 is shown in Table I. Note that our work does not rely on a specific cost function.

[1]For simplicity, we ignore node weights in this paper, and our approach can support node weights.

Rank	Knodes			Graph	Cost	Center
	a	b	c			
1	v_4	v_8	v_6	R_3	7	$\{v_4, v_7\}$
2	v_{13}	v_8	v_9	R_4	10	$\{v_9\}$
3	v_{13}	v_8	v_{11}	R_5	11	$\{v_{11}, v_{12}\}$
4	v_4	v_2	v_3	R_1	14	$\{v_1\}$
5	v_4	v_2	v_9	R_2	15	$\{v_5\}$

TABLE I
RANKING

Problem Statement: In this paper we study two interrelated problems for an l-keyword query against a database graph G_D, with a user-given radius Rmax, namely, finding all communities and finding the top-k communities, for a given l-keyword query. We denote them as *COMM-all* and *COMM-k*, respectively. For both, the resulting communities must be complete and duplication-free. By complete, we mean that we explore all combinations of the keywords to identify communities based on all possible cores. By duplication-free, we mean for any two communities R and R', $\mathsf{core}(R) \neq \mathsf{core}(R')$. In other words, let $C = [c_1, c_2, \cdots, c_l]$ and $C' = [c'_1, c'_2, \cdots, c'_l]$ be the two cores for the two communities, R and R', $C[i] \neq C'[i]$ for some i. We define duplication with the following consideration. A community is uniquely determined by its core, because otherwise the cost of checking whether two graphs are the same is too expensive based on graph isomorphism.

III. An Overview

In this section, we discuss several possible solutions and address the efficiency we want to achieve. For processing an l-keyword query, $\{k_1, k_2, \cdots, k_l\}$, with a Rmax, against G_D, let V_i be the set of nodes in $V(G_D)$ that contain the keyword k_i, and let $|V_i|$ be the number of nodes in V_i, for $1 \leq i \leq l$. Because a core C uniquely determines a community, we discuss how to find all and duplication-free cores in this section, and address the efficiency problems.

First, we consider a naive approach using the 3-keyword query in Example 2.1. For processing *COMM-all*, it needs a nested loop to check all combinations of nodes that contain keywords as follows.

```
1: for v_i ∈ V_1 do
2:   for v_j ∈ V_2 do
3:     for v_k ∈ V_3 do
4:       form a core candidate C[v_i, v_j, v_k];
5:       output C if there is a center c which connects every nodes
         in C within Rmax;
```

The three for-loops together check every combination of the three sets, V_1, V_2, and V_3, and compute every possible cores, C. The complexity is $O(|V_{max}|^3)$, where $|V_{max}| =$

$\max\{|V_1|, |V_2|, |V_3|\}$. In general, for an l-keyword query, it is in nature exponential $O(n^l)$ where $n = |V(G_D)|$. Since it checks every distinct combination of nodes that contain the three keywords, the result is complete and duplication-free.

In order to find communities based on the user-given keywords, an expanding approach can be adopted to solve our problem, which is to expand from nodes, step-by-step, until they can identify communities.

First, we can expand from all the nodes in V_i that contain the keyword, k_i, for $1 \leq i \leq l$. We call it bottom-up expanding, and outline it below.

1: $V' \leftarrow V_1 \cup V_2 \cup V_3$;
2: let each $u \in G_D$ maintain l-sets where each set, $u.V_i$, keeps the nodes $v \in V_i$ that u can reach within Rmax;
3: **repeat**
4: find a new node, w, that is expanded from a $v_i \in V'$ within Rmax;
5: add v_i into $w.V_i$ if v_i contains the keyword k_i;
6: **if** $w \neq \emptyset$ **and** all $w.V_i$ are non-empty **then**
7: output new cores found;
8: **until** $w = \emptyset$

During the expansion process, when a keyword node, $v_i \in V_i$ expands to a node, u, it implies that u can reach v_i, and we maintain v_i in a set denoted $u.V_i$ at the node u. In other words, the set $u.V_i$ maintains all the nodes containing keyword k_i that can be reached from u within Rmax. If all $u.V_i$, for $1 \leq i \leq l$, are non-empty, there exist cores of communities. The number of core candidates at node u is $O(|u.V_{max}|^3)$, where $|u.V_{max}| = \max\{|u.V_1|, |u.V_2|, |u.V_3|\}$. When it is to be output, the algorithm first checks if the candidate is a duplication. For doing so, the algorithm maintains a pool of the already output cores. When a new core is to be output, it checks whether it has already been in the pool. If it does not exist in the pool, the algorithm will output it and add it into the pool.

Second, in a similar fashion, we can expand from any node $u \in V(G_D)$ in a top-down fashion up to Rmax, and check whether it can contain cores of communities in a similar way as to maintain $u.V_i$ as used in the bottom-up expanding approach. Note that both top-down/bottom-up expanding approaches can find all communities. In other words, the results are complete and duplication-free.

A. Enumeration Delay

In this paper, we investigate new novel enumeration algorithms [9] for supporting *COMM-all* and *COMM-k* queries. To our problem, an enumeration algorithm, $\mathcal{A}$, outputs all/top-k communities, $\mathcal{O} = (R_1, R_2, \cdots, R_{|\mathcal{O}|})$, for an l-keyword query against a database graph G_D, with a radius Rmax. We consider the graph G_D and the l-keywords, as the input, and denote it as $\mathcal{I}$. The size of input, $\mathcal{I}$, is $|\mathcal{I}| = n + m + l$, where n and m are the numbers of nodes and edges of G_D, respectively. Note that Rmax is a constraint rather than an input data. The size of output is $|\mathcal{O}|$, where in $\mathcal{O}$ $R_i \neq R_j$ if $\mathsf{core}(R_i) \neq \mathsf{core}(R_j)$ (duplication-free).

First, consider the *COMM-all* queries. The naive approach (nested loop), in worst case, takes $\exp(\mathcal{I})$ time, that is $O(n^l)$, to output all the results of size $|\mathcal{O}|$. The complexity of the naive approach is irrelevant to the output size, $|\mathcal{O}|$, which can be even much larger than the input $|\mathcal{I}|$. Therefore, even an enumeration algorithm, $\mathcal{A}$, is not polynomial to the input $|\mathcal{I}|$, it may be seen as reasonable, because, when $|\mathcal{O}|$ dominates, all algorithms need to output $\mathcal{O}$. Therefore, it is requested to consider the complexity by taking both input, $\mathcal{I}$, and output, $\mathcal{O}$, into consideration. In the literature [9], [11], there are several categories of enumeration algorithms, namely, polynomial total time, incremental polynomial time, and polynomial delay. Here, *polynomial total time* means that the processing time of the algorithm, $\mathcal{A}$, is polynomial to both sizes of input and output, $|\mathcal{I}| + |\mathcal{O}|$. The *incremental polynomial time* implies that the processing time to output the o-th answer, which does not necessarily follow any ranking order, is polynomial to the combined size of the input and the first $o-1$ answers output already, $|\mathcal{I}| + o$. And, the *polynomial delay* implies that the o-th output is output in time which is polynomial only to $|\mathcal{I}|$. Obviously, the best algorithm is a polynomial delay algorithm.

A bottom-up/top-down expanding algorithm is not a polynomial delay algorithm, because it needs to check the already output results in order to ensure duplication-free. A bottom-up/top-down expanding algorithm is an incremental polynomial time algorithm, because it is polynomial to the combined size of input and the size of results that have been generated.

Second, consider the *COMM-k* queries, which are to output the top-k communities in an order (ranking). In [12], Lawler gives a procedure (Lawler's procedure) to compute the k best solutions to discrete optimization problems, and shows that if the number of computational steps required to find an optimal solution to a problem with l $(0, 1)$ variables is $c(l)$, then the amount of computation required to obtain the k best solutions is $O(l \cdot c(l))$. Kimelfeld et al. propose a polynomial delay algorithm that adopts the Lawler's procedure to find top-k steiner trees for keyword search problems [4]. Based on the Lawler's procedure, for *COMM-k* queries, it is straightforward to obtain an algorithm of the time complexity, $O(l \cdot c(l))$, where $c(l)$ is the time complexity to compute the top-1 community. However, in this paper, we propose new algorithms to compute *COMM-k* queries, which is $O(c(l))$ instead of $O(l \cdot c(l))$.

B. New Enumeration Delay Algorithms

We highlight the main ideas of our novel polynomial delay algorithms for processing *COMM-all*/*COMM-k* queries followed by detail discussions in the following sections.

1: find the first best core, C;
2: **while** $C \neq \emptyset$ **do**
3: output the community based on C;
4: $C \leftarrow Next(\cdot)$;

First, we discuss our algorithm for processing *COMM-all* queries. As shown above, it first finds the first best core C. In the while loop, it outputs the community based on C which is

duplication-free. Here, *Next*() is a precedure to determine the next core. The main issue is how to enumerate all (complete). We explain it using the 3-keyword query in Example 2.1. Suppose that the first core determined is $C = [v_a, v_b, v_c]$ for an l-keyword query. Here, v_a, v_b, and v_c contain the keyword, a, b, and c, respectively. We need to ensure that such C will not be enumerated again. In doing so, we divide the entire search space, $V_1 \times V_2 \times V_3$, into 4 subspaces $(l+1)$: S_1: $\{v_a\} \times \{v_b\} \times \{v_c\}$, S_2: $(V_1 - \{v_a\}) \times V_2 \times V_3$, S_3: $\{v_a\} \times (V_2 - \{v_b\}) \times V_3$, and S_4: $\{v_a\} \times \{v_b\} \times (V_3 - \{v_c\})$. It is important to know the following facts. (a) S_1 is the current core found. (b) $V_1 \times V_2 \times V_3 = S_1 \cup S_2 \cup S_3 \cup S_4$. It implies that we can enumerate all cores (complete). (c) $S_i \bigcap S_j = \emptyset (i \neq j)$ (duplication-free). In order to enumerate all, we propose a depth-first traversal algorithm. Conceptually, there exists a virtual root node, which represents the entire search space, and, as shown in this example, it has 4 child nodes (S_1, S_2, S_3, S_4) representing 4 subspaces. Suppose that we find the next best core in one of the subspaces, say S_4. With the same procedure, we further divide S_4 into 4 subspaces in the similar way in a depth-first traversal fashion in traversing the virtual tree. The time complexity of our algorithm is $O(l \cdot (n \cdot \log(n) + m))$ using space $O(l \cdot n + m)$. The similar idea can be easily extended to support any l-keyword queries.

Below, we outline our algorithm for *COMM-k* queries.

```
1: H ← ∅;
2: find the first best core, C;
3: H.enheap(C);
4: while H ≠ ∅ do
5:    g ← H.deheap();
6:    output the community based on g.C;
7:    Next(·);
```

For *COMM-k* queries, we need to output communities following its ranking order. In doing so, we use a Fibonacci heap ($\mathcal{H}$). We explain our main idea using the same example. We find the first best core, C, in the entire space $V_1 \times V_2 \times V_3$, and we ensure the first core found is the core for the top-1 community. We enheap C with other information into the heap $\mathcal{H}$, and enter the while loop. In the while loop, first, we deheap the core C, with the smallest cost, from $\mathcal{H}$. We compute its community and output it. Then, we try to call *Next*(). In *Next*(), we attempt to find the next best core in each of the three subspaces, S_2, S_3, and S_4, individually. If we find the best core, C_i, in S_i, for $1 < i \leq 4$, we enheap C_i to $\mathcal{H}$. With $\mathcal{H}$, the next best core, with the smallest cost, can be selected in the next iteration from all the cores kept in $\mathcal{H}$. We repeatedly deheap one core, C', from $\mathcal{H}$, identify the subspace where C' is in, say S', further divide S' into $l = 3$ subspaces, find the best cores in the 3 subspaces, and enheap them into $\mathcal{H}$ for finding the next best core. The time complexity of our algorithm is also $O(l \cdot (n \cdot \log(n) + m))$ using space $O(l^2 \cdot k + l \cdot n + m)$.

Algorithm 1 *COMM-all* (G_D, $\{k_1, k_2, \cdots, k_l\}$, Rmax)

Input: the database graph (G_D), the set of keywords $\{k_1, k_2, \cdots, k_l\}$, a radius Rmax.
Output: all communities.

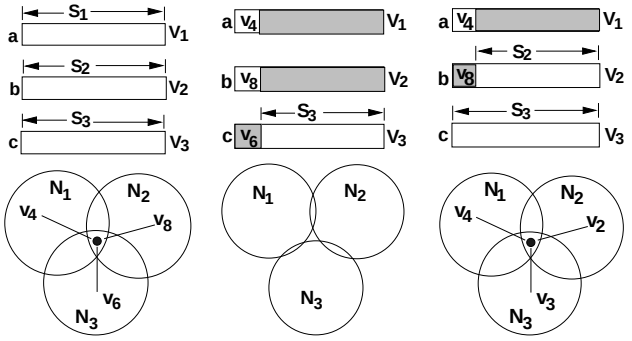
```
1: for i = 1 to l do
2:    V_i ← the set of nodes in G_D containing k_i;
3:    S_i ← V_i;
4:    N_i ← Neighbor(G_D, S_i, Rmax);
5: (C, cost) ← BestCore(N_1, N_2, ··· , N_l);
6: while C ≠ ∅ do
7:    R' ← GetCommunity(G_D, C, Rmax);
8:    output R';
9:    C ← Next(G_D, C, Rmax);

10: Procedure Next(G_D, C, Rmax)
11: for i = 1 to l do
12:    N_i ← Neighbor(G_D, {C[i]}, Rmax);
13: for i = l downto 1 do
14:    S_i ← S_i − {C[i]};
15:    N_i ← Neighbor(G_D, S_i, Rmax);
16:    (C', cost') ← BestCore(N_1, N_2, ··· , N_l);
17:    if C' ≠ ∅ then
18:       return C';
19:    S_i ← V_i;
20:    N_i ← Neighbor(G_D, S_i, Rmax);
21: return ∅;
```

IV. Find All Communities

The algorithm for processing *COMM-all* is shown in Algorithm 1, where it takes three inputs, the database graph G_D, the set of keywords, $\{k_1, k_2, \cdots, k_l\}$, and the radius Rmax. First, for every keyword k_i, it finds all nodes in $V(G_D)$ that contain k_i, and assigns them into V_i (line 2), which can be done using the full text index [1] efficiently. For every V_i, we introduce S_i which represents the currently available subset of V_i, as candidates, for finding next community. Initially, S_i is set to be V_i (line 3). For S_i, it finds the subset of $V(G_D)$, called neighborSet of S_i and denote as N_i, by calling a procedure *Neighbor*() (Algorithm 2). In N_i, every node v_j must have at least one neighbor $v_k \in S_i$ such as $\mathsf{dist}(v_j, v_k) \leq$ Rmax. Note that $S_i \subseteq V_i \subseteq N_i \subseteq V(G_D)$. Then, it attempts to output all communities in the rest of the algorithm. It finds the first core of a community, C, associated with a $cost$, by calling *BestCore*() (Algorithm 3) with all neighborSets, N_i, for $1 \leq i \leq l$ (line 5). In the while loop, the unique community, R', for a non-empty core C is determined by *GetCommunity*() (Algorithm 4). The while loop repeats by calling the *Next*() procedure, which finds the next core, C.[2] In *Next*(), there are two main parts: the preparation phase (line 11-12), and the search phase to find the next core (line 13-20). We explain the two main parts below. Recall that C is represented as a list of l nodes, $C = [c_1, c_2, \cdots, c_l]$, where $c_i = C[i]$ contains the keyword k_i. It attempts to find the next core, $C' = [c'_1, c'_2, \cdots, c'_l]$ which contains at least one node $c'_i \neq c_i \in C$. It is important to note that $C' \neq C$, because

[2]For simplicity, we assume that all variables, V_i and S_i, for $1 \leq i \leq l$, used are global variables, and we do not need to pass them to the procedure *Next*().

in the last iteration, at least $c_i \neq c'_i$ since c_i is removed from S_i (line 14). In addition, C' is not only different from the current, C, but also different from any core found up to this stage (duplication free). Finally, the procedure *Next*() search the entire search space, and does not miss any possible cores. We will explain these issues in detail later, after showing an example.



(a) initial (b) after $[v_4, v_8, v_6]$ (c) $[v_4, v_2, v_3]$

Fig. 6. Finding cores

Reconsider the database graph G_D in Fig. 4 for the 3-keyword query, $k_1 = \mathsf{a}$, $k_2 = \mathsf{b}$, and $k_3 = \mathsf{c}$, with $\mathsf{Rmax} = 8$. Initially, after line 4, $S_1 = V_1 = \{v_4, v_{13}\}$, $S_2 = V_2 = \{v_8, v_2\}$, and $S_3 = V_3 = \{v_6, v_3, v_9, v_{11}\}$. Also, the three neighborSets are: $N_1 = \{v_1, v_4, v_5, v_7, v_8, v_9, v_{11}, v_{12}, v_{13}\}$, $N_2 = \{v_1, v_2, v_4, v_5, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}\}$, and $N_3 = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_9, v_{11}, v_{12}\}$. The *BestCore*() will identify a core based on the nodes in the intersection of $N_1 \cap N_2 \cap N_3 = \{v_1, v_4, v_5, v_7, v_9, v_{11}, v_{12}\}$, because only a node in the intersection of the neighborSets can possibly serve as a center to connect the nodes that contain all three keywords. Suppose that *BestCore*() identifies a core $C = [v_4, v_8, v_6]$ centered at v_7 with a cost of 7 (line 5) (Refer to Fig. 6(a)). The first community based on the core C is uniquely identified as R_3 in Table I (Fig. 5 (c)), and is output (line 7-8). Then, it calls *Next*() to find the next core (line 9).

In *Next*(), it attempts to find the next core based on the current core $C = [v_4, v_8, v_6]$. It computes the three neighborSets, N_1, N_2, and N_3, for the three nodes in C regarding them as the center, respectively. After line 12, $N_1 = \{v_1, v_4, v_5, v_7\}$, $N_2 = \{v_4, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}\}$, and $N_3 = \{v_4, v_6, v_7\}$. Then, in the for loop, initially $i = 3$. Note that $C = [v_4, v_8, v_6]$, $C[3] = v_6$, and $S_3 = \{v_3, v_9, v_{11}\}$ after removing $C[3] = v_6$ from S_3 (line 14). It implies that the next core should not contain v_6 in S_3. It recomputes N_3 using S_3, N_3 is reset to be $N_3 = \{v_1, v_2, v_3, v_5, v_9, v_{11}, v_{12}\}$ (line 15). Then, it attempts to find the next core by calling *BestCore*() using the three newly computed neighborSets, N_1, N_2, and N_3. However, as can be seen, the intersection of $N_1 \cap N_2 \cap N_3 = \emptyset$, therefore, it is impossible to find a core. *BestCore*() will return an empty C' (line 16)(Refer to Fig. 6(b)). Because $C' = \emptyset$, it will move to the next iteration. Before returning to the main while loop, it resets $S_3 \leftarrow V_3$ (line 19), because any new combination, to form a new core C' in the next iteration, can possibly contain any node in the entire V_3. It also recomputes $N_3 = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_9, v_{11}, v_{12}\}$. In the next iteration, for $i = 2$, it repeats the similar procedure starting from $S_2 \leftarrow S_2 - \{v_8\} = \{v_2\}$ because $C[2] = v_8$. The new neighborSet N_2 becomes $\{v_1, v_2, v_5\}$ (line 15), and $C' = [v_4, v_2, v_3]$ (line 16). Since $C' \neq \emptyset$, it returns the new core C' in line 18 (Refer to Fig. 6(c)). Fig. 6 shows the main ideas. The three sets, V_i, for $1 \leq i \leq 3$, are represented as three rectangles. In a rectangle, the shaded part is the subset of V_i that does not need to be searched in an iteration. The circles represent neighborSets.

Algorithm 2 *Neighbor*(G_D, V_i, Rmax)

Input: G_D is the database graph, and $V_i \subseteq V(G_D)$
Output: neighborSet of V_i within Rmax.

1: let $G_t(V_t, E_t)$ be a virtual graph such as $V_t = V(G_D) \cup \{t\}$ and $E_t = E(G_D) \cup \{(v, t) | v \in V_i\}$ where every $w_e((v, t)) = 0$;
2: run the Dijkstra's algorithm to find the shortest paths from all nodes in V_t to t; {consider $(u, v) \in E_t$ as (v, u) (reverse order)}
3: $N_i \leftarrow \{u \mid \mathsf{dist}(u, t) \leq \mathsf{Rmax} \wedge u \in V(G_D)\}$;
4: **return** N_i;

A. The Three Subproblems

In this section, we discuss the details of the three procedures used in Algorithm 1, namely, *Neighbor*(), *BestCore*(), and *GetCommunity*().

The algorithm for *Neighbor*() is shown in Algorithm 2. It takes three inputs: the database graph G_D, the set of nodes V_i where every node $v \in V_i$ contains the keyword k_i, and the radius Rmax. The *Neighbor*() will find the neighborSet of V_i, denoted N_i, such that every node $u \in N_i$ has at least a node $v \in V_i$ where $\mathsf{dist}(u, v) \leq \mathsf{Rmax}$. The nodes in N_i have the potential to be a *cnode* in a community. Obviously, $V_i \subseteq N_i$.

In Algorithm 2, it constructs a virtual graph $G_t(V_t, E_t)$ where $V_t = V(G_D) \cup \{t\}$ and $E_t = E(G_D) \cup \{(v, t) | v \in V_i\}$ (line 1). In other words, the virtual graph G_t has one additional sink node, t, and additional edges from every $v \in V_i$ to t. For a newly added edge (v, t), the weight, $w_e((v, t))$, is set to be zero. Then, it runs Dijkstra's algorithm to find the shortest paths from the newly added sink node t to all nodes in V_t, by consider every edge $(u, v) \in E_t$ as (v, u) (reverse order) (line 2). Then, it identifies the neighborSet of V_i, as $N_i = \{u \mid \mathsf{dist}(u, t) \leq \mathsf{Rmax} \wedge u \in V(G_D)\}$. It is interesting to note that, because the shortest path from any node $u \in N_i$ to the sink node t must bypass a node $v \in V_i$ and the weight from, $v \in V_i$, to t is zero, if $\mathsf{dist}(u, t) \leq \mathsf{Rmax}$, for $u \in N_i$, the node u must have at least a near node $v \in V_i$ such as $\mathsf{dist}(u, v) \leq \mathsf{Rmax}$.

The time complexity of Algorithm 2 is the time complexity of Dijkstra algorithm $O(n \cdot log(n) + m)$ where $n = |V(G_D)|$ and $m = |E(G_D)|$ for a given database graph G_D. For every node, u, in the computed neighborSet, N_i, we store the nearest node $v \in V_i$ that contain the keyword k_i, and the shortest distance, and denote them as $src(N_i, u)$ and $min(N_i, u)$, respectively. The space complexity for N_i is $O(n)$.

Algorithm 3 *BestCore*($\{N_1, N_2, \cdots, N_l\}$)

Input: all l neighborSets.
Output: the best core and its cost.

1: $C \leftarrow \emptyset$; $best \leftarrow +\infty$;
2: $\mathcal{N} \leftarrow \bigcap_{i=1}^{l} N_i$;
3: **for all** $u \in \mathcal{N}$ **do**
4: $\quad c \leftarrow \mathsf{nearestCore}(u)$;
5: $\quad$ **if** $\mathsf{cost}(c) < best$ **then**
6: $\quad\quad C \leftarrow c$; $best \leftarrow \mathsf{cost}(c)$;
7: **return** $(C, best)$;

The *BestCore*() algorithm is shown in Algorithm 3. It takes l neighborSets as input, and finds the best core $C = [c_1, c_2, \cdots, c_l]$ where c_i contains the keyword k_i. Note that $V_i \subseteq N_i$, for $1 \leq i \leq l$. It computes the intersection of all neighborSets N_i, for $1 \leq i \leq l$, denoted $\mathcal{N}$ (line 2). Every node $u \in \mathcal{N}$ must be able to serve as a center to form a core $C = [c_1, c_2, \cdots, c_l]$ because $\mathsf{dist}(u, c_i) \leq \mathsf{Rmax}$ for every $c_i \in C$. In a for loop (line 3-6), the core C with the smallest $\mathsf{cost}(C)$ is determined. Here, $\mathsf{nearestCore}(u)$ identifies the core centered at u, $\mathsf{cost}(c)$ computes the cost of the core c.

We achieve $O(n)$ time to find the best next core with some preparation which is done by sharing the computational cost done in *Neighbor*() using additional data structures. In implementation, we maintain a data structure, with three elements, for every node $u \in V(G_D)$. The first element is a list of l pairs. The i-th pair maintains the nearest node of u, say v_i, that contains the keyword k_i as well as the total weight along the path between u and v_i ($\mathsf{dist}(u, v_i)$). For the i-th pair, we record $(v_i, \mathsf{dist}(u, v_i))$ if there exits $v_i \in V_i$ and $\mathsf{dist}(u, v_i) \leq \mathsf{Rmax}$, otherwise we record $(\emptyset, +\infty)$. The second element maintains the total weight $\sum_{i=1}^{l} \mathsf{dist}(u, v_i)$ if $\mathsf{dist}(u, v_i) \leq \mathsf{Rmax}$. The third element keeps how many v_i, for $1 \leq i \leq l$, $\mathsf{dist}(u, v_i) \leq \mathsf{Rmax}$. If the counter is l, it implies that the corresponding u can be a possible center in a community. The space for the table is $O(l \cdot n)$. We update the data structure while computing neighborSets without additional cost in terms of time complexity. With such a data structure available, in *BestCore*(), we only need to scan the data structure once and then find the core with the smallest cost.

The algorithm for *GetCommunity*() is shown in Algorithm 4. It takes three inputs, the database graph G_D, a core C, and Rmax, and uniquely determines a community, $\mathcal{R}(\mathcal{V}, \mathcal{E})$, based on the core C. Note that a community based on a core C is an induced subgraph $\mathcal{R}(\mathcal{V}, \mathcal{E})$ where $\mathcal{V} = V_c \cup V_l \cup V_p$. Here V_l is the set of *knode*s and $V_l = C$. We need to determine the set of *cnode*s V_c and the set of *pnode*s V_p.

First, we identify the set of *cnode*s, V_c, where each *cnode* $v \in V_c$ can reach every *knode* $c \in C$, such as $\mathsf{dist}(v, c) \leq \mathsf{Rmax}$ (line 1). In order to find the set of *cnode*s, V_c, we construct a virtual graph $G'(V', E')$ where $V' = V(G_D)$ and $E' = \{(u, v) | (v, u) \in E(G_D)\}$, for a given database graph G_D. In our implementation, for every node, $u \in V(G_D)$ $(= V(G'))$, we keep a pair of numbers, namely, $u.sum$ and $u.count$. Both are initialized zero. Then, for each *knode*, $c \in C$, we compute the shortest paths from c to all the other nodes using Dijkstra's algorithm. For every $u \in V(G_D)$, if $\mathsf{dist}(u, c) \leq \mathsf{Rmax}$, we update $u.sum \leftarrow u.sum + \mathsf{dist}(u, c)$ and $u.count \leftarrow u.count + 1$. There are l *knode*s in C, and we run Dijkstra's algorithm l times. If $u.count = l$, for $u \in V(G_D)$, it indicates that u can be added into the set of centers, V_c.

Algorithm 4 *GetCommunity*(G_D, C, Rmax)

Input: the graph G_D, a core C, and the radius Rmax.
Output: a community uniquely determined by the core C.

1: find the set of *cnode*s, V_c, where each *cnode* $v \in V_c$ can reach every $c \in C$ within Rmax;
2: let $G_s(V_s, E_s)$ be a virtual graph such as $V_s = V \cup \{s\}$ and $E_s = E \cup \{(s, v) | v \in V_c\}$ where every $w_e((s, v)) = 0$;
3: run the Dijkstra's algorithm to find the shortest paths from s to all nodes in V_s;
4: let $G_t(V_t, E_t)$ be a virtual graph such as $V_t = V \cup \{t\}$ and $E_t = E \cup \{(u, t) | u \in C\}$ where every $w_e((u, t)) = 0$;
5: Let every $(u, v) \in E_t$ be (v, u), run the Dijkstra's algorithm to find the shortest paths from t to all nodes in V_t;
6: $\mathcal{V} \leftarrow \{u \mid \mathsf{dist}(s, u) + \mathsf{dist}(u, t) \leq \mathsf{Rmax} \wedge u \in V(G_D)\}$;
7: construct an induced subgraph $\mathcal{R}$ in G_D including all nodes in $\mathcal{V}$;
8: **return** $\mathcal{R}$;

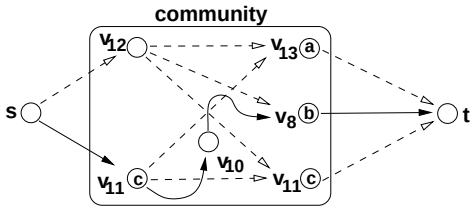


Fig. 7. Finding the community for a core

Second, based on $V_l = C$ and V_c computed, we compute $\mathcal{V}$ as follows. (1) We construct a virtual graph $G_s(V_s, E_s)$ where $V_s = V(G_D) \cup \{s\}$ and $E_s = E \cup \{(s, v) | v \in V_c\}$. The weight for each newly added edge (s, v) is set to be zero. Like *Neighbor*(), it runs Dijkstra's algorithm to compute the shortest paths from s to all the other nodes (line 2-3). After line 3, every node, u, is associated with a counter recording the distance $\mathsf{dist}(s, u)$, if $\mathsf{dist}(s, u) \leq \mathsf{Rmax}$. (2) We construct another virtual graph $G_t(V_t, E_t)$ where $V_t = V(G_D) \cup \{t\}$ and $E_t = E \cup \{(v, t) | v \in C\}$. The weight for each newly added edge (v, t) is set to be zero. We treat the graph G_t as a reversed graph by virtually dealing with $(v, u) \in E_t$ as (u, v). Again, like *Neighbor*(), we run Dijkstra's algorithm to compute the shortest paths from t to all the others (line 4-5). Every node, u, is associated with another counter recording the distance $\mathsf{dist}(u, t)$ if $\mathsf{dist}(u, t) \leq \mathsf{Rmax}$. Then $\mathcal{V}$ is computed by selecting the nodes $u \in V(G_D)$ if $\mathsf{dist}(s, u) + \mathsf{dist}(u, t) \leq \mathsf{Rmax}$ (line 6). Note that $V_c \subseteq \mathcal{V}$, $C \subseteq \mathcal{V}$. The total time complexity for Algorithm 4 is $O(l \cdot (n \cdot log(n) + m))$.

We explain Algorithm 4 using the database graph G_D in Fig. 4 for the 3-keyword query, $k_1 = \mathsf{a}$, $k_2 = \mathsf{b}$, and $k_3 = \mathsf{c}$, with $\mathsf{Rmax} = 8$. Let $\mathcal{R}(\mathcal{V}, \mathcal{E})$ be the community for a core $C = [v_{13}, v_8, v_{11}]$. Here, $\mathcal{V} = V_c \cup V_l \cup V_p$. V_l is the given set

of *knode*s, C, $V_c = \{v_{11}, v_{12}\}$, and $V_p = \{v_{10}\}$. The set of edges, $\mathcal{E}$, can be easily identified by scanning $E(G_D)$. Fig. 7 shows the community found, where s and t are two virtual nodes used in *GetCommunity*().

B. The Time/Space Complexity

Theorem IV.1: *Algorithm 1 enumerates communities in polynomial delay time, $O(l \cdot (n \cdot \log(n) + m))$, with the space complexity of $O(l \cdot n + m)$.* □

Proof Sketch: The time complexity to get a community from a core (line 7) is $O(l \cdot (n \cdot \log(n) + m))$, we only need to prove that the complexity to get the next core (line 9) is $O(l \cdot (n \cdot \log(n) + m))$. Lines 11-12 invoke l times of Neighbor(), which costs $O(l \cdot (n \cdot \log(n) + m))$. Lines 13-20 loop for at most l times. In each iteration, we invoke *Neighbor*() 2 times, which costs $O(n \cdot \log(n) + m)$ and 1 time of *BestCore*(). Note that *BestCore*() is $O(n)$. So the total time complexity for *Next*() becomes $O(l \cdot (n \cdot \log(n) + m))$ and the total time complexity for Algorithm 1 to enumerate each answer is also $O(l \cdot (n \cdot \log(n) + m))$. For the space complexity, we need to record l values for each center v_c, the best core centered at v_c, using space $O(n \cdot l)$, and load the database graph, G_D, using space $O(n + m)$. All S_i and N_i ($1 \le i \le l$) cost space $O(n \cdot l)$. The total space complexity is $O(n \cdot l + m)$. □

V. Find Top-K Communities

The algorithm for *COMM-k* is shown in Algorithm 5. It takes four inputs, the database graph G_D, the set of keywords, $\{k_1, k_2, \cdots, k_l\}$, the radius Rmax, and an integer $\mathsf{k} > 0$, and outputs the top-k communities.

We first compute the set of nodes, S_i, that contain the keyword k_i, and compute its neighborSet for S_i, for $1 \le i \le l$ (line 1-3). And we compute the first and the best core, C (line 4).

In order to find the top-k communities, we use a data structure, called *can-list*, to maintain a list of core candidates among them the top-k core and its community can be identified. The maximum size of the pool is $l \cdot k$ at most, which we will explain later in detail. A candidate core is kept in a 4-element tuple, called can-tuple, in the form of $(C, cost, pos, prev)$. Here, C is the core of a community, $cost$ is the minimum total weight from the nearest center, denoted u, of C, to every node $c_i \in C$, for $1 \le i \le l$, such as $\sum_{i=1}^{l} \mathsf{dist}(u, c_i)$. The $prev$ points to its previous candidate in the can-list. We explain pos using an example. Consider two can-tuple, $x = (C, cost, \cdots, \cdots)$ and $x' = (C', cost', i, x)$, and suppose $C = [c_1, c_2, \cdots, c_i, \cdots, c_l]$ and $C' = [c'_1, c'_2, \cdots, c'_i, \cdots, c'_l]$. The $prev$ in the can-tuple x' points to the can-tuple x. The position, $pos = i$, in x', means that, by comparing C and C' kept in the two candidates, $c_j = c'_j$ if $1 \le j < i$, $c_i \ne c'_i$, and c_j and c'_j may or may not be the same if $j > i$. Over the can-list, we use a Fibonacci heap, denoted $\mathcal{H}$, which is initialized to be empty (line 5). In the following, when we enheap a can-tuple to $\mathcal{H}$, we mean to insert it into the can-list, and then keep a pointer in $\mathcal{H}$ pointing to the can-tuple on the can-list. When we deheap a can-tuple from $\mathcal{H}$, we simply remove the pointer from $\mathcal{H}$, but still maintain the can-tuple on the can-list. The complexity for $\mathcal{H}.enheap()$ and $\mathcal{H}.deheap()$ is $O(1)$ and $O(\log n)$, respectively.

Algorithm 5 *COMM-k* (G_D, $\{k_1, k_2, \cdots, k_l\}$, k, Rmax)

Input: the database graph (G_D), the set of keywords $\{k_1, k_2, \cdots, k_l\}$, a radius Rmax, and $\mathsf{k} > 0$
Output: the top k communities.

```
1: for i = 1 to l do
2:    S_i ← the set of nodes in G_D containing k_i;
3:    N_i ← Neighbor(G_D, S_i, Rmax);
4: (C, cost) ← BestCore(N_1, N_2, ··· , N_l);
5: H ← ∅;
6: H.enheap(C, cost, 1, ∅);
7: while H ≠ ∅ do
8:    g ← H.deheap(); {g = (C, cost, pos, prev)}
9:    G' ← GetCommunity(g.C);
10:   output G';
11:   k ← k − 1;
12:   if k = 0 then
13:      return;
14:   Next(g);

15: Procedure Next(g)
16: for i = 1 to l do
17:   N_i ← Neighbor({g.C[i]});
18:   S_i ← the set of nodes in G_D containing k_i;
19: h ← g;
20: while h ≠ ∅ do
21:   i ← h.pos;
22:   S_i ← S_i − {h.C[i]};
23:   h ← h.prev;
24: for i = l downto g.pos do
25:   S_i ← S_i − {g.C[i]};
26:   N_i ← Neighbor(S_i);
27:   (C', cost) ← BestCore(N_1, N_2, ··· , N_l);
28:   if C' ≠ ∅ then
29:      H.enheap(C', cost, i, g);
30:   S_i ← S_i ∪ {g.C[i]};
31:   N_i ← Neighbor(S_i);
```

It first enheaps a can-tuple for the first found core C with its $cost$ (line 6). Because it is the first can-tuple to be maintained in the can-list, its $prev = \emptyset$, and its $pos = 1$. The following while loop repeats when $\mathcal{H}$ is non-empty (line 7-14). When $\mathcal{H} \ne \emptyset$, it deheaps $\mathcal{H}$ and assigns it to a can-tuple, g (line 8). Because $\mathcal{H}$ is maintained in an ascending order, the can-tuple, g, is with the smallest cost among others in $\mathcal{H}$. It will call *GetCommunity*() to output the community for the core $g.C$ (line 9). Then, it decreases k by 1, and checks if it has already output k communities (line 11-13). If $\mathsf{k} \ne 0$, it then add more can-tuples into H, by calling the procedure *Next*() (line 14).

In the procedure *Next*(), it conducts three main things. First, in a preparation phase, it computes the neighborSet for every node in the core of the deheaped can-tuple g, $g.C[i]$, for $1 \le i \le l$, and it also recomputes S_i as to include all nodes in G_D that contain the keyword k_i. Second, it removes those candidates that have been considered before (line 20-23). This process limits the search space to a specific subspace out of l subspaces. Third, it adapts the similar idea used in Algorithm 1 to find the next l candidates, and enheap them into $\mathcal{H}$.

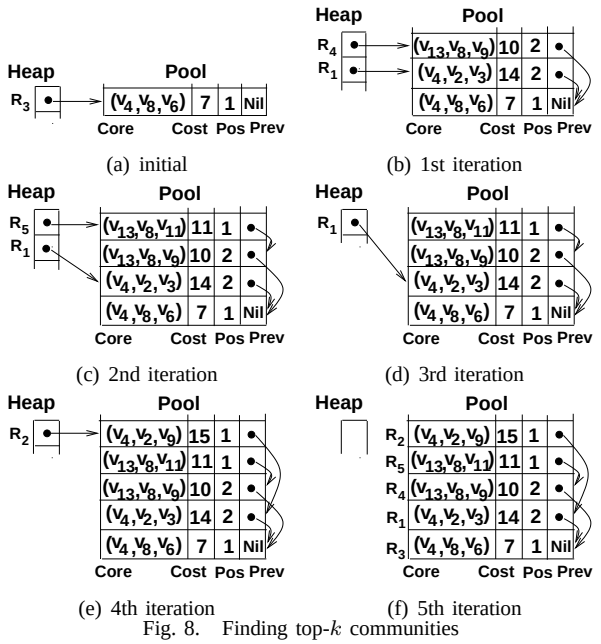


Fig. 8. Finding top-k communities

Fig. 8 shows the pool (can-list) and heap $\mathcal{H}$ when finding the top-5 communities in Example 2.1. The 5 communities, R_i, for $1 \leq i \leq 5$, are listed in Fig. 5.

Theorem V.1: *The time complexity for Algorithm 5 is* $O(l \cdot (n \cdot \log(n) + m))$ *using* $O(l^2 \cdot k + l \cdot n + m)$ *space.* □

Proof Sketch: For the time complexity, we only need to prove that the heap operations (line 8 and line 29) do not have impacts on the complexity of the algorithm and Lines 20-23 is done in complexity $O(l \cdot n)$. The complexity for the other parts is all the same as in Algorithm 1. In Algorithm 5, every iteration, we deheap a can-tuple from $\mathcal{H}$ and enheap at most l can-tuples into $\mathcal{H}$ in order to get the next high ranked community. Suppose we have output p communities already. There are at most $p \cdot l$ can-tuples in $\mathcal{H}$. Note that in total $p \leq n^l$. Using the Fibonacci heap, the $deheap()$ costs $O(\log(p \cdot l)) \leq O(\log(n^l \cdot l)) = O(l \cdot log(n) + \log(l)) \leq O(l \cdot n)$ time. The $enheap()$ only costs $O(1)$ time. Therefore, the heap operations do not affect the time complexity of the algorithm. Lines 20-23 remove some already used nodes from S_i, whose time cost is at most $\sum_{i=1}^{l} S_i = O(n \cdot l)$. It does not affect the time complexity of Algorithm 5. The time complexity to get each community is $O(l \cdot (n \cdot \log(n) + m))$.

For the space complexity, we need to maintain up to $p \cdot l$ can-tuples in the pool, where p is the number of currently output communities. The space complexity for the other parts is all the same as in Algorithm 1. For each generated community, we have to record its core using space $O(l)$, so the total space to record all the generated cores is $O(l^2 \cdot p)$, and the total space complexity for Algorithm 5 to generate the k-th best answer is $O(l^2 \cdot k + l \cdot n + m)$. □

VI. Indexing and Graph Projection

As stated before, the time complexity for finding all/top-k communities for a user-given l-keyword query against a database graph, G_D, with Rmax, is polynomial delay of $O(l \cdot (n \cdot log(n) + m))$. However, when G_D is large in size, it is still costly to process *COMM-all*/*COMM-k* queries. In order to reduce the search space, in this section, we introduce an index that can be used to project a small graph $G_P \subseteq G_D$ to support l-keyword queries with radius up to R, which is the largest Rmax users can use. In brief, the result for a given l-keyword query against G_D is the same as the result for the same query against the projected database graph G_P. For an l-keyword query, the projected graph G_P can be much smaller than G_D.

Algorithm 6 *GraphProjection*($\{k_1, k_2, \cdots, k_l\}$, Rmax)

Input: the set of keywords $\{k_1, k_2, \cdots, k_l\}$, a radius Rmax.
Output: a projected graph G_P ($\subseteq G_D$).

1: $V' \leftarrow \emptyset$; $E' \leftarrow \emptyset$; $V_c \leftarrow \emptyset$; $W' = \emptyset$;
2: **for** $i = 1$ **to** l **do**
3: $W_i \leftarrow getNode(\mathsf{invertedN}, k_i)$;
4: $E_i \leftarrow getEdge(\mathsf{invertedE}, k_i)$;
5: $V_i \leftarrow W_i \cup \{u \mid (u, v) \in E_i \vee (v, u) \in E_i\}$;
6: $W' \leftarrow W' \cup W_i$;
7: $E' \leftarrow E' \cup E_i$;
8: $V' \leftarrow V' \cup V_i$;
9: $V_c \leftarrow V_i$ if $i = 1$, otherwise $V_c \cap V_i$;
10: let $G_s(V', E')$ be a virtually graph with a new virtual node s, and a set of new edges from (s, v), for $v \in V_c$, where $w_e((\mathsf{s}, v)) = 0$;
11: compute the shortest paths from s to others over G_s;
12: let $G_t(V', E')$ be a virtually graph with a new virtual node t, and a set of new edges from (v, t), for $v \in W'$, where $w_e((v, \mathsf{t})) = 0$;
13: compute the shortest paths from t to others, on G_t, by virtually considering $(u, v) \in E'$ as (v, u) (reverse the order);
14: $V_P \leftarrow \{v \mid v \in V' \wedge \mathsf{dist}(\mathsf{s}, v) + \mathsf{dist}(v, \mathsf{t}) \leq \mathsf{Rmax}\}$;
15: $E_P \leftarrow \{(u, v) | u \in V_P \wedge v \in V_P \wedge (u, v) \in E'\}$;
16: **return** $G_P(V_P, E_P)$;

We use two inverted indexes, invertedN and invertedE. For each keyword w in the database graph G_D, in the invertedN, it maintains an invert list to store the set of nodes, denoted V_w, where every node $v \in V_w$ contains the keyword w, and in the invertedE, it maintains the set of edges, (u, v), such that both u and v nodes are within R from at least one node in V_w. The node/edge weights are kept with the nodes and the edges in the two inverted indexes.

Next, we show how to use the two inverted indexes to project a database graph for an l-keyword query. Note that with the two inverted indexes, we do not need to use the underneath graph G_D, or in other words, the entire G_D can be constructed using the two inverted indexes.

The algorithm to project a subgraph of G_D for an l-keyword query within Rmax is shown in Algorithm 6. The main idea is the same as to find a community for a given core, as illustrated in Fig. 7 for a given set of *cnode*s (v_{12} and v_{11}) and a given set of *knode*s (v_{13}, v_8, and v_{11}). When projecting a graph G_P, in Algorithm 6, the set of *cnode*s becomes V_c and the set of *knode*s becomes W'. As shown in Algorithm 6, in a for loop (line 2-9), for every keyword k_i, it obtains the set of nodes that contain k_i, W_i, using $getNode(\mathsf{invertedN}, k_i)$ (line 3); and it obtains the set of edges, E_i, using $getEdge(\mathsf{invertedE}, k_i)$,

Parameter	Range	Default
KWF	.0003, .0006, .0009, .0012, .0015	.0009
l	2, 3, 4, 5, 6	4
Rmax	4, 5, 6, 7, 8	6
k	50, 100, 150, 200, 250	150

TABLE II

PARAMETERS FOR DBLP DATASET

KWF	Keywords
.0003	scalable, protocols, distance, discovery
.0006	space, graph, routing, scheme
.0009	environment, database, support, development, optimization, fuzzy
.0012	dynamic, application, modeling, logic
.0015	web, parallel, control, algorithms

TABLE III

KWF AND THE KEYWORDS USED IN DBLP

Parameter	Range	Default
KWF	.0003, .0006, .0009, .0012, .0015	.0009
l	2, 3, 4, 5, 6	4
Rmax	9, 10, 11, 12, 13	11
k	50, 100, 150, 200, 250	150

TABLE IV

PARAMETERS FOR IMDB DATASET

KWF	Keywords
.0003	summer, bride, game, dream
.0006	Friday, heaven, street, party
.0009	star, death, all, girl, lost, blood
.0012	city, American, blue, world
.0015	night, story, king, house

TABLE V

KWF AND THE KEYWORDS USED IN IMDB

where both ends of an edge are reachable from a node in W_i (line 4). Note that V_i is the neighborSet of W_i (line 5). After the for loop, it can project a subgraph $G'(V', E')$ of G_D to answer the given l-keyword query if Rmax $\leq$ R. But, it is still considered as large. Note that in the for loop, we also compute the set of centers, V_c ($\subseteq V'$), where every node $v \in V_c$ can reach at least a node v_i ($\in W_i$) which contains the keyword k_i. Based on the set of centers V_c, a smaller graph G_P is constructed using the similar ideas given in Algorithm 4. We omit the discussions due to the limit of space.

VII. PERFORMANCE STUDIES

We implemented two polynomial delay algorithms, Algorithm 1 and Algorithm 5, to find all/top-k communities. We denote them as PDall and PDk below. We also implemented four expanding-based algorithms. Two are based on bottom-up expanding, we denote them as BUall and BUk, for finding all/top-k communities, respectively. Similarly, two are based on top-down expanding, we denote them as TDall and TDk. All the algorithms were written in C++.

We tested the algorithms using two real datasets, DBLP (DBLP 2008) (http://dblp.uni-trier.de/xml/) and IMDB (http://www.grouplens.org/node/73). Both are used in the reported studies to test l-keyword queries. We use the same edge-weight function $w_e()$ as used in [2], [7], [3], $w_e((u, v)) = log_2(1 + N_{in}(v))$, where $N_{in}(v)$ is the in degree of node v.

For DBLP, there are 4 tables, Author(Aid, Name), Paper(Pid, Title, Other), Write(Aid, Pid, Remark), Cite(Pid1, Pid2). The numbers of tuples of the 4 tables are, $597K$, $986K$, $2426K$, and $112K$, respectively. The whole DBLP dataset consists of $4,121,120$ tuples and $5,076,826$ references. The database graph consists of $4,121,120$ nodes and $10,153,652$ directed edges (bi-directed). The parameters with their default values are shown in Table II. The l-keywords are selected from the keyword sets shown in Table III, with the associated KWF (keyword frequency).

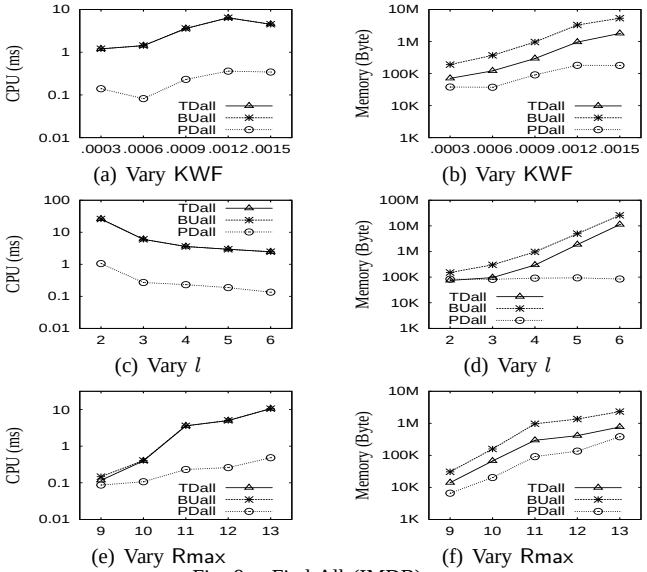
For IMDB, there are 3 tables: Users(UserID, Gender, Age, Occupation, Zip-code), Movies(MovieID, Title, Genres), and Ratings (UserID, MovieID, Rating, Timestamp). The numbers of tuples of the 3 tables are, $6.04K$, $3.88K$ and $1,000.21K$, respectively. The database graph consists of $1,010,132$ nodes and $4,000,836$ directed edges, which is denser then the DBLP graph. The parameters used and their default values are shown in Table IV. The keyword sets we used are shown in Table V, with their associated KWF.

The characteristics for the two datasets are different. In DBLP dataset, each author writes 4.06 papers on average while each paper is written by 2.46 authors on average. In IMDB dataset, each user evaluates 165.60 movies on average while each movie is evaluated by 257.59 users on average. This fact explains why we set the default Rmax to be 6 for DBLP and 11 for IMDB.

All experiments were conducted on a 1.60GHz Intel(R) Xeon(R) CPU and 2GB memory PC running windows server 2003. For all algorithms to be tested, we first project a database subgraph, for an l-keyword query, using the two inverted indexes (invertedN and invertedE), and test the algorithms. The maximum and average size of the projected graphs are 1.2% and 0.4% of the DBLP graph, and 1.8% and 0.5% of the IMDB graph, respectively. We significantly reduce the search space using the two inverted indexes. The elapsed time for constructing inverted indexes for DBLP and IMDB are 355 seconds and 24 seconds, respectively. The sizes of the total inverted indexes for DBLP and IMDB are 1,216 MB and 84 MB respectively, compared with the sizes of the raw datasets, 445 MB and 24 MB.

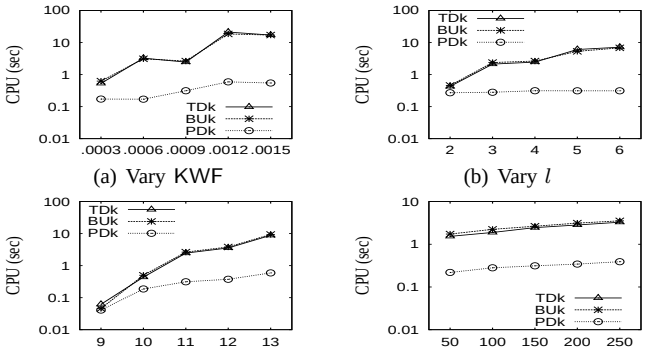
For testing algorithms PDall, BUall, and TDall, we report the average-delay, which is the total CPU time divided by the number of communities found, as used in [11] for the purpose of testing polynomial delay algorithms. For testing PDk, BUk, and TDk, we report the total CPU time for finding all k communities. We also report the maximum memory used in testing.

Exp-1 (IMDB): We compare PDall with BUall and TDall for finding all communities against IMDB. Fig. 9(a) and 9(b) show that, the more frequent the keyword is, the longer the average-delay is, and the larger memory it consumes. PDall is 10 times faster than BUall and TDall algorithms, and consumes the least memory. BUall consumes more memory than TDall does,



(a) Vary KWF (b) Vary KWF
(c) Vary l (d) Vary l
(e) Vary Rmax (f) Vary Rmax

Fig. 9. Find-All (IMDB)

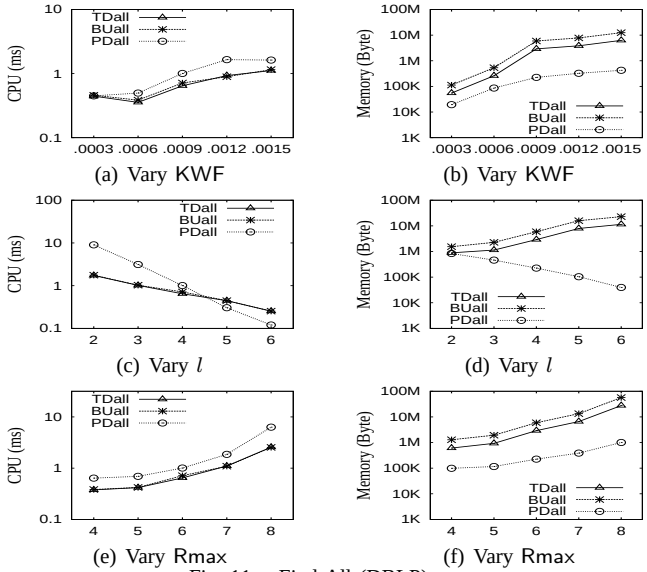


(a) Vary KWF (b) Vary l
(c) Vary Rmax (d) Vary k

Fig. 10. Find-TopK (IMDB)

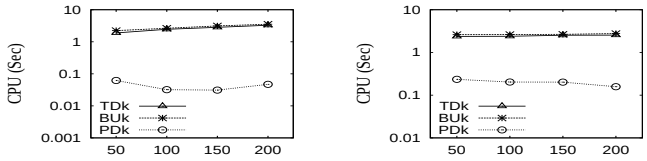
because each center node is associated with keyword node sets, which are the sets of keyword nodes that can be reached from the center. BUall needs to maintain all these sets whereas TDall can free the memory after outputs the communities found. When increasing l from 2 to 6, as shown in Fig. 9(c) and 9(d), the average-delay for all algorithms decreases, as expected. PDall is also faster than both BUall and TDall. The memory cost increases using BUall and TDall, because, when l increases, the number of resulting communities increases, both BUall and TDall need to maintain all the resulting communities. PDall consumes least memory and does not vary much, even when l increases. Fig. 9(e) and 9(f) show that, when Rmax increases, both the average-delay and the memory consumption increases for all three algorithms. PDall performs best among all.

We compare PDk with BUk and TDk for finding top-k communities against IMDB. As shown in Fig. 10(a), when KWF increases, the total time to get the top-k communities increases in most cases for all three algorithms, PDk performers best. Fig. 10(b) shows that, when l increases, the time for BUk and TDk increases, because the number of temporary results generated increases. PDk is consistent. In Fig. 10(c) and 10(d), when Rmax or k increases, the total time to get the



(a) Vary KWF (b) Vary KWF
(c) Vary l (d) Vary l
(e) Vary Rmax (f) Vary Rmax

Fig. 11. Find-All (DBLP)



(a) Vary k (IMDB) (b) Vary k (DBLP)

Fig. 12. Interactive TopK Test

top-k communities increases, for all three algorithms. PDk performs best. The memory consumptions for all tests are not large and do not change much. Due to space limit, we do not show the memory consumptions. As an indicator, the memory consumption for the default values of three algorithms are 80.47 KB (TDk),111.2 KB (BUk), and 91.16 KB (PDk).

Exp-2 (DBLP): We compare PDall with BUall and TDall for finding all communities against DBLP. In Fig. 11(a), 11(b), 11(e) and 11(f), for the memory consumption, PDall performs best, but for the average-delay, PDall is slower than both BUall and TDall, because, in the DBLP dataset, the probability for a set of keyword nodes to be centered at multiple nodes is very small, and most of the results have only one center. In this situation, the number of duplications generated by BUall and TDall is very small, which makes them faster to enumerate all communities. When KWF or Rmax increases, the average-delay and memory consumption for all three algorithms increases. Fig. 11(c) shows that, when the number of keywords l increases, the average-delay, for all three algorithms, decreases. PDall decreases faster. In Fig. 11(d), when l increases, the memory consumption for BUall and TDall increases, because they have to maintain all the results generated, and the number of results will increase when l increases. PDall consumes smaller memory when l becomes larger, because, when l increases, the size of the projected graph decreases for the DBLP dataset. We compare PDk with BUk and TDk for finding top-k communities against DBLP. They show the similar trends as they do for finding all communities due to the same reasons that the number of duplications is small in DBLP.

Exp-3 (Interactive TopK Test): We show that our algorithm (PDk) allows users to dynamically reset k values, when they are not sure how many communities they want to explore. With PDk, due to the nature of polynomial delay, we can continue the search and output more communities if user resets k to be a large value. For the other algorithms, BUk and TDk, they need to recompute a new l-keyword query with a larger k from the scratch. The reason is that BUk and TDk need to use the pruning rule to make it perform fast. But because BUk and TDk prune the communities if their ranking is lower than k, they cannot continue to find any communities when a large k is reset. Fig. 12 shows that PDk significantly outperforms BUk and TDk on both real datasets, when querying answers interactively. The setting is, a user will initially issue an l-keyword query and expect to find the top-k communities where k values are indicated on the x-axis in Fig. 12, and then the user wants to find the next 50 communities. PDk can continue to find 50 more. But, BUk and TDk need to do the same query twice, one is for the initial top-k, and the other is for top-(k+50).

VIII. Related Work

Keyword Search in RDB: Keyword search in *RDB* has been extensively studied recently. Most of the works concentrate on finding minimal connected tuple trees from *RDB* [2], [3], [1], [13], [14], [15], [4], [7], [5], [16]. There are two basic approaches. One uses SQL to find the trees in *RDB* [1], [13], [14], [15], [5]. The other materializes the relational database as a graph, and finds the trees over the graph [2], [3], [4], [7], [16]. In this work, we take the latter approach. As an exception, here is a recent reported study to find single-center graphs from *RDB* [17] using an index. The algorithm is not designed to find multi-center graphs as we studied in this paper. Their algorithm may miss some high ranked graphs if they are included in other graphs that are ranked low. And they do not consider duplication-free.

Finding Communities: There has in recent years been a great interest in finding communities in different kinds of networks, i.e. WWW, citation networks, email networks, and social networks. Gibson et al. in [18] view a community as containing a core of central, "authoritative" pages directly linked together by "hub" pages, which can be computed by HITS [19]. A survey can be found in [8] for finding Web communities. The problem we are studying in this paper is different. We find communities within a radius Rmax. In other words, in our problem, an authoritative and a hub may be indirectly connected over some paths.

Polynomial Delay: In [9], Johnson et al. studied the enumeration algorithms to enumerate all the result with duplication-free. In database community, the polynomial delay was discussed in [10], [11], [4] for different problems. In this paper, we propose new polynomial-delay algorithms for finding multi-center communities.

IX. Conclusion

In this paper, we focus on finding communities which are multi-center induced subgraphs for an l-keyword query over a database graph, G_D. Such a database graph is a materialized view of a relational database. A community contains much more information about how the tuples are connected than a connected tree does. We proposed new novel polynomial delay enumeration algorithms, for finding all/top-k communities. Our algorithms for finding both all/top-k communities are in the worst case, $O(l \cdot (n \cdot \log n + m))$, where n and m are the numbers of nodes and edges. The space complexity for finding all communities and finding top-k communities are $O(l \cdot n + m)$ and $O(l^2 \cdot k + l \cdot n + m)$, respectively. We also proposed an index approach to project a small subgraph out of G_D to process an l-keyword query, and allowed users to interactively enlarge k at run-time. We conducted extensive performance studies using two real datasets (DBLP and IMDB), and confirmed the efficiency of our new algorithms.

Acknowledgment: This work was supported by grants of RGC, Hong Kong SAR, China (No. 418206, CUHK 4161/07, and CUHK 4173/08).

References

[1] V. Hristidis and Y. Papakonstantinou, "Discover: Keyword search in relational databases," in *Proc. of VLDB'02*, 2002.
[2] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, "Keyword searching and browsing in databases using banks," in *Proc. of ICDE'02*, 2002.
[3] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar, "Bidirectional expansion for keyword search on graph databases," in *Proc. of VLDB'05*, 2005.
[4] B. Kimelfeld and Y. Sagiv, "Finding and approximating top-k answers in keyword proximity search," in *Proc. of PODS'06*, 2006.
[5] Y. Luo, X. Lin, W. Wang, and X. Zhou, "Spark: top-k keyword query in relational databases," in *Proc. of SIGMOD'07*, 2007.
[6] A. Markowetz, Y. Yang, and D. Papadias, "Keyword search on relational data streams," in *Proc. of SIGMOD'07*, 2007.
[7] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin, "Finding top-k min-cost connected trees in databases," in *Proc. of ICDE'07*, 2007.
[8] Y. Zhang, J. X. Yu, and J. Hou, *Web Communities: Analysis and Construction*. Springer Berlin Heidelberg, 2006.
[9] D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis, "On generating all maximal independent sets," *Inf. Process. Lett.*, vol. 27, no. 3, pp. 119–123, 1988.
[10] B. Kimelfeld and Y. Sagiv, "Efficiently enumerating results of keyword search," in *Proc. of DBPL'05*, 2005, pp. 58–73.
[11] S. Cohen, I. Fadida, Y. Kanza, B. Kimelfeld, and Y. Sagiv, "Full disjunctions: Polynomial-delay iterators in action," in *Proc. of VLDB'06*, 2006, pp. 739–750.
[12] E. L. Lawler, "A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem," *Management Science*, vol. 18, no. 7, pp. 401–405, 1972.
[13] V. Hristidis, L. Gravano, and Y. Papakonstantinou, "Efficient ir-style keyword search over relational databases," in *Proc. of VLDB'03*, 2003.
[14] S. Agrawal, S. Chaudhuri, and G. Das, "Dbxplorer: A system for keyword-based search over relational databases," in *Proc. of ICDE'02*, 2002.
[15] F. Liu, C. T. Yu, W. Meng, and A. Chowdhury, "Effective keyword search in relational databases," in *Proc. of SIGMOD'06*, 2006.
[16] H. He, H. Wang, J. Yang, and P. S. Yu, "Blinks: ranked keyword searches on graphs," in *Proc. of SIGMOD'07*, 2007.
[17] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou, "Ease: Efficient and adaptive keyword search on unstructured, semi-structured and structured data," in *Proc. of SIGMOD'08*, 2008.
[18] D. Gibson, J. M. Kleinberg, and P. Raghavan, "Inferring web communities from link topology," in *Hypertext*, 1998, pp. 225–234.
[19] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," in *Proc. of SODA'98*, 1998, pp. 668–677.