Duplication Free and Minimal Keyword Search in Large Graphs

Mehdi Kargar, Aijun An and Xiaohui Yu

Technical Report CSE-2013-02

February 4 2013

Department of Computer Science and Engineering
4700 Keele Street, Toronto, Ontario M3J 1P3 Canada
Duplication Free and Minimal Keyword Search in Large Graphs
Mehdi Kargar, Aijun An and Xiaohui Yu

Abstract—Keyword search over a graph searches for a subgraph that contains a set of query keywords. A problem with most existing keyword search methods is that they may produce duplicate answers that contain the same set of content nodes (i.e., nodes containing a query keyword) although these nodes may be connected differently in different answers. Thus, users may be presented with many similar answers with trivial differences. In addition, some of the nodes in an answer may contain query keywords that are all covered by other nodes in the answer. Removing these nodes does not change the coverage of the answer but can make the answer more compact. The answers in which each content node contains at least one unique query keyword are called minimal answers in this paper. We define the problem of finding duplication-free and minimal answers, and propose algorithms for finding such answers efficiently. Extensive performance studies using two large real data sets confirm the efficiency and effectiveness of the proposed methods.

Index Terms—Keyword Search, Graph Data, Polynomial Delay, Approximation Algorithm.

1 INTRODUCTION

Keyword search is a well known method for extracting relevant knowledge from a set of documents in information retrieval. Given a graph where nodes are associated with text, keyword search over the graph finds a subgraph that contains a set of query keywords. Due to the fact that many types of data can be represented by graphs, keyword search over graphs has received much attention in recent years. Most of the work in this area find minimal connected trees (e.g, Steiner trees with the minimum sum of edge weights [1], [2], [3], [4], [5]) or subgraphs that minimize a proximity function (e.g., the sum of distances from the nodes in the answer to a center node [6]). However, these methods may generate many trees or subgraphs with the same set of content nodes (i.e., nodes containing at least one query keyword) even though these answers may have different intermediate nodes connecting the content nodes.

The following example illustrates the duplication problem for a tree-based method. Suppose the nodes in an input graph are web pages. Two nodes are connected by an edge if there is a link from one page to the other. Consider Figure 1. The user is interested in finding pages that contain keywords $k_1$ and $k_2$. Two nodes $m_{k_1}$ and $n_{k_1}$ contain keyword $k_1$ and another two nodes $m_{k_2}$ and $n_{k_2}$ contain keyword $k_2$. The left graph in the figure contains 4 trees that cover $m_{k_1}$ and $m_{k_2}$, where each branch from $m_{k_1}$ to $m_{k_2}$ is a tree. The right graph contains a single tree that covers $n_{k_1}$ and $n_{k_2}$. Assume that the weight on each edge is the same. According to the ranking function used in the tree approaches, the tree that contains $n_{k_1}$ and $n_{k_2}$ in the right graph is produced after the first four trees that cover $m_{k_1}$ and $m_{k_2}$ on the left, because it has more edges than the other four trees. However, all the four trees on the left have the same set of content nodes.

1 If a user wants to explore different relationships among the content nodes, the method in [7] can be used to produce a set of Steiner trees that connect a set of specified nodes together.
where a node represents a store and contains the list of items that the store sells, and an edge between two nodes is weighted by the distance between the two stores. The customer issues a query specifying the set of items he/she wants to buy. It would be better that the search result is a list of stores in which each store has at least one unique item in the query that other stores do not have because there is no need to go to a store that does not have a unique item in the query. Another example is to determine required textbooks that together cover all the topics in a course. Assume that an online bookstore (e.g., Amazon.com) maintains its product information in an underlying graph where a node represents a book and contains the topics the book covers, and two books are connected by an edge if they share an author. Assume that the topics for a course are Smartphone Programming, Java Fundamentals, Object Oriented Programming and ASP.NET. A search over the graph allows us to find a set of books that not only covers all the topics but may also share the same author(s), which is preferred because the writing style of the books may be consistent. A possible answer to this query is shown in Fig. 2, where the three books share the same authors and together cover all the topics. But the topics covered by "Java How to Program" are also covered by the two other books. Thus, from the money-saving prospective it is not necessary to require the students to buy this book. In this type of applications, minimal answers are desired.

The following example shows the existing graph keyword search methods generate duplicate and non-minimal answers. Consider a small part of the DBLP dataset, which contains four authors and four papers. The paper titles, author names and a weighted graph that connect the authors and papers are shown in Figure 3. The edge weights are computed in the same way as in [6], [8]. Assume that the input keywords are $k_1$:dynamic, $k_2$:fuzzy, $k_3$:logic, $k_4$:design and $k_5$:optimization. Among all the subsets of the nodes, only $\{p_2,p_3\}$ covers all the input keywords and is also minimal. Other subsets either do not cover all the input keywords or are not minimal. The top-5 answers of the dynamic programming algorithm in [2] for finding Steiner trees are given in Table 1, which shows that all the answers contain the same set of content nodes, although they have different roots connecting the content nodes. The top-5 answers of the BLINKS algorithm [4] are shown in Table 2, which shows that the sets of content nodes of the last three answers are exactly the same. In addition, none of the five answers is minimal. The top-5 answers of the community-finding method [6] are shown in Table 3. The second column of the table presents the association of each keyword with a node in the answer and the third column shows the set of content nodes. Clearly, some of these top-5 answers are duplicated and some of them are not minimal.

In this paper, we first propose a new approach to keyword search in graphs that produces duplication-free answers. Each answer produced by our approach has a unique set of content nodes. We also define minimal answers, in which each node contains at least one input keyword that other nodes do not. We propose two algorithms that convert an answer to a minimal answer. We prove that the problem of finding a minimal answer while minimizing the proximity function that we use is NP-hard. Thus, one of the algorithms we propose is a greedy algorithm that searches for a sub-optimal minimal answer. We prove that this greedy algorithm has a bounded approximation ratio. Finally, for finding top-$k$ duplication-free and minimal answers, we propose two approaches. The first approach is faster but may miss some answers.
The second approach takes more time in theory but can produce all the answers if needed. Our extensive experiments show the efficiency and effectiveness of the proposed methods.

In the next section we discuss related work. In section 3, we give formal problem statements. In section 4, a procedure for finding duplication free answers in polynomial delay is presented. An algorithm for finding the best answer in each search space is given in section 5. Finding minimal answers is discussed in section 6. Other issues including graph indexing and presenting the answers are discussed in section 7. Experimental results are given in section 8. Section 9 concludes this work.

2 RELATED WORK

Most of the approaches to keyword search over graphs find trees as answers. In [1], a backward search algorithm for producing Steiner trees is introduced. A dynamic programming procedure for finding Steiner trees in graphs are presented in [2]. In [3], the authors propose algorithms that produce Steiner trees with polynomial delay. The algorithms follow the Lawler’s procedure [9]. Since finding Steiner trees is an NP-hard problem, producing trees with distinct roots is introduced in [5]. BLINKS improves the work in [5] by using an efficient indexing structure [4]. However, distinct root tree methods may miss some answers when top-k or all answers need to be produced because only one tree rooted at a node is considered in the algorithms. If another tree rooted at the same node also has a better weight than the trees rooted at other nodes, this tree is not considered.

To show that distinct root tree approach is not complete and might miss some combination of content nodes, an example is presented in Fig. 4. \(a_1\) and \(a_2\) contain keyword \(k_1\) and \(b_1\) and \(b_2\) contain keyword \(k_2\). The graph has the following five nodes: \(\{r, a_1, a_2, b_1, b_2\}\). Thus, the number of answers is at most five. Five answers with the associated root, the set of content nodes and their weights are presented in Table 4. The tree which has \(\{a_2, b_2\}\) as the content nodes is not produced. The reason is that any tree that has \(\{a_2, b_2\}\) as the content nodes has the weight of at least 4. Thus, this combination of content nodes is never produced using the distinct root semantics.

Table 2

<table>
<thead>
<tr>
<th>No.</th>
<th>Root</th>
<th>Leaf Nodes (Content Nodes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>((a_1, p_1), (a_2, p_4))</td>
<td>(a_1, a_2, p_1, p_4)</td>
</tr>
<tr>
<td>2</td>
<td>((a_1, p_2), (a_2, p_4))</td>
<td>(a_1, a_2, p_2, p_4)</td>
</tr>
<tr>
<td>3</td>
<td>((a_1, p_1), (a_2, p_1))</td>
<td>(a_1, a_2, p_1, p_1)</td>
</tr>
<tr>
<td>4</td>
<td>((a_1, p_2), (a_2, p_2))</td>
<td>(a_1, a_2, p_2, p_2)</td>
</tr>
<tr>
<td>5</td>
<td>((a_1, p_2), (a_2, p_4))</td>
<td>(a_1, a_2, p_2, p_4)</td>
</tr>
</tbody>
</table>

Table 3

<table>
<thead>
<tr>
<th>No.</th>
<th>Keyword-Node Association</th>
<th>Content Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>((k_1, p_1), (k_2, p_4))</td>
<td>(p_1, p_2, p_4)</td>
</tr>
<tr>
<td>2</td>
<td>((k_1, p_2), (k_2, p_4))</td>
<td>(p_1, p_2, p_4)</td>
</tr>
<tr>
<td>3</td>
<td>((k_1, p_1), (k_2, p_1))</td>
<td>(p_1, p_1, p_1)</td>
</tr>
<tr>
<td>4</td>
<td>((k_1, p_2), (k_2, p_2))</td>
<td>(p_2, p_2, p_2)</td>
</tr>
<tr>
<td>5</td>
<td>((k_1, p_2), (k_2, p_4))</td>
<td>(p_2, p_4, p_4)</td>
</tr>
</tbody>
</table>

Fig. 4. An example for showing the incompleteness of distinct root trees approach (e.g. BLINKS [4]). \(a_1\) and \(a_2\) contain keyword \(k_1\) and \(b_1\) and \(b_2\) contain keyword \(k_2\). Using distinct root semantics, the answer which has \(a_2\) and \(b_2\) as the content nodes is never produced.

Table 4

<table>
<thead>
<tr>
<th>No.</th>
<th>Root Node</th>
<th>Leave Nodes (Content Nodes)</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>r</td>
<td>({a_1, b_1})</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>(a_1)</td>
<td>({a_1, b_1})</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>(b_1)</td>
<td>({a_2, b_1})</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>(a_2)</td>
<td>({a_2, b_2})</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>(b_2)</td>
<td>({a_1, b_2})</td>
<td>3</td>
</tr>
</tbody>
</table>

Recently, the BROAD system is proposed to find diversified answers for keyword search on graphs [11]. The system is built on top of a keyword search engine and partitions the answer trees produced by the engine into dissimilar clusters. The dissimilarity between answers is measured based on the structural and semantic information of the given trees. The structural dissimilarity is measured based on the sub-tree kernel introduced in [12]. The semantic information is added to the kernel by merging the textual content of the nodes using the well known TF-IDF weighting scheme. A hierarchical browsing
method is further proposed to help users navigate and browse the results. Our effort of finding duplication-free answers can be considered as a special case of finding diversified answers, where each answer must have a different set of content nodes. Our method has cheaper computational cost due to its problem simplicity. We find such “diversified” answers during the search process, while BROAD does it as a post-processing process. BROAD can be applied to the results of our method to further diversify the answers using the BROAD’s dissimilarity measures.

Keyword search in graphs is closely related to finding a team of experts in social networks [13], [14]. Given a set of required skills, the purpose is to find a set of experts that together cover all of the required skills and also be able to communicate efficiently. The experts are connected together in a social network which is modeled as a graph. Our problem is also related to the graph pattern matching problem [15]. However, finding duplication free and minimal answers is not discussed in this area.

3 PROBLEM STATEMENT

Given a data graph whose nodes are associated with text and a query consisting of a set of keywords, the problem of keyword search in a graph is generally to find a subgraph that contains all or part of the keywords. The data graph can be directed or undirected. The edges and/or nodes may have weights on them. In this work, the same as [10], [2], [8], we consider undirected graphs with weighted edges, where two nodes are connected by an edge if there is a relationship between them and the edge weight represents the distance between the two nodes. Undirected graphs can be used to model different types of unstructured, semi-structured and structured data, such as web pages, XML documents and relational datasets. It should be noted that our approach is adaptable to work with directed graphs

Definition 1: (Answer) Given a graph $G$ and a set of query keywords ($Q = \{k_1, k_2, \ldots, k_l\}$), an Answer to $Q$ in $G$ is a set of content nodes in $G$ that together cover all of the input keywords in $Q$.

An Answer has a weight which can be defined according to the application need based on the weights of the edges in $G$ that connect the nodes in the Answer. The above definition does not require the nodes in an Answer to be connected with each other either directly or indirectly in $G$, but Answers with nodes connected to each other can be preferred over those with disconnected nodes by using a weight function.

Problem 1: (Duplication free keyword search) Given a graph $G$, an integer $k$ and a set $Q$ of query keywords, find top-$k$ unique Answers of $Q$ in $G$ whose weights are optimal.

An Answer is unique if it appears at most once in the top-$k$ list. The next definition deals with the minimality of the Answer.

Definition 2: (minAnswer) Given a graph $G$ and a set of query keywords ($Q = \{k_1, k_2, \ldots, k_l\}$), a minAnswer of $Q$ in $G$ is an Answer of $Q$ in $G$ in which each content node covers at least one query keyword that other content nodes do not cover.

In other words, each content node of a minAnswer uniquely contributes to cover at least one query keyword.

Problem 2: (Duplication free and minimal keyword search) Given a graph $G$, an integer $k$ and a set $Q$ of input keywords, find top-$k$ unique minAnswers of $Q$ in $G$ whose weights are optimal.

To focus on the generality of the above keyword search problems, we intentionally avoided defining the weight of an Answer in Definitions 1 and 2. Below we give two definitions of weight functions, used in [8], [4], [6], to measure the proximity of the nodes in an Answer. Note that other weight functions can be used with our definitions. Also, most of the algorithms proposed in this paper (i.e., Algorithms 1, 3, 5 and 6) are independent of the weight function. Only Algorithms 2 and 4 depend on the weight function.

Definition 3: (sumDistance) Suppose that the set of nodes in an Answer in graph $G$ is denoted as $V = \{v_1, v_2, \ldots, v_l\}$. The sumDistance of the Answer is defined as

$$\text{sumDistance} = \sum_{i=1}^{l} \sum_{j=i+1}^{l} \text{dist}(v_i, v_j)$$

where $\text{dist}(v_i, v_j)$ is the shortest distance between $v_i$ and $v_j$ in $G$, i.e., the sum of weights on the shortest path between $v_i$ and $v_j$ in $G$ [8].

Definition 4: (centerDistance) Suppose that the set of nodes in an Answer in graph $G$ is denoted as $V = \{v_1, v_2, \ldots, v_l\}$. The centerDistance of the Answer is defined as

$$\text{centerDistance} = \min_{c \in G} \sum_{i=1}^{l} \text{dist}(c, v_i)$$

where $\text{dist}(c, v_i)$ is the shortest distance between a node $c$ in $G$ and $v_i$. The node in $G$ that achieves the minimum distance is called the center of the Answer.

The centerDistance is used in [4], [6]. In [4], the center is the root of the answer tree. Note that the center may/may not be a node in the Answer.

When using sumDistance or centerDistance to define the weight of an Answer, Answers with smaller weights are considered to be better because the nodes in an Answer are closer to each other when its weight is smaller.


<table>
<thead>
<tr>
<th>Alg.</th>
<th>Complexity Type</th>
<th>Dup. Free</th>
<th>Minimal</th>
<th>Minimize $\text{sumDistance}$</th>
<th>Approx. Ratio</th>
<th>Complete</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alg. 2</td>
<td>Polynomial</td>
<td>Yes</td>
<td>No</td>
<td>Yes, bounded approx.</td>
<td>Yes</td>
<td>Yes</td>
<td>$O(l^2 \times</td>
</tr>
<tr>
<td>Alg. 3</td>
<td>Polynomial</td>
<td>No with Alg. 2</td>
<td>Yes</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
<td>$O(n^4)$</td>
</tr>
<tr>
<td>Alg. 4</td>
<td>Polynomial</td>
<td>No with Alg. 2</td>
<td>Yes</td>
<td>Yes, bounded approx.</td>
<td>(log $n$) $\frac{</td>
<td>max_k - min_k</td>
<td>}{k-1}$</td>
</tr>
<tr>
<td>Alg. 5</td>
<td>Polynomial</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A in general</td>
<td>N/A</td>
<td>No</td>
<td>$O(l^2 \times</td>
</tr>
<tr>
<td>Alg. 6</td>
<td>FPT</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A in general</td>
<td>N/A</td>
<td>Yes</td>
<td>$O([\delta_1(k)] \times l \times</td>
</tr>
</tbody>
</table>

Note: N/A means not applicable, $l$ is the number of query keywords, $n(\leq l)$ is the number of nodes in the input answer for Alg. 4, $d_{max}$ and $d_{min}$ are the max and min distances between any pair of nodes in the input answer for Alg. 4, $\sum_{i=1}^k |K_i| < l$, $D_{max}$ is the number of nodes in graph.

### 3.1 An Overview of the Proposed Algorithms

In this paper, we propose six algorithms to solve Problems 1 and 2. Algorithm 1 is a general framework for generating top-$k$ duplication-free answers by wisely dividing the search space. It calls Algorithm 2 (which is a $2$-approximation algorithm for finding a single answer that minimizes $\text{sumDistance}$) to find top-$k$ duplication-free answers in polynomial delay. Thus, Algorithms 1 and 2 together solve Problem 1.

To generate $\text{minAnswers}$, Algorithms 3 and 4 are proposed to convert the answers generated by Algorithm 2 into a $\text{minAnswer}$. Algorithm 3 does not optimize a weight function, while Algorithm 4 finds a $\text{minAnswer}$ that also minimizes the $\text{sumDistance}$ function with a bounded approximation ratio. However, simply converting Algorithm 2’s answer to a $\text{minAnswer}$ with Algorithm 3 or 4 may lead to generation of duplicate answers in the top-$k$ procedure.

To generate top-$k$ duplication-free $\text{minAnswers}$ (i.e., to solve Problem 2), Algorithms 5 and 6 are proposed to replace Algorithm 2 in Algorithm 1. Both algorithms are general frameworks for confining or dividing the search space to ensure minimality and no duplication in the top-$k$ answers generated by Algorithm 1. They call a modified version of Algorithm 2 (which calls Algorithm 4) to generate a $\text{minAnswer}$ that also minimizes $\text{sumDistance}$. The difference between Algorithms 5 and 6 is that Algorithm 5 is faster (completely polynomial) but may miss some answers, while Algorithm 6 is complete (i.e., it allows all the possible answers to be considered), but is a fixed-parameter tractable (FPT) algorithm.

### 4 Finding Top-$k$ Duplication Free Answers in Polynomial Delay

An efficient search engine should satisfy three properties [3]. First, it should be able to generate all answers without missing them. Second, the answers should be presented in an order with better answers ranked higher. Third, the search engine should produce the answers efficiently. Assume that the maximum number of nodes containing a query keyword in the input graph is $m$. Based on the definition of $\text{Answer}$, the total number of $\text{Answer}$s might be up to $m^l$, where $l$ is the number of query keywords. Apparently, producing all of the $\text{Answer}$s may overwhelm the user since $m$ and/or $l$ can be large. Thus, it is important to produce top-$k$ $\text{Answer}$s (or all the answers if fewer than $k$ answers exist) in a ranked order. The efficiency of a search engine is commonly measured based on the delay between producing two consecutive answers. If this delay is polynomial based on the input data, the algorithm is called a polynomial delay algorithm [16], [3].

Our algorithm for producing top-$k$ duplication-free answers is an adaption of Lawler’s procedure [9] for finding top-$k$ answers to discrete optimization problems. Lawler generalized Yen’s algorithm in [17] which finds the $k$ shortest loopless paths in a graph. In Lawler’s procedure, the search space is divided into disjointed sub-spaces. The best answer in each sub-space is found and used to produce the current best global answer. The sub-space that produces the best global answer is further divided into sub-subspaces and the best answer among its sub-subspaces is used to compete with the best answers in other sub-spaces in the previous level to find the next best global answer. Two main issues in this procedure are how to divide a space into subspaces and how to find the best answer within a (sub)space. To have duplication free answers, the procedure for dividing the search space into sub-spaces must produce disjoint sub-spaces so that the same answer cannot be generated from different sub-spaces.

Lawler’s procedure has been used to generate top-$k$ answers in graph keyword search in [6], [8], in which a search (sub)space is represented by $C_1 \times C_2 \times \cdots \times C_l$, where $C_i$ is the set of nodes containing query keyword $k_i$, and the space is divided by taking away certain node(s) from $C_i$ to form a subspace based on the best answer in the space being divided. A problem with this strategy is that a node taken away from $C_i$ may appear in $C_j$ (where $i \neq j$) if the node contains more than one query keyword (i.e., it belongs to more than one $C_i$ for $1 \leq i \leq l$), and thus the same set of content nodes may be generated from different subspaces if a node contains more than one query keyword, although different answers have different keyword-node associations. Since we aim at generating unique sets of content nodes, a different strategy for dividing a search space is needed to avoid duplicate answers.

We first illustrate our idea of dividing the search space into disjoint subsets using an example.

---

**TABLE 5**

An overview of Algorithms 2-6.
Given a set of input keywords, we first use the FindBestAnswer procedure (to be described later) to find the best answer \{a, b, c\} in the input graph G, where a, b, and c are nodes in G. Then we divide the set of remaining answers to be found into three subsets: (1) the answers that contain a and b but no c, (2) the ones that contain a but no b, and (3) the ones that contain no a. Clearly, (1), (2) and (3) are disjointed, and they, together with \{a, b, c\}, comprise the set of all possible answers. Each subset has constraints, which can be represented using an inclusion set containing the nodes that must be included and an exclusion set containing the nodes that must be excluded. Table 6 shows the constraints of these three subsets.

After dividing the search space into disjoint subsets based on the global best answer, the best Answer in each subspace is found using the FindBestAnswer procedure. These best Answers are inserted into a priority queue, where the Answers are ranked in ascending order of their weights. Obviously, the second best Answer is the one at the top of the priority queue. Suppose that this Answer is taken from \(SB_2\) and contains \(p\) content nodes. After returning the second best answer, \(SB_2\) is divided into \(p\) subspaces in the way similar to the one shown in Table 6. In each subspace, the best Answer is found and is added to the priority queue. At this state, the priority queue has 2 + \(p\) elements: two elements from the first step and \(p\) elements from this new step. Then, the top Answer is returned and removed from the queue, its corresponding space is divided into subspaces and the best Answer (if any) in each new subspace is added to the priority queue. This procedure continues until the priority queue becomes empty or top-\(k\) Answers are found.

The pseudocode of this procedure for enumerating top-\(k\) Answers is described in Algorithm 1. The algorithm first computes the set \(C\) of nodes that contain at least one input keyword. This can be easily done using a pre-built inverted index. In line 5, procedure FindBestAnswer (to be described in the next section) is called to find the best answer from the whole search space (i.e., \(C\)). It takes input graph \(G\), query \(Q\), \(C\), an inclusion set and an exclusion set as input, and returns the best answer in the search space specified by \(C\). Since the first best answer is found in the whole search space, empty inclusion and exclusion sets are passed to the procedure in line 5. If the best answer exists (i.e., \(A \neq \emptyset\)), \(A\), together with the inclusion and exclusion sets (the constraints for the space from which \(A\) is generated), are inserted into \(Queue\) in line 7. The \(Queue\) is maintained in the way that its elements are ordered in ascending order of their weights. The while loop starting at line 8 is executed until the \(Queue\) becomes empty or \(k\) answers have been outputted. In line 9, the top of the \(Queue\) is removed, which contains the best answer (\(A\)) in the \(Queue\) and its inclusion (\(Inc\)) and exclusion (\(Exc\)) sets. The answer in \(A\) is outputted. Then, if the number of answers has not reached \(k\), the nodes in \(A\) are assigned to \(n_1, n_2, \ldots, n_p\) where \(p\) is the number of nodes in \(A\). In lines 15-21, \(p\) new inclusion and exclusion sets are produced based on the nodes in \(A\) and the inclusion and exclusion sets for the space \(A\) was generated from. The new subspaces are specified by these new constraints. For each new subspace, if the intersection of its inclusion and exclusion sets is empty, the best answer is found and it is inserted into the \(Queue\) with the constraints of its related subspace. Clearly, if procedure FindBestAnswer finds the best answer in a search (sub)-space. In addition, the sub-spaces produced based on answer \(A\) are all disjoint and none of them contains \(A\). Therefore, they do not lead to

<table>
<thead>
<tr>
<th>Subspace</th>
<th>Inclusion set</th>
<th>Exclusion set</th>
</tr>
</thead>
<tbody>
<tr>
<td>SB₁</td>
<td>Inc₁ = {a, b}</td>
<td>Exc₁ = {c}</td>
</tr>
<tr>
<td>SB₂</td>
<td>Inc₂ = {a}</td>
<td>Exc₂ = {b}</td>
</tr>
<tr>
<td>SB₃</td>
<td>Inc₃ = {}</td>
<td>Exc₃ = {a}</td>
</tr>
</tbody>
</table>

Algorithm 1: Generate Duplication Free Top-\(k\) Answers

**Input:** the input graph \(G\); the query \(Q = \{k₁, k₂, \ldots, kₗ\}\); \(k\)

**Output:** the top-\(k\)-ordered Answers printed with polynomial delay

1. \(C \leftarrow \emptyset\) for storing content nodes
2. for \(i = 1\) to \(k\) do
3.   add the nodes in \(G\) containing \(k\) to \(C\)
4. \(Queue \leftarrow \emptyset\) an empty priority queue
5. \(A \leftarrow \text{FindBestAnswer}(G, Q, C, \emptyset, \emptyset)\)
6. if \(A \neq \emptyset\) then
7.   insert \((A, \emptyset, \emptyset)\) into \(Queue\)
8. while \(Queue \neq \emptyset\) do
9.   \((A, Inc, Exc) \leftarrow \text{top element of Queue}\)
10. print(\(A\))
11.   \(k \leftarrow k - 1\)
12.   if \(k = 0\) then
13.      return
14.   \{n₁, n₂, \ldots, nₚ\} \leftarrow \text{content nodes of} \(A\)
15.   for \(i = 1\) to \(p\) do
16.     \(Inc_i \leftarrow Inc \cup \{n₁, \ldots, n_{i-1}\}\)
17.     \(Exc_i \leftarrow Exc \cup \{n_{i+1}, \ldots, nₚ\}\)
18.     if \(Inc_i \cap Exc_i = \emptyset\) then
19.       \(A_i \leftarrow \text{FindBestAnswer}(G, Q, C, Inc_i, Exc_i)\)
20.      if \(A_i \neq \emptyset\) then
21.         insert \((A_i, Inc_i, Exc_i)\) into the right place of \(Queue\)

3. The idea of using the inclusion and exclusion sets to represent constraints is inspired by [18]. However, the constraints in [18] are described using edges (instead of nodes as in our approach) for finding a different type of answers.

4. This assumes that all of the subspaces contain at least one Answer. In some cases, the subspace does not have any Answer.
the same answer and the set of produced answers is duplication free. In addition, this duplication free search procedure is independent of the procedure for finding the best answer and the weight function used to measure the quality of an answer.

5 FINDING THE BEST ANSWER IN EACH SEARCH SPACE

Algorithm 1 calls the FindBestAnswer procedure to find the best answer in a search space specified by a set of content nodes and the constraints (i.e., the inclusion and exclusion sets). The best answer must contain the nodes in the inclusion set, excluding the nodes in the exclusion set and also have an optimal weight. Depending on the weight function used, FindBestAnswer can be designed differently. Below, we present an algorithm that produces an answer satisfying the constraints and minimizing the sumDistance function. We present a modification of this algorithm which minimizes the centerDistance later in this section.

5.1 Minimizing the sumDistance function

In [8] we proved that minimizing sumDistance is an NP-hard problem, with respect to the number of query keywords, and proposed an approximation algorithm that finds an answer with an approximation ratio of 2. The search space in that algorithm is a Cartesian product $C_1 \times C_2 \times \cdots \times C_l$, where $C_i$ is a subset of nodes containing keyword $k_i$ and excluding certain nodes. However, a node excluded from $C_i$ may appear in $C_j$ if the node contains both $k_i$ and $k_j$. Since our answers must completely exclude the nodes specified by the exclusion set, we modify the algorithm in [8] to consider the constraints specified by the inclusion and exclusion sets.

The pseudo-code of the modified algorithm, FindBestAnswer, is presented in Algorithm 2. It takes an input graph $G$, a query $Q$, a set of content nodes $C_i$, and the inclusion and exclusion sets (Inc and Exc) as input and produces the best (approximate) Answer as output in polynomial time. The algorithm approximates the sumDistance of an answer using the sum of distances from each node in the answer to a center node within the answer. In the pseudo-code, set $F$ is the search space, which consists of all the nodes in the inclusion set and the set of content nodes containing the query keywords not covered by the inclusion set and not belonging to the exclusion set. In the code, $D_i$ is the set of nodes that contain keyword $k_i$ (which is not covered by the inclusion set) but do not belong to the exclusion set. For each node $f_i$ in $F$, an answer is formed by using $f_i$ as the center and including all the nodes in the inclusion set and adding the node in each $D_i$ that is closest to $f_i$. The final answer is the one with the least sum of distances between each node in the answer and its center. In the code, $d(x,y)$ is the shortest distance between nodes $x$ and $y$, which can be efficiently obtained by consulting a pre-built index (described in [8])

Clearly, the answer produced by this algorithm satisfies the inclusion and exclusion constraints. Since all the nodes in $F$ have been considered as a center candidate, it can be proved that the sumDistance of the produced answer is no more than $2 \times (l-1) \times |F|$, where $l$ is the number of query keywords. Thus, the produced answer has a weight that is at most twice that of an optimal answer. The proof is similar to the one in [8]. We omit it here due to the space limit. The complexity of this algorithm is $O(|F| \times l \times D_{max})$ where $|F|$ is the size of the set $F$, $l$ is the number of query keywords and $D_{max}$ is the maximum size of $D_i$ for $1 \leq i \leq l$. Since $|F| \leq (l \times D_{max})$ and $|Inc| \leq (l-1) \times |F| = O(l \times D_{max})$. Thus, the complexity of Algorithm 2 is $O(l^2 \times D_{max})$.

5.2 Minimizing the centerDistance function

Authors of [4], [6] proposed algorithms to minimize the centerDistance function. Here, we briefly describe how to modify Algorithm 2 to work with the centerDistance function. In line 11 of Algorithm 2, all of the content nodes of the uncovered keywords and the inclusion set are checked for finding the best approximate Answer that minimizes sumDistance.

5. Using a pre-built index to obtain the shortest distance between nodes has been used in [10], [6], [8].
Algorithm 3: ConvertToMinAnswerGeneral - General Procedure

Input: the set of content nodes as Answer; the query Q
Output: a minAnswer
1: for each node ni in Answer do
2: K = ∅
3: for j = 1 to i − 1 do
4: K = K ∪ keywords(ni)
5: for j = i + 1 to size(Answer) do
6: K = K ∪ keywords(ni)
7: if keywords(ni) ⊆ K then
8: remove ni from Answer
9: return Answer

However, for minimizing centerDistance, each node in the input graph G should be considered as a center of a possible answer. Therefore, instead of browsing only the nodes in set F in line 11, the loop iterates through all of the nodes in graph G and checks each of them for finding the best center and its associated answer. Thus, the time complexity of the algorithm becomes \(O(N \times l \times |D_{max}|)\), where \(N\) is the number of nodes in graph G. Since \(N > F\), the revised Algorithm 2 for minimizing centerDistance is slower than Algorithm 2 for minimizing sumDistance. Note that the same as [4], [6], the modified Algorithm 2 for minimizing centerDistance returns the exact answer in polynomial time.

6 FINDING MINIMAL ANSWERS

Some of the Answers returned by Algorithm 2 and existing algorithms may not be a minAnswer. That is, the input keywords in some nodes of an Answer may all be covered by other nodes in the answer. If these nodes are removed from the answer, the remaining set of nodes still covers all the input keywords. Below we first present two algorithms for converting an Answer to a minAnswer. However, the converted minAnswer may violate the inclusion constraint for finding duplication-free answers. We then propose two approaches to solve the problem.

6.1 Generating Minimal Answers

The problem of finding a minimal answer from an Answer can be solved in polynomial time as shown in Algorithm 3. The algorithm checks each node in the Answer to see if the input keywords the node contains are all covered by other nodes. If yes, it removes the node. The complexity of this algorithm is \(O(n^2)\) where \(n\) is the number of nodes in the input Answer.

Lemma 1: Algorithm 3 produces a minAnswer in which each node contains at least one unique input keyword. In addition, all the input keywords are covered in the minAnswer.

Proof: Let \(n_i\) be a node in the answer produced by the algorithm. Assume that when the algorithm checks whether \(n_i\) should be removed, \(T\) was the intermediate answer at that time. Since \(n_i\) was not removed, \(\text{keywords}(n_i) \not\subseteq \text{keywords}(T - \{n_i\})\), where keywords\((n_i)\) is the set of input keywords \(n_i\) contains and \(\text{keywords}(T - \{n_i\})\) is the set of input keywords contained in \(T - \{n_i\}\). Since the output minAnswer is a subset of \(T\), minAnswer - \(\{n_i\}\) must be a subset of \(T - \{n_i\}\). Thus, \(\text{keywords}(n_i) \not\subseteq \text{keywords}(\text{minAnswer} - \{n_i\})\), which means that \(n_i\) contains at least one unique keyword that the rest of nodes in \(\text{minAnswer}\) does not contain. Also, since the algorithm only removes a node when its input keywords are completely covered by the rest of nodes in \(T\), the set of input keywords covered by \(T\) does not change after a node is removed from \(T\). Thus, the final minAnswer covers the same set of input keywords as the input Answer.

An Answer may contain multiple minAnswers. The answer returned by Algorithm 3 may not be optimal with respect to a weight function such as sumDistance. Below we first prove that the problem of finding a minAnswer with the minimum sumDistance is an NP-hard problem, and then present a greedy algorithm to solve the problem.

Theorem 1: The problem of producing a minAnswer from an Answer while minimizing sumDistance is NP-hard.

Proof: We prove the theorem by a reduction from the set cover problem. Given a set of \(m\) elements (universe) and \(n\) sets whose union is the universe, the set cover problem is to identify the smallest number of sets whose union still contains all elements in the universe. Consider the set of input keywords in our problem as a universe. The nodes in an Answer can be considered as the sets of keywords whose union is the universe because they cover all the input keywords. Assume that the shortest distance between each pair of nodes in an Answer is the same. Then finding a minAnswer from the Answer is equivalent to finding the minimal number of nodes that cover all the input keywords (i.e., the universe). This is because a minAnswer with a smaller number of nodes has a smaller sumDistance when the shortest distance between each pair of nodes is the same. Since the set cover problem is NP-hard [19], finding a minAnswer while minimizing sumDistance is NP-hard.

Theorem 2: The problem of producing a minAnswer from an Answer while minimizing centerDistance is NP-hard.

Proof: We prove the theorem by a reduction from the set cover problem. Given a set of \(m\) elements (universe) and \(n\) sets whose union is the universe, the set cover problem is to identify the smallest number of sets whose union still contains all elements in the universe. Consider the set of input keywords in our problem as a universe. The nodes in an Answer can be considered as the sets of keywords whose union is the universe because they cover all the input keywords. Assume that the shortest distance between each pair of nodes in the Answer and the center is the same. Then finding a minAnswer from the Answer is equiv-
Algorithm 4 \textbf{ConvertToMinAnswer} - Greedy Procedure for Minimizing \textit{sumDistance}

\textbf{Input:} the set of content nodes as $Answer$; the query $Q$  
\textbf{Output:} a \textit{minAnswer} with \textbf{(sub)optimal} \textit{sumDistance}

1: $A \leftarrow \emptyset$  
2: while $Q \neq \emptyset$ do  
3: \hspace{1em} select a node $n \in Answer$ that maximizes $|\text{keywords}(n) \cap Q|$ 
4: \hspace{1em} $Answer \leftarrow Answer \setminus \{n\}$ 
5: \hspace{1em} $Q \leftarrow Q \setminus \text{keywords}(n)$ 
6: \hspace{1em} $A \leftarrow A \cup \{n\}$ 
7: \hspace{2em} for each node $n_i$ in $A$ do  
8: \hspace{3em} calculate $n_i$’s sum of distances to all the other nodes in $A$ 
9: \hspace{2em} sort nodes in $A$ based on their sum of distances to other nodes in descending order and put them in a list $T$. 
10: \hspace{2em} $\text{minAnswer} = \text{ConvertToMinAnswerGeneral}(T, Q)$ 
11: return $\text{minAnswer}$

alient to finding the minimal number of nodes that cover all the input keywords (i.e., the universe). This is because a $\text{minAnswer}$ with a smaller number of nodes has a smaller \textit{centerDistance} when the shortest distance between each node and the \textit{center} is the same. Since the set cover problem is NP-hard [19], the problem of finding a $\text{minAnswer}$ while minimizing \textit{centerDistance} is NP-hard. 

Since the problem is NP-hard, we design a greedy algorithm to find a \textit{minAnswer} that may be sub-optimal in minimizing \textit{sumDistance}. The algorithm is presented in Algorithm 4. It first uses a greedy set-covering procedure (Lines 1-6) to reduce the number of nodes in $Answer$ while still covering all the input keywords. The procedure chooses nodes to form an answer $A$ as follows: at each stage, choose the node that contains the largest number of uncovered keywords. However, $A$ may not be a $\text{minAnswer}$ because the above procedure is a greedy procedure for minimizing the number of nodes. Thus, we further sort the nodes in $A$ based on their sum of distances to other nodes in descending order, and then call $\text{ConvertToMinAnswerGeneral}$ (i.e., Algorithm 3) to convert $A$ into a $\text{minAnswer}$. 

The complexity of the algorithm is $O(n^2)$, where $n$ is the number of nodes in the input $Answer$. Also, since the set-covering procedure (Lines 1-6) chooses nodes from $Answer$ until all the input keywords are covered and Lemma 1 states that the $\text{minAnswer}$ produced by $\text{ConvertToMinAnswerGeneral}$ covers all the input keywords, the $\text{minAnswer}$ produced by this algorithm covers all the input keywords. 

Theorem 3: Algorithm 4 generates a $\text{minAnswer}$ that minimizes $\textit{sumDistance}$ with the approximation ratio of $(\log n) \frac{d_{\text{max}}}{d_{\text{min}}}$ where $n$ is the number of nodes in the input $Answer$ and $d_{\text{max}}$ and $d_{\text{min}}$ are the maximum and minimum distances between any pair of nodes in $Answer$. 

Proof: Assume that the number of nodes of an optimal $\text{minAnswer}$ that minimizes $\textit{sumDistance}$ is $\text{opt}_n$ and the number of nodes of the $\text{minAnswer}$ produced by Algorithm 4 is $\text{approx}_n$. Also assume that the number of nodes of an optimal answer that minimizes the number of nodes (which is the objective of the set cover problem) is $\text{opt}_{\text{scn}}$ and the number of nodes of the approximate answer produced by lines 1-6 (i.e., the greedy set cover procedure) is $\text{approx}_{\text{scn}}$. It has been proved that the number of nodes of the answer obtained by the greedy set cover algorithm is at most $\log n$ times that of the optimal answer [19], where $n$ is the number of nodes in the input $Answer$. That is, $\text{approx}_{\text{scn}} \leq \log n \times \text{opt}_{\text{scn}}$. Since the later steps of Algorithm 4 may further reduce the number of nodes from the answer generated by the greedy set-cover procedure, $\text{approx}_n \leq \text{approx}_{\text{scn}}$. Also, it is obvious that $\text{opt}_{\text{scn}} \leq \text{opt}_n$. Thus, we have $\text{approx}_n \leq \log n \times \text{opt}_n$. For a query with $l$ keywords, the $\text{sumDistance}$s of the optimal and the approximation answers satisfy the following inequalities: 1) $\text{sumDistance}_{\text{opt}} \geq (\frac{l}{2} - l \cdot \text{opt}_n) \times d_{\text{min}}$ and 2) $\text{sumDistance}_{\text{approx}} \leq (\frac{l}{2} - l \cdot \text{approx}_n) \times d_{\text{max}}$ where $d_{\text{max}}$ and $d_{\text{min}}$ are the maximum and minimum distances between any pair of nodes in the $Answer$, respectively. Since $\text{approx}_n \leq \log n \times \text{opt}_n$, we have:

$$\frac{\text{sumDistance}_{\text{approx}}}{\text{sumDistance}_{\text{opt}}} \leq \frac{(\frac{l}{2} - l \cdot \text{opt}_n) \times d_{\text{max}}}{(\frac{l}{2} - l \cdot \text{approx}_n) \times d_{\text{min}}}$$

Therefore, the following can be easily derived:

$$\frac{\text{sumDistance}_{\text{approx}}}{\text{sumDistance}_{\text{opt}}} \leq (\log n) \frac{d_{\text{max}}}{d_{\text{min}}}$$

It should be noted that Algorithm 4 is guaranteed to generate a $\text{minAnswer}$. The approximation is in terms of minimizing the weight of $\text{minAnswers}$. 

Since the weight of a $\text{minAnswer}$ may be smaller than that of the $\text{Answer}$ the $\text{minAnswer}$ is generated from, Algorithm 4 should be called after line 26 of Algorithm 2 using $\text{answer} \leftarrow \text{ConvertToMinAnswer} (\text{answer}, Q)$. After that, the weight of the answer should be updated as well. Thus, in Algorithm 2 the generated $\text{minAnswer}$ of each candidate is used to compete with the $\text{minAnswers}$ of other candidates so that the $\text{minAnswer}$ with the smallest weight among the candidates can be returned by Algorithm 2. 

Since the number of nodes in $\text{Answer}$ is at most the number of input keywords, the time complexity of Algorithm 2 becomes $O(|F| \times (|D_{\text{max}}| \times l + l^2))$, where $l$ is the number of input keywords, $|D_{\text{max}}|$ is the maximum size of $D_i$ (the set of the nodes containing keyword $k_i$) and $|F|$ is the size of set $F$. As we discussed in previous section, $|F| = O(l \times |D_{\text{max}}|)$. Therefore, the time complexity of Algorithm 2 becomes $O(l^2 \times D_{\text{max}} \times (D_{\text{max}} + l))$. Since $l$ can be much smaller than $|D_{\text{max}}|$ ($l \ll |D_{\text{max}}|$), time complexity of Algorithm 4 is the same as Algorithm 2 and is equal to $O(l^2 \times |D_{\text{max}}|^2)$. 

The same strategy can be applied for minimizing \textit{centerDistance} in Algorithm 4. The only difference is that in line 8, the distance to the center node is taken into account and the sorting of line 9 is based on this distance. Similar to Theorem 3, the approximation ratio of the algorithm for minimizing the \textit{centerDistance} is $d_{\text{max}}' \frac{d_{\text{max}}}{d_{\text{min}}}$ where $d_{\text{max}}'$ and $d_{\text{min}}'$ are
the maximum and minimum distances between any nodes in the Answer\(^6\) and the center node, respectively.

**Theorem 4:** The above modification of Algorithm 4 generates a minAnswer that minimizes \( \text{centerDistance} \) with the approximation ratio of \( d_{\text{max}}' \) where \( n \) is the number of nodes in the input Answer and \( d_{\text{max}}' \) and \( d_{\text{min}}' \) are the maximum and minimum distances between any nodes in the Answer and the center node, respectively.

**Proof:** For a query with \( l \) keywords, the \( \text{centerDistances} \) of the optimal and the approximation answers satisfy the following inequalities: 1) \( \text{centerDistance}_{\text{opt}} \geq l \times d_{\text{min}} \) and 2) \( \text{centerDistance}_{\text{approx}} \leq l \times d_{\text{max}}' \) where \( d_{\text{max}}' \) and \( d_{\text{min}}' \) are the maximum and minimum distances between any nodes in the Answer and the center node, respectively. If center node is part of the Answer, it is excluded for computing \( d_{\text{min}}' \) because in that case \( d_{\text{min}}' \) is equal to zero. Therefore, the following can be easily derived:

\[
\frac{\text{sumDistance}_{\text{approx}}}{\text{sumDistance}_{\text{opt}}} \leq \frac{d_{\text{max}}'}{d_{\text{min}}'}
\]

\( \Box \)

### 6.2 Producing Top-k / All Minimal Answers

To generate all or top-\( k \) duplication-free minAnswers, Algorithm 1 is needed to divide the search space and call Algorithms 2 and 4 to find a minAnswer in each subspace. This procedure works fine for finding the first best minAnswer in the whole search space. However, for finding subsequent answers, the search space is divided into subspaces, each with inclusion and exclusion constraints, and the best answer from each subspace is generated to compete for the next best answer. This requires that the minAnswer generated from each subspace contains all the nodes in the inclusion set of that subspace. However, when generating a minAnswer from an Answer and when the inclusion set is not empty, Algorithm 4 may delete some of the inclusion nodes if their keywords are covered by other nodes in the Answer. This may lead to generating duplicate answers by Algorithm 1. The problem can be illustrated with the following example.

Consider the graph in Figure 5. It contains 6 nodes: \( a, b, c, d, e \) and \( f \). Assume that the query consists of 4 keywords \( k_1, k_2, k_3 \) and \( k_4 \). The first best answer generated by Algorithm 2 is \( \{a, b\} \). Since it is a minAnswer, the algorithm for finding an minAnswer also returns \( \{a, b\} \) as the first best minimal answer. For finding the second best answer, the search space is divided into two subspaces. The first subspace has the constraints \( \text{Inc}_1 = \{a\} \) and \( \text{Exc}_1 = \{b\} \) and the second one \( \text{Inc}_2 = \{\} \) and \( \text{Exc}_2 = \{a\} \). The procedure for finding the best answer in the first subspace returns \( \{a, c, d\} \) as the best Answer, which is then converted to \( \{c, d\} \) by Algorithm 4 as the minAnswer. Since node \( a \) is removed from the answer, the \( \text{Inc}_1 \) constraint is violated.

To solve this problem, we change Algorithm 2 so that all the inclusion nodes in the Answer produced by the algorithm must contain at least one unique input keyword. In this way, the inclusion nodes in the answer cannot be removed when converting the Answer to a minAnswer. Below we propose two approaches that use this strategy. The first one is called the incomplete approach. It is faster but may miss some answers. The second approach is called the complete approach. It considers all the answers but has higher time complexity than the first approach. The algorithms for both approaches are named FindMinimalAnswer below. They are called in Algorithm 1 at the places where Algorithm 2 was called. Both approaches are independent of the weight function used to measure the quality of the answer.

#### 6.2.1 Incomplete Approach

Based on the way the search space is divided in Alg. 1, the nodes in the inclusion set of a subspace are part of a previously-generated minAnswer. Thus, each node in an inclusion set has at least one unique keyword among other nodes in the set. If in Alg. 2 each \( D_i \) contains only the nodes that do not contain any keyword that an inclusion node contains, the inclusion nodes will keep their uniqueness and will not be removed when converting the Answer to a minAnswer. This is the idea of the incomplete approach.

The pseudo-code of this approach is presented in Algorithm 5. Its inputs are the same as the ones for Algorithm 2. It first collects the keywords covered by the inclusion nodes into \( \text{CovKeywords} \). Then it calls procedure \( \text{FindBestAnswerCovConstraint} \) to generate a minAnswer. Procedure \( \text{FindBestAnswerCovConstraint} \) is similar to procedure \( \text{FindBestAnswer} \) (i.e., Algorithm 2) with two differences. The first difference is that in addition to other inputs, it also takes set \( \text{CovKeywords} \) as input and in line 4 of procedure \( \text{FindBestAnswer} \)
the algorithm also excludes from $D_i$ all the nodes that contain a keyword in $CovKeywords$. Since $D_i$s store the candidate nodes to be added to the answer, this exclusion guarantees that no node with a keyword in $CovKeywords$ is added to the answer. The second difference is that the procedure calls Algorithm 4 after line 26 to convert a candidate answer to a minAnswer and then calculates the weight of the minAnswer. The best minAnswer is returned. In section 6.1, we have showed that calling Algorithm 4 within Algorithm 2 does not change the complexity of Algorithm 2. Thus, the time taken by Algorithm 5 is the same as Algorithm 2 and is equal to $O(l^2 \times |D_{max}|^2)$.

Algorithm 5 FindMinimalAnswer, Incomplete Approach

Input: the input graph $G$; the query $Q$; the set of content nodes $C$; the set of inclusion nodes $Inc$; the set of exclusion nodes $Exc$.

Output: the best minAnswer satisfying both $Inc$ and $Exc$ constraints.

1. CovKeywords $\leftarrow$ set of keywords covered by $Inc$.
2. minAnswer $\leftarrow$ FindBestAnswerCovConstraint($G, Q, C, Inc, CovKeywords$).
3. return minAnswer.

However, Algorithm 5 may miss some answers because it puts a too strong constraint on the search space and removes some good candidate nodes. Consider the example in Figure 5. The best answer in the first subspace ($Inc_1 = \{a\}$ and $Exc_1 = \{b\}$) is set $\{a,e\}$. However, since $a$ belongs to the inclusion set, Algorithm 5 removes all of the nodes that contain a keyword in $a$, i.e. $k_1$ or $k_2$. Thus, $e$ is removed from the search space because it contains $k_2$. Therefore, Algorithm 5 is not able to produce answer $\{a,e\}$. It produces $\{a,f\}$, which has higher weight than $\{a,e\}$.

6.2.2 Complete Approach

To solve the missing-answer problem of the incomplete approach, we propose the complete approach. Since each node in the inclusion set has at least one unique keyword, we first compute the set of unique keywords for each node in the inclusion set and then calculate the Cartesian product of these sets. For example, if $Inc = \{a,b\}$ and $a$ and $b$ uniquely contain $\{k_1, k_2\}$ and $\{k_3, k_4\}$ respectively, the Cartesian product of $\{k_1, k_2\}$ and $\{k_3, k_4\}$ is $\{k_1, k_3\}, \{k_1, k_4\}, \{k_2, k_3\}$ and $\{k_2, k_4\}$. Then, for each set $s$ in the Cartesian product, procedure FindBestAnswerCovConstraint is called with $s$ as the input value for CovKeywords to generate a minAnswer whose non-inclusion nodes do not contain any keyword in $s$. Among all of the minAnswers (each generated based on an element in the Cartesian product), the best minAnswer is returned as the solution.

The pseudo-code of the complete approach is presented in Algorithm 6. It first gets the set of inclusion nodes as $\{n_1, n_2, \ldots, n_s\}$. Then, for each content node $n_i \in Inc$, it gets the unique keywords covered by $n_i$ and stores them in $K_i$. The Cartesian product of $\{K_1, K_2, \ldots, K_s\}$ is calculated and stores in CovKeywords in line 3. For each member CovKeywords$_i$ of CovKeywords, a minAnswer is found by calling FindBestAnswerCovConstraint in line 8. Procedure FindBestAnswerCovConstraint is the same as the one used in the incomplete approach. It finds a minAnswer and makes sure that its non-inclusion nodes do not contain any keywords in CovKeywords$_i$. If the minAnswer is not NULL and its weight outperforms previous minimal answers, leastWeight and bestMinAnswer are updated accordingly. The algorithm returns the minAnswer with the smallest weight among all the minAnswers corresponding to the members of the Cartesian product.

Algorithm 6 FindMinimalAnswer, Complete Approach

Input: the input graph $G$; the query $Q$; the set of content nodes $C$; the set of inclusion nodes Inc; the set of exclusion nodes Exc.

Output: the best minAnswer satisfying both Inc and Exc constraints.

1. $\forall i, 1 \leq i \leq s, K_i$ -- unique keywords of $n_i$.
2. CovKeywords $\leftarrow$ Cartesian product of $\{K_1, K_2, \ldots, K_s\}$.
3. leastWeight $\leftarrow$ $\infty$.
4. bestMinAnswer $\leftarrow$ NULL.
5. for $i$ in size(CovKeywords) do
6. CovKeywords$_i$ $\leftarrow$ CovKeywords.$get(i)$
7. minAnswer $\leftarrow$ FindBestAnswerCovConstraint($G, Q, C, Inc, CovKeywords$).
8. if minAnswer $\neq$ NULL then
9. weight $\leftarrow$ weight of minAnswer.
10. if weight $\leq$ leastWeight then
11. leastWeight $\leftarrow$ weight.
12. bestMinAnswer $\leftarrow$ minAnswer.
13. return bestMinAnswer.

Since in each element CovKeywords$_i$ of the Cartesian product, each inclusion node has a unique keyword, the keyword will remain unique in the Answer generated by FindBestAnswerCovConstraint because the nodes containing that keyword will not be added to the Answer. Hence, the inclusion nodes in the Answer cannot be removed when converting the Answer to the minAnswer. Therefore, Algorithm 6 does not violate the inclusion constraint. In addition, since all possible combinations of the unique keywords of the nodes in the inclusion set are evaluated, no answer is missed. For the example in Figure 5, the inclusion set of the first subspace is $Inc_1 = \{a\}$. Since $a$ contains keywords $k_1$ and $k_2$, the Cartesian product CovKeywords is $\{\{k_1\}, \{k_2\}\}$. When $\{k_1\}$ is used as the value for CovKeywords when calling FindBestAnswerCovConstraint, $\{a,e\}$ is returned as the minAnswer, which is the best answer in the subspace that was missed by the incomplete approach.

The time complexity of the algorithm is $O((|\prod_{i=1}^{s} |K_i|) \times l^2 \times |D_{max}|^2)$, where $s$ is the average number of nodes in an inclusion set and $|K_i|$ is the number of unique input keywords in the $i$th inclusion node. Note that $\sum_{i=1}^{s} |K_i| \leq l - 1$, where $l$ is the number of input keywords. When the number of input keywords is small, the maximum cardinality of the Cartesian product is small. For example, for six keywords, the worst case happens when the inclusion set contains two nodes, one containing 3
unique keywords and the other containing 2 unique keywords. In this case, \( \prod_{i=1}^{l} K_i = |K_1| \times |K_2| = 6 \). Similarly, when \( l = 3, 4, 5 \) or 7, \( \prod_{i=1}^{l} K_i \) is at most 1, 2, 4, or 8, respectively. Thus, since the number of query keywords is usually small in practice, Algorithm 6 is fixed-parameter tractable (FPT) [20].

7 Discussion of Some Issues

7.1 Graph Indexing

In Algorithms 2 and 4, we need to compute the shortest distance between two nodes in the input graph. Calculating the shortest path while searching for the best answer is expensive. Since the shortest distance between any two nodes in a graph is independent of the query, we pre-built an index that stores the shortest distances between nodes. A straight forward indexing method is to calculate and store the shortest path between each pair of nodes. However, this index needs \( O(n^2) \) storage, where \( n \) is the number of nodes in graph \( G \). This index is very large and not feasible for graphs with a large number of nodes. We use the neighbor indexing method in [8] to pre-compute and store the shortest distances and paths for the pairs of nodes whose shortest distance is within a certain threshold \( r_{max} \). Details on this indexing method can be found in [8]. Note that the idea of indexing the graph using a distance threshold has also been used in [6], [10].

7.2 Presenting the Answers

The Answers and minAnswers produced by our algorithms are a set of content nodes. Often it is important to see how these nodes are connected to each other in the input graph. The neighbor index that we use stores not only the shortest distances but also the shortest paths between nodes. Thus, the relations between the nodes can be revealed using the index. In this work, we use two approaches to revealing the relationships between nodes in an answer. In the first approach, a Steiner tree that connects the nodes in an answer with the minimum weight is created after the answer is generated and the user indicates that he/she would like to see the connections. Figures 11 and 12 depict two trees created by our tree-generating procedure for two answers used in our user study (to be described later). Note that generating a Steiner tree from an answer is much faster than generating a tree directly from the input graph. We use an algorithm in [8], [21] to generate the Steiner tree. In the second approach a multi-center sub-graph is generated to reveal the relations among content nodes in an answer. A center for each answer is any node in the graph with the distance up to \( r \) to any content node in the answer [6]. A path between each pair of content nodes and centers is added to the sub-graph. The advantage of using multi-center graphs rather than trees is that it reveals more relations. The disadvantages is that it might add some irrelevant nodes to the answer [8] and the size of an answer can be overwhelmingly large. Some other methods can also be used.

8 Experimental Results

We implemented all the algorithms presented above. In addition, for the purpose of comparison and showing that previous approaches produce duplicate and non-minimal answers, we implemented four algorithms in the literature: Dynamic [2], BLINKS [4], Community [6], and \( r \)-clique [8]. All of the algorithms are implemented in Java. The experiments are conducted on an Intel(R) Core(TM) i7-2720QM 2.20GHz computer with 16GB of RAM. 7.

8.1 Data Sets and Queries

Two real world data sets, DBLP and IMDb, are used in our experiments. The DBLP graph is produced from the DBLP XML data8. The dataset contains information about a collection of papers and their authors. It also contains the citation information among papers. Papers and authors are connected together using the citation and authorship relations. The numbers of tuples of the 4 relations author, paper, authorship and citation are 947K, 2,578K, 5,221K, and 112K respectively. Keyword search over a DBLP graph is useful to find, e.g., a set of papers related to an author that covers a list of topics. The papers found in this way are more likely related to each other via authors. We used two approaches for assigning weights to the edges of the graph. In the first approach, the weight of the edge between two nodes \( v \) and \( u \) is \( \log_2 (1 + v_{deg}) + \log_2 (1 + u_{deg})/2 \), where \( v_{deg} \) and \( u_{deg} \) are the degrees of nodes \( v \) and \( u \) respectively. This approach is called logarithmic edge weight and was used in [8], [6], [5], [2]. The second approach simply assigns the uniform weight of 1 to each edge. It is called uniform edge weight and was used in [10]. The set of input keywords used in our experiments and their frequencies in the input DBLP graph are shown in Table 7. The queries used in our experiments

7. The reason for using a 16GB RAM is that the Dynamic method stores the whole graph in the main memory. Other methods including ours use proper indexing, for which smaller RAM can be used.
8. http://dblp.uni-trier.de/xml/
are randomly generated from this set of keywords with the constraint that in each query all keywords have the same frequency (in order to better observe the relationship between run time and keyword frequency). Note that the input keywords shown in Table 7 were generated by the authors of [6] and used to generate queries in [6], [8]. We use the same set of input keywords and the same way to generate queries to make our results comparable to others.

The IMDb dataset contains the relations between movies and the users of the IMDb website that rate the movies. The numbers of tuples of 3 relations user, movie and rating are 6.04K, 3.88K and 1,000.21K, respectively. The edges of the graph are weighted in the same way as for the DBLP graph. Due to the space limit, for the IMDb dataset we only present the results of query with keywords house, king, night, city, city, world and story. The same set of input keywords is used in [6], [8].

8.2 Duplication of Previous Approaches

Our top-k method is guaranteed to generate duplication-free answers. In this section, we show the rates of duplicate answers of previous methods. Figure 6 shows the percentage of duplicate answers for four previous methods on the DBLP dataset with two different edge weights, different values of keyword frequency, different numbers of query keywords and different k values. Two answers are considered duplicates if they have the same set of content nodes. The rate of duplicate answers in the Dynamic method [2] is higher than BLINKS [4], Community [6] and r-clique [8]. This is because it finds minimum cost connected trees, and in most of the cases, the same set of content nodes are connected via different connections. BLINKS also has a high rate of duplication. It is due to its policy of defining trees based on a unique root. The same set of content nodes may have a different root. The Community and r-clique methods have the smallest rate of duplication among the existing methods because they divide the search space more wisely. But they still have some duplications. By increasing the frequency of keywords, the duplication rate of Dynamics and BLINKS increases. By increasing the number of keywords, the duplication rate generally decreases for Dynamic and BLINKS. Changing the value of k does not have a significant effect on the duplication rate. All these previous methods have duplications for any value of k in the top-k answers.

The percentage of duplicate answers for 4 different methods on the IMDb dataset is shown in Figure 7, in which the edge weights are logarithmic and $r_{max}$ is 11 for the Community and r-clique methods. The Community and r-clique methods do not produce any duplicate answer for the queries used due to small numbers of content nodes (e.g., only 23 nodes contain keyword house). In addition, for 5 and 6 keywords, the duplication rate of all methods is close to zero due to small numbers of content nodes.

8.3 Non-Minimality of Previous Approaches

Both the complete and incomplete approaches proposed in this paper are guaranteed to generate only minAnswers. In Figure 8 we show the rates of non-minimal answers of four previous methods on the DBLP dataset with two different edge weights. The rates of non-minimal answers in Community and r-clique are higher than those of BLINKS and Dynamic. This is because for each keyword, Community finds the closest keyword holder to the center of the community. However, the keyword may be covered by another node associated with the another keyword in the answer. This leads to non-minimal answers. The similar scenario occurs for the results of r-clique. In Dynamic, when merging two
trees, their keywords cannot overlap. This leads to a very small rate of non-minimality. This is also valid on the IMDB dataset (Figure 7).

8.4 Run-time Comparison

One way to produce duplication-free answers is to post-process the answers generated from a keyword search method by removing the duplicates. In this section, we would like to see if our approach (which avoids generating duplicates) is faster than using the post-pruning method. Below we compare the run time of our methods to that of the r-clique and Community methods with the post-pruning procedure. When comparing with Community, we use the centerDistance function which is the weight function used in Community. Note that minimize the centerDistance function is slower than minimizing sumDistance. Note that the centerDistance weight function is also used in BLINKS. We do not compare with Dynamic because Dynamic is too slow for its results to be put into the same graph with others. We do not directly compare with the original BLINKS algorithm because BLINKS generates much fewer answers than others. That is, if we allow all the methods to generate all the possible answers, BLINKS only generates a subset of them while ours generates them all (i.e., BLINKS misses some answers). Thus, due to the incompleteness of BLINKS, we do not compare with the post-pruning version of original BLINKS, but its weight function is used in the Community method to compare with our approach with a modified Alg. 2 that minimizes BLINKS' weight function.

Figure 9 shows the run time of different methods on DBLP with the logarithmic edge weight. The first method is r-clique (or Community in the second chart) which may generate duplicate and non-minimal answers. PP-Dup-Free refers to the r-clique (or Community in the second chart) method that post-prunes duplicate answers. PP-Dup-Free&Minimal refers to the r-clique (or Community in the second chart) method that post-prunes both duplicate and non-minimal answers. Dup-Free refers to our procedure for finding duplication free answers (i.e., Algorithms 1 and 2). The last two methods refers to our two approaches for finding duplication-free and minimal answers: the incomplete and complete approaches. To make fair comparisons, all of the methods use the same indexing method described in [8]. All the run times are the average time for producing one answer and presented in the logarithmic scale.

The run time of r-clique and Community are slower than our duplication free method (Dup-Free) for all the three different settings. Since they both use the same proximity measure, it seems to be a surprise. However, it is due to the fact that our method divides the search space into sub-spaces more wisely. The number of subspaces is usually smaller in our method. For example, for four keywords, assume that the best answer A contains only two nodes. The r-clique and Community methods divides the search space into four subspaces (equal to the number of keywords). But our procedure divides the search space into two subspaces (equal to the number of nodes in A). Since the number of nodes is always no larger than the number of keywords, we gain better performance.

The results show that finding duplication free answers with post-processing is two to four times slower than our procedure. Finding duplication free and minimal answers using post processing is three to ten times slower than each of our approaches. By increasing the frequency of keywords, the number of keywords or the value of \( r_{max} \), the run time increases. In addition, the run time (for producing one answer) does not change when the value of \( k \) changes. It shows that they all scale well with any number of required answers.

8.5 Incomplete vs. Complete Approaches

The incomplete approach is faster in theory but it may miss some answers. On the other hand, the complete approach can produce all answers, but is slower. Figure 9 shows that for up to 6 keywords, the run time difference between the two approaches is less than 5% (which may be hard to see on the log scale in Figure 9). This is due to the small cardinality of the Cartesian product when the number of keywords is small and also because the worse case rarely happens in practice. For 7 to 10 keywords, our experiments show that the run time difference is up to 20%. In terms of missing answers, based on our experiments, the incomplete approach misses few answers for up to six keywords.
8.6 The Quality of the Approximation Algorithm for Producing Minimal Answers

To evaluate the quality of the $\text{minAnswer}$ generated by the greedy Algorithm 4, we used exhaustive search to find the optimal (exact) answer that minimizes $\text{sumDistance}$. Figure 10 shows the average weight of the answers produced by the exact and greedy algorithms for different values of $k$. The results show that the difference of the two algorithms is at most 10% in practice, suggesting the high quality of the proposed greedy algorithm. Similar results are obtained for the $\text{centerDistance}$ function.

8.7 The Quality of the Minimal Answers

Finding duplication free answers is well motivated. Clearly, users prefer answers without duplication. However, it may be unclear whether users prefer minimal (more compact) over non-minimal (less compact) answers. To investigate this issue, we conducted a user study that compares the duplication free and non-duplication free answers. The results show that users prefer answers without duplication. Clearly, finding duplication free answers is well motivated.
this purpose we used 4 meaningful queries for the DBLP dataset as shown in Table 8 and applied our Algorithms 1 and 2 to find duplication-free answers. We collected the first 10 non-minimal answers from the top-100 answers for each query, and used Algorithm 4 to convert them into minimal answers. We asked 8 users (who are graduate students in computer science but not involved with this work) to compare each pair of non-minimal and minimal answers by giving each answer a relevance score between 0 and 1 with 1 meaning completely relevant and 0 completely irrelevant to the query. Each answer is presented to the user as a Steiner tree generated using the first answer presentation method discussed in Section 3. Figure 11 shows a tree generated from a non-minimal answer for the first query (i.e. "parallel graph optimization algorithm") and Figure 12 shows the tree for its corresponding minimal answer.

For each answer we use the average of the relevance scores of both of the answer sets are very close. The result suggests that the IR scores of the minimal answers are generally higher than the non-minimal answers (except for the third query in which the IR-scores of both of the answer sets are very close.).

9 Conclusion

We have proposed novel and efficient methods for keyword search in graphs. A problem with existing approaches is that they may produce duplicate answers that have the same set of content nodes with trivial differences in their connections. To address this problem, we introduced a procedure that produces duplication free answers by wisely dividing the search space. In addition, since users are usually interested in exploring more compact answers [8] and in some applications (such as textbook selection) answers with unique contributions from each node are preferred, we defined minimal answers and proposed two algorithms for converting an answer to a minimal answer and two approaches to finding top-k or all duplication free and minimal answers. Our algorithms are guaranteed to generate duplication-free and minimal answers. We presented the rates of duplicate and non-minimal answers produced by previous approaches. We compared the run-time of our proposed methods to that of using post-pruning techniques to remove duplicate answers. We showed that our approaches are faster than post-pruning techniques. We also showed that our greedy algorithm for minimizing the weight of a minimal answer produces minimal answers whose weights are close to the optimal weights produced by the exact algorithm. Finally, we show

\[
\text{CombinedScore}(A, Q) = \frac{\sum_{i=1}^{p} \text{Score}(v_i, Q)}{p}
\]

where, for a word \( k \) that appears in both \( v \) and \( Q \), \( tf \) is the frequency of \( k \) in \( v \), \( df \) is the number of nodes of the same type as \( v \) that contains \( k \), \( cs \) is the size of \( v \) in characters, \( AV_{cs} \) is the average size of all of the nodes with the same type as \( v \) in characters, \( N \) is the total number of nodes with the same type as \( v \) and \( s \) is a constant. The same as in [22], we set \( s \) to 0.2. Then, the combined score of the answer \( A \) that contains \( p \) content nodes is calculated as follows:
that the minimal answers have higher quality than non-minimal answers through a user study and a state-of-the-art IR weighting function. Our user study indicates that users prefer minimal answers to non-minimal ones.

REFERENCES