Compact Group Discovery in Attributed Graphs and Social Networks

Abeer Khan[*], Lukasz Golab[*],
Mehdi Kargar[+,#], Jaroslaw Szlichta[$], Morteza Zihayat[+]

[*] *University of Waterloo, Canada*
[+] *Ted Rogers School of Management, Ryerson University, Canada*
[$] *Ontario Tech University, Canada*

[#] *corresponding author: kargar@ryerson.ca*

Abstract

Social networks and many other graphs are attributed, meaning that their nodes are labelled with textual information such as personal data, expertise or interests. In attributed graphs, a common data analysis task is to find subgraphs whose nodes contain a given set of keywords. In many applications, the size of the subgraph should be limited (i.e., a subgraph with thousands of nodes is not desired). In this work, we introduce the problem of compact attributed group (AG) discovery. Given a set of query keywords and a desired solution size, the task is to find subgraphs with the desired number of nodes, such that the nodes are closely connected and each node contains as many query keywords as possible. We prove that finding an optimal solution is NP-hard and we propose approximation algorithms with a guaranteed ratio of two. Since the number of qualifying AGs may be large, we also show how to find approximate top-k AGs with polynomial delay. Finally, we experimentally verify the effectiveness and efficiency of our techniques on real-world graphs.

Keywords: Attributed Graphs, Social Networks, Graph Data Management, Approximation Algorithms

1. Introduction

Graphs have become a significant part of modern data analysis due to their ability to express relationships among entities [1]. Examples include social networks

such as Facebook, LinkedIn, Twitter and GitHub, where nodes are persons and edges model various types of connections between persons; webpages, where edges correspond to hyperlinks; e-commerce, where graphs can model interactions among products and customers; and relational databases, where nodes are tuples edges correspond to foreign key connections.

Many graphs are node-attributed, also known as node-labelled, meaning that each node is associated with descriptive text. For example, nodes in the LinkedIn graph may be labelled with persons' skills and locations, and nodes in a product graph may be labelled with product attributes. Query processing over attributed graphs has recently attracted a great deal of attention [2, 3, 4, 5, 6, 7, 8]. Here, a query consists of a set of keywords and an answer is a subgraph whose nodes contain the query keywords. A node that contains at least one query keyword is called a *content node*; however, there may be other nodes in the answer whose purpose is only to connect the content nodes.

In this work, we introduce a new search problem over attributed graphs. Given a set of query keywords and a size range (in terms of the number of content nodes), we want to find closely-connected groups of content nodes whose number is within the desired range and such that each content node covers as many query keywords as possible. We call such a group a compact ***attributed group***, abbreviated **AG**. In contrast to prior work (details in Section 2), the number of content nodes in an AG is bounded and there is no need to specify a node to build around. Furthermore, each content node in an AG includes many query keywords. Our problem is motivated by the following real-world applications:

- **Targeted Marketing in Social Networks.** Suppose a social media company wants to advertise rock climbing tours to groups of users. It constructs a social network of its users, with each node (person) labelled with their hobbies or interests. The company wants to target groups of people who already know each other (i.e., are connected in the social network), are interested in rock climbing, and would motivate each other to attend the tour. Due to budget limitations, ads cannot be sent to every possible group of users. Instead, the

company can find AGs containing keywords such as “rock climbing” whose sizes are within the minimum and maximum tour size (e.g., between 20 and 30 people).

- **Hiring Experts in Expert Networks.** Suppose we want to hire a group of experts to work on a project. Assume the project requires expertise in certain areas (e.g., databases and ontologies), and that budgetary and logistic constraints dictate a lower and an upper bound on the size of the group. To ensure the team works efficiently together, they should have mutual collaborators. We are given a network of experts that captures their past collaborations (e.g., DBLP or GitHub) and their skill sets. To solve the problem, we can find AGs in the network that are closely connected (i.e., have collaborated in the past or have mutual collaborators), have the desired size, and cover as many required skills as possible.

- **Photo and Post Recommendations in Social Networks.** In social networks such as Instagram, where users share photos (and posts), one might be interested in finding groups of similar photos. The underlying connections among users (whether they follow each other or like each others' photos) provide the graph structure. Each photo is tagged with a set of keywords called hashtags. It has been shown that hashtags and visual features can help extract communities of related images [9]. Given a set of query keywords and the required number of photos, we can search for groups of photos that were taken by related users and whose hashtags contain the query keywords. Note that screen size, or paper size if the photos are to be printed, restricts the number of photos in a group. The same method applies to social media posts, e.g., on Twitter.

Our contributions are as follows.

1. We define the novel attributed group (AG) query problem over attributed graphs and we propose new objective functions to rank the results.

2. We prove that optimizing these objectives is NP-hard. Thus, we propose approximation algorithms with a guaranteed ratio of two to find the answers ef-

ficiently.

3. Since the total number of answers is exponential in the number of query keywords and the size of the group, we propose a procedure to find approximate top-k groups with polynomial delay.

4. We experimentally compare our methods against existing approaches on real-world large graphs, showing that our algorithms efficiently find compact groups with the desired size and keyword coverage.

The remainder of this paper is organized as follows. Related work is presented in Section 2. Section 3 formally defines the problem and Section 4 introduces the approximation algorithms. Our method for enumerating top-k answers with polynomial delay is presented in Section 5. Experiments are presented in Section 6, and Section 7 concludes the paper.

2. Related Work

We first discuss the differences between our problem definition and existing work in community search and keyword search. Next, we provide a detailed discussion of recent related work in these and other related areas. In community search, the input consists of a specific node from the graph plus a set of keywords, and the output is a dense subgraph, built around the given node, each of whose nodes contains as many query keywords as possible. One problem with community search is that the output may be very large, perhaps thousands of nodes. On the other hand, keyword search over attributed graphs returns a subgraph, often in the form of a tree, in which each query keyword is covered by at least one node. Here, the problem is that the number of content nodes is at most equal to the number of input keywords, which might not be enough for some applications. Furthermore, each content node usually cover only one keyword and multiple keyword coverage per content node is not supported.

To illustrate the difference between this work and existing techniques, we present an attributed graph in Figure 1. This graph shows a portion of the GitHub network

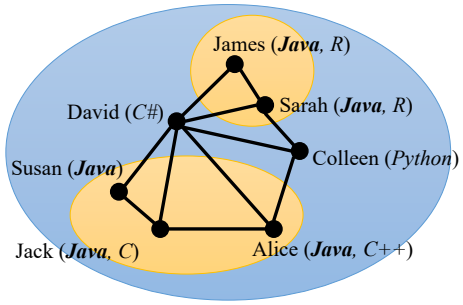

Figure 1: Compact groups vs. communities in attributed graphs.

with seven developers and the programming languages of their expertise. Two developers are connected if they worked on the same project in the past. Suppose we want to find a group of five *Java* developers. A community search (CS) algorithm such as the one in [4] could give two possible answers depending on the node it starts with, neither of which has size five: {Susan, Jack, Alice} or {James, Sarah}. Some CS algorithms additionally require a minimum degree of each node in a community as input. If the minimum degree were set to three, then no output would be generated. On the other hand, if we pass the query *Java* to a graph keyword search algorithm such as the one in [10], it would return a minimal subgraph where each query keyword is covered by at least one node, i.e., a single *Java* expert such as Alice. In contrast, our method returns a group with all five *Java* developers in the network as they are either directly connected or connected through mutual collaborators (similar to the notion of "friends of friends").

The attributed graph in Figure 2 represents researchers and their areas of expertise (portion of the DBLP dataset). Two researchers are connected if they have co-authored a paper. Suppose we have three query keywords: Databases (DB), Information Retrieval (IR), and Machine Learning (ML). We are interested in finding a compact subgraph with three nodes. If the minimum degree is set to 1, 2, or 3, CS returns the entire graph, which is too large. If it is set to 4, CS does not return anything. Graph keyword search returns two nodes that cover all three input keywords. In contrast, our method returns the subgraph coloured in blue, which contains three nodes, as required.

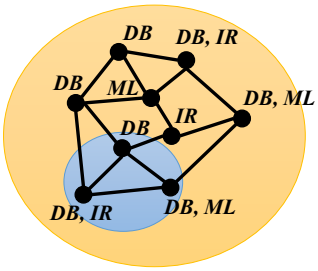

Figure 2: Compact groups vs. communities in attributed graphs.

We present a summary of related work in Table 1, showing that no previous work can satisfy all the requirements of our motivating applications: handling attributed graphs, ensuring the resulting groups are compact and well connected (optimizing distance), have the desired size, and that each node covers as many query keywords as possible (multiple keyword coverage). Below, we discuss each related work in detail.

Community Detection finds *all* densely connected subgraphs in an input graph [11]. Early solutions use network structure to find communities [12, 13] but do not consider attributed graphs. Some recent work focuses on finding communities in attributed graphs and considers structural and textual similarity, meaning that nodes within a community must be closely connected and must have similar textual content; see, e.g., [14, 15, 16]. Furthermore, Hajiabadi et. al propose an algorithm to extract overlapping communities [17].

Community detection algorithms are generally slow and unsuitable for online community retrieval since they usually enumerate *all* the communities in the input graph [4]. Community detection has major differences with this work. First, there is no limit on the number of nodes: a community might contain hundreds or thousands of nodes. Second, algorithms for finding communities usually do not consider the distance between all nodes (or, in attributed graphs, between content nodes) within a community. Instead, they minimize density-based objectives such as degree or edge betweenness. In such algorithms, some nodes in the same community might be far away from each other although they all satisfy the density objective

Table 1: Comparison of related work

Topic	Attributed Graphs	Optimizing Distance	Controlled Size	Multiple Keyword Coverage
Community Detection	[14, 15, 16]	No	No	[14, 15, 16]
Community Search	[4, 18]	No	No	[4, 18]
Keyword Search	[19, 20, 21]	[19, 20, 21]	Not Explicitly	No
Compact Attributed Groups	Yes	Yes	Yes	Yes

(e.g., minimum degree).

Community Search is recent approach for obtaining *specific* communities, namely those centered around a specific query node [22]. To determine structural relevance, most algorithms use density-based metrics such as minimum degree. For example, several algorithms such as those in [23, 24] find k'-cores around the query node (a k'-core is a subgraph in which each node has a degree of at least k'). Furthermore, Li et al. [22] recently defined a new model to capture the influence of a community and proposed an algorithm to search for communities with high influence. A few works have recently explored the problem of finding spatially-aware communities that satisfy structural (using degree) and geo-spatial constraints [25, 26]. However, these works study non-attributed graphs and therefore do not consider textual information attached to nodes. Li et al. [27] developed a community search algorithm to find influential communities, where each node has multiple *numeric* attributes representing influence.

Fang et al. [4] recently proposed a community search algorithm for attributed graphs. The graph is queried with a set of keywords T, an input node n, and a minimum degree k'. The results are the connected components of the largest dense subgraph around n whose nodes contain the keywords in T. Each connected component is considered to be a community (i.e., k'-core). However, these communities may be very large. Similarly, the authors of [18] proposed a community search algorithm that receives a *group* of nodes N as input and builds a community around N. In [18], k'-truss is used to measure the density of a subgraph (a k'-truss is a subgraph in which each edge is part of at least $(k'-2)$ triangles within the subgraph). In contrast, we search the entire graph and do not receive an input node n (or a set

of nodes N). We also control the size of the output and optimize the distance between content nodes (rather than density). Finally, we explicitly take a parameter k to return approximate top-k best AGs.

Keyword Search in Attributed Graphs finds subgraphs whose nodes collectively cover all (or most) of the query keywords. Solutions to this problem can be categorized by the type of answers they produce: tree-based methods and graph-based methods.

Bhalotia et al.[28] proposed a backward search algorithm and Ding et al. [29] presented a dynamic programming approach to find answer trees. These methods have exponential time complexity in the number of keywords, but polynomial time complexity in the number of nodes of the input graph. To find trees with distinct roots, an algorithm was proposed in [30] and was improved by He et al. [31]. For graphs that do not fit in memory and use the distinct root semantics, Dalvi et al. [32] created a smaller graph on top of the input graph that resolves the memory limitations of previous work.

Li et al., [19] find r-radius Steiner graphs and [20] finds multi-centered subgraphs called communities. Each community contains some center nodes such that there exists a path between each center node and each content node that is shorter than a given threshold. This parameter controls the size of a community. The authors of [10] find r-cliques as answers to keyword search over graphs. In an r-clique, all content nodes are guaranteed to be close to each other [10]. Recently, the authors of [21] find duplication free and minimal answers. Authors of [33] propose a framework to group and summarize answers to graph keyword search based on similarity in content and structure.

Keyword search in graphs is significantly different from this work. First, in keyword search, each input keyword is covered by exactly one content node[1]. This is undesirable in applications where users prefer content nodes that cover as many keywords as possible. Thus, we provide more flexible keyword coverage as we allow

[1] A keyword may be covered by more than one content node, but the goal is to assign one keyword to one content node.

multiple content nodes (not just one content node) to cover each query keyword. Second, unlike our model, the number of content nodes is not explicitly specified. Graph keyword search algorithms could generate answers with few content nodes, much smaller than the required size. The same as our model, the algorithms for graph keyword search do not explicitly optimize node degree but optimize proximity of the answer subgraph (e.g., minimizing the diameter of the answer subgraph or minimizing the number of edges of the answer tree).

3. Preliminaries and Problem Statement

Given a node-labeled graph G, a range for the required number of content nodes (i.e., the size of the group), and a set of query keywords, we want to find and rank a set of compact ***attributed groups*** **(AG)** which are subgraphs of G, contain the desired number of content nodes, and are close to each other. Furthermore, each content node should contain as many query keywords as possible. For now, as long as a content node contains at least one query keyword, it could be part of a group R. We build on this later by assigning a score to each content node that depends on the number of query keywords it covers.

Definition 1. ***Attributed Group (AG):*** *Given a graph G, a set of query keywords $T = \{t_1, t_2, \ldots, t_p\}$, and a range for the required number of content nodes containing at least one keyword each ($size_{min}$ and $size_{max}$), an* ***attributed group (AG)*** *is a subgraph of G that is composed of a set of q content nodes ($size_{min} \leq q \leq size_{max}$) that are close to each other. Each pair of content nodes in the AG is connected via their shortest path. The nodes and the edges on these shortest paths are part of the AG subgraph.*

Throughout this paper, $\boldsymbol{p}$ indicates the number of input keywords and $\boldsymbol{q}$ indicates the number of content nodes of an AG. The graph could be weighted or unweighted. In weighted graphs, weights model the semantic importance of connections between the two end points. For any pair of nodes n_i and n_j, the distance between them (i.e., $dist(n_i, n_j)$) is defined as the shortest path between them in G. This corresponds to the proximity of the two nodes. The function $c(n, t)$ determines

whether a node n contains a keyword t. If it does, $c(n,t) = 1$, otherwise $c(n,t) = 0$. We assume that G is undirected (although our approach can also work with directed graphs[2]).

The content nodes in an AG might be connected to each other via some "middle" nodes that do not contain any query keywords. However, and unless stated otherwise, when we refer to the nodes in an AG, we are referring to its content nodes. Each AG has a weight corresponding to the weights of the edges in G that connect the constituent content nodes. Below, we define a weight function to measure the proximity of the content nodes in an AG.

Definition 2. ***Proximity Weight:*** *For a given set of query keywords $T = \{t_1, t_2, \ldots, t_p\}$, let $\{n_1, n_2, \ldots, n_q\}$ be the set of content nodes in an AG R in which $\forall n_i \; \exists t_r \; | c(n_i, t_r) = 1; 1 \leq i \leq q; 1 \leq r \leq p$. The proximity weight (PW) of R is: $PW(R) = \frac{\sum_{i=1}^{q} \sum_{j=i+1}^{q} dist(n_i, n_j)}{\binom{q}{2}}$ where $dist(n_i, n_j)$ is the distance (shortest path) between nodes n_i and n_j in graph G.*

Proximity weight returns the average distance between every pair of content nodes. Since the size of an AG can vary between $size_{min}$ and $size_{max}$, we use average distance, not the sum of the distances, to avoid penalizing large AGs. AGs with smaller weights are more desirable since their content nodes are closer to each other. Now, we define the following problem.

Problem 1. *Given a graph G, a range for the required number of content nodes ($size_{min}$ and $size_{max}$) and a set of query keywords $T = \{t_1, t_2, \ldots, t_p\}$, find an AG R with q content nodes in G (in which $size_{min} \leq q \leq size_{max}$) with the minimum proximity weight $PW(R)$.*

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

[2] In some dense group applications (e.g., photo recommendation in Twitter), we can convert a directed graph into an undirected graph by removing the directions from the edges. In these applications, similarity is defined in both ways: if there is an edge between two nodes, this implies the end nodes are related, regardless of the direction. Another way to find the length of a shortest path in a directed graph is by using edges in both directions. In this case, the length of a shortest path could be the average length in both direction [21].

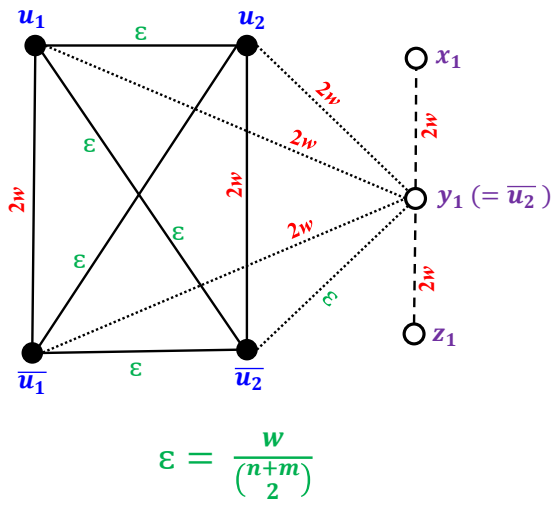

Figure 3: Proving the NP-hardness of Problem 1. Note that some edges are omitted for clarity. There are three types of edges, each shown with a different line style. We want to see whether there exists an AG with proximity weight w. Clearly, any edge with weight $2w$ violates this requirement.

Proof 1. *We prove that the decision version of this problem is NP-hard, implying that Problem 1 is also NP-hard. The decision version asks whether there exists an AG R with proximity weight w for some constant w. We prove the theorem by reduction from 3-satisfiability (3-SAT).*

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 define an instance of the above problem as follows. For each pair of variables* u_s *and* $\overline{u}_s$, *two nodes are created (i.e.,* u_s *is the negation of* $\overline{u}_s$*). The distance between each pair of* u_s *and* $\overline{u_s}$ $(1 \le s \le n)$ *is set to* $2w$. *We set the distance between other variables to* $\frac{w}{\binom{n+m}{2}}$. *Furthermore, for every clause* D_k, *we create three nodes,* x_k, y_k *and* z_k. *The distance between each of these nodes to their associated variable node is set to* $\frac{w}{\binom{n+m}{2}}$ *(e.g., if* $y_1 = \overline{u_2}$, *the distance between the node that represents* y_1 *to the node that represents* $\overline{u_2}$ *is* $\frac{w}{\binom{n+m}{2}}$*). Their distance to other nodes is set to* $2w$. *Therefore, we have* $(2.n + 3.m)$ *nodes in total. We assume all of these nodes contain at least one query keyword* $t_r; 1 \le r \le p$ *(i.e., all of them are content nodes) and the number of required content nodes is* $n + m$ *(i.e.,* $size_{min} = size_{max} = q = n + m$*). A sample of this mapping graph is presented in Figure 3.*

A feasible solution to this problem with weight at most w is any set of content nodes such that from each pair of nodes associated with u_s and $\overline{u_s}$, exactly one is selected, and from each triplet of nodes associated with x_k, y_k and z_k, one is selected. This is because selecting two pairs, u_s and $\overline{u_s}$, in one answer gives a sum of distances exceeding w; recall that the distance between u_s and $\overline{u_s}$ is set to $2w$. Using the same logic, if we select two nodes from one clause D_k in one answer, the sum of distances exceeds w since the distance between members of D_k is set to $2w$. Therefore, if there exists a subset with weight at most w, then we find a satisfying assignment for $D_1 \wedge D_2 \wedge \cdots \wedge D_m$. Also, a satisfying assignment is equivalent to a feasible solution, which is a set of nodes with weight at most w. This completes the proof. □

Now, we also want to optimize query keyword coverage. We define the following score for each node based on the query keywords.

Definition 3. ***Keyword Score of a Node:*** *Given p query keywords $T = \{t_1, t_2, \ldots, t_p\}$, the keyword score of a node n is defined as follows: $s(n) = 1 - \frac{\sum_{r=1}^{p} c(n,t_r)}{p}$.*

The smaller the keyword score of n, the more keywords are covered by it, which gives a minimization problem (similar to minimizing the proximity weight). Note that $0 \leq s(n) \leq 1$.

Now, we define the keyword score for an entire AG.

Definition 4. ***Keyword Score of an AG:*** *Let $\{n_1, n_2, \ldots, n_q\}$ be the list of content nodes in an AG R in which $\forall n_i \; \exists t_r \; | c(n_i, t_r) = 1; 1 \leq i \leq q; 1 \leq r \leq p$. The keyword score (KS) of R is defined as $KS(R) = \frac{\sum_{i=1}^{q} s(n_i)}{q}$, where $s(n_i)$ is the keyword score of node n_i from Definition 3.*

As in the proximity weight, we take the average of keyword scores. We now define the following minimization problem to optimize the keyword score of an AG.

Problem 2. *Given a graph G, the desired AG size ($size_{min}$ and $size_{max}$), and p query keywords $T = \{t_1, t_2, \ldots, t_p\}$, find an AG R with q content nodes ($size_{min} \leq q \leq size_{max}$) in G with minimal keyword score $KS(R)$.*

We can find an optimal solution to this problem in polynomial time by choosing $size_{min}$ content nodes from G that contain the most query keywords. However, the content nodes might be far away from each other in G since we have ignored the distance between them. Therefore, we are interested in finding an AG that minimizes both proximity weight and keyword score. This is a bi-objective optimization problem.

One way to solve a multi-objective optimization problem is to optimize a weighted sum of the objective functions. This converts the original multi-objective optimization problem into a single-objective optimization problem. This approach is called the **Weighted Sum Method** and has been shown to be effective in solving bi-objective optimization problems [34, 35]. Note that the individual objective functions may not be related (e.g., optimizing hourly rate and workload of employees) [34], but must be normalized to allow a trade-off parameter combine them into a single objective function. In this work, we use the weighted sum method to solve our bi-objective optimization problem. We define a single objective function that combines the proximity weight and keyword score of an AG with a trade-off parameter λ as follows.

Definition 5. ***Combined Score:*** *Let $\{n_1, n_2, \ldots, n_q\}$ be the list of content nodes in an AG R in which $\forall n_i \; \exists t_r \; | c(n_i, t_r) = 1; 1 \leq i \leq q; 1 \leq r \leq p$. Let $0 < \lambda < 1$ be a trade-off parameter between the proximity weight and keyword score of the AG R. The combined score (CS) of R is defined as follows*

$$CS(R) = (1 - \lambda).KS(R) + \lambda.PW(R)$$

The parameter λ varies from 0 to 1 and indicates the trade-off between the proximity weight and keyword score of an AG. λ is application-dependent and we leverage domain expert feedback to set its value. Furthermore, edge weights and keyword scores of nodes might have different scales. As mentioned earlier, they should be normalized before being used in the combined score function.

Given the combined cost function, we define the following problem.

Problem 3. *Given a graph G, the desired AG size ($size_{min}$ and $size_{max}$), p query keywords $T = \{t_1, t_2, \ldots, t_p\}$, and a trade-off parameter λ ($0 < \lambda < 1$), find an AG R with q content nodes ($size_{min} \leq q \leq size_{max}$) in G with minimum combined score $CS(R)$.*

Theorem 2. *Problem 3 is NP-hard.*

Proof 2. *We proved Theorem 1 by showing that finding an AG R from graph G while minimizing the proximity weight ($PW(R)$) is an NP-hard problem. Since $PW(R)$ is linearly related to $CS(R)$, minimizing $CS(R)$ is also an NP-hard problem.* □

4. Algorithms

We now present two approximation algorithms to solve the NP-hard problems introduced in previous section (Problems 1 and 3). We also prove that our algorithms achieve an approximation ratio of two.

4.1. Optimizing the Proximity Weight

Algorithm 1 is our solution to Problem 1 for finding an AG R that minimizes the proximity weight function. The algorithm takes as input a graph G, the required range for the size of the AG ($size_{min}$ and $size_{max}$), and a set of query keywords T. It returns an AG whose weight is at most twice that of an optimal AG when the distance function satisfies the triangle inequality. We assume the existence of an inverted index on the keywords associated with nodes.

The intuition behind our algorithm is as follows. For each node cn that contains at least one query keyword, we form a subgraph around cn by adding other content nodes that are close to cn. The number of selected content nodes depends on the required range for the size of the AG. As we explain (and prove) later, this process produces answers with an approximation ratio of two based on the proximity weight defined in Definition 2.

In line 1, the set of nodes, B, that contain at least one query keyword is obtained from the inverted index. Line 2 initializes the best AG to the empty set and its weight

Algorithm 1 Finding the Best Approximate Attributed Group (AG)

Input: graph G; $size_{min}$; $size_{max}$; query keywords $T = \{t_1, t_2, \ldots, t_p\}$
Output: best approximate AG

1: $B \leftarrow$ set of nodes that contain at least one query keyword (from an inverted index)
2: $bestGroup \leftarrow \emptyset$; $leastWeight \leftarrow +\infty$
3: **for** $i \leftarrow 1$ **to** $|B|$ **do**
4: $cn_1 \leftarrow B$.get(i) ; $group \leftarrow \emptyset$
5: $group$.add(cn_1); $totalWeight \leftarrow 0$
6: **for** $j \leftarrow 2$ **to** $size_{max}$ **do**
7: $minPathValue_j \leftarrow ew_{close}(cn_1, B, j-1)$; $cn_j \leftarrow no_{close}(cn_1, B, j-1)$
8: **if** $cn_j \neq \emptyset$ **then**
9: $totalWeight \leftarrow totalWeight + minPathValue_j$; $group$.add(cn_j)
10: **if** $j \geq size_{min}$ **then**
11: $weight \leftarrow \frac{totalWeight}{j}$
12: **if** $weight < leastWeight$ **then**
13: $leastWeight \leftarrow weight$; $bestGroup \leftarrow group$
14: **return** $bestGroup$

to infinity. In lines 3 to 13, for each content node in B, a group is constructed around it. The current content node, labelled cn_1, in line 4 is the first content node in the current group. For each cn_1, a new $group$ is initialized in line 4. In line 5, cn_1 is added to the $group$ and the total weight of the current $group$ is set to 0. Then, we add more content nodes (up to the maximum size) around cn_1 to the current $group$ by finding the nodes in B that are closest to cn_1 (by adding the closest content node, and then the second closest content node and so on). These nodes are called cn_2, $cn_3, \ldots, cn_q$ ($size_{min} \leq q \leq size_{max}$). In line 7, we find the $j-1$th closest node and its distance to cn_1. Note that $ew_{close}(n, N, k)$ denotes the value of the kth nearest distance between node n and a set of nodes N. Similarly, $no_{close}(n, N, k)$ denotes the kth nearest node in N to n. If the algorithm finds such a node (line 8), the current weight and the current group are updated in line 9. If the current group contains at least $size_{min}$ content nodes, the current weight is compared to the lowest weight so far. If it is smaller than the lowest weight, the lowest weight and the best group are updated in line 13. The best approximate group is returned in line 14. Note that the weight of every group around cn_1 that is within the desired size range is compared with the lowest weight so far.

The above operations can be computed in polynomial time. We use the 2-hop

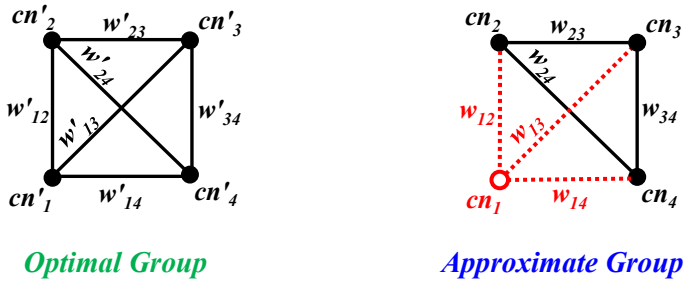

Figure 4: Optimal and approximate groups for four required content nodes. Note that w_{ij} and w'_{ij} are the shortest distances between the associated content nodes. In this example, and without loss of generality, we assume that the approximation algorithm forms the best approximate answer around node cn_1.

cover index to find shortest paths [36]. Let $Dist_{time}$ be the time to return the distance between a pair of nodes using the 2-hop cover index [3]. The complexity of Algorithm 1 is $O(size_{max}.|B|^2.Dist_{time})$, where $|B|$ is the number of content nodes. Note that $Dist_{time}$ is very small (under 2 μs in our experiments).

The shortest path between each pair of nodes in G can be found using Dijkstra's algorithm in graphs with positive edge weights. In addition, in graphs with positive edge weights, the shortest path function (also known as the distance function) is a metric and satisfies the triangle inequality. In this case, Algorithm 1 produces nearly-optimal groups with a ratio of two. We first provide an example and then the proof.

Figure 4 shows an optimal group and an approximate group produced by Algorithm 1 assuming the required number of content nodes is exactly four. Note that these groups have similar structure (both have four content nodes) but the content nodes (cn_i in the approximate group vs cn'_i in the optimal group) and the shortest paths (w_{ij} in the approximate group vs w'_{ij} in the optimal group) are **different**.

The weight of the optimal group is defined as follows:

$$optimal\ weight = w'_{12} + w'_{13} + w'_{14} + w'_{23} + w'_{24} + w'_{34} \tag{1}$$

[3] The complexity of answering distance queries between two nodes s and t using the 2-hop cover index is $O(L(s) + L(t))$, where $L(s)$ and $L(t)$ are the sizes of the associated sets of nodes for s and t in the 2-hop cover index. See [36] for details.

Without loss of generality, assume the best approximate group is produced around content node cn_1 (i.e., the node in line 4). The sum of the edge weights (i.e., shortest distance values) of nodes connected to cn_1 (i.e. w_{12}, w_{13} and w_{14}) in the approximate group is the smallest among all the content nodes. Therefore, the following four statements are valid[4] (this is being optimized in Algorithm 1):

$$\begin{cases} cn_1 \text{ vs. } cn'_1: & w'_{12} + w'_{13} + w'_{14} \geq w_{12} + w_{13} + w_{14} \\ cn_1 \text{ vs. } cn'_2: & w'_{12} + w'_{23} + w'_{24} \geq w_{12} + w_{13} + w_{14} \\ cn_1 \text{ vs. } cn'_3: & w'_{13} + w'_{23} + w'_{34} \geq w_{12} + w_{13} + w_{14} \\ cn_1 \text{ vs. } cn'_4: & w'_{14} + w'_{24} + w'_{34} \geq w_{12} + w_{13} + w_{14} \end{cases}$$

Note that the sum of the shortest paths from content node cn_1 to the other three content nodes is the **smallest** comparing to all other content nodes in the graph (including the content nodes of the optimal answer: $\{cn'_1, cn'_2, cn'_3, cn'_4\}$). This value is being optimized in the approximation algorithm (Algorithm 1). That is why the above four equations are all valid since we are comparing the weights of the shortest paths from cn_1 versus the weights of the shortest paths from cn'_1, cn'_2, cn'_3, and cn'_4. Then, we sum up both sides of these four inequalities:

$$2(w'_{12} + w'_{13} + w'_{14} + w'_{23} + w'_{24} + w'_{34}) \geq 4(w_{12} + w_{13} + w_{14})$$

This equation can be written as:

$$\frac{2}{4}(w'_{12} + w'_{13} + w'_{14} + w'_{23} + w'_{24} + w'_{34}) \geq (w_{12} + w_{13} + w_{14})$$

Using the above equation and Equation 1, we have:

[4] For simplicity, we do not divide each side of the inequality by the size of the group (i.e., four in this example).

$$\frac{2}{4}(optimal\ weight) \geq (w_{12} + w_{13} + w_{14}) \tag{2}$$

Recall that the shortest path is used as the distance function and it satisfies the triangle inequality. Based on the triangle inequality, these equations hold:

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

The weight of the group that is generated by the approximation algorithm is:

$$approximate\ weight = w_{12} + w_{13} + w_{14} + w_{23} + w_{24} + w_{34}$$

Below, we replace parts of the above approximate weight with the three inequalities in Equation 3. Using the first inequality in Equation 3 ($w_{12} + w_{13} \geq w_{23}$), we have:

$$w_{12} + w_{13} + w_{14} + (w_{12} + w_{13}) + w_{24} + w_{34} \geq approximate\ weight$$

Note that here we replace w_{23} with the larger value of $w_{12} + w_{13}$. Using the second inequality in Equation 3 ($w_{12} + w_{14} \geq w_{24}$), we have:

$$w_{12} + w_{13} + w_{14} + (w_{12} + w_{13}) + (w_{12} + w_{14}) + w_{34} \geq approximate\ weight$$

Finally, using the third inequality in Equation 3 ($w_{13} + w_{14} \geq w_{34}$), we have:

$$w_{12} + w_{13} + w_{14} + (w_{12} + w_{13}) + (w_{12} + w_{14}) + (w_{13} + w_{14}) \geq approximate\ weight$$

By simplifying the left hand side of this inequality, we have:

$$3 \times (w_{12} + w_{13} + w_{14}) \geq approximate\ weight$$

This can be written as:

$$(w_{12} + w_{13} + w_{14}) \geq \frac{1}{3}(approximate\ weight) \tag{4}$$

Based on Equation 2 and Equation 4, the following inequality holds:

$$\frac{2}{4}(optimal\ weight) \geq \frac{1}{3}(approximate\ weight)$$

This can be written as:

$$\frac{2 \times 3}{4}(optimal\ weight) \geq approximate\ weight \tag{5}$$

The value of the right side of this inequality (Equation 5) is the weight of the approximate group. The value of the left side of this inequality (Equation 5) is at most twice the weight of the optimal group ($2 > \frac{2\times3}{4}$) . Therefore, the weight of the approximate group is twice the weight of the optimal group in the worst case scenario. The complete proof is presented in the following theorem.

Theorem 3. *Algorithm 1 finds an AG R in G that minimizes the weight from Definition 2 with an approximation ratio of two as long as the distance function satisfies the triangle inequality.*

Proof 3. *Consider two groups, an* optimal group *and the* approximate group *produced by Algorithm 1. We prove that the proximity weight of the* approximate group *is at most twice the proximity weight of the* optimal group.

Let the size of the optimal group be equal to q, where $size_{min} \leq q \leq size_{max}$. Let the best node *be the node in the* approximate group *with the smallest sum of weights of the edges connected to it (i.e., the sum of the weights of the $q-1$ edges connected to the* best node *in the* approximate group *is the smallest among all the content nodes in the input graph G). The edge weights of the* best node *are denoted as $w_{12}, w_{13}, \ldots, w_{1q}$. Therefore, based on Algorithm 1, the value of $\sum_{i=2}^{q} w_{1i}$ is the smallest among all content nodes that contain at least one of the input keywords. Each node in the* optimal group *has $q-1$ neighbors. Also, $q-1$ edges are connected to each node. For each content node in the* optimal group*, the following equation holds:*

$$w'_{1j} + w'_{2j} + \cdots + w'_{j-1j} + w'_{jj+1} + \cdots + w'_{jq} \geq w_{12} + w_{13} + \cdots + w_{1q}$$

The above equation can be summarized as:

$$\sum_{i=1}^{j-1} w'_{ij} + \sum_{i=j+1}^{q} w'_{ji} \geq \sum_{i=2}^{q} w_{1i}$$

Here, w'_{ij} $i < j$ (w'_{ji} $i > j$) is the weight of the shortest path between content nodes i and j in the optimal group. If this equation is written for all q content nodes of the optimal group *and is summed up on both sides of the inequalities, we have:*

$$2 \times \sum_{i=1}^{q} \sum_{j=i+1}^{q} w'_{ij} \geq q \times \sum_{i=2}^{q} w_{1i}$$

On the left side, each edge appears twice because it connects two content nodes. Note that the left side of this inequality is twice the proximity weight of the optimal group. Therefore, it can be written as follows:

$$2 \times (optimal\ weight) \geq q \times \sum_{i=2}^{q} d_{1i} \tag{6}$$

Recall that the shortest path is used as the distance function and therefore the triangle inequality is satisfied as follows:

$$w_{ij} \leq w_{1i} + w_{1j}, i \neq j \neq 1$$

The weight of the approximate group *can be calculated using this equation:*

$$approximate\ weight = \sum_{i=1}^{q} \sum_{j=i+1}^{q} w_{ij} = \sum_{i=2}^{q} w_{1i} + \sum_{i=2}^{q} \sum_{j=i+1}^{q} w_{ij}$$

Since $w_{ij} \leq w_{1i} + w_{1j}$*, this inequality holds:*

$$\sum_{i=2}^{q} w_{1i} + \sum_{i=2}^{q} \sum_{j=i+1}^{q} w_{ij} \leq \sum_{i=2}^{q} d_{1i} + \sum_{i=2}^{q} \sum_{j=i+1}^{q} (w_{1i} + w_{1j})$$

On the right side of the above inequality, each weight w_{1i} *appears exactly* $q-1$ *times. Therefore, this holds:*

$$\sum_{i=2}^{q} w_{1i} + \sum_{i=2}^{q} \sum_{j=i+1}^{q} (w_{1i} + w_{1j}) = (q-1) \times \sum_{i=2}^{q} w_{1i}$$

Furthermore, this holds:

$$approximate\ weight \leq (q-1) \times \sum_{i=2}^{q} w_{1i} \tag{7}$$

Based on equations 6 and 7, we have:

$$\frac{2 \times (q-1)}{q} (optimal\ weight) \geq approximate\ weight$$

This complete the proof that the proximity weight of the approximate group *is at most twice the proximity weight of the* optimal group. □

4.2. Optimizing the Combined Score

In this section, we propose an approximation algorithm for solving Problem 3 that minimizes the combined score function with an approximation ratio of two. The idea is to convert the distance function from line 7 of Algorithm 1 to a new distance function that takes proximity and keyword score into account. Assume the shortest distance between two content nodes n_i and n_j in G is $dist(n_i, n_j)$. Note that both n_i and n_j contain at least one query keyword, i.e., $s(n_i) < 1$ and $s(n_j) < 1$. The new distance function between n_i and n_j with the given trade-off parameter λ

is defined as follows:

$$dist'(n_i, n_j) = (1-\lambda).(s(n_i) + s(n_j)) \; + \; 2.\lambda.dist(n_i, n_j) \tag{8}$$

Since $0 \leq s(n_i), s(n_j) \leq 1$, the value of $dist$ should be normalized to lie between 0 and 1 before calculating $dist'$. Below we show that the combined score of an AG R with q content nodes is the same as the sum of the distances of the q content nodes using the new distance function $dist'$.

Lemma 1. *For any AG R with q content nodes and for |T| query keywords, the following holds:*

$$\sum_{i=1}^{q} \sum_{j=i+1}^{q} dist'(n_i, n_j) = CS(R)$$

Proof 4. *Let the AG R be composed of q content nodes* $\{n_1, n_2, \ldots, n_q\}$. *Based on Definition 8, we have:*

$$\sum_{i=1}^{q} \sum_{j=i+1}^{q} dist'(n_i, n_j) =$$

$$\sum_{i=1}^{q} \sum_{j=i+1}^{q} ((1-\lambda).(s(n_i) + s(n_j)) + 2.\lambda.dist(n_i, n_j)) =$$

$$(1-\lambda) \sum_{i=1}^{q} \sum_{j=i+1}^{q} (s(n_i) + s(n_j)) \; + \; 2.\lambda \sum_{i=1}^{q} \sum_{j=i+1}^{q} dist(n_i, n_j) =$$

$$(q-1).(1-\lambda) \sum_{i=1}^{q} s(n_i) \; + \; 2.\lambda.PW(R) =$$

$$(q-1).(1-\lambda).KS(R) \; + \; 2.\lambda.PW(R) = \; CS(R)$$

□

The sum of distances of AG R with q content nodes using the new distance function $dist'$ is equal to the proximity weight of R using $dist'$. In other words, the following equation holds:

$$\sum_{i=1}^{q}\sum_{j=i+1}^{q} dist'(n_i,n_j) = PW_{dist'}(R)$$

in which $PW_{dist'}(R)$ is the proximity weight of AG R when the distance function $dist'$ is used to find the distance between content nodes n_i and n_j. Therefore, based on the above lemma, finding an AG R that minimizes the combined score function (Definition 5) is equivalent to finding an AG that minimizes the proximity weight function (Definition 2) using distance function $dist'$.

In Section 4.1, we proposed an algorithm to find an AG R that minimizes the proximity weight function with an approximation ratio of two when the distance function used in Definition 2 satisfies the triangle inequality. Below we show that the new distance function $dist'$ satisfies the triangle inequality.

Lemma 2. *The distance function $dist'$ that is defined in Equation 8 satisfies the triangle inequality.*

Proof 5. *The function $dist$ from Definition 8 is the shortest distance between two nodes n_i and n_j. The shortest distance satisfies the triangle inequality and therefore, the function $dist$ also satisfies the triangle inequality. Therefore, this holds: $dist(n_i,n_j) \leq dist(n_i,n_k) + dist(n_k,n_j)$, where n_i, n_j and n_k could be any three nodes in the input graph. Since $0 < \lambda < 1$, this statement is valid:*

$$2.\lambda.dist(n_i,n_j) \leq 2.\lambda.dist(n_i,n_k) + 2.\lambda.dist(n_k,n_j)$$

Therefore,

$$(1-\lambda).(s(n_i)+s(n_j)) + 2.\lambda.dist(n_i,n_j) \leq$$
$$(1-\lambda).(s(n_i)+s(n_j)) + 2.\lambda.dist(n_i,n_k) + 2.\lambda.dist(n_k,n_j) \leq$$
$$2.(1-\lambda).s(n_k) + (1-\lambda).(s(n_i)+s(n_j)) + 2.\lambda.dist(n_i,n_k) + 2.\lambda.dist(n_k,n_j)$$

Based on the definition of $dist'$, the last inequality states:

$$dist'(n_i, n_j) \leq dist'(n_i, n_k) + dist'(n_k, n_j)$$

Since n_i, n_j and n_k may be any nodes in G, $dist'$ satisfies the triangle inequality.

□

Since $dist'$ satisfies the triangle inequality, we can use the 2-approximation algorithm from Section 4.1 to find an AG R that minimizes the proximity weight with distance function $dist'$. Based on Lemma 1, such an AG also minimizes the combined score $CS(R)$.

Theorem 4. *Algorithm 1 with the new distance function $dist'$ finds an AG R in G that minimizes the weight defined in Definition 5 with an approximation ratio of two.*

Proof 6. *In Theorem 3, we proved that Algorithm 1 is a 2-approximation algorithm for finding an AG that minimizes the proximity weight objective when the distance function satisfies the triangle inequality. Since $PW_{dist'}(R)$ is equivalent to $CS(R)$ (according to Lemma 1), Algorithm 1 is a 2-approximation algorithm for finding an AG R that minimizes $CS(R)$.* □

5. Finding Approximate top-k Attributed Groups (AGs) with Polynomial Delay

Let B be the set of nodes containing at least one query keyword. Based on the definition of an AG, the total number of AGs is up to $(size_{max} - size_{min} + 1) \times \Big((|B|) \times (|B|-1) \times \cdots \times (|B| - size_{min} + 1) \Big)$ which is $O((size_{max} - size_{min} + 1).|B|^{size_{min}})$. Generating and presenting all AGs to the user is not feasible because $|B|$ (and also $size_{min}$) may have large values for large graphs. Since the number of AGs is exponential in the number of required content nodes (i.e., at least $size_{min}$), producing all AGs and then ranking them is not feasible. On the other hand, generating ranked top-k (e.g., $k = 50$) AGs is a feasible solution. The performance of an algorithm that produces a list of ranked answers (i.e., AGs in this work) can be evaluated based on the delay between generating two consecutive answers. When this delay is polynomial [37, 38], we get a *polynomial delay algorithm*.

Table 2: Dividing the search space into sub-spaces. The best AG is $\{n_1, n_2, n_3\}$ and the sub-spaces are disjoint.

Subspace	Inclusion set	Exclusion set
#1	$Inc_1 : \{n_1, n_2\}$	$Exc_1 : \{n_3\}$
#2	$Inc_2 : \{n_1\}$	$Exc_2 : \{n_2\}$
#3	$Inc_3 : \{\emptyset\}$	$Exc_3 : \{n_1\}$

Our solution for generating top-k AGs is an adaption of Lawler's technique for producing top-k answers [39]. At the beginning, the first best AG is produced from the entire search space (i.e., the entire graph). Then, based on the first best AG, the search space is divided into sub-spaces. The best AG in each sub-space is produced. Then, these AGs compete and the best one among them is used as the second best AG. The sub-space that generates the second best AG is divided into sub-sub-spaces and the best AG among its sub-sub-spaces is compared with the best AGs in other sub-spaces that were produced previously. The one with the lowest weight is generated as the third best AG. One important aspect of this process is to ensure that duplicate AGs are not generated from different search spaces, meaning that the search spaces must be disjoint. The first challenge is how to find the best AG in each sub-space. We address this in Sections 4.1 and 4.2. The second challenge is how to divide the graph into disjoint sub-spaces. We address this challenge in this section.

We first present an example and then provide the algorithm to find AGs with polynomial delay. After receiving the required number of nodes and a set of query keywords, we first use Algorithm 1 (or its variation from Section 4.2) to find the best approximate AG in the entire graph. Let the best approximate AG in the entire graph be composed of three content nodes, $\{n_1, n_2, n_3\}$. The remaining AGs are divided into the following three disjoint subsets (i.e., sub-spaces):

1. AGs that contain n_1 and n_2 but not n_3
2. AGs that contain n_1 but not n_2
3. AGs that do not contain n_1

The three subsets are disjoint. Furthermore, the union of AGs that are produced from these three subsets along with the best AG $\{n_1, n_2, n_3\}$ comprise the set of all

Algorithm 2 Generating top-k Attributed Groups (AGs) with Polynomial Delay

Input: graph G; $size_{min}$; $size_{max}$; query keywords $T = \{t_1, t_2, \ldots, t_p\}$; value of k
Output: top-k best approximate AGs

1: $Queue \leftarrow$ initialize an empty priority queue
2: $D \leftarrow$ **FindGroupConstraint**(G, $size_{min}$, $size_{max}$, T, $\emptyset$, $\emptyset$)
3: **if** $D \neq \emptyset$ **then**
4: insert $\langle D, \emptyset, \emptyset \rangle$ into $Queue$
5: **while** $Queue \neq \emptyset$ **do**
6: $\langle D, Inc, Exc \rangle \leftarrow$ remove and return top AG and its respected constraints from $Queue$
7: **output**(D)
8: $k \leftarrow k - 1$
9: **if** $k = 0$ **then**
10: **return**
11: $\{n_1, n_2, \ldots, n_q\} \leftarrow$ content nodes of AG D
12: **for** $i \leftarrow 1$ **to** q **do**
13: $Inc_i \leftarrow Inc \cup \{n_1, \ldots, n_{q-i}\}$
14: $Exc_i \leftarrow Exc \cup \{n_{q-i+1}\}$
15: **if** $Inc_i \cap Exc_i = \emptyset$ **then**
16: $D_i \leftarrow$ **FindGroupConstraint**(G, $size_{min}$, $size_{max}$, T, Inc_i, Exc_i)
17: **if** $D_i \neq \emptyset$ **then**
18: insert $\langle D_i, Inc_i, Exc_i \rangle$ into the $Queue$ according to D_i's score

AGs in the input graph. Each subset is defined by some **constraints**. These constraints are enforced using an **inclusion set** and an **exclusion set**. The inclusion set is comprised of nodes that must be included in any AG that is generated from that subset (i.e., node n_1 must be part of any AG produced from the first subset). On the other hand, the nodes in the exclusion set must not be part of any AG produced from that subset (e.g., node n_1 is not part of any AG produced from the third subset). Table 2 represents the constraints (i.e., the inclusion sets and exclusion sets) of these subsets.

After dividing the entire graph into three sub-spaces using the best AG, the next step is to find the best approximate AG in each sub-space while satisfying the constraints of the respective sub-space. This can be done by using a variation of Algorithm 1 that is presented in Section 5.1 to find the best approximate answer under the inclusion and exclusion constraints. Since we have three sub-spaces, three new AGs are generated from each subspace[5]. We initialize a priority queue and insert

[5] Note that in practice, there might be some sub-spaces that do not contain any answer

these AGs into the queue. Note that the AG with the best score is always at the top of the priority queue. In this work, the score is either the proximity weight or the combined score.

The next best approximate AG is now at the top of the priority queue. Assume this AG is generated from the second sub-space. We first return this AG to the user and then divide its sub-space (i.e., the second sub-space) into sub-sub-spaces. Again, in each sub-sub-space of the second sub-space, the best approximate AG is found and is inserted into the priority queue. Then, we return the best AG from the top of the priority queue, and divide its search space into sub-spaces. The best approximate AGs in the new sub-spaces are produced and inserted into the queue. This process is repeated until the top-k AGs are generated or the queue becomes empty.

Enumerating the top-k AGs with polynomial delay is done by Algorithm 2. An empty priority queue is initialized in line 1. In line 2, the procedure for finding the best approximate AG (i.e., FindGroupConstraint) considers the entire search space. This procedure is Algorithm 1 or its modified version presented in Section 5.1. The last two parameters of this procedure determine the inclusion and exclusion sets. When finding the first best AG in the entire graph, these sets are empty. If the best AG exists (i.e., $D \neq \emptyset$), D and the inclusion and exclusion sets are inserted into the variable $Queue$ in line 4. Note that $Queue$ is a priority queue such that the AG with the minimum score is always at the top. The while loop in line 5 is executed until the $Queue$ becomes empty. However, as soon as the top-k AGs are generated, even if $Queue$ is not empty, this procedure is terminated in line 10. In line 6, the AG at the top of $Queue$ along with its constraints (i.e., inclusion and exclusion sets) are removed. The best AG (D) is returned to the user in line 7. As mentioned before, the procedure terminates in line 10 if the (top) k answers have been generated. If the top-k answers have not been generated yet, the content nodes in D are assigned to $n_1, n_2, \ldots, n_q$. Note that $size_{min} \leq q \leq size_{max}$. In lines 12-18, q new constraints (as inclusion and exclusion sets) are created based on these q content nodes. As we previously discussed, each set of constraints is associated with a new sub-space. The next step is to find the best AG in each new search space. For each new search space, if the inclusion and exclusion sets have no content nodes in common (line

Algorithm 3 Finding the Best Attributed Group (AG) under Constraints

Input: graph G; $size_{min}$; $size_{max}$; query keywords $T = \{t_1, t_2, \ldots, t_p\}$; inclusion set Inc; exclusion set Exc

Output: best approximate AG

$B \leftarrow$ the set of nodes that contain at least one query keyword (from an inverted index)
$B' \leftarrow \{B - Inc - Exc\}$; $leastWeight \leftarrow +\infty$; $bestGroup \leftarrow \emptyset$
for $i \leftarrow 1$ **to** $|B'|$ **do**
  $totalWeight \leftarrow 0$; $group \leftarrow \emptyset$;
  $cn_1 \leftarrow B'$.get(i); $group$.add(cn_1)
  for $k \leftarrow 1$ **to** $|Inc|$ **do**
    $inc_i \leftarrow Inc$.get(i);
    $group$.add(inc_i);
    $totalWeight \leftarrow totalWeight + dist(cn_1, inc_i)$
  for $j \leftarrow 2$ **to** $(size_{max} - |Inc|)$ **do**
    $minPathValue_j \leftarrow ew_{close}(cn_1, B', j-1)$;
    $cn_j \leftarrow no_{close}(cn_1, B', j-1)$
    if $cn_j \neq \emptyset$ **then**
      $totalWeight \leftarrow totalWeight + minPathValue_j$;
      $group$.add(cn_j)
      if $j \geq (size_{min} - |Inc|)$ **then**
        $weight \leftarrow \frac{totalWeight}{j+|Inc|}$
        if $weight < leastWeight$ **then**
          $leastWeight \leftarrow weight$;
          $bestGroup \leftarrow group$
for $i \leftarrow 1$ **to** $|Inc|$ **do**
  $totalWeight \leftarrow 0$; $group \leftarrow \emptyset$;
  $cn_1 \leftarrow Inc$.get(i); $group$.add(cn_1)
  for $k \leftarrow 1$ **to** $|Inc|$ **and** $i \neq k$ **do**
    $inc_k \leftarrow Inc$.get(k);
    $group$.add(inc_k)
    $minPathValue \leftarrow ew(cn_1, inc_k)$;
    $totalWeight \leftarrow totalWeight + minPathValue$
  for $j \leftarrow 1$ **to** $(size_{max} - |Inc|)$ **do**
    $minPathValue_j \leftarrow ew_{close}(cn_1, B', j-1)$;
    $cn_j \leftarrow no_{close}(cn_1, B', j-1)$
    if $cn_j \neq \emptyset$ **then**
      $totalWeight \leftarrow totalWeight + minPathValue_j$;
      $group$.add(cn_j)
      if $j \geq (size_{min} - |Inc|)$ **then**
        $weight \leftarrow \frac{totalWeight}{j+|Inc|}$
        if $weight < leastWeight$ **then**
          $leastWeight \leftarrow weight$; $bestGroup \leftarrow group$
return $bestGroup$

15), the best AG is found under the associated constraints in line 16. If the best AG is not empty, it is inserted into the $Queue$. Since the procedure FindGroupConstraint runs in polynomial time, Algorithm 2 produces ranked AGs with polynomial delay.

For each best approximate AG D, the union of the sub-spaces produced based on D and the set of content nodes in D are equivalent to the original search space (i.e., all possible AGs) used to generate D. Therefore, no AG is omitted. Hence, Algorithm 2 produces the top-k AGs (or it produces all the AGs that exist in the input graph if the total number of AGs is less than k). Furthermore, the sub-spaces that are created based on the D are disjoint so that none of them can possibly generate D itself. Thus, they do not generate the same AG. Therefore, the set of AGs is duplicate-free. In other words, no AG has the same set of content nodes as any other AG.

5.1. Finding the Best AG Under Constraints

When finding the top-k AGs, each AG (except the first one) satisfies some inclusion and exclusion constraints. Here, we present a modified version of Algorithm 1 that receives two extra parameters as input: the inclusion set (Inc), and the exclusion set (Exc). The returned AG is guaranteed to contain all the content nodes in Inc and to not contain any content nodes in Exc. The details are presented in Algorithm 3. First, we define B' as the set of content nodes that contain at least one query keyword. Note that none of the content nodes in Inc and Exc are in set B'. The idea is to construct answers around two sets of content nodes: the first set is B' (lines 3-20) and the second set is Inc (lines 32-38). The AGs that are constructed from these two sets compete to form the best approximate AG satisfying the constraints. Since $|B'| = O(|B|)$ and $|Inc| << |B|$, the complexity of Algorithm 3 is the same as that of Algorithm 1, which is $O(size_{max}.|B|^2)$, where $|B|$ is the number of content nodes.

5.2. Discussion of the Polynomial Delay Procedure

Our method generates **approximate** top-k AGs. The first AG is produced from the entire graph (with an approximation ratio of two). The next best AG is produced from disjoint (i.e., non-overlapping) sub-spaces, taking their inclusion and exclusion constraints into account. Therefore, the second AG is also an approximation

of the second best AG in the entire graph. By continuing this process, we generate approximate top-k AGs.

Note that different orders of content nodes of the first answer generate different sub-spaces in the next round of the process. In other words, the initialization of the second round (and all other subsequent rounds) varies depending on the order of the content nodes in the current best approximate answer. However, the order of content nodes does not affect the ranking of the answers. This is because the sub-spaces are disjoint (i.e., the same answer cannot be produced from two different subspaces) and the union of the subspaces form the entire search space. Thus, regardless of how we order the content nodes, the second best answer is the same (and the same logic applies to the third best answer and so on). Note that the number of subspaces is polynomial, unlike the number of answers, which can be exponential.

6. Experiments

In this section, we test the performance of our algorithms. The algorithm that solves Problem 1 and optimizes the proximity weight is labelled PW. The algorithm that solves the bi-objective Problem 3 and optimizes the combined objective of proximity weight and keyword score is labelled CS. Note that CS also receives the trade-off parameter λ. We include the value of λ when we refer to the results of CS (e.g., CS-0.25 refers to the results of CS when $\lambda = 0.25$). Unless otherwise stated, we return top-5 answers and report the average results. Since our specific problem has not been studied before, we do not have a direct competitor. To the best of our knowledge, the algorithm from Fang et al. [4] to find attributed communities (communities structurally cohesive with keywords *in common* with the query keyword), abbreviated AC, is the closest to ours. We implemented two versions of this algorithm and refer to them as AC-Inc-All and AC-Inc-Last. In the next section, we describe how we modified the algorithm in [4] to return AGs. Unless stated otherwise, for each experiment, we use 100 randomly generated queries and report the average results. All algorithms are implemented in Java 8 and executed on an Intel Xeon 2.40GHz with 64 GB of RAM. We use several criteria to evaluate the quality of answers. We

first describe the algorithms we compare against and the datasets we used in our experiments. We then present our results.

6.1. Extending Prior Work to Discover AGs

In this section, we describe how we modified the algorithm from Fang et al. [4], denoted AC, to return a compact group. The input to AC is a query node nq, a set of query keywords T, and the minimum degree of each node k' (corresponding to the k' in k'-core). The output is a set of k'-cores in which each node is a content node. A k'-core is a subgraph in which the degree of each node is at least k'. Furthermore, each k'-core is built around nq. The size of an AC (i.e., the number of content nodes) is not explicitly controlled. Since we have already used the variable k to refer to the top-k answers, we use k' to represent the minimum degree in a k'-core.

Fang et al. propose two strategies for producing ACs: an incremental bottom-up algorithm and a decremental top-down algorithm. In our experiments, the decremental algorithm was slow, so we only present the results of the incremental algorithm. In each iteration of the incremental algorithm, the size of the subgraph becomes smaller. However, the AC of the last iteration might be too small to satisfy our size requirements. Therefore, we evaluate two versions of the algorithm. The first one evaluates the subgraphs of all iterations (i.e., AC-Inc-All) and the second one only evaluates the subgraph of the last iteration (i.e., AC-Inc-Last).

We modify the AC algorithm to produce attributed groups (AGs) as follows. Recall that we receive a set of query keywords T and the range of $size_{min}$ and $size_{max}$ as the desired size of the group. For each content node cn, we run the algorithm and pass cn as the query node (so each cn becomes a query node nq). The subgraphs built around each content node cn are filtered so that their size falls within the required range. The AC algorithm also receives the minimum degree (i.e., k') to form k'-cores. The authors state that $k' = 6$ produces the best AC in terms of keyword relevance. However, the value of $k' = 6$ produces ACs with few content nodes. Therefore, we run the AC algorithm for the following values of k': $k' = \{1, 2, 4, 6\}$. All ACs with different minimum degrees that satisfy our size constraints are valid AGs. These ACs are then sorted based on keyword relevance to form the top AGs. If the

algorithm does not terminate in one hour, we stop it, and the ACs that are produced within the one hour are used to form top AGs.

6.2. Datasets and Settings

We create attributed graphs from the DBLP[6] and GitHub[7] datasets. DBLP is a publication dataset and we use it to build a network of experts. Each expert (i.e., author) is a node in the graph, and if two authors published a paper together, there is an edge between them. This produces a DBLP graph with 1M nodes and 3.3M edges. The expertise (i.e., attributes) of authors are extracted from the titles of their publications. GitHub is a web-based hosting service for IT projects that also provides version control. We use the GitHub API to download information about projects in the network. Nodes of the GitHub graph are experts (i.e., developers or IT consultants) that participate in different projects. If two experts participated in the same project, their respective nodes are connected. The expertise (i.e., keywords) of an expert is extracted from the titles of the projects they participated in and the programming languages they listed in their profiles. This produces a graph with 0.5M nodes and 8M edges.

For both DBLP and GitHub, we assume that two nodes are connected if there is either a direct edge between them, or if there is only one other node on the shortest path between them. The former models the friendship relation in social networks and the latter models the friends of friends relation. We observed that DBLP and Github differ in the distribution of keywords across their nodes. Keywords tend to be rarer in Github than in DBLP, i.e., for each keyword in the graph, there are fewer nodes with the same keyword. Thus, groups with more than 20 content nodes are rare in Github. When experimenting over Github, we set the maximum group size to 20.

[6] *http://dblp.uni-trier.de/xml/*
[7] *https://github.com/*

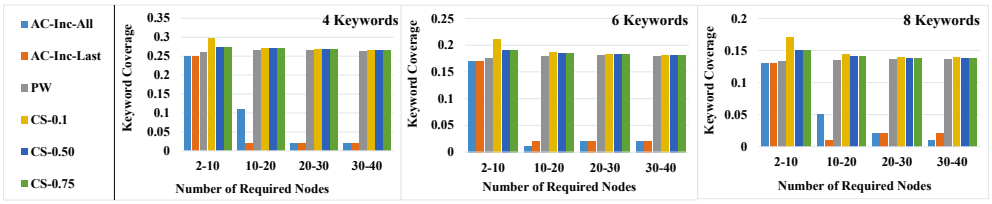

Figure 5: Keyword coverage over DBLP.

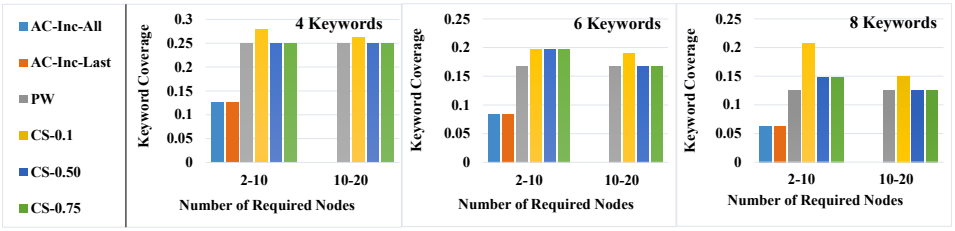

Figure 6: Keyword coverage over GitHub. AC does not produce any answer for groups of size 10-20.

6.3. *Keyword Coverage*

For a given AG R with q content nodes and p query keywords, $T = \{t_1, t_2, \ldots, t_p\}$, the keyword coverage of R is defined as follows:

$$KC(R) = \frac{\sum_{i=1}^{q} |K(n_i) \cap T|}{p \times q}$$

where $K(n_i)$ determines the keywords that appear in node n_i. Clearly, the higher the keyword coverage, the better (i.e., more keywords are covered by the nodes in the AG). Keyword coverage varies between 0 and 1.

Figures 5 and 6 show the keyword coverage over the DBLP and GitHub datasets for different numbers of keywords when the number of required nodes (range of nodes) changes. As expected, (1) CS outperforms PW since PW does not optimize the keyword score. (2) Decreasing the value of λ increases keyword coverage, since smaller values of λ emphasize keyword relevance more than proximity. When the required size of the answer is small (i.e., 2 to 10 nodes), the keyword coverage of

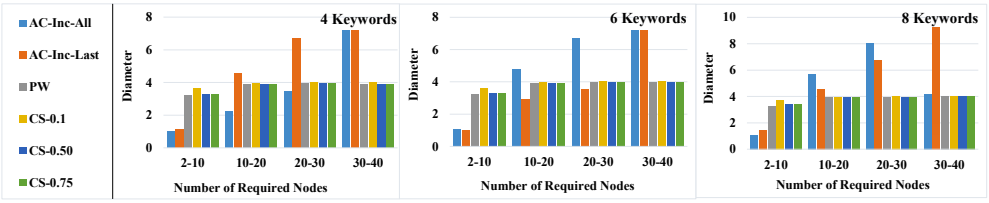

Figure 7: Diameter over DBLP.

CS-0.1 is significantly higher than other methods. However, this difference is less noticeable when the required size increases (i.e., 20 to 10 nodes). This is because when we require fewer nodes, there is a higher chance of finding a small group of nodes that are close to each other and also have high keyword coverage. Note that in most of our motivating applications (e.g., expert hiring or targeted marketing), we do not need many nodes in the answer. We observed that changing λ by less than 0.1 does not significantly affect keyword coverage.

The advantage of CS over PW is more noticeable in small AGs (e.g., group sizes 2-10). For example, in the the DBLP dataset, the *KC* of CS is 4% higher than the *KC* of PW. In Github, the *KC* of CS is 8% higher than that of PW. AC-Inc-All and AC-Inc-Last have the lowest keyword coverage. The reason is that every node in an AC subgraph must contain at least one keyword. AC removes any subgraph with high coverage even if few middle nodes exist. Note that by definition, AC does not allow the existence of middle nodes (the notion of friends of friends in this work) in the answer. This is because AC uses the notion of k-cores for forming answers. In a k-core, all nodes must contain at least one input keyword. However, since we explicitly optimize proximity and allow middle nodes in the answer (through the notion of friends of friends), the answers produced by our algorithms cover more keywords. This is one of the advantages of our model versus community search models.

6.4. Diameter (Compactness)

We now evaluate the quality of the answers in terms of their compactness. A well known measure for evaluating the proximity of a subgraph is its diameter, which is the largest shortest distance between any two nodes in the subgraph. The smaller

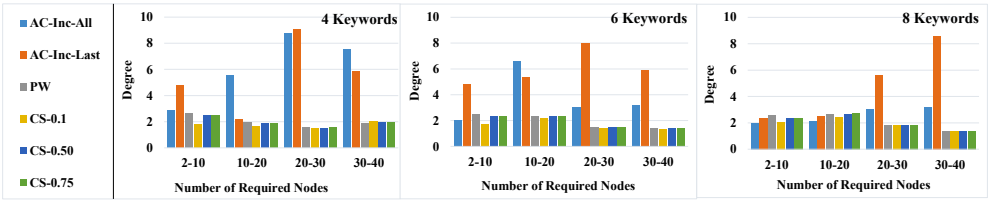

Figure 8: Degree of subgraphs over DBLP.

the diameter, the closer the nodes are to each other. We use all the nodes in the subgraph (including nodes that are not content nodes but connect content nodes to each other) to calculate the diameter. Figure 7 shows the diameter of answers over DBLP (GitHub shows a similar trend and its results are omitted for brevity). Our algorithm consistently produces compact AGs for different numbers of required nodes. On the other hand, AC produces compact groups when the number of required nodes is small, but for larger sizes (20-30 and 30-40), it does not perform well. This is expected as we explicitly optimize proximity. Also note that keyword coverage of groups with a small number of nodes is low (see experiments on keyword coverage). Therefore, AC does not produce high quality answers.

6.5. Degree of Nodes (Density)

The degree of the answer subgraphs for the DBLP dataset is presented in Figure 8. As expected, the degree of subgraphs produced by AC is higher than those produced by PW and CS. The reason is that AC explicitly optimizes the degree. However, and as we discussed before, the proximity (i.e., compactness) and keyword coverage of AC is not guaranteed.

6.6. Group Size Requirement

Figure 9 shows the percentage of answers produced by AC that satisfy various desired size requirements. For example, if the required size is 2 to 10 content nodes, we measure the percentage of communities produced by AC that have 2 to 10 nodes. This percentage is usually around 2%. The other 98% of the communities do not

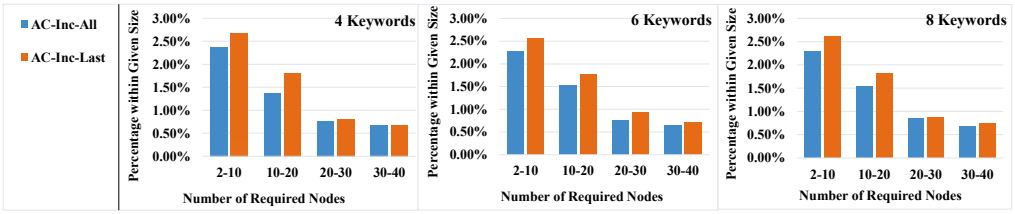

Figure 9: Percentage of groups returned by AC that satisfy the given size requirement over DBLP.

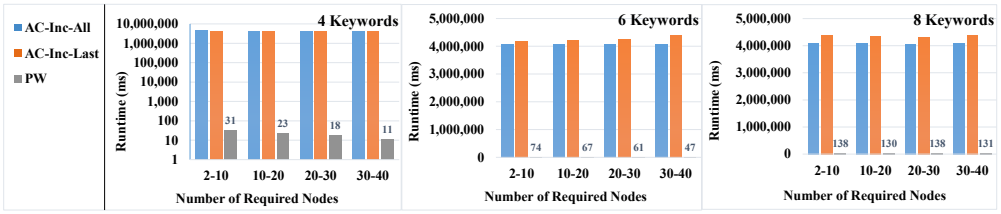

Figure 10: Runtime over DBLP.

qualify as valid groups. This result is not surprising as AC does not optimize for size. On the other hand, all AGs satisfy the size requirements.

6.7. Runtime

The runtime of different algorithms over the DBLP dataset is shown in Figure 10 (GitHub has similar runtime). CS has the same runtime as PW since they use the same base algorithm and only differ in the distance function they optimize, which does not affect the runtime. AC is significantly slower than our algorithm. The reason is that AC was not optimized for attributed graphs with nodes that are associated with many keywords (this is also noted in [4] in the scalability experiments of ACs w.r.t. keywords). The other reason is that AC was optimized to construct communities around a single query node. However, to find the best AG over the entire graph, we need to run AC over every content node. While we run AC over with four different values of k', we note that even if we run AC for a single value of k', it is still significantly slower than PW (and CS). Notably, PW (and CS) scale well with the number of required nodes and query keywords. We return answers in milliseconds, which

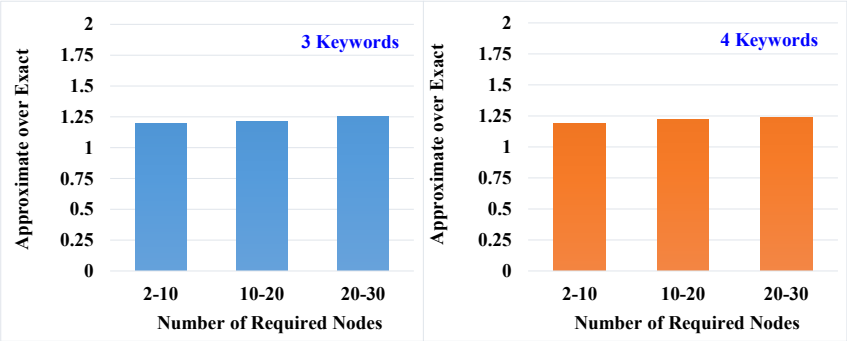

Figure 11: Comparison with exhaustive search.

makes our solution practical for interactive applications.

6.8. Comparison with Exhaustive Search

In this experiment, we compare the results of our approximation algorithm with optimal solutions obtained by exhaustive search (in practice, exhaustive search is not feasible for large problem instances). For each number of keywords and required size range, we randomly generate 100 queries. Exhaustive search did not terminate after one hour for most queries with over four keywords. It also did not terminate after one hour for most queries with four keywords and the required size of 30-40 nodes. Therefore, we only report results for queries with three and four keywords and maximum required nodes under 30. We report the proximity weights of the top answer generated by our approximation algorithm divided by the respective values of the exhaustive algorithm. Results for DBLP are shown in Figure 11; results for GitHub show similar trends and are omitted for brevity. Our approximate results are only about 25% worse than optimal, which outperforms the worst case theoretical approximation ratio of two. We also compare the top-5 answers that are produced by our procedure that generates approximate top-k answers in polynomial delay with the top-5 answers of the exact algorithm. The results show that on average, top-5 approximate answers are also only around 25% worse than optimal answers.

7. Conclusions

We formalized the problem of finding compact attributed groups (AGs) in attributed graphs and social networks. The input is a set of query keywords and a required range for the size of the AG. The output is a subgraph with the required number of content nodes that cover the query keywords. We proposed new ranking objectives based on the proximity of the answer subgraph and keyword coverage. We proved that optimizing these new objectives is NP-hard and proposed approximation algorithms with a provable approximation ratio of two. Since the total number of answers is exponential in the number of query keywords, we proposed a procedure that produces approximate top-k answers with polynomial delay. Experiments on real datasets showed that the answers produced by our algorithms are compact and have high keyword coverage, especially when the required number of nodes is small (around ten).

Another way to optimize both proximity and keyword coverage is to find a set of Pareto-optimal answers. In future work, we plan to develop algorithms to find Pareto answers, rank them based on relevance, and compare their results with this work.

8. Acknowledgement

We would like to thank Yixiang Fang for providing the source code of the algorithms in [4] that find attributed communities.

References

[1] N. Jayaram, A. Khan, C. Li, Querying Knowledge Graphs by Example Entity Tuples, IEEE Transactions on Knowledge and Data Engineering 27 (10) (2015) 2797–2811.

[2] L. Chunlin, B. Jingpan, W. Zhao, X. Yang, Community Detection Using Hierarchical Clustering based on Edge Weighted Similarity in Cloud Environment, Information Processing & Management 55 (2019) 91–109.

[3] Z. Meng, H. Shen, Dissimilarity-constrained Node Attribute Coverage Diversification for Novelty-enhanced Top-k Search in Large Attributed Networks, Knowl.-Based Syst. 150 (2018) 85–94.

[4] Y. Fang, R. Cheng, S. Luo, J. Hu, Effective Community Search for Large Attributed Graphs, PVLDB 9 (12) (2016) 1233–1244.

[5] M. Neshati, Z. Fallahnejad, H. Beigy, On Dynamicity of Expert Finding in Community Question Answering, Information Processing & Management 53 (2017) 1026–1042.

[6] M. Kargar, A. An, N. Cercone, P. Godfrey, J. Szlichta, X. Yu, Meaningful Keyword Search in Relational Databases with Large and Complex Schema, in: ICDE, 2015, pp. 411–422.

[7] M. Kargar, L. Golab, J. Szlichta, eGraphSearch: Effective Keyword Search in Graphs, in: CIKM, 2016, pp. 2461–2464.

[8] M. Zihayat, A. An, L. Golab, M. Kargar, J. Szlichta, Authority-based Team Discovery in Social Networks, in: EDBT, 2017, pp. 498–501.

[9] S. Papadopoulos, C. Zigkolis, G. Tolias, Y. Kalantidis, P. Mylonas, Y. Kompatsiaris, A. Vakali, Image clustering through community detection on hybrid image similarity graphs, in: ICIP, 2010, pp. 2353–2356.

[10] M. Kargar, A. An, Keyword Search in Graphs: Finding r-cliques, PVLDB 4 (10) (2011) 681–692.

[11] L. Bai, J. Liang, H. Du, Y. Guo, A Novel Community Detection Algorithm based on Simplification of Complex Networks, Knowl.-Based Syst. 143 (2018) 58–64.

[12] S. Fortunato, Community Detection in Graphs, Physics Reports 486 (3) (2010) 75–174.

[13] M. Newman, M. Girvan, Finding and Evaluating Community Structure in Networks, Physical Review E 69.

[14] Y. Zhou, H. Cheng, J. X. Yu, Graph Clustering Based on Structural/Attribute Similarities, PVLDB 2 (1) (2009) 718–729.

[15] Y. Ruan, D. Fuhry, S. Parthasarathy, Efficient Community Detection in Large Networks using Content and Links, in: WWW, 2013, pp. 1089–1098.

[16] Z. Xu, Y. Ke, Y. Wang, H. Cheng, J. Cheng, A Model-based Approach to Attributed Graph Clustering, in: SIGMOD, 2012, pp. 505–516.

[17] M. Hajiabadi, H. Zare, H. Bobarshad, IEDC: An Integrated Approach for Overlapping and Non-overlapping Community Detection, Knowl.-Based Syst. 123 (2017) 188–199.

[18] X. Huang, L. V. Lakshmanan, Attribute-Driven Community Search, PVLDB 10 (9) (2017) 949–960.

[19] G. Li, B. Ooi, J. Feng, J. Wang, L. Zhou, EASE: Efficient and Adaptive Keyword Search on Unstructured, Semi-Structured and Structured Data, in: SIGMOD, 2008, pp. 903–904.

[20] L. Qin, J. Yu, L. Chang, Y. Tao, Querying Communities in Relational Databases, in: ICDE, 2009, pp. 724–735.

[21] M. Kargar, A. An, X. Yu, Efficient Duplication Free and Minimal Keyword Search in Graphs, IEEE Transactions on Knowledge and Data Engineering 26 (7) (2014) 1657–1669.

[22] R. H. Li, L. Qin, J. X. Yu, R. Mao, Influential Community Search in Large Networks, PVLDB 8 (5) (2015) 509–520.

[23] M. Sozio, A. Gionis, The Community-search Problem and How to Plan a Successful Cocktail Party, in: KDD, 2010, pp. 939–948.

[24] W. Cui, Y. Xiao, H. Wang, W. Wang, Local Search of Communities in Large Graphs, in: SIGMOD, 2014, pp. 991–1002.

[25] K. Wang, X. Cao, X. Lin, W. Zhang, L. Qin, Efficient Computing of Radius-bounded k-cores, in: ICDE, 2018, pp. 233–244.

[26] Y. Fang, R. Cheng, X. Li, S. Luo, J. Hu, Effective Community Search over Large Spatial Graphs, PVLDB 10 (6) (2017) 709–720.

[27] R.-H. Li, L. Qin, F. Ye, J. X. Yu, X. Xiao, N. Xiao, Z. Zheng, Skyline Community Search in Multi-valued Networks, in: SIGMOD, 2018, pp. 457–472.

[28] G. Bhalotia, C. Nakhe, A. Hulgeri, S. Chakrabarti, S. Sudarshan, Keyword Searching and Browsing in Databases using BANKS, in: ICDE, 2002, pp. 431–440.

[29] B. Ding, J. Yu, S. Wang, L. Qin, X. Zhang, X. Lin, Finding top-k Min-Cost Connected Trees in Databases, in: ICDE, 2007, pp. 836–845.

[30] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, H. Karambelkar, Bidirectional Expansion for Keyword Search on Graph Databases, in: VLDB, 2005, pp. 505–516.

[31] H. He, H. Wang, J. Yang, P. Yu, BLINKS: Ranked Keyword Searches on Graphs, in: SIGMOD, 2007, pp. 305–316.

[32] B. B. Dalvi, M. Kshirsagar, S. Sudarshan, Keyword Search on External Memory Data Graphs, PVLDB 1 (1) (2008) 1189–1204.

[33] Y. Wu, S. Yang, M. Srivasta, A. Iyengar, X. Yan, Summarizing Answer Graphs Induced by Keyword Queries, PVLDB 6 (14) (2013) 1774–1785.

[34] R. T. Marler, J. S. Arora, Survey of Multi-objective Optimization Methods for Engineering, Structural and Multidisciplinary Optimization 26 (2004) 369–395.

[35] R. T. Marler, J. S. Arora, The Weighted Sum Method for Multi-objective Optimization: New Insights, Structural and Multidisciplinary Optimization 41 (2010) 853–862.

[36] T. Akiba, Y. Iwata, Y. Yoshida, Fast Exact Shortest-Path Distance Queries on Large Networks by Pruned Landmark Labeling, in: SIGMOD, 2013, pp. 349–360.

[37] D. Johnson, M. Yannakakis, C. Papadimitriou, On Generating All Maximal Independent Sets, Info. Proc. Lett. 27 (3) (1998) 119–123.

[38] K. Golenberg, B. Kimelfeld, Y. Sagiv, Keyword Proximity Search in Complex Data Graphs, in: SIGMOD, 2008, pp. 927–940.

[39] 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) (1972) 401–405.