Keyword Search in Graphs: Finding r-cliques

Mehdi Kargar and Aijun An
Department of Computer Science and Engineering
York University, Toronto, Canada
{kargar,aan}@cse.yorku.ca

ABSTRACT

Keyword search over a graph finds a substructure of the graph containing all or some of the input keywords. Most of previous methods in this area find connected minimal trees that cover all the query keywords. Recently, it has been shown that finding subgraphs rather than trees can be more useful and informative for the users. However, the current tree or graph based methods may produce answers in which some content nodes (i.e., nodes that contain input keywords) are not very close to each other. In addition, when searching for answers, these methods may explore the whole graph rather than only the content nodes. This may lead to poor performance in execution time. To address the above problems, we propose the problem of finding r-cliques in graphs. An r-clique is a group of content nodes that cover all the input keywords and the distance between each two nodes is less than or equal to r. An exact algorithm is proposed that finds all r-cliques in the input graph. In addition, an approximation algorithm that produces r-cliques with 2-approximation in polynomial delay is proposed. Extensive performance studies using two large real data sets confirm the efficiency and accuracy of finding r-cliques in graphs.

1. INTRODUCTION

Keyword search, a well known mechanism for retrieving relevant information from a set of documents, has recently been studied for extracting information from structured data. Structured data are usually modeled as graphs. For example, considering IDREF/ID as links, XML documents can be modeled as graphs. Relational databases can also be modeled using graphs, in which tuples are nodes of the graph and foreign key relationships are edges that connect two nodes (tuples) to each other [6, 12]. In such models, keyword search plays a key role in finding useful information for the users. Users usually do not have sufficient knowledge about the structure of data. In addition, they are not familiar with query languages such as SQL. Thus, they need a simple system that receives some keywords as input and returns a set of nodes that together cover all or part of the input keywords. A node that contains one or more keywords is called a *content node*.

Proceedings of the VLDB Endowment, Vol. 4, No. 10

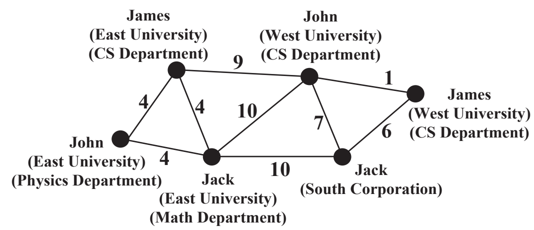

Figure 1: A sample graph. The shortest distance between a pair of nodes is shown on their edge.

Most of the work in keyword search over graphs finds minimal connected trees that contain all or part of the input keywords [15]. A tree that covers all the input keywords with the minimum sum of edge weights is called *Steiner tree*[1]. Tree-based methods produce succinct answers. Recently, methods that produce graphs are proposed, which provide more informative answers [12, 13]. However, these tree or graph based methods have the following problems. First, while some of the content nodes in the resulting trees or graphs are close to each other, there might be content nodes in the result that are far away from each other, meaning that weak relationships among content nodes might exist in the found trees or graphs. We argue that, assuming all the keywords are equally important, results that contain strong relationships (i.e., short distances) between each pair of content nodes should be preferable over the ones containing weak relationships. Second, current graph or tree based methods explore both content and non-content nodes in the graph while searching for the result. Since there may be thousands or even millions of nodes in an input graph, these methods have high time and memory complexity.

In this paper, we propose to find r-cliques as a new approach to the keyword search problem. An r-clique is a set of content nodes that cover all the input keywords and whose shortest distance between each pair of nodes is no larger than r. The benefits of finding r-cliques are as follows. First, in an r-clique all pairs of the content nodes are close to each other (i.e, within r distance). Second, there is no need to explore all the nodes in the input graph when finding r-cliques if a proper index is built. This reduces the search space by orders of magnitude. To illustrate the differ-

[1]In some literature, it is called *minimal Steiner tree.*

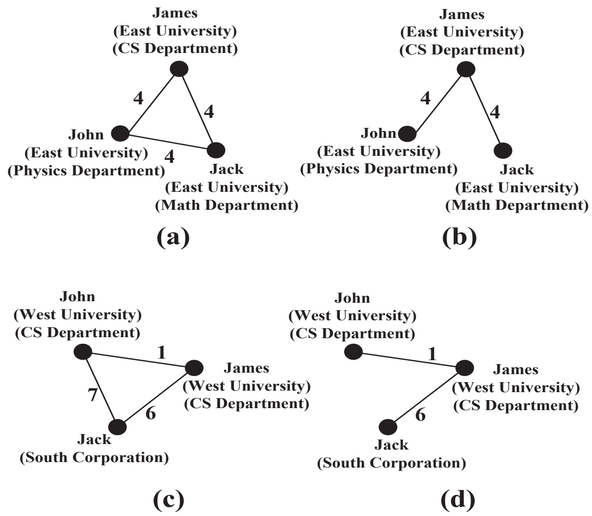

Figure 2: Two different answers (a) and (c) and their Steiner trees (b) and (d) over the sample graph.

ences between r-clique and other approaches (e.g., Steiner tree [2, 6] and community [13]), an example is given below.

Suppose the nodes in an input graph are web pages of researchers and their organizations. Two nodes are connected by an edge if there is a link from one page to the other. Assume the weight on each edge is 1. Let's assume that the user would like to find a collection of pages that contain *James*, *John* and *Jack* and each page is reasonably close to each of the other pages. Given such input keywords, our method will reduce the size of the input graph by keeping only the nodes that contain at least one of the input keywords. Assume that there are only 6 nodes containing the above keywords and the reduced graph is shown in Figure 1, in which the shortest distance between each pair of nodes is shown in the edge that connects the two nodes. Assuming $r = 10$, edges with distance larger than 10 are ignored. Our method will produce ranked r-cliques, two of which are shown in Figure 2 (a) and (c). In Figure 2 (a) each node contains one input keyword, all the pages are from the same university, and each pair of the pages are related to each other via 3 other pages (which are not shown). A Steiner tree that covers these nodes is presented in Figure 2 (b). The r-clique in Figure 2 (c) also contains three pages, two of which from the same university but the other one is from a different organization. A Steiner tree covering these three nodes is shown in Figure 2 (d). Our method will rank the answer in (a) ahead of the one in (c) because the sum of the distances between each pair of nodes in (a) is 12, while the one in (c) is 14. On the other hand, a method that produces Steiner trees would rank the result in (d) ahead of the one in (b) because the total distance on the tree paths in (d) is 7, while the one in (b) is 8. Since the web pages in (a) and (b) are from the same organization and all close to each other, the ranking produced by our method is reasonably better.

The closest work in the literature to our work is [13]. The authors proposed to find communities as the results of keyword search over graphs. In each community, there are some center nodes which are close to the content nodes. The answers are ranked based on the sum of the distances of content nodes to the centers. In the above example, node *James* is considered as the center node in both answers (a) and (c) because it has the least sum of the distances to other content nodes. The community method will rank the answer in (c) ahead of the one in (a) because the sum of the distances from the center node to the content nodes in (c) is 7, while the one in (a) is 8. However, (a) is better than (c) because the three nodes in (a) are from the same university.

The contributions of this paper are summarized as follows:

1. We propose a new model for keyword search in graphs that produces r-cliques in which all pairs of content nodes are reasonably close to each other.

2. We prove that finding the r-clique with the minimum weight is an NP-hard problem.

3. An exact algorithm based on Branch and Bound is proposed for finding all r-cliques.

4. An approximation algorithm that produces r-cliques with 2-approximation is proposed. The algorithm can produce all or top-k r-cliques in polynomial delay in ascending order of their weights.

5. To reveal the relationship between the nodes in a found r-clique, we propose to find a Steiner tree in the graph that connects the nodes in the r-clique. Using a tree instead of a graph reduces the chance of including irrelevant nodes in the final answer.

The paper is organized as follows. Related work is discussed in section 2. In section 3, a formal problem statement is given. In section 4, an algorithm based on Branch and Bound for finding all r-cliques is introduced. An algorithm that produces r-cliques with a 2 approximation ratio in polynomial delay is presented in section 5. A method for presenting an r-clique is given in section 6. Experimental results are given in section 7. Section 8 concludes the paper. The appendix contains theorem proofs, pseudo codes, a graph-indexing method and the information on data sets.

2. RELATED WORK

Most of the approaches to keyword search over graphs find trees as answers[2]. In [2], a backward search algorithm for producing Steiner trees is presented. A dynamic programming approach for finding Steiner trees in graphs are presented in [3]. Although the dynamic programming approach has exponential time complexity, it is feasible for input queries with small number of keywords. In [5], the authors proposed algorithms that produce Steiner trees with polynomial delay. The algorithms follows the Lawler's procedure [11]. Due to the NP-completeness of the Steiner tree problem, producing trees with distinct roots are introduced in recent years [7]. BLINKS improves the work of [7] by using an efficient indexing structure [6].

There are two methods that find subgraphs rather than trees for keyword search over graphs [12, 13]. The first method finds r-radius Steiner graphs that contain all of the input keywords [12]. Since the algorithm for finding r-radius graphs index them regardless of the input keywords, if some of the highly ranked r-radius Steiner graphs are included in other larger graphs, this approach might miss them. In addition, it might produce duplicate and redundant results [13].

[2]A survey on keyword search in databases and graphs can be found in [15].

The second method finds multi-centered subgraphs, called communities [13]. In each community, there are some center nodes. There exists at least a single path between each center node and each content node such that the distance is less than R_{max}. Parameter R_{max} is used to control the size of the community. The authors of [13] propose an algorithm that produces all communities in an arbitrary order and another algorithm that produces ranked communities in polynomial delay. The rank of a community is based on the minimum value among the total edge weights from one of the centers to all of the content nodes. Finding communities as the answer for keyword search over graph data has three problems. While some of the content nodes might be close to each other, the others might not. In addition, for finding each community, the algorithm considers all of the nodes within R_{max} distance from every content node as a candidate for a center node. This leads to poor run-time performance. Finally, while including center and intermediate nodes in the answers can reveal the relationships between the content nodes, these center and intermediate nodes may be irrelevant to the query, which makes some answers hard to interpret. Our proposed model improves the community method by (1) finding r-cliques in which all the content nodes are close to each other, (2) improving the run-time by exploring only the content nodes during search, and (3) reducing the irrelevant nodes by producing a Steiner tree (instead of a graph) to reveal the relationship between the content nodes in an r-clique.

Finding r-cliques is closely related to *Multiple Choice Cover* problem introduced in [1] and used in [10] for finding a team of experts in social networks. These approaches find a single best answer with the smallest diameter. In comparison, we find all or top-k r-cliques with polynomial delay. Our problem is apparently more challenging. In addition, we use the sum of the weights between each pair of nodes as the ranking function. Previous works use other functions such as the diameter of the graph to evaluate the answers.

Keyword search in graphs is also related to the graph pattern matching problem. The concept of bounded graph simulation for finding maximum matches in graphs was recently introduced in [4]. The authors extended the definition of patterns in the graphs. In a pattern of [4], each node indicates a search condition and each edge specifies the connectivity in the graph with a predefined distance. The authors proposed algorithms for finding the maximum match in a graph based on the new definition of matches. The r-clique defined in this paper can be considered as an input pattern in [4]. Also, the output of our algorithm is different from the one in [4]. Their algorithm finds one maximum match in a graph which contains all the nodes in the graph that match with a node in the query. Our top-k r-clique algorithm finds matches that cover all the input keywords but minimize the sum of distances between each two nodes.

3. PROBLEM STATEMENT

Given a data graph and a query consisting of a set of keywords, the problem of keyword search in a graph is to find a set of connected subgraphs that contain all or part of the keywords. It is preferred that the answers are presented according to a ranking mechanism. The data graph can be directed or undirected. The edges or nodes may have weights on them. In this work, we only consider undirected graphs with weighted edges. 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 easily adaptable to work with directed graphs[3].

The problem tackled in this paper is to find a set of r-cliques, preferably ranked in ascending order of their weights. An r-clique and its weight are defined below.

Definition 1. **(r-clique)** Given a graph G and a set of input keywords ($Q = \{k_1, k_2, \ldots, k_l\}$), an r-clique of G with respect to Q is a set of content nodes in G that together cover all the input keywords in Q and in which the shortest distance between each pair of the content nodes is no larger than r. The shortest distance between two nodes is the sum of the weights of the edges in G on the shortest path between the two nodes.

Definition 2. **(Weight of r-clique)** Suppose that the nodes of an r-clique of a graph G are denoted as $\{v_1, v_2, \ldots, v_l\}$. The weight of the r-clique is defined as

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

where $dist(v_i, v_j)$ is the shortest distance between v_i and v_j in G, i.e., the weight on the edge between the two nodes in the r-clique.

r-cliques with smaller weights are considered to be better in this paper. Thus, the core of our problem can be stated below in Problem 1.

Problem 1. Given a distance threshold r, a graph G and a set of input keywords, find an r-clique in G whose weight is minimum.

THEOREM 1. *Problem 1 is NP-hard.*

PROOF. We prove the theorem by a reduction from 3-satisfiability (3-SAT). The detailed proof is presented in Appendix A. □

4. BRANCH AND BOUND ALGORITHM

We present a branch and bound algorithm for finding all r-cliques in a graph. The algorithm is based on systematic enumeration of candidate solutions and at the same time using the distance constraint r to avoid generating subsets of fruitless candidates. Note that this method does not rank the r-cliques by their weights. The ranking, if needed, can be done as a post-processing process. This method is used as a baseline to compare with the polynomial delay approximation algorithm proposed in the next section.

The pseudo-code of the algorithm is presented in Algorithm 1 in Appendix D. In the first step, the set of nodes that contain each keyword is extracted. This can be easily done using a pre-built inverted index that stores a mapping from a word in the dataset to the list of nodes containing the word. The set of nodes containing keyword k_i is stored in set C_i. C_i^j specifies the jth element of set C_i. The candidate partial r-cliques are stored in a list called $rList$. The

[3]For directed graphs, the shortest distance between two nodes in an r-clique should be no larger than r in both directions.

basic idea of the algorithm is as follows. First, the content nodes containing the first keyword are added to $rList$. Then, for the second keyword, we compute the shortest distance between each node in C_2 and each node in $rList$. If the distance $\leq r$, a new candidate that combines the corresponding nodes in C_2 and $rList$ is added to a new candidate list called $newRList$. After all pairs of nodes in C_2 and $rList$ have been checked, the content of $rList$ is replaced by the content of $newRList$. The process continues in the same way to consider all of the remaining keywords. The final content of $rList$ is the set of all r-cliques.

To speed up this process, an index (described in Appendix F) is pre-built to store the shortest distance between each pair of nodes. Thus, the shortest path computation is at the unit cost. Assume that the maximum size of C_i ($1 \leq i \leq l$ where l is the number of keywords) is $|C_{max}|$. The complexity of the algorithm is $O(l^2|C_{max}|^{l+1})$.

5. POLYNOMIAL DELAY ALGORITHM

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

5.1 Main Procedure

Our approximation algorithm is an adaption of Lawler's procedure [11] for calculating the top-k answers to discrete optimization problems. Lawler generalized Yen's algorithm in [14] which finds the k shortest loopless paths in a network. In Lawler's procedure, the search space is first divided into disjoined sub-spaces; then the best answer in each subspace is found and used to produce the current global best 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.

We first informally describe the idea of dividing the search space into subspaces using an example[4]. Suppose that the input query consists of four keywords, i.e., $\{k_1, k_2, k_3, k_4\}$. Let C_i be the set of nodes in graph G that contains input keyword k_i. Thus, the search space that contains the best answer can be represented as $C_1 \times C_2 \times C_3 \times C_4$. From this space, we use the $FindTopRankedAnswer$ procedure (to be described later in this section) to find the best (approximate) answer in polynomial time in the size of the database and the number of keywords. Assume that the best answer is (v_1, v_2, v_3, v_4), where v_i is a node in graph G containing keyword k_i, and $\sum_{i=1}^{4}\sum_{j=i+1}^{4} d_{ij}$ is the weight of the answer, where d_{ij} is the shortest distance between nodes i and j in graph G. Based on this best answer, the search space is divided into 5 subspaces SB_0, SB_1, SB_2, SB_3 and SB_4 as shown in Table 1, where SB_0 contains only the best answer. The union of the subspaces cover the whole search space.

After finding the best answer and dividing the search space into subspaces, the best answer in each subspace except subset SB_0 is found using the $FindTopRankedAnswer$ procedure. These best answers are inserted into a priority queue, where the answers are ranked in ascending order according to 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. After returning the second best answer, SB_2 is divided into 5 subspaces in the way similar to the one shown in Table 1. In each subspace (except the first subspace), the best answer is found and is added to the priority queue. At this state, the priority queue has seven elements: three elements from the first step and four elements from this new step[5]. 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.

Table 1: An Example of dividing the search space.

Subspace	Representative set
SB_0	$\{v_1\} \times \{v_2\} \times \{v_3\} \times \{v_4\}$
SB_1	$[C_1 - \{v_1\}] \times C_2 \times C_3 \times C_4$
SB_2	$\{v_1\} \times [C_2 - \{v_2\}] \times C_3 \times C_4$
SB_3	$\{v_1\} \times \{v_2\} \times [C_3 - \{v_3\}] \times C_4$
SB_4	$\{v_1\} \times \{v_2\} \times \{v_3\} \times [C_4 - \{v_4\}]$

The pseudo-code of algorithm $GenerateAnswers$ is presented in Algorithm 2 in Appendix D. The main body of the algorithm is similar to other polynomial delay algorithms discussed in [5, 13]. It is modified to perform in the setting of producing ranked r-cliques from a graph. The algorithm takes graph G, query $\{k_1, k_2, \dots, k_l\}$, the distance threshold r and k as input. It searches for answers and outputs top-k of them in ascending order according to their weights. In lines 1 and 2, the algorithm computes sets C_i, the set of the nodes containing keyword k_i. This can be easily done using a pre-built inverted index. Then, the collection of sets C_i is called C in line 3. It should be noted that C is the whole search space that contains keyword nodes and the first best answer should be found in this space. In line 5, procedure $FindTopRankedAnswer$ (to be discussed later) is called to find the best answer in space C in polynomial time. If the best answer exits (i.e., A on line 6 is not empty), A, together with the related space C, is inserted into $Queue$ in lines 6 and 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. The top of the $Queue$ contains the best answer in the $Queue$ and the space that this answer is produced from. We assign this space to S and the best answer to A. The answer in A is outputted in line 10. Then if the number of answers has not reached k, l sets of content nodes are generated based on space S, each set corresponding to an input keyword (lines 14 and 15). In line 16, procedure $ProduceSubSpaces$ produces l new subspaces based on the current answer A and sets S_1, S_2, ..., S_l. These subspaces are shown by SB_i. In lines 17-20, these new subspaces are explored. For each subspace, the best answer is found and it is inserted into the $Queue$ with its

[4]Our approach to dividing a search space is similar to the idea used in [13].

[5]This is based on the assumption that all of the subspaces contain at least one r-clique. In some cases, the subspace does not have any answer.

related subspace[6]. If procedures $FindTopRankedAnswer$ and $ProduceSubSpaces$ terminate in polynomial time, then algorithm $GenerateAnswers$ produces answers with polynomial delay. Below, we explain these two procedures.

5.2 Finding Best Answer from a Search Space

The pseudo-code of algorithm $FindTopRankedAnswer$ is presented in Algorithm 3 in Appendix D. It takes the current space, S, as the input and produces the best (approximate) answer in S as the output in polynomial time. In lines 1 and 2, it produces the set of nodes containing each keyword, S_i. Variable s_i^j denotes the j-th node of set S_i. $d(s_i^j, k)$ denotes the shortest distance between s_i^j and set S_k, which is the distance between s_i^j and the node in S_k that is closest to s_i^j. $n(s_i^j, k)$ denotes the node in S_k which has the shortest distance to s_i^j. In lines 3-6, the distance of a node to its own set is set to 0. In lines 7-16, the values of $d(s_i^j, k)$ and $n(s_i^j, k)$ for all the nodes in S_i $(1 \le i \le l)$ are calculated. That is, for each node s_i^j in S_i for $1 \le i \le l$ and $1 \le j \le |S_i|$, its distance to S_k and its nearest node in S_k are computed. This is done by comparing all distances and choosing the smallest one. Then, in lines 17-26 for each node s_i^j, the algorithm checks to see if $d(s_i^j, k) \le r$ for all k values. If yes, which means the distance from s_i^j to its nearest node in each set S_k is less than r, then the set of nodes consisting of s_i^j and its nearest nodes in all other sets S_k (for $1 \le k \le l$ and $k \ne l$) is considered as a candidate for best answer. The sum of distances from s_i^j to all its nearest nodes in other sets S_k is calculated and used to compete with other candidates for the best answer in space S. The candidate with the lowest sum is outputted as the best answer (denoted as *topAnswer* in the pseudo-code).

Clearly, all of the above operations can be done in polynomial time. Since a pre-built index (described in Appendix F) is used for finding the shortest path between each pair of nodes, the shortest path computation is at the unit cost. Thus, the complexity of this algorithm is $O(l^2|S_{max}|^2)$, where $|S_{max}|$ is the maximum size of S_i for $1 \le i \le l$.

It should be noted that the answer returned by this approximation algorithm might not be an r-clique. In the worst cast, the distance between a pair of nodes in the answer is $2 \times r$, as stated in the following theorem.

THEOREM 2. *The distance between each pair of nodes in the answer produced by procedure $FindTopRankedAnswer$ is at most $2 \times r$.*

PROOF. A proof is provided in Appendix B. □

[6]Note that, unlike tree-based methods, this procedure produces duplication-free answers (i.e., the set of content nodes in an answer is unique compared to other answers in the top-k list) if no content node contains more than one input keyword in the input graph. But if a node contains more than one input keyword, the procedure may produce duplicate answers although the answers are unique in terms of keyword-node couplings. In this regard, our result is the same as the top-k result in [13], where the authors consider such answers duplication-free because of different keyword-node couplings in the top-k answers. If a user prefers a completely duplication-free set of answers with respect to the set of content nodes in an answer, a post-pruning process can be applied to remove an answer if the set of nodes in the answer is the same as an answer already generated in the top-k list. Since k is usually small, the post-pruning process is fast.

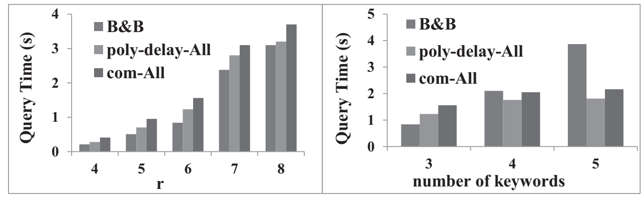

Figure 3: Run time on DBLP for the algorithms that produce all answers. The number of keywords in the left graph is 3. The r value on the right graph is 6. The frequency of keywords is 0.0009.

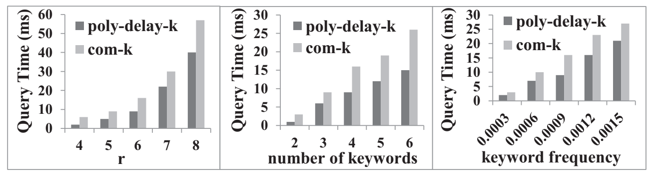

Figure 4: Run time on DBLP for algorithms that produce top-k answers with polynomial delay. When not changing, the number of keywords is set to 4, r is set to 6 and keyword frequency is 0.0009.

Also, in the worst case, the *weight* of an answer produced by the above algorithm is twice the *weight* of the optimal answer. However, as we will show in the experimental results, in practice the difference in *weight* between the optimal answer and the one produced by this approximation algorithm is small, much less than the difference in the worst case scenario. For the convenience reason, we still refer to an answer from this algorithm as r-clique. The following theorem proves that this procedure produces r-cliques with 2-approximation.

THEOREM 3. *Procedure $FindTopRankedAnswer$ produces an r-clique with 2-approximation.*

PROOF. In the worst case situation, the following expression is stated for l query keywords.

$$\frac{2 \times (l-1)}{l}(optimal\ weight) \ge candidate\ weight$$

A formal and detailed proof with an example is presented in Appendix C. □

The pseudo-code of algorithm $ProduceSubSpaces$ is presented in Algorithm 4 in Appendix D. It takes the best answer of the previous step, A, and the set of content nodes, $S1, \ldots, S_l$, as input. It produces l new subspaces, $\langle SB_1, \ldots, SB_l \rangle$. In the procedure, SB_i^j specifies the j-th position in vector SB_i. It is a polynomial procedure and runs in $O(l^2)$.

6. PRESENTING R-CLIQUES

Each r-clique is a unique set of content nodes that are close to each other and cover the input keywords. However, sometimes it is not sufficient to only show the set of content nodes discovered. It is also important to see how these nodes are connected together in the input graph. To show the relationship between the nodes in an r-clique, we further

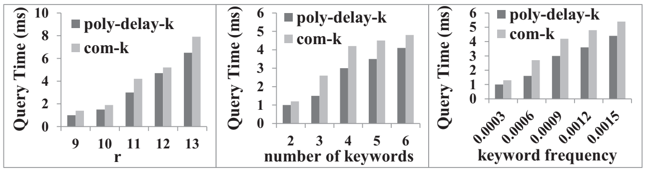

Figure 5: Run time on the IMDb data set of algorithms producing top-k answers with polynomial delay. When not changing, the number of keywords is 4, r is 11 and keyword frequency is 0.0009.

find a Steiner tree from the input graph which connects the nodes in the r-clique with the minimum sum of edge weights. The tree contains all the nodes in the r-clique. Its leaves are the content nodes of the r-clique, and its internal node can be a content node in the r-clique or an intermediate node that connects the content nodes. The algorithm for finding the tree given an r-clique is presented in Appendix E and is based on the algorithm presented in [9].

The reason for choosing a Steiner tree instead of a graph to present an r-clique is that it potentially minimizes the number of intermediate nodes, which decreases the chance of having irrelevant nodes in the answer presented to the user. In the next section, we will show that the community-based method [13] (which returns a graph) tends to include more irrelevant nodes in its answer. Compared to other methods that return trees, our approach returns fewer duplicate answers with respect to the set of content nodes in an answer and is faster because finding r-cliques from only the content nodes in the graph and then finding a Steiner tree that covers all the nodes in an r-clique are together much faster than finding Steiner trees directly from the input graph based on the input keywords. In other words, we take the advantage of trees for presenting the answers with fewer irrelevant nodes than the graph-based methods while preventing the disadvantages of the tree-based methods.

7. EXPERIMENTAL RESULTS

We implemented the Branch and Bound algorithm for finding all r-cliques and two versions of the polynomial delay algorithm, one finding all r-cliques and the other finding top-k r-cliques. For the purpose of comparison we also implemented the algorithms presented in [13]. All of the algorithms were implemented using Java. To keep the comparison fair, all of the algorithms use the same graph indexing structures. The experiments are conducted on an Intel(R) Core(TM) i7 2.86GHz computer with 3GB of RAM. In this section, the results of the algorithms and the factors affecting the performance of the algorithms are presented. The factors include the value of r, the number of keywords (l) and the frequency of keywords. Throughout this section, the Branch and Bound algorithm is called *B&B* and our polynomial delay algorithms that produce all and top-k answers are called *poly-delay-All* and *poly-delay-k* respectively. In addition, the algorithm in [13] that produces all communities is called *com-All* and the algorithm that produces top-k communities with polynomial delay is called *com-k*.

Two data sets are used in the evaluation: DBLP and IMDb. The sets of input keywords and parameters used in the evaluation are the same as the ones in [13]. The data sets and keyword queries are described in Appendix G. Be-

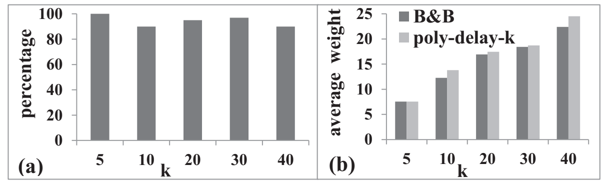

Figure 6: Quality of the *poly-delay-k* algorithm on DBLP. The number of keywords is set to 4, r is 8 and the frequency of keywords is 0.0009.

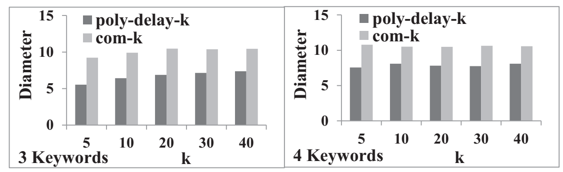

Figure 7: The average diameter of answers on DBLP. The r is set to 8 and the frequency of keywords is 0.0009.

tween the two datasets, DBLP is larger and contains more textual information and relations. Due to the space limit, some of results on IMDb are not presented.

7.1 Search Efficiency

For the algorithms that produce top-k answers, the average time for producing one answer in finding top-50 answers is used as their run time. For the algorithms that produce all answers, the run time is the total time the program takes. If there is no answer for the query, the time of completing the program is considered as the run time. Since the number of communities is usually more than that of r-cliques given the same value for parameter r, to keep the comparison fair, we stop *com-All* when it produces the same number of results as *poly-delay-All*. For r-clique methods, the time also includes the time for generating Steiner trees as final answers.

The run time of different algorithms on the DBLP dataset that produce all answers is presented in Figure 3. The left graph shows how the run time varies with the value of r, while the number of keywords is set to 3. The right graph shows how the run time changes with the number of keywords while r is set to 6. These two figures show that for producing all answers the Branch and Bound algorithm outperforms others when the number of input keywords is 3. But when the number of keywords becomes larger, its runtime increases significantly and is much higher than *poly-delay-All* and *com-All*. Comparing *poly-delay-All* and *com-*

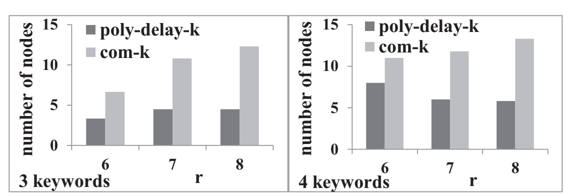

Figure 8: The average number of nodes in final answers of *poly-delay-k* and *com-k* on DBLP. k is set to 10 and the frequency of keywords is 0.0009.

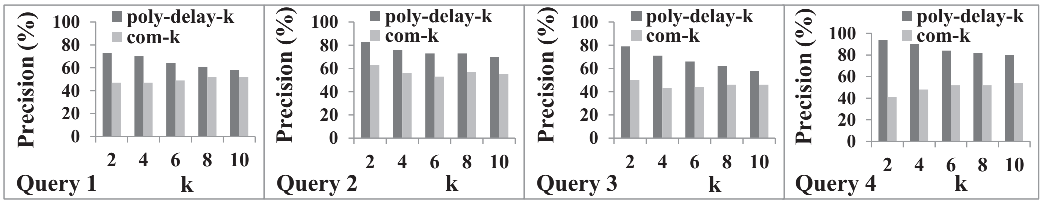

Figure 9: Top-k precision of answers with different values of k.

All, *poly-delay-All* is faster. By increasing the value of r, the run time of all algorithms increase. This is because there are more nodes to evaluate as candidates for generating answers.

The run time of polynomial delay algorithms for producing top-k answers on the DBLP and IMDb datasets is shown in Figures 4 and 5. We can see that *poly-delay-k* produces results faster than *com-k*. By increasing the value of r and the frequency of keywords, the run time of both algorithms increases. This is because there are more candidates and nodes to evaluate. These results agree with the findings in [13]. By increasing the number of keywords, the average run time of both algorithms for producing one answer also increases. This means that average delay increases when the number of keywords increases. This is because more nodes need to be evaluated in each step. It should be mentioned that this result does not agree with the results presented in [13] for generating top-k communities.

7.2 The Quality of the Approximation Algorithm Compared with B&B

In this section, the quality of the answers generated by the approximation algorithm is evaluated. We compare the answers from Branch and Bound algorithm with those of the *poly-delay-k* algorithm. Figure 6 (a) shows the percentage of answers produced by the approximation algorithm which are actually r-cliques. The results suggest that at least 90% of the answers are r-cliques. Figure 6 (b) shows the average weight of the answers produced by the *B&B* and *poly-delay-k* algorithms for different k values. To get the top-k results for *B&B*, we rank the answers from *B&B* based on their weight. Although in theory the weight of an answer from *poly-delay-k* can be twice that of the corresponding answer from *B&B*, our results show that the difference is small in practice (only 11% in the worst case when k=10). These results suggest the high quality of the proposed approximation algorithm.

7.3 Comparing the Compactness of r-cliques with that of Communities

In this section, we evaluate the quality of the answers produced by *poly-delay-k* and *com-k* in terms of their compactness. A well known measure for estimating the proximity of a subgraph is the *diameter* of the subgraph, defined as the largest shortest distance between any two nodes in the subgraph. Generally, the smaller the diameter, the closer the nodes are to each other. When calculating the diameter for *poly-delay-k*, we use all the nodes in the final answer, i.e., the nodes in the Steiner tree presented to the user. The average diameters of the answers produced by the *poly-delay-k* and *com-k* algorithms are shown in Figure 7, which shows that the nodes in an answer produced by *poly-delay-k* are closer to each other than those from *com-k* for different k values and different numbers of keywords. The average number of

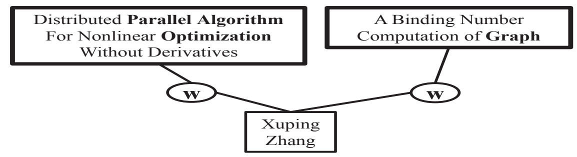

Figure 10: Best r-clique answer to the query consisting of *parallel*, *graph*, *optimization* and *algorithm*.

nodes in the answers produced by each algorithm is shown in Figure 8. Since a community includes all of the nodes whose distance to each content node is no larger than r , the number of nodes in a community is higher than that in the r-cliques that use trees to present the final answers.

7.4 Search Accuracy from a User Study

We further compare the *poly-delay-k* and *com-k* algorithms in terms of how relevant their answers are to the query. A common metric of relevance used in information retrieval is top-k precision, defined as the percentage of the answers in the top-k answers that are relevant to the query. To evaluate the top-k precision of the algorithms, we conducted a user study. We designed 4 meaningful queries from the lists of keywords used in [13] for the DBLP dataset in order for human users to be able to evaluate the search results. The four queries are listed in Table 3 in the Appendix. For example, the first query is "parallel graph optimization algorithm". In the experiment, r is set to 8 and top-10 answers are produced for each query from each algorithm.

We asked 8 graduate students in computer science and electrical engineering at two universities to judge the relevancy of the answers. The users are asked to evaluate the answers using two methods. In the first method, for each answer, the user assigns a score between 0 and 1 to each paper (i.e., node) in the answer where 1 means completely relevant and 0 means completely irrelevant to the query. Then, the average score of the papers in an answer is calculated as the relevancy score of the answer. This score may vary among the users. We use the average of the relevancy scores from the 8 users as the final relevancy score of the answer. The top-k precision is computed as the sum of the relevancy scores of the top-k answers divided by k. In the second method, users assign a score between 0 and 1 to the whole answer based on the relevancy and understandability of the answer. The results and trends from both methods are very close to each other. Due to the space limit, we only report the results of the first method.

The top-2 to top-10 precisions for each query are presented in Figure 9. Clearly, *poly-delay-k* achieves better precisions than *com-k* in all the queries for all the k values. The reason

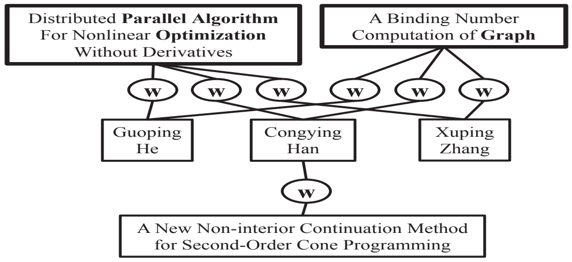

Figure 11: Best community answer to the query consisting of *parallel*, *graph*, *optimization* and *algorithm*.

for the community method to have a lower precision is that a community may contain some center nodes and these centers are determined only based on their distance to the content nodes. If a node's distance to each of the content nodes is within a threshold, it is included in the community as a center. However, such a node may not be relevant to the query. By looking at the individual answers, we find that the community method indeed returns papers that are considered irrelevant to the query by the users.

7.5 Qualitative Evaluation

We compare the *poly-delay-k* and *com-k* algorithms via an example. The top answer returned by *poly-delay-k* for the first query in the user study is shown in Figure 10. The two boxes at the top are content nodes, each containing the title of a paper. The node at the bottom is the mediator node generated by our Steiner tree algorithm given the two content nodes. It is a common author of the two papers. The "W" symbol on an edge indicates the "writing" relationship. Clearly, our r-clique based method is able to reveal a relationship between the two content nodes. Figure 11 illustrates the top answer from the *com-k* algorithm. The top two nodes are content nodes, and the others are center nodes because each of them is within r distance from each of the content nodes. As can be seen, the community contains more nodes than the answer from the r-clique method. The three middle nodes are the three common authors of the two papers and the bottom node is another paper written by one of the authors, which is not relevant to the query. The advantage of this answer is that it reveals more common authors of the two papers (assuming this is useful for the user), but the disadvantage is that it also includes an irrelevant node. Having irrelevant nodes in an answer can make the answer hard to understand. Most of the users in our user study prefers the answer in Figure 10 over this one.

8. CONCLUSIONS

We have proposed a novel and efficient method for keyword search on graph data. A problem with existing approaches is that, while some of the nodes in the answer are close to each other, others might be far from each other. To address this problem, we introduced the concept of r-cliques as the answer for keyword search in graphs. A benefit of finding r-cliques is that only content nodes need to be explored during the search process, which leads to significant runtime improvement. We proposed an exact algorithm that produces all r-cliques using the Branch and Bound strategy and a polynomial delay algorithm that produces r-cliques with 2 approximation ratio. To reveal the relationship between the nodes in an r-clique, a Steiner tree is generated based on the r-clique and presented to the user. Our experimental results showed that finding r-cliques is more efficient and produces more compact and more relevant answers than the method for finding communities [13]. We also showed that quality of the answers from the proposed approximation algorithm is high in terms of the percentage of r-cliques and the sum of weights in the top ranked answers.

9. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their comments that helped improve the quality of this paper.

10. REFERENCES

[1] E. M. Arkin and R. Hassin. Minimum-diameter covering problems. *Networks*, 36(3):147–155, 2000.

[2] G. Bhalotia, C. Nakhe, A. Hulgeri, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *Proc. of ICDE'02*, pages 431–440, 2002.

[3] B. Ding, J. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *Proc. of ICDE'07*, pages 836–845, 2007.

[4] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractable to polynomial time. In *Proc. of VLDB'10*, pages 264–275, 2010.

[5] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *Proc. of SIGMOD'08*, pages 927–940, 2008.

[6] H. He, H. Wang, J. Yang, and P. Yu. Blinks: ranked keyword searches on graphs. In *Proc. of SIGMOD'07*, pages 305–316, 2007.

[7] 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*, pages 505–516, 2005.

[8] R. Karp. Reducibility among combinatorial problems. *Compexity of computer computations*, 1972.

[9] L. Kou, G. Markowsky, and L. Berman. A fast algorithm for steiner trees. *Acta Informatica*, 15(2):141–145, 1981.

[10] T. Lappas, K. Liu, and E. Terzi. Finding a team of experts in social networks. In *Proc. of KDD'09*, pages 467–475, 2009.

[11] E. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7):401–405, 1972.

[12] 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*, pages 903–914, 2008.

[13] L. Qin, J. Yu, L. Chang, and Y. Tao. Querying communities in relational databases. In *Proc. of ICDE'09*, pages 724–735, 2009.

[14] J. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.

[15] J. Yu, L. Qin, and L. Chang, editors. *Keyword Search in Databases*. Morgan and Claypool Publisher, 2010.

APPENDIX

A. PROOF OF THEOREM 1

In this section, we formally prove that Problem 1 (finding an r-clique with the minimum weight) is NP-hard. We prove that the decision version of the problem presented below is NP-hard. Thus, as a direct result, Problem 1 is NP-hard too. The decision problem is specified as follows.

Problem 2. Given a distance threshold r, a graph G and a set of input keywords $S_1, \ldots, S_l$, determine whether there exists an r-clique with weight w, for some constant w. The weight of the r-clique is defined in Definition 2.

THEOREM 4. *Problem 2, a decision version of Problem 1, is NP-hard.*

PROOF. The problem is obviously in NP. We prove the theorem by a reduction from 3-satisfiability (3-SAT)[7]. First, consider a set of m clauses $D_k = x_k \vee y_k \vee z_k$ $(k = 1, \ldots, m)$ and $\{x_k, y_k, z_k\} \subset \{u_1, \overline{u}_1, \ldots, u_n, \overline{u}_n\}$. We set the distance between each variable and its negation (i.e. u_i and $\overline{u}_i$) to $2 \times w$. The distance between other variables is set to $\frac{w}{\binom{n+m}{2}}$. The distance of each variable to itself is set to zero. We define an instance of the above problem as follows. First, r is set to $2 \times w$. For each pair of variables u_i and $\overline{u}_i$, two nodes are created. Thus, we have $2 \times n$ nodes. For each pair of variables u_i and $\overline{u}_i$, we create one keyword S_i $(i = 1, \ldots, n)$. Thus, u_i and $\overline{u}_i$ have keyword S_i and the only holders of S_i are u_i and $\overline{u}_i$. In addition, for every clause D_k, we create one keyword S_{n+k} $(k = 1, \ldots, m)$ such that the holders of keyword s_{n+k} consists of the triplet of nodes associated with those of x_k, y_k and z_k. Therefore, the number of required keywords is $n + m$.

A feasible solution to the above problem with the weight at most w is any set of nodes such that from each pair of nodes corresponding to u_i and $\overline{u}_i$, exactly one is selected and from each triplet of nodes corresponding to x_k, y_k and z_k, one is selected. Thus, if there exists a subset of the weight at most w, then there exists a satisfying assignment for $D_1 \wedge D_2 \wedge \cdots \wedge D_m$. On the other hand, a satisfying assignment apparently determines a feasible set of nodes with the weight at most w. Therefore, the proof is complete.

□

B. PROOF OF THEOREM 2

We prove that the upper bound on the distance between any pair of nodes in an answer produced by the approximation algorithm is $2 \times r$.

PROOF. In the answer produced by algorithm $FindTopRankedAnswer$, there is a content node (s_i^j in line 10 of Algorithm 3) that has distance less than or equal to r to each of the other nodes in the answer. Assume this node is called a. The distance between a and any of other nodes is less than or equal to r. We want to show that the distance between two other nodes b and c in the answer is at most $2 \times r$. Since shortest distances satisfy the triangle inequality, we have:

$$d_{bc} \leq d_{ab} + d_{ac}$$

where d_{bc} is the shortest distance between nodes b and c and so on. Also, as we mentioned above, the distance between nodes a and b and the distance between nodes a and c are both less than or equal to r (i.e., $d_{ab} \leq r$ and $d_{ac} \leq r$). Thus, based on the above equation, we have:

$$d_{bc} \leq d_{ab} + d_{ac} \leq r + r \leq 2 \times r$$

□

C. PROOF OF THEOREM 3

To prove Theorem 3, we first give an example and then present a formal proof. Consider the example presented in Fig.12 with four input keywords. One of the answers is the optimal answer and the other one is the candidate answer produced by procedure $FindTopRankedAnswer$. Without the loss of generality, we assume that the node for keyword k_1 is the best candidate node (i.e., the best s_i^j) selected by the procedure. Since the sum of the weights on edges connected to k_1, i.e. d_{12}, d_{13} and d_{14}, in the selected candidate is the smallest among all the content nodes whose connected edges have a weight less than or equal to r, the following expressions hold:

$$\begin{cases} k_1: & o_{12} + o_{13} + o_{14} \geq d_{12} + d_{13} + d_{14} \\ k_2: & o_{12} + o_{23} + o_{24} \geq d_{12} + d_{13} + d_{14} \\ k_3: & o_{13} + o_{23} + o_{34} \geq d_{12} + d_{13} + d_{14} \\ k_4: & o_{14} + o_{24} + o_{34} \geq d_{12} + d_{13} + d_{14} \end{cases} \tag{1}$$

Summing up both sides of the above equations, we have:

$$2(o_{12} + o_{13} + o_{14} + o_{23} + o_{24} + o_{34}) \geq 4(d_{12} + d_{13} + d_{14}) \tag{2}$$

Since the distance between each pair of nodes is the shortest distance between them, the triangle inequality is satisfied and the following equations hold:

$$\begin{cases} d_{12} + d_{13} \geq d_{23} \\ d_{12} + d_{14} \geq d_{24} \\ d_{13} + d_{14} \geq d_{34} \end{cases} \tag{3}$$

The weight of the selected candidate produced by procedure $FindTopRankedAnswer$ is $d_{12} + d_{13} + d_{14} + d_{23} + d_{24} + d_{34}$. Based on Equation 3, the candidate weight is at most $3 \times (d_{12} + d_{13} + d_{14})$. Thus, after some basic calculations and based on Equation 2, the following is valid:

$$\frac{2 \times 3}{4}(o_{12} + o_{13} + o_{14} + o_{23} + o_{24} + o_{34}) \geq 3 \times (d_{12} + d_{13} + d_{14}) \tag{4}$$

The left side of the equation is at most twice the weight of the optimal answer and the right side of the equation is at most the weight of the selected candidate. Thus, in the worst case, the weight of the selected candidate is twice the weight of the optimal answer. Now we are ready to present the formal proof in detail.

C.1 Formal Proof

We prove that procedure $FindTopRankedAnswer$ produces r-cliques with an approximation ratio of 2. Consider two answers, one *optimal answer* and the answer produced

[7]It should be noted that the same approach is used in [1] for proving the NP-hardness of *multiple choice cover* problem.

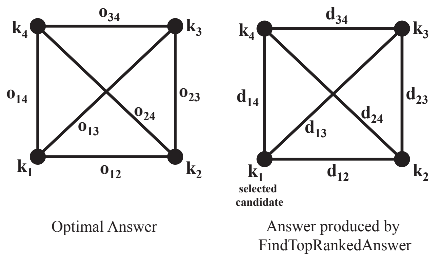

Figure 12: The optimal answer and the answer produced by procedure $\boldsymbol{FindTopRankedAnswer}$**.**

by $FindTopRankedAnswer$ (denoted here as *candidate answer*). Our purpose is to show that the weight of the *candidate answer* is at most twice the weight of the *optimal answer*.

Assume that the number of input keywords are l. We denote a node in *candidate answer* that has the smallest sum of weights on the edges connected to it as *candidate node*. In other words, the sum of the weights on the $l-1$ edges connected to *candidate node* in *candidate answer* is the smallest among all other content nodes of all keywords in the input graph. Without loss of generality, assume that the *candidate node* is the node related to the first keyword, i.e. k_1. Let's call the edges of the *candidate node* $d_{12}, d_{13}, \ldots, d_{1l}$. Thus, based on the $FindTopRankedAnswer$ procedure, $\sum_{i=2}^{l} d_{1i}$ has the smallest value among all other content nodes in the graph. Each node in the *optimal answer* also has $l-1$ neighbors and $l-1$ edges are connected to it. For each node containing $k_j : 1 \leq j \leq l$ of the *optimal answer*, we have:

$$o_{1j} + o_{2j} + \cdots + o_{j-1j} + o_{jj+1} + \cdots + o_{jl} \geq d_{12} + d_{13} + \cdots + d_{1l} \tag{5}$$

In other words,

$$\sum_{i=1}^{j-1} o_{ij} + \sum_{i=j+1}^{l} o_{ji} \geq \sum_{i=2}^{l} d_{1i} \tag{6}$$

In the above equation, o_{ij} $i < j$ (o_{ji} $i > j$) is the weight of the edges between i and j in the optimal answer. If we write the above equation for all l content nodes of the *optimal answer* and sum up both sides of the inequalities, we have:

$$2 \times \sum_{i=1}^{l} \sum_{j=i+1}^{l} o_{ij} \geq l \times \sum_{i=2}^{l} d_{1i} \tag{7}$$

This is because we have l content nodes. Also, since each edge is connected to two content nodes, each edge appears in the left side of the equation twice. The left side of the above equation is twice the weight of the optimal answer. Thus, the following is valid:

$$2 \times (\textit{optimal weight}) \geq l \times \sum_{i=2}^{l} d_{1i} \tag{8}$$

Since the distance between each pair of nodes in the *candidate answer* is the shortest distance between them, the triangle inequality is satisfied:

$$d_{ij} \leq d_{1i} + d_{1j}, i \neq j \neq 1 \tag{9}$$

The weight of the *candidate answer* is as follows:

$$\textit{candidate weight} = \sum_{i=1}^{l} \sum_{j=i+1}^{l} d_{ij} = \sum_{i=2}^{l} d_{1i} + \sum_{i=2}^{l} \sum_{j=i+1}^{l} d_{ij} \tag{10}$$

Since we have $d_{ij} \leq d_{1i} + d_{1j}$, the following is valid:

$$\sum_{i=2}^{l} d_{1i} + \sum_{i=2}^{l} \sum_{j=i+1}^{l} d_{ij} \leq \sum_{i=2}^{l} d_{1i} + \sum_{i=2}^{l} \sum_{j=i+1}^{l} (d_{1i} + d_{1j}) \tag{11}$$

In the right side of the above equation, each edge d_{1i} is appeared exactly $l-1$ times. Thus, we have:

$$\sum_{i=2}^{l} d_{1i} + \sum_{i=2}^{l} \sum_{j=i+1}^{l} (d_{1i} + d_{1j}) = (l-1) \times \sum_{i=2}^{l} d_{1i} \tag{12}$$

As a result, we have:

$$\textit{candidate weight} \leq (l-1) \times \sum_{i=2}^{l} d_{1i} \tag{13}$$

Based on equations 8 and 13, we have:

$$\frac{2 \times (l-1)}{l} (\textit{optimal weight}) \geq \textit{candidate weight} \tag{14}$$

It proves that the weight of the *candidate answer* is at most twice the weight of the *optimal answer*.

D. PSEUDO-CODE OF ALGORITHMS

Algorithm 1 Branch and Bound Algorithm

Input: the input graph G; the query $\{k_1, k_2, \ldots, k_l\}$ and r
Output: the set of all r-cliques

```
for i ← 1 to l do
    C_i ← the set of nodes in G containing k_i
rList ← empty
for i ← 1 to size(C_1) do
    rList.add(C_1^i)
for i ← 2 to l do
    newRList ← empty
    for j ← 1 to size(C_i) do
        for k ← 1 to size(rList) do
            if ∀ node ∈ rList_k dist(node, C_i^j) ≤ r (where rList_k is the kth element of rList) then
                newCandidate ← C_i^j.concatenate(node)
                newRList_k.add(newCandidate)
    rList ← newRList
return rList
```

E. FINDING STEINER TREES

We present an algorithm for finding a Steiner tree for an r-clique. The purpose of finding a Steiner tree for an r-clique is to reveal the relationship among the content nodes in the r-clique via their relationships to other nodes.

Algorithm 2 GenerateAnswers Algorithm

Input: the input graph G; the query $\{k_1, k_2, \ldots, k_l\}$; r and k

Output: the set of top-k ordered r-cliques printed with polynomial delay

for $i \leftarrow 1$ **to** l **do**
  $C_i \leftarrow$ the set of nodes in G containing k_i
$C \leftarrow \langle C_1, C_2, \ldots, C_l \rangle$
$Queue \leftarrow$ an empty priority queue
$A \leftarrow$ **FindTopRankedAnswer**(C, G, l, r)
if $A \neq \emptyset$ **then**
  $Queue$.**insert**($\langle A, C \rangle$)
while $Queue \neq \emptyset$ **do**
  $\langle A, S \rangle \leftarrow Queue$.**removeTop**()
  print(A)
  $k \leftarrow k - 1$
  if $k = 0$ **then**
    return
  for $i \leftarrow 1$ **to** l **do**
    $S_i \leftarrow$ S.**get**(i)
  $\langle SB_1, SB_2, \ldots, SB_l \rangle \leftarrow$ **ProduceSubSpaces**($\langle A, S_1, \ldots, S_l \rangle$)
  for $i \leftarrow 2$ **to** l **do**
    $A_i \leftarrow$ **FindTopRankedAnswer**(SB_i, G, l, r)
    if $A_i \neq \emptyset$ **then**
      $Queue$.**insert**($\langle A_i, SB_i \rangle$)

Given a set of nodes, S, that belong to graph G, the Steiner tree problem is to find a tree of graph G that spans S with the minimal total distance on the edges of the tree. This is a well known NP-hard problem [8]. A heuristic algorithm was introduced in [9] to find a Steiner tree from a graph G given a set S of nodes in G. The nodes in S are called Steiner points. The algorithm in [9] first finds the shortest path in G between each pair of nodes in S and builds a complete graph, G_1, whose nodes are the nodes in S and whose edge between each two nodes is weighted by the total distance on the shortest path between the two nodes in G. It then finds a minimal spanning tree, T_1, of G_1, and constructs a subgraph G_2 of G by replacing each edge of T_1 by its corresponding shortest path in G. Finally, it finds a minimal spanning tree, T_2, of G_2, and constructs a Steiner tree from T_2 by deleting leaves and their associated edges from the tree so that all the leaves are Steiner points.

We make use of this procedure to find a Steiner tree for an r-clique. The input to our procedure is an r-clique, which is a complete graph whose weight on each edge is the shortest distance between the two corresponding nodes in the graph G from which the r-clique was generated. The set of Steiner points is the set of nodes in the r-clique. The output of the algorithm is a Steiner tree of G that spans all the nodes in the r-clique. The pseudo-code of the algorithm is presented in Algorithm 5. The Steiner tree produced by this heuristic algorithm is not necessarily minimal, but its total distance on the edges is at most twice that of the optimal Steiner tree [9]. The algorithm terminates in polynomial time [9].

A major difference of our method from other keyword search methods that generate Steiner trees is that we generate a Steiner tree based on an r-clique, which contains a very small subset of content nodes in the original graph G. The number of nodes in an r-clique is no more than the number of input keywords. Other tree-based keyword search methods need to explore at least all the content nodes in G or the entire graph to find a Steiner tree to cover the input keywords. Since our r-clique finding algorithm is also fast due to the fact that only the content nodes are explored during the search, the total time spent on finding r-cliques and then trees is much less than finding Steiner trees directly from G.

Algorithm 3 FindTopRankedAnswer Procedure

Input: the search space S; the input graph G; the number of query keywords l and r

Output: the best r-clique in the search space S

for $i \leftarrow 1$ **to** l **do**
  $S_i \leftarrow$ S.**get**(i)
for $i \leftarrow 1$ **to** l **do**
  for $j \leftarrow 1$ **to** **size**(S_i) **do**
    $d(s_i^j, i) \leftarrow 0$
    $n(s_i^j, i) \leftarrow s_i^j$
for $i \leftarrow 1$ **to** l **do**
  for $j \leftarrow 1$ **to** **size**(S_i) **do**
    for $k \leftarrow 1$ **to** l ; $k \neq i$ **do**
      $\langle dist, nearest \rangle \leftarrow$ shortest path from s_i^j to S_k
      if $dist \leq r$ **then**
        $d(s_i^j, k) \leftarrow dist$
        $n(s_i^j, k) \leftarrow nearest$
      else
        $d(s_i^j, k) \leftarrow \infty$
        $n(s_i^j, k) \leftarrow \emptyset$
$leastWeight \leftarrow \infty$
$topAnswer \leftarrow \emptyset$
for $i \leftarrow 1$ **to** l **do**
  for $j \leftarrow 1$ **to** **size**(S_i) **do**
    if $\forall k : [1 \ldots l], d(s_i^j, k) \leq r$ **then**
      $weight \leftarrow \sum_{h=1}^{l} d(s_i^j, h)$
      if $weight < leastWeight$ **then**
        $leastWeight \leftarrow weight$
        $topAnswer \leftarrow \langle n(s_i^j, 1), \ldots, n(s_i^j, l) \rangle$
return $topAnswer$

F. NEIGHBOR INDEXING METHOD

In the above algorithms, we need to compute the shortest distance between each pair of nodes. Calculating the shortest path on the fly is not feasible and it increases the running time of the algorithm. An index that stores the shortest distance and path between nodes improves the performance of the algorithm. 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 a simple and fast indexing method that pre-computes and stores the shortest distances for only the pairs of nodes whose shortest distance is within a certain threshold R. The index is called *neighbor index*. The value of R should be bigger than the value of r used in the r-clique finding algorithms. This requires the estimation of possible r values based on the graph structure and user preferences and may be estimated using the domain knowledge. At the same time, we should keep it as small as possible to keep the index in a feasible size. The idea of indexing the graph using a distance threshold has been used in [13, 12].

Table 2: Keywords used in DBLP data set.

Frequency	Keywords
0.0003	distance, discovery, scalable, protocols
0.0006	graph, routing, space, scheme
0.0009	fuzzy, optimization, development, support, environment, database
0.0012	modeling, logic, dynamic, application
0.0015	control, web, parallel, algorithms

Table 3: Set of queries used for finding the accuracy of the results in DBLP data set.

Query	Keywords
1	parallel, graph, optimization, algorithm
2	dynamic, fuzzy, logic, algorithm
3	graph, optimization, modeling,
4	development, fuzzy, logic, control

Algorithm 4 ProduceSubSpaces Procedure

Input: the best answer of previous step, $A = \langle v_1, v_2, \dots v_l \rangle$, and the sets of content nodes, $S_1, \dots, S_l$

Output: l new subspaces

for $i \leftarrow 1$ **to** l **do**
    for $j \leftarrow 1$ **to** $i - 1$ **do**
        $SB_i^j \leftarrow \{v_j\}$
    $SB_i^i \leftarrow S_i - \{v_i\}$
    for $j \leftarrow i + 1$ **to** l **do**
        $SB_i^j \leftarrow S_j$
return $\langle SB_1, \dots, SB_l \rangle$ where $SB_i = SB_i^1 \times \dots \times SB_i^l$

Algorithm 5 Generating Steiner Tree Algorithm based on an algorithm introduced in [9]

Input: an r-clique generated from graph G

Output: the Steiner tree of G that spans the nodes in the r-clique

1: Let G_1 be the input r-clique.
2: Find the minimal spanning tree T_1 of G_1.
3: Create graph G_2 by replacing each edge in T_1 by its corresponding shortest path in G. The shortest path can be obtained by using the neighbor index on G described in the next section.
4: Find the minimal spanning tree T_2 of G_2.
5: Create an Steiner tree from T_2 by removing the leaves (and the associated edges) that are not in the r-clique.

The *neighbor index* of a graph G with respect to the distance threshold R is structured as follows. For each node n, a list is created to contain the nodes that are within R distance from node n. This list is called the *neighbor list* of n. In each node m of the *neighbor list* of node n, the shortest distance between n and m is stored and also a pointer to the node right before m on the shortest path from n to m is stored. The pointed node p must be within R distance from n, is thus on n's neighbor list and contains a pointer to the node right before p on the shortest path between n and p. The space complexity of this index is $O(mn)$, where n is the number of nodes in G and m is the average number of nodes on a neighbor list. To build the index we use the Dijkstra's algorithm to compute the shortest path between each pair of nodes.

When finding r-cliques, both inverted and neighbor indexes are used to retrieve the shortest distance from a node, n, containing keyword k_1, to a node, m, containing keyword k_2 by first looking up the inverted index list for k_1 to locate the entry for node n and then search the neighbor list of n for node m. If the neighbor list contains node m, the stored shortest distance is returned. Otherwise, nodes n and m are not within R distance from each other. The shortest path between n and m (which is used in the Steiner tree finding algorithm) can be found by following the pointer stored in the m node in n's neighbor list, which points to the node right before m on the shortest path. In our experiments, the whole neighbor index is loaded into the main memory. For larger data sets or larger R values, the index may need to be disk resident. A performance study that distinguishes between cold/warm cache timings is an item of future work.

G. DATA SETS AND QUERIES IN EXPERIMENTS

To evaluate the proposed algorithms, we use the DBLP and IMDb data sets. The input graphs are undirected and weighted. 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 [13, 7, 3].

The DBLP graph is produced from the DBLP XML data (http://dblp.uni-trier.de/xml/). 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 613K, 929K, 2,375K, and 82K respectively. The set of input keywords and their frequencies in the input graph are presented in Table 2. The queries used in our experiments are 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). Noted that the set of input keywords and the way to generate queries are the same as the ones in [13].

To evaluate the search accuracy through a user study, four queries are formed from the set of keywords in Table 2. The set of four queries are presented in Table 3. These four queries are formed to be meaningful so that it is more convenient for the users to evaluate the relevancy of the search results.

The IMDb dataset contains the relations between movies and the users of the IMDb website that rate the movies (http://www.grouplens.org/node/73). The numbers of tuples of 3 relations *user*, *movie* and *rating* are 6.04K, 3.88K and 1,000.21K, respectively. The set of input keywords and the frequencies are presented in Table 4. Note that the set of input keywords is the same as the one used in [13].

Table 4: Keywords used in IMDb data set.

Frequency	Keywords
0.0003	game, summer, bride, dream
0.0006	Friday, street, party, heaven
0.0009	girl, lost, blood, star, death, all
0.0012	city, world, blue, American
0.0015	king, house, night, story