Interium report for C-Explorer

Student: Kang Yunfan

UID: 3035140603

Date of submission: 2017/11/30

Abstract

Community search algorithms have been one of the hottest research topics in graph mining. To facilitate community search algorithms, we proposed C-Explorer. C-Explorer enables users to formulate community search queries to retrieve and view communities that they are interested in. The module for comparing the efficiency of different community search algorithms is also provided. Besides, C-Explorer supports attributed graphs. Each vertex in attributed graphs is associated with a set of attributes. C-Explorer can help to look for attributed communities, in which vertices are cohesive both structurally and semantically. Interfaces are provided for researchers to plug in different algorithms for testing or visualization. After building C-Explorer, we on the extended research problem of edge-attributed community search. The inspiration of the research topic is gained from the feature implemented for C-Explorer. We implemented the EACQ algorithm based on ACQ algorithm and carried out experiments to test its performance.

Acknowledgement

I would like to extend my sincere thanks to those who help me to make the project possible and better. First, I hope to express my gratitude to my supervisor Dr. Reynold C.K Cheng. Dr. Cheng and I discussed a lot about the expect features of the project and helps me to develop a clearer direction. I also wish to say thank you to Dr. Yixiang Fang, who provided me with the original idea of C-Explorer and the basic knowledges of community search algorithms. Both of them help me a lot in testing the program.

- **Table of Contents**

- **List of Figures**

- **Abbreviations**

ACQ attributed community query
AC attributed community
CMF Community member frequency
CPJ Community pair-wise Jaccard
EACQ Edge-attributed community query

1. Introduction

Social networks are becoming increasingly popular and important in our everyday life and they generate massive valuable data that draws the attention of researchers. Social networks can be treated as attributed graphs. Users are modeled as vertices and the relationship between users is represented by edges. Each vertex has a set of attributes that are linked with certain properties [1]. Subgraphs whose vertices are cohesively connected are defined to be communities. Users in a community are usually related to several other users and can share common attributes or geographically close to each other. Hence communities are valuable in numerous aspects such as commercial promotion and social science study. Due to the importance of communities, discovering communities has attracted much attention and becomes one of the well-studied graph-mining problems. In the last few years, a query dependent variant of community detection problem has been raised and it is called attributed community search problem. It takes a set of query vertices and attributes as input and retrieves communities that consist of the query vertices and each vertex in the community has the attributes specified.

Sozio et al. suggest that attributed community search is useful in solving many real-life problems and three examples that community search can do in different areas are given in [2]. We hypothesize that having a platform that facilitates formulating community search query and provides interfaces for community visualization and change of dataset might help researchers in testing and demonstrating their algorithms and it can also be easily extended to real applications. As is pointed out in [3], there are systems that provide user interfaces for users to compose graph queries and run algorithms on different datasets, such as AutoG[4] and VIIQ[5]. However, these platforms are not customized for community search algorithms and using general graph queries to implement community search algorithms is considered unstraightforward in [3]. Therefore, we introduce C-Explorer, a web-based platform that facilitates community search graph query, result visualization and efficiency comparison between community retrieval (CR) algorithms. Interfaces for plugging in different algorithms and datasets are also provided.

In addition to the program, the research problem of edge-attributed community search will also be explored. In the current attributed community search algorithm, the attribute sets of vertices are considered when measuring the keyword cohesiveness of the community. However, Guo et al. pointed out in [7] that edge attributes may contain more information than vertex attributes.

Hence, we will explore the edge-attributed community search problem in the next step.

The remainder of the report proceeds as follows. First, we offer description of the structure of the C-Explorer and the UI design. The components in UI design are discussed in detail and justifications on how they serve the purpose of the project are given. Then the current progress is presented. The tests we performed on the program are given with the results commented. We then discuss about the motivation of doing edge-attributed community search problem and the formal problem definition. Detailed description of algorithm and justification are also proposed. Following that we give the experiments on the EACQ algorithm and analyze the results. Next, the difficulties encountered will be listed and described. We also give the current mitigation strategies or possible solutions for each difficulty. Finally, we close up with a summary and provide inspirations that may lead to future work.

2. Methodology

The C-Explorer is almost finished except for the standard APIs for other researchers to plugging in algorithms and datasets. In this section, the structure of the program and the techniques used will be discussed. Then we discuss the user interface and how it facilitates community search algorithm queries, community browsing and comparison of community search algorithms. Finally, we introduce the dataset assumptions and the data structures of the dataset loaded.

2.1 Structure of C-Explorer

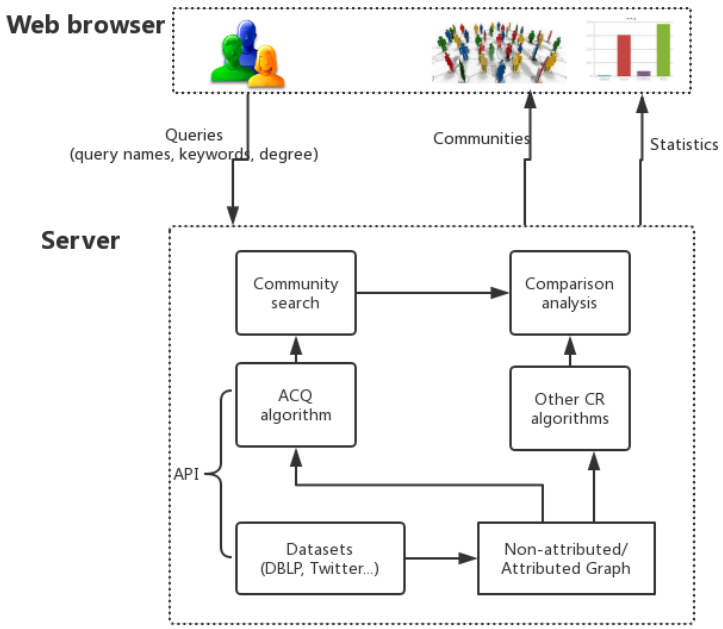


Figure 1 Structure of C-Explorer

C-Explorer adopts a browser-server architecture (see Figure 1.1). The front end is accessible through URL and it provides users with user interfaces to issue queries and view the communities and results of comparison returned. The queries issued by users are sent to the server side. If the query is a community search query, the community search algorithm embedded will be called to retrieve the communities. In our current design, the default community search algorithm embedded is the ACQ algorithm. Then the server sends the communities retrieved to the browser for display. If the query is issued for comparing the efficiency of different algorithms, algorithms embedded will be called with the same input specified in the query and the resulting communities and statistics of each algorithm will be sent to the browser. By default, the algorithms used for comparison is Global, Local, CODICIL, and ACQ. With the browser-server architecture, calculations that require large memory space and CPU powers are kept on the server side. Once the communities are sent to the browser, all other interactions with the communities are then handled by JavaScript. This separation of logic is expected to balance the load and reduce the traffic between the client and server.

JSP (JavaServer Pages) framework is used for implementing the C-Explorer. It is one of the most famous standard technologies for creating dynamically generated web pages. In addition, JSP uses Java programming language. With the interface feature of Java language, replacing

algorithms and datasets can be done easily and should not affect other functions.

2.2 User interfaces

On the browser side, user interfaces for issuing community search queries and comparing different algorithms are provided.

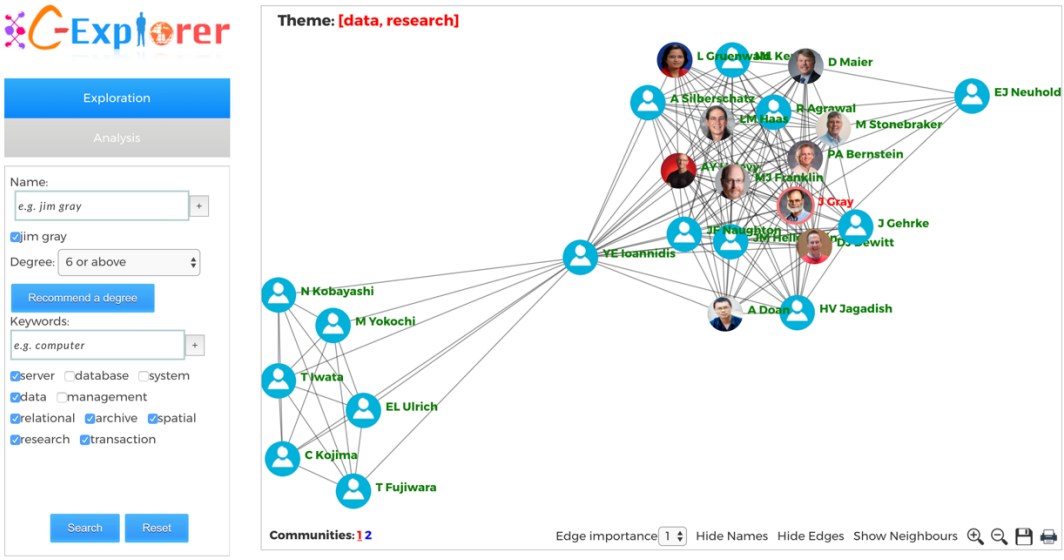


Figure 2.1 Community search user interface

2.2.1 Exploration

The "Exploration" page (see Figure 2.1) is for formulating attributed community search queries and viewing the communities returned. The left division is for query formulation and the user interface is designed specifically for community search queries. Users can type in names in the "Name" text field and click the "+" button on the right to add a query name. Because the number of vertices in a social network can be very large, a list of candidate names will show up as the name candidates according to the input of the text field with the hope that users can find the target name faster. Once a query name is added, attributes associated with that name is retrieved and the union of attributes of each query name will be displayed at the bottom. Users can also type in new keywords and add that to the set of attributes. "Degree" specifies the minimum degree of each vertex in the communities to be retrieved and it is a parameter need for k-core related algorithms. After specifying the query names, degree and the set of attributes, users can click on the "Search" button at the bottom to send the query to the server. If the "Reset" button is clicked, the query names and attributes will be deleted to start a new search.

The resulting communities will be displayed in the right division of the page. The “Theme” section at the top shows the set of attributes that all vertex share and it should be a subset of the attributes specified by the user in the query. If the result contains more than one community, users can choose to view different community by clicking on the community at the bottom. Communities are drawn using Scalable Vector Graphics (SVG). SVG is XML-based and it integrates with DOM. Compared with Canvas, SVG is easier to perform vector operations to change the position of elements or interact with a certain element dynamically. This feature makes the functions for interacting with the community easy to implement.

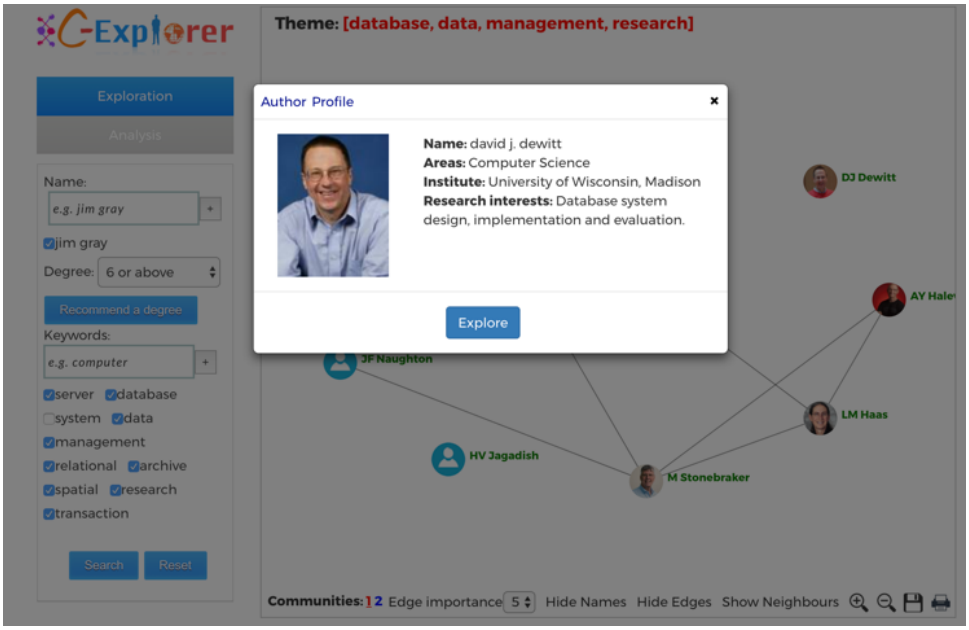
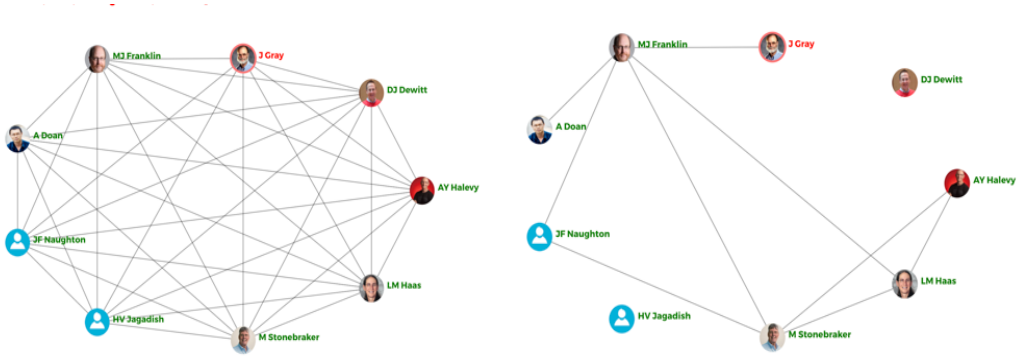


Figure 2.2 Exploring a vertex

To facilitate users to better view and interact with the communities, some functions are provided. When the user clicks on one of the vertices, the profile of that vertex will be shown (Figure 2.2). If the "Explore" button is clicked, the communities that vertex belongs to will be retrieved and displayed. Other functions are placed below the community displayed. Users can perform zoom-in and zoom-out or save or print the community by clicking on the icons at the right bottom corner. In addition, users can use arrow keys on the keyboard to move to the part of the community they want after zoom-in. We hypothesize that “Hide/Show Names” and “Hide/Show edges” are self-explanatory, but the setting of “Edge importance” might be worth a description.



(a) Edge importance = 1

(b) Edge importance = 5

Figure 2.3 After the "Edge importance" is changed from 1 to 5, the edges with importance < 5 is hidden.

We define edge importance as an attribute for edges. It represents the closeness of the two vertices that the edge connects. Its meaning can vary from case to case depending on the choice of the researchers and the type of the graph. For example, in a community of game players, edge importance can be defined to be the time two players have played together. In Figure 2.3, the dataset is retrieved from DBLP and edge importance is defined to be the number of co-authorship between two researchers. By selecting a value of edge importance, all edges with a lower edge importance will be hidden. This feature might help to make the potential relationship in the communities clearer by further reducing the number of the edges that are less important. Meanwhile, this feature is independent of the community search algorithm used. One additional file recording the "Edge importance" between each pair of vertices is needed to make use this feature but the algorithms need not be modified.

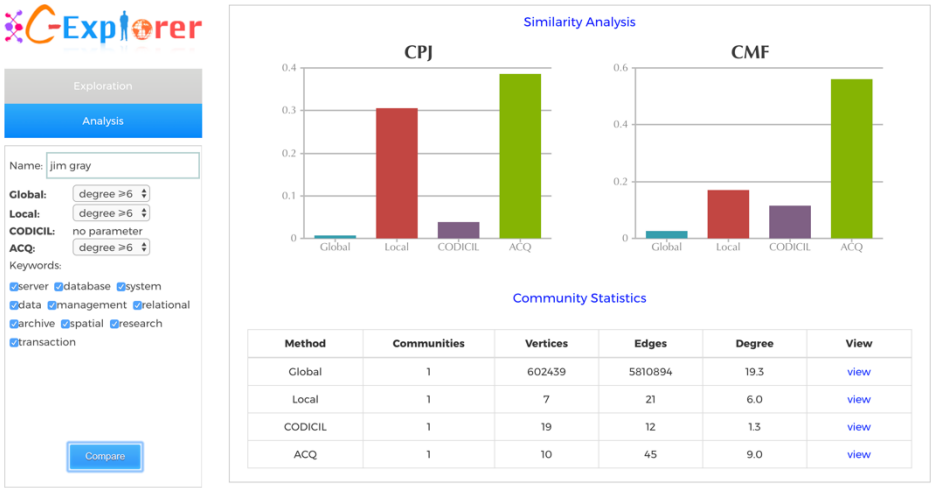


Figure 2.4 User Interface for "Analysis"

2.2.2 Analysis

The layout of "Analysis" (See Figure 2.4) is similar to that of "Exploration". Because some of

the community search algorithms do not support more than one query vertex, only one query name can be inputted. Once the text field loses its focus, the keywords of the name in the text field will be loaded and users can click on the “Compare” button after the selection of keywords. Two metrics (CMF and CPJ) are used to evaluate the overall effectiveness of algorithms. The two metrics are considered effective in measuring the cohesiveness of a community in [6]. Usually, communities with better cohesiveness will score higher in these two metrics. The result is put into two charts and shown in the upper half of the right division. The statistics of each algorithm are shown in the table under “Community Statistics” for users to check and compare.

2.3 Dataset and Data Structure

We assume that the dataset contains at least the graph structure and methods should be provided to load the graph into an adjacency list for further process by the algorithm. The adjacency list is one of the two major ways to represents undirected graphs and it requires smaller memory space. Although adjacency matrix guarantees better performance compared with adjacency list, we tend to choose adjacency list because the graph is of large size and the memory cost for using adjacency matrix is too much.

Attribute sets are required for attributed graphs and we assume that the corresponding data can be loaded into a HashMap. The keys of the HashMap is the ids of the vertices and the values are the list of attribute set related to that vertex. Using HashMap enables us to achieve average O(1) time complexity in retrieving the attribute set for a given vertex. This might be the best performance we can expect.

In addition, to make use of the additional features of edge importance and viewing the profile of a vertex on right click, the corresponding files and method to load them are needed. The data recording edge importance is stored in the format of HashMap<vertexId1, HashMap<vertexId2, edge importance>> so that the time complexity for retrieving the edge importance for a given edge is O(1). Profiles are loaded and stored similarly to attributes. But because profiles are not needed by algorithms and are aimed for providing additional information to users that they might be interested in, the profiles may not cover every vertex in the graph. Default avatar and the name of the vertex will be displayed if the profile of the vertex is missing.

3. Progress and Interim Results

The major features of “Exploration” and “Analysis” have been implemented. The program embedded with ACQ algorithm and DBLP dataset was demoed in VLDB conference by Dr. Reynold C.K. Cheng. The dataset is retrieved from the XML file released by DBLP on June 30th, 2017. The attribute set for each vertex is the set of 10 most frequent keywords associated with each author. Before the demonstration, we prepared the profiles for 220 first authors whose paper is accepted by PVLDB volume 10. We formulate queries for each of these 229 authors with each author being the single query name and the set of it keywords being the set of attributes to test the system. There are 12 authors whose community cannot be found by the ACQ algorithm. For the rest 208 queries, the retrieved communities and the profiles are displayed properly. The respond time for comparing the four default algorithms with the queries varies and it can take up to 30 seconds for the result to show up. Though the performance is unacceptable, it is still within the expectation because running four algorithms and drawing the communities retrieved by each of them can take a long time. C-Explorer was then sent to Dr. Reynold Cheng and Dr. Yixiang Fang to test the other features related to the browsing the communities. Both of them are familiar with the community search algorithms and the feedbacks from them are positive.

During the demo, the audiences at VLDP commented about the performance and response time. According to Dr. Cheng, the demo is in general successful except that retrieving name candidates can take several seconds in some cases. We examined this and figured out it is caused by the poor performance of the method for doing dynamic retrieving name candidates. This problem was not identified before because the computer we used to develop C-Explorer is better than the one used for the demo at the conference and the response time is acceptable. Because we expect C-Explorer to be a tool for researchers, it should be able to run smoothly on different devices. Hence the long response time was considered a serious threat to the user experience. So, we modified the improved the algorithm and tested the new program with random 100 authors in the DBLP dataset. The current response time for retrieving the list of name candidates are less than 100ms when it runs on the FYP virtual machine provided by the CS department. Considering that the performance of the virtual machine is roughly estimated to be poorer than average laptops, we think the current performance should be acceptable.

The research on edge-attributed community search has also been started. We carried out

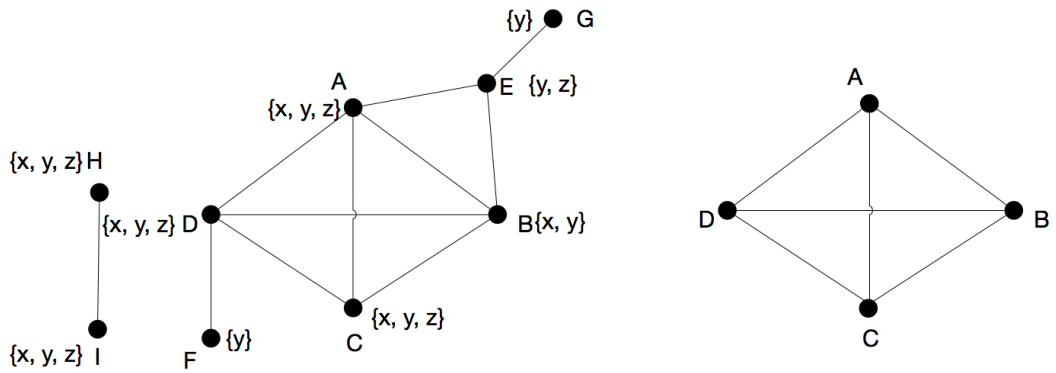
literature review on related topics such as edge-attributed community detection, attributed community search and multi-layer graph cluster algorithms. The motivation and problem definition have been developed and the details will be given in the next section. We will continue to work on the preliminary solutions.

4. Edge-attributed search algorithm

In Section 2.2.1, the feature of Edge importance is presented. Edge importance is designed to represent the closeness of the relationship represented by the edge. However, the relationship may not be related to the theme of the community and it makes less sense to discuss how important the relationship is for the community. In this section, we first present the potential drawback of the current attributed community search algorithm ACQ. Then we propose an alternative way to model the social networks as edge-attributed graphs and give potential solutions on retrieving edge-attributed communities.

4.1 Motivation

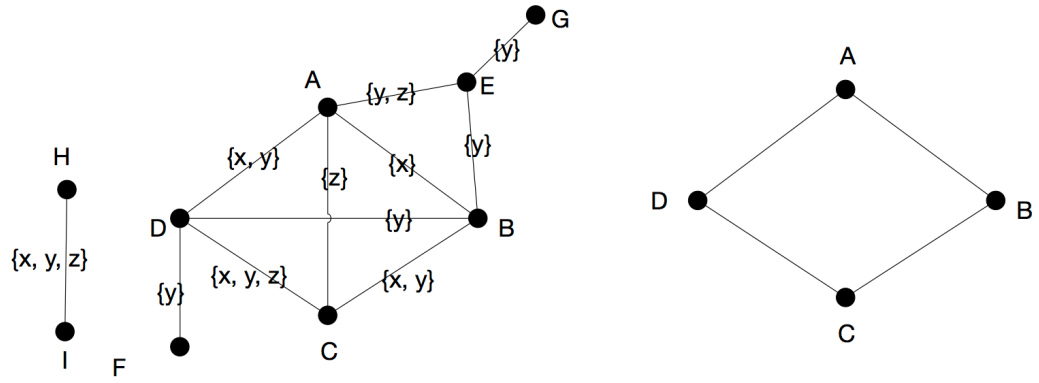
In [1], Fang et al. proposed attributed community query (ACQ). Algorithms are also given to retrieve attributed community (AC) from an attributed graph. The attributed graph in ACQ problem is defined to be the graph with vertex attributes. An AC satisfies two constraints: structure cohesiveness and keyword cohesiveness. Structure cohesiveness requires that vertices of the AC are closely linked with each other, while keyword cohesiveness requires that vertices have common attributes. Hence, a community retrieved by ACQ represents a group of users that interact actively with each other and all of them share some common interests.



(a)Vertex-attributed graph (b)Resulting community of query (A, {x})

Figure 4.1 Example of Vertex-attributed graph representation the resulting community

However, it could be the case that the interactions between two users are not associated with the keywords we specify. Figure 4.1(a) gives a simple example of a vertex-attributed graph representation of a social network. We may assume that it is an E-mail network. If two users have a communication about a certain topic, an edge will be drawn between the vertices representing the two users and the keyword of the topic will be added to the attribute list of the two vertices. Figure 4.1(b) gives the result of the query (A, 1, {x}) on the graph constructed. It retrieves the community containing the vertex A and each vertex in the community has the keyword x. The query also requires that the degree of each vertex is at least 1. Edge A-C is included in the result because A and C both have the keyword x and they have communications. However, it is possible that A only communicates with C on the topic z, but A communicates with D about x and C usually discusses x with B. In this case, we assume that the edge A-C makes no contribution to the theme of resulting community because we are interested in the topic x but A-C reflects the relationship about y.



(a)Edge-attributed graph (b)Resulting community of query (A, 1, {x})

Figure 4.2 Example of Edge-attributed graph representation the resulting community

Using edge attributes can solve this problem. Figure 4.2(a) models the same email network as the previous example. In this model, if there is a communication between two users, one edge is drawn between these two users but the keyword of the letter is added to the keyword set of the edge. Vertices do not have keyword sets. When running the query (A, 1, {x}), only edges associated with keyword x will be considered as valid. The resulting community is shown in Figure 4.2(b). In the resulting community, edge A-C is not included because it is not associate with {x}, which is the theme of the community.

Edge attributes also contain more information than vertex attributes. If we assign a certain

attribute to a vertex given that it has an edge with this attribute, the new graph will satisfy the definition of the attributed graph in ACQ. For example, in Figure 3.2(a), the attribute sets of the edges of A are {x, y}, {z}, {x} and {y, z}, so the corresponding attribute set for the vertex A in the vertex-attributed graph is {x, y, z}. The inferred graph would be the same as Figure 3.1(a). Because the characteristics of users are actually reflected by the interactions, edge attributes not only contain the information carried by vertex attributes but also provide information about the relationships. Therefore, we hypothesized that communities retrieved by community search algorithm considering edge attributes instead of vertex attributes will still guarantee the structure cohesiveness and keyword cohesiveness. Meanwhile, the retrieved communities are associated more closely with the given theme.

4.2 Problem definition:

The graph is assumed to be undirected in most community detection and community search works. We also examine the undirected graph G(V, E), with vertex set V and edge set E. Each edge e is associated with a set of attributes denoted by K(e). We denote the set of attributes of a vertex to be the union of all attribute sets of all edges associated with that vertex. The set of attributes of a vertex v is denoted by Kv(v). Symbols used in this paper are listed in the table below:

Symbol	**Meaning**
G(V, E)	A graph with vertex set V and edge set E
K(*e*)	The attribute set of edge e
$K_v(v)$	The attribute set of vertex *v*. Defined as the union of the attribute sets of all attribute set of the edges associated with *v*.
$deg_G(v)$	The degree of vertex *v* in G
G[S']	The largest connected subgraph of G s. t. the query vertex $q \in G[S']$ and $\forall e \in G[S'], S' \subseteq K(e)$

$G_k[S']$	The largest connected subgraph of G s. t. the query vertex $q \in G[S']$, $\forall e \in G[S']$, $S' \subseteq K(e)$ and $\forall v \in Gk[S']$, $deg_{Gk[S']}(v) \geq k$

The definition of edge-attributed community query (EACQ) is similar to that of ACQ proposed by Fang et al. in [1]. The only difference is that keyword cohesiveness is imposed on edge attributes instead of vertex attributes. The formal definition is given as follow:

Problem 1 (EACQ): Given an undirected graph G(V,E), a positive integer k, a vertex $q \in V$ and a set of keywords $S \subseteq W(q)$, where W(q) is the union of all keyword sets of edges linked with q, return a set G of graphs, such that $\forall Gq \in G$, the following properties hold:

- **Connectivity.** $Gq \subseteq G$ is connected and contains q;
- **Structure cohesiveness.** $\forall v \in Gq$, $degGq(v) \geq k$;
- **Keyword cohesiveness.** The size of L(Gq, S) is maximal, where L(Gq, S) is the set of keywords shared in S by all edges of Gq.

In the definition given above, we call Gq edge-attributed community (or EAC). By imposing that L(Gq, S) to be maximal, we wish that EAC(s) retrieved only contain the most desired edges in terms of the number of shared keywords. Because the different edge of q can be labeled with different sets of attributes, we may expect to get more communities with smaller L(Gq, S) compared with ACQ.

4.3 Dataset and preprocessing

A number of datasets are available. The datasets are acquired from large social networks such as DBLP, Tencent, and Facebook and are widely studied. In addition, the algorithm for preprocessing the datasets and model them as vertex-attributed graphs is also present. We can assume that the algorithm can be adapted to generate edge-attributed graphs.

In pure EACQ problem, we assume that all keywords are generated by the interaction between users. Hence the DBLP dataset is one of the most intuitive datasets that are of this form. The xml file of DBLP is available online*. In the DBLP dataset, every record is an academic

publication. Co-authorship can be viewed as the relationship between authors that cooperate on a paper and the keywords of that publication can be extracted as the attributes of the edge that represents the relationship.

We keep an array of users, for each user, we use the data structure *HashMap<Integer, Set<String>>* to store the information of its edges when parsing the xml file. The key of the hash map is the id of the neighbors of the user. For a given user, we can use the id of one of its neighbor to locate the edge between the user and that neighbor. The value of the map is a set of keywords. We obtain the following procedure for parsing the records and contract the user list:

Algorithm 1 Preprocess DBLP XML

Input: DBLP XML file
Output: The structure of the graph and the keyword sets of each edge

```
1: function PREPROCESS
2:     Initialize HashMap <Integer, Set<String>>[Max] users; //Contruct a sufficiently large user array
3:     Initialize HashMap<String, Integer> userMap; //For maping the user name to its id;
4:     Initialize userNum = 0; //Keep track of the number of authors registered
5:     for each record <title, authorList> do
6:         for each author in authorList do
7:             if userMap.get(author's name) is null then
8:                 userMap.add(author's name, userNum)
9:                 userNum++
10:            end if
11:        end for
12:        for author in authorList do
13:            for neighbor in authorList do
14:                if neighbor != author then
15:                    userId = userMap.get(author)
16:                    neighborId = userMap.get(neighbor)
17:                    if !users[userId].keySet.contains(neighborId) then
18:                        users[userId].add(neighborId, new Set<String>)
19:                    end if
20:                    users[userId].get(neighborId).add(extractKeywords(title))
21:                end if
22:            end for
23:        end for
24:    end for
25:    return users
26: end function
```

The returned array contains the structure information and keyword information of the whole graph. When contrasting the keyword set for each edge, we can also set the threshold for selecting keywords with the highest frequency. The array is stored in a file or database for

future processing.

4.4 Algorithm

In graph theory, a k-core, denoted by Hk, of a graph G is defined to be the largest subgraph of G with $\forall v \in Hk$, $deg_{Hk}(v) \geqslant k$. The structure cohesiveness is defined using the definition of k-core in the above problem definition. Therefore, for each subset S' of keyword set S, we can first traverse the whole graph and remove edges that do not contain S'. Then we retrieve the k-core containing the query vertex q as a candidate community. Finally, the candidate community with largest |S| will be returned. However, because there are $(2k - 1)$ nonempty subsets of S if $|S| = k$, the method becomes impractical when k gets large. Due to the poor performance of the intuitive approach, special algorithms that can retrieve edge-attributed communities more efficiently are needed.

We observe that the EACQ problem can be solved more efficiently by first split the graph through adopting the structural constraint then combine the keyword constraint and the structural constraint to finally get the targeting community. Hence, we may attack the EACQ problem following the similar route of the ACQ algorithm. In the solution to ACQ problem, a space-efficient data structure called CL-tree is designed based on the property of k-core. Algorithms making use of the properties of attribute sets are presented. The algorithm first makes us of the CL-tree to locate the k-core containing the query vertex and then efficiently retrieve the optimal community. Because both ACQ problem and EACQ problem use k-core as the measurement of structural cohesiveness, the differences reside in be the properties of the attribute sets. We assume that if the edge-attributed graphs can be proved to have similar prosperities with the vertex attributed graphs in terms of attribute sets, the EACQ problem can be solved by adapting the ACQ algorithms.

4.4.1 Implementation

The first approach is to attack the EACQ problem by adapting the ACQ solution. In the solution to ACQ problem, a space-efficient data structure called CL-tree is designed based on the property of k-core. Algorithms making use of the properties of attribute sets are presented. The algorithm first makes use of the CL-tree to locate the k-core containing the query vertex and then efficiently retrieve the optimal community. Because both ACQ

problem and EACQ problem use k-core as the measurement of structural cohesiveness, the differences reside in be the properties of the attribute sets. We assume that if the edge-attributed graphs can be proved to have similar prosperities with the vertex attributed graphs in terms of attribute sets, the EACQ problem can be solved by adapting the ACQ algorithms.

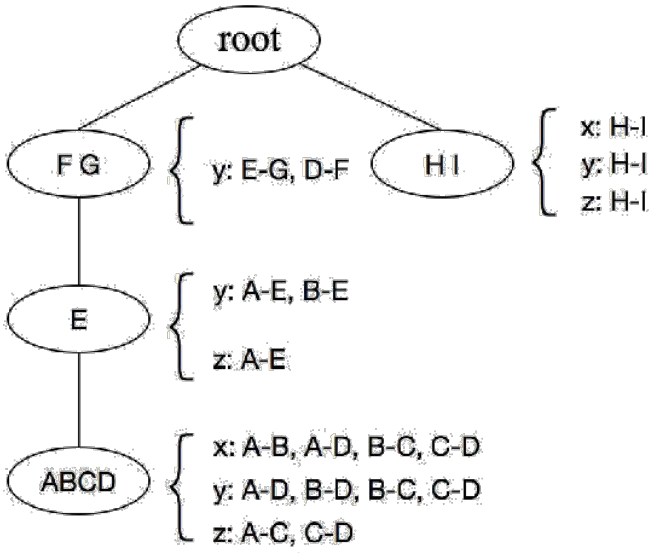


Figure 4.3 CL-tree for the edge-attributed graph

The whole ACQ procedure can be divided into two steps. First, given a vertex-attributed graph, construct the corresponding CL-tree to store the information extracted from the graph. Second, for the given query vertex, use the CL-tree constructed to locate the k-core that contain the vertex then find the optimal community iteratively. We first show how to construct the CL-tree for an EACQ community. For ACQ problems, each CL-tree node contains four components: *coreNum*, *vertexSet*, *invertedList* and *childList*. The *coreNum* stores the k value of the k-core that the present node represents. The *vertexSet* is a set of graph vertices that are contained by the k-core represented by the node but not contained by the *vertexSets* of the nodes rooted at the present node. The *clildList* simply store the pointers to the child tree nodes. These three components are solely associated with the structure of the graph. The *invertedList* is a list of <*key, value*> pairs, where the *key* is an attributed contained by at least one vertex in the *vertexSet*, and the *value* is the set of vertices possessing key in the *vertexSet*. The goal of the *invertedList* is to store the information of

attribute association. Both basic and advanced algorithm for constructing the CL-tree construct the tree structure first and then traverse the tree to construct the *invertedList*. Since edge-attributed graph and vertex-attributed have the same structural properties and in both ACQ and EACQ problems the k-core is used to measure the structural cohesiveness, we can safely assume that for EACQ problems, the tree structure and the three components associated with the structural properties can be constructed safely. We then present the definition of the *invertedList* in the EACQ problem and how to construct it.

For a CL-tree constructed based on an edge-attributed graph, we define the *invertedList* to be a list of <key, value> pairs where the key is an attribute and the value is a set of edges that connect two vertices in the *vertexSet* or connect one vertex from the *vertexSet* with another vertex contained in the *vertexSet*s of the subtree rooted at the presented tree node.

We defined a class based on the TNode class from ACQ algorithm and named it ETNode. Meanwhile, because Java does not have tuple data type, we defined the class TwoTuple by ourselves. Then we represent an edge using TwoTuple <Integer, Integer> and the two Integers are the vertex Id of the two end points.

The CL-tree constructed based on the edge-attributed graph given in Figure 2(a) is shown in Figure 3 as an example. The procedure of building the CL-tree can be divided into two major steps. The first step is to construct the tree according to the structural property. The second step is to build the inverted list for each tree node. Because edge-attributed graph and vertex attributed graph have the same structure, the first step of two methods follow the same procedure. The detailed procedure can be found in [6]. We then describe the detail of constructing inverted lists and we can safely assume that the tree structure has been constructed (see Algorithm 2).

Algorithm 2 Build CL-Tree for the edge-attributed graph

Input: $int[][]graph; String[][]nodes; ETNoderoot;$

Output: Return the pointer to the root of the tree after attatching keywords

```
1: //Define the function for checking whether the vertex set of the tree node and its subtree contain the give vertex
2: function CONTAINS(ETNODE node, INT id)
3:     if node.getNodeSet().contains(id)) then return true
4:     end if
5:     for childNode in node.getChildList() do
6:         if contains(childNode, id) then return true
7:         end if
8:     end for
9:     return false
10: end function
11:
12: Initialize HashMap < Integer, Set < Integer >> visited //To keep track of the edges visited
13: function ATTATCHKWS(ETNODE ROOT)
14:     for tree node in root.getChildList() do
15:         attatchKws(node) //Traverse the tree in post order and build the inverted list for each tree node
16:     end for
17:     Initialize Map < String, Integer > kwMap
18: Keep track of the keywords for the current tree node.
19:     Initialize kwIndex = 0
20:     Initialize List < List < TwoTuple < Integer, Integer >>> invertList
21:     for vertex v in root.getNodeSet() do
22:         for edge e in v.getEdges() do
23:             int neighborId < − id of the other end point of the edge e
24:             if !contains(root, neighborId) then
25:                 continue
26:             end if
27:             if e is visited then
28:                 continue
29:             end if
30:             for keyword in the keyword set of e do
31:                 if kwMap does not have key = keyword then
32:                     kwMap.put(keyword, kwIndex)
33:                     kwIndex + +
34:                     Initialize List < TwoTuple < Integer, Integer >> list
35:                     list.add(e)
36:                     invertedList.add(list)
37:                 else
38:                     Initialize intindex = kwMap.get(keyword)
39:                     invertedList.get(index).add(e)
40:                 end if
41:             end for
42:             mark e as visited
43:         end for
44:         Initialize Map < String, TwoTuple < Integer, Integer > [] > invertMap
45:     end for
46:     for each entry < keyword, keywordId > in kwMap do
47:         invertMap.put(keyword, invertedList.get(keywordId).toArray())
48:     end for
49:     root.setKwMap(invertMap)
```

Because the union of vertexSet of each CL-tree node is the set containing all vertices, all vertices and their edges are visited once when traversing the CL-tree. All edges and their corresponding attributes are stored in the inverted lists. After building the tree structure andthe inverted list, the k-core for a given vertex *q* can be located by finding the subtree containing *q* with the k value of the root tree node being k.

We then move on to the second step. In [1], the *Inc-T* algorithm for retrieving ACs makes use of one property of vertex-attributed graph. Let Gk'[S] denote the largest connected subgraph of a vertex-attributed graph G' s.t. q $\in$ Gk'[S], and $\forall$v $\in$ Gk'[S], $\deg_{Gk'[S]}$ (v) $\geqslant$ k and S $\subseteq$ W(v). Given Gk'[S1] and Gk'[S2], it can be proved that Gk'[S1 $\cup$ S2] $\subseteq$ Gk'[S1] $\cap$ Gk'[S2]. *Inc-T* algorithm first find the Gk'[S] for all S containing one of the keyword of the query vertex *q*. It then makes use of the property to find the community for a larger set of keywords iteratively. The algorithm stops when no feasible communities with larger attribute set can be found and return the communities found in the last round. Hence, if we can prove that edge-attributed graphs also have the similar property, i.e. Gk[S1 $\cup$ S2] $\subseteq$ Gk[S1] $\cap$ Gk[S2], we can modify the Inc-T algorithm to retrieve the EACs from the CL-tree. The proof goes as follows:

1. Let S1 and S2 be two set of keywords and S1 $\subseteq$ W(*q*), S2 $\subseteq$ W(*q*).
2. Let S be S1 $\cup$ S2. If Gk[S] exists, then Gk[S] contains q and each edge of Gk[S] contains S.
3. So, each edge contains S1 because S1 $\subseteq$ S.
4. Hence Gk[S] $\cup$ Gk[S1] contains *q* and every edge contains S1.
5. Meanwhile, since the core number of Gk[S] and Gk[S1] are at least k, the core number of Gk[S] $\cup$ Gk[S1] is at least k.
6. By the definition of Gk[S1], we haveGk[S] $\cup$ Gk[S1] $\subseteq$ Gk[S1].
7. Hence, we have Gk[S] $\subseteq$ Gk[S1].
8. Similarly, we also have Gk[S] $\subseteq$ Gk[S2]. Since Gk[S] $\subseteq$ Gk[S1] and Gk[S] $\subseteq$ Gk[S2] where S = S1 $\cup$ S2, we get Gk[S1 $\cup$ S2] $\subseteq$ Gk[S1] $\cap$ Gk[S2]. The proof is completed.

By the proof, we conclude that we can correctly obtain the community that we are interested in. Following that, we also give the algorithm of querying for the community for a given vertex and a set of keywords.

Algorithm 3 Query function of EACQ

Input: $int[][]graph; String[][]nodes; ETNoderoot; IntqueryId, Stringkws[];$
Output: Return a subgraph which is the desirable community

```
1: function QUERY( int[][]graph; String[][]nodes; ETNoderoot; IntqueryId, Stringkws[], intk)
2:     if The tree node containing the vertex v with queryId < k then return
3:     end if
4:     Traverse the CL-Tree using BFS, locate the CL-tree node subRoot such that its subtree contains
       v and has degree > k
5:     Define List < TwoTuple < Integer, Integer >> edgeList
6:     booleanlistInitialized = false
7:     for node in the subtree rooted at subRoot do
8:         for keyword in kws do
9:             if node.getKwMap().keySet() contains the keyword then
10:                if !listInitialized then
11:                    Initialize the edgeList to be node.getKwMap().get(keyword)
12:                else
13:                    edgeList = intersection of edgeList and node.getKwMap().get(keyword)
14:                end if
15:            end if
16:        end for
17:    end for
18:    Initialize int[][]subGraph = The graph spanned by edges in edgeList
19:    Locate the k-core containing v in subGraph
20: end function
```

We have shown that an edge-attributed graph can be represented by a modified CL-tree and the Inc-T algorithm can be adapted to retrieve EACs using the CL-tree. The algorithm has also been implemented. However, there are still some question to answer. Since usually there are significantly more edges than vertices in a graph, the efficiency of the approach need to be determined as all edges need to be traversed to construct the CL-tree. Meanwhile, we ought to explore more about the properties associated with the edge-attribute sets. There might be properties based on which more efficient second step approaches can be designed, like the Dec algorithm for retrieving ACs.

5. Experiment of EACQ and Results

We decide to test the algorithm in two aspects. The first aspect is the cohesiveness of the community retrieved. Since we impose the structural cohesiveness constraint when searching for the desirable community, we focus on measuring the keyword cohesiveness in the experiments. We also analysis the time efficiency of the EACQ algorithm and compare it with ACQ algorithm by executing the two algorithms on graphs extracted from the same dataset with the same query. Finally, we analyze the difference between communities retrieved by two algorithms and explore the reason.

5.1 Setup

Because the correspondence of edge-keywords need to be maintained for every edge, parsing the original dataset to get the edge-attributed requires large amount of memory. We observe that we would run out of memory when parsing the first half of records of DBLP dataset with 7GB of memory and it is relative. Hence, we decided to do the experiment with the graphs generated by the first 500000, 1000000 and 2000000 records respectively. The details of the graphs are listed in the following table. For each set of records, we use the preparation functions to generate the vertex-attributed graph and edge-attributed graph. The value k for k-core extraction is set to 2 for both algorithms. For each node in vertex-attributed graph and edge in edge-attributed graph, select the top 20 keywords as its keyword set.

Records	Vertices	Edges	Average degree
500000	322781	1686328	5.224
1000000	527461	2975352	5.640
2000000	864354	6077928	7.031

5.2 Structural cohesiveness

In the “Analysis” module of C-Explorer, we used two matrices to measure the keyword cohesiveness. They are community member frequency (CMF) and community pair-wise Jaccard (CPJ). The CMF measures the occurrence frequencies of the query vertex’s keywords in the community retrieved to determine the degree of cohesiveness and the CPJ calculates the

similarity between the keyword sets of any pair of vertices of community retrieved. The detailed description can be found in [6]. In general, higher scores of CMF and CPJ indicate that the community retrieved has higher keyword cohesiveness.

For each graph, we randomly select 5 vertices. For each vertex selected, we form the query using the vertex as the query vertex and select 2 of its most frequent keywords as the keyword set. The queries are issued to both ACQ and EACQ and we measure the CPJ and CMF of the communities returned.

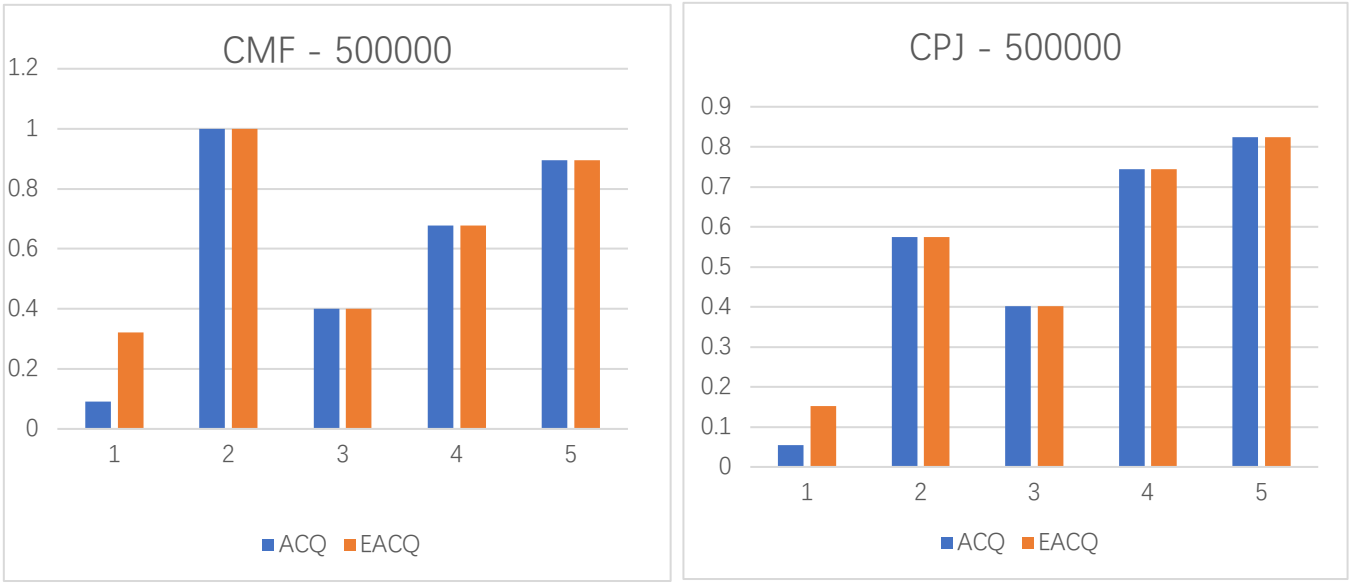


Figure 5.1 CMF and CPJ sample for the graph generated by the first 500000 records

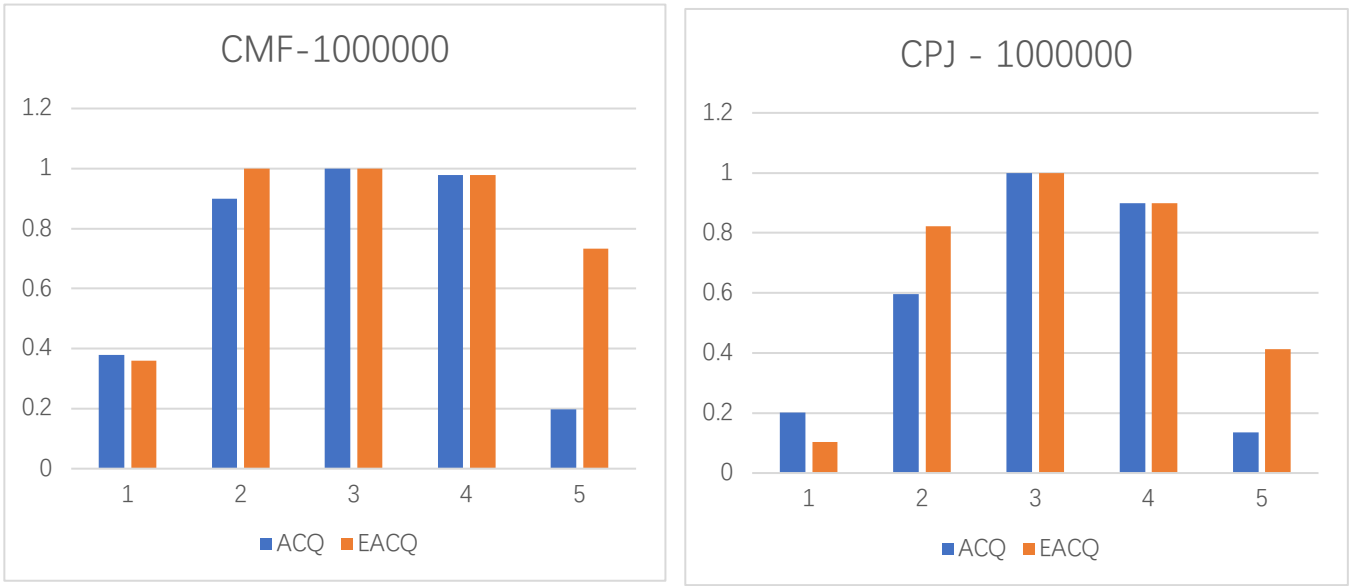
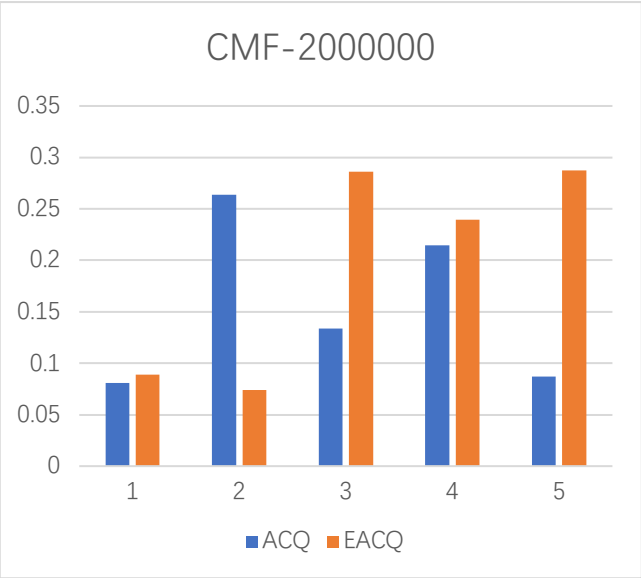


Figure 5.2 CMF and CPJ sample for the graph generated by the first 1000000 records



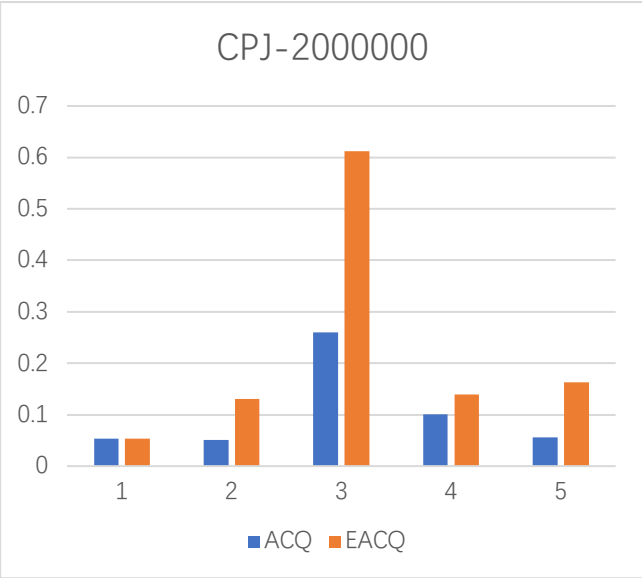


Figure 5.3 CMF and CPJ sample for the graph generated by the first 2000000 records

According to [6] and our previous experience of implementing the "Analysis" module of C-Explorer, ACQ usually scores higher than other community search algorithms. From figure 5.1, 5.2 and 5.3, we observe that the scores of EACQ is usually the same as those of ACQ when the graph is relatively simple.

There are also cases where EACQ and ACQ retrieve difficult communities. In those cases, the community retrieved by EACQ is usually a subgraph of the community retrieved by ACQ. Meanwhile, the CMF and CPJ scores are also higher. If the graph is more complicated because the increased number of records, EACQ tends to retrieve smaller while more cohesive communities.

However, there are also cases when EACQ retrieves larger graph than ACQ. After checking the keyword set of the corresponding vertices and edges, we find that nodes that are present in the EACQ community but absent from ACQ community all have 20 keywords in the extracted vertex-attributed graph. As described in 5.1, we select the 20 keywords with highest frequency. In edge-attributed graph, because we store at most 20 keywords for each edge and every vertex has multiple edges, the keyword capacity for each vertex is obviously larger than that for a vertex in the vertex-attributed graph. Hence, his situation should not appear if we set no limit to the number of keywords. However, more memory and time would be required if the size of keyword sets is unlimited and it may not be practical in real applications.

5.3 Efficiency

Because the EACQ algorithm contains two major steps and multiple query can processed after the construction of CL-tree, we analyze the time for building the CL-tree and time for performing the community search using the CL-tree separately.

5.3.1 Building the CL-tree

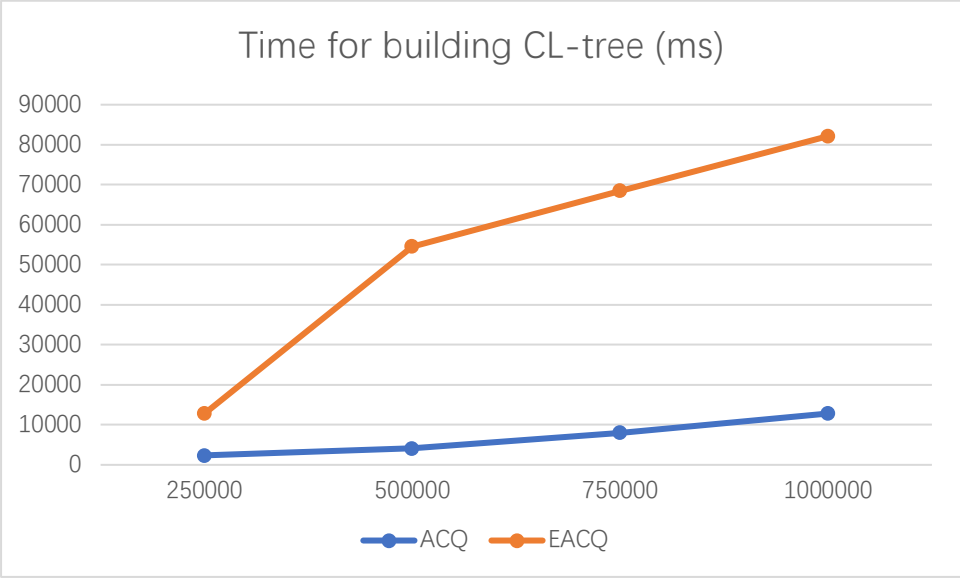


Figure 5.4 Time cost for building CL-tree

We build the CL-tree for the graph generated using 250000, 500000, 750000 and 1000000 DBLP records and measure the time for ACQ and EACQ respectively. From figure 5.1, we observe that the increase of time requirement for EACQ is quite large with the increase of records. We also tried to build the CL-tree for 1500000 records and it takes 271912ms to complete. The dramatic increase of time cost is caused by the explosion of edges.

Although CL-tree is only built once during start up, we still think that time cost is too high for us to accept given that the number of vertices and edges in real-world social networks are even bigger.

5.3.2 Time cost for query

Compared with the time required for building the CL-tree, we find that the time required for

completing an EACQ query is kind of “random”. It cannot be estimated solely by the size of the graph. Instead, it depends on the number of edges involved during the search. Or in other words, it depends more on the nature of the query vertex and the keyword set. Since we cannot predict which vertex or what keywords a user wants to use to issue a query, we cannot predict beforehand the time needed.

During experiments, we recorded time consumed for completing an EACQ query. In average, it takes around 369ms to complete a query. However, it may only take down to 7ms if the k-core of the query vertex is simple and up to 2451ms if the number of edges involved in search is extremely large. However, we think that the time cost for querying is acceptable for online applications. Meanwhile, we ought to keep improving the algorithm to make it less time consuming.

6 Difficulties Encountered

In this section, we discuss about the difficulties we encountered when doing this project. We first present the difficulties when implementing the C-Explorer and then we move on to the difficulties of implementing EACQ algorithm.

6.1 Difficulties concerning C-Explorer

6.1.1 <datalist> not supported by Safari

As is described in the methodology section, when the content of the text field under “Name” is changed, the system will guess the possible names that the user might want to type in. The candidates are shown using the HTML5 <datalist> tag (See Figure 3\2.1). We tested the system with the four mainstream browsers. The newest version of Chrome, Microsoft Edge, Internet Explorer and Mozilla Firefox support the tag but Safari does not. Because the list will have some content as long as there are some names that contain the input string, the feature of showing name candidates might help users to find the target name quickly

Figure 6 <datalist> is used for showing the recommended query name candidates. This is a screen capture under Chrome.

and also get to know whether a name is acceptable or not before clicking on the “+” button. Hence, we hypothesize that this feature can enhance the user experience. There might be substitution technology to achieve the same feature on Safari because search engines like Google can achieve similar keyword recommendation features. We will continue to search for substitutions and see if it can be implemented in the next stage of the project.

6.1.2 Unpredictable memory space requirement

In our original design, we planned to enable researchers to upload their dataset to our server and run algorithms directly on the dataset uploaded. However, we find that this might not be easy to achieve. Currently, the dataset we use for demonstration is retrieved from DBLP. To load this dataset, the machine needs to have at least 4GB of memory. Considering that this dataset is much smaller than some other datasets retrieved from popular social networks such as Facebook or Tencent, the memory requirement for loading the dataset uploaded by researchers is difficult to estimate and it might also be unaffordable to provide a virtual machine to accommodate such a requirement. Hence, we decide not to implement this feature. Instead, standard interfaces will be designed for plugging in dataset locally. The program will be made open source so that interested researchers can download and run the program on a suitable machine.

6.1.3 Unacceptable user experience when the community displayed has large number of vertices

It is highly likely that the communities returned have more than 40 vertices. We observe that if a community with more than 40 vertices is displayed, the names and avatar of vertices tend to overlap each other. Furthermore, if there are more than 1000 vertices, the large number of DOM elements in the SVG causes the change of coordinates for all elements to take more than 2 seconds to complete. This delay makes zoom-in and zoom-out almost unusable when the number of vertices is too large. The JUNG library will also spend longer time to calculate the layout for large communities and users might feel that the system fails to respond to a certain query. To compensate for the readability of the communities displayed and the overall performance, we decided to adopt "2-hop" strategy. If there are more than 40 vertices, the query vertices and their direct neighbors will be put in the result. Then the 2-hop neighbors (neighbor of neighbors) will be added until the number of vertices reaches 40 or all the 2-hop neighbors

are added. In this way, we expect to limit the number of vertices in a community displayed to be lower than 40 while still keeping the most important vertices in the community with respect to the query vertices.

6.1.4 Lack of community search algorithm in multi-layer graphs

In section 4.4, we proposed to use multi-layer graph clustering methods to attack the EACQ problem. However, multi-layer clustering is not query based. Given a multi-layer graph, the goal of multi-layer clustering is to partition the graph into cohesive clusters. As a result, multi-layer graph clustering algorithms are widely studied in the field of community detection. Because community search problems usually model social networks as simple undirected graphs with no attributes, there is no community search algorithm on multi-layer graph to the best of our knowledge.

One straightforward way to find the target community for the given vertex is to first partition the whole graph, then traverse the resulting clusters to find the target community. However, this procedure is considered unacceptable for community search applications because the overhead caused by clustering the graph globally is significant. In order to mitigate the time cost so that the algorithm can run in an on-line manner, a special algorithm is required to search for the community locally. Although we do not have the exact solution at the present stage, we think it possible to adapt the existing communication detection algorithm for multi-layer graphs to an efficient community search algorithm. Because community search problem is a query-based variant of community detection problem, a number of community search algorithms are inspired by community detection algorithms. We will do further literature review in this direction to explore the possible solutions.

6.2 Difficulties concerning EACQ

6.3.1 Insufficient memory for preprocessing

When implementing the C-Explorer system, we used the whole DBLP dataset to generate the vertex-attributed graph. The whole dataset contains about 10400000 records. However, our algorithm for generating edge-attributed can only process up to half of the dataset due to memory limitation. The algorithm is adapted from the preprocessing algorithm of ACQ and

the procedure and data structure used is not of great difference. Hence, we think the difference of preprocessing capability is caused by the increased number of keyword sets.

Because keyword sets are associated with edges and they are key for the edge-attributed community search algorithm, we cannot reduce memory requirement by reducing the size of keyword sets. Instead, we would like to explore the possibility of updating the preprocessing algorithm to reduce the amount of memory required.

7 Conclusion

We have presented C-Explorer, a web-based platform facilitating community search algorithms in terms of query formulation, community visualization, and algorithm comparison. The basic functions have been implemented and demoed and the response from the audience is positive in general except for some performance issue. The performance problem is also solved in the current version. Meanwhile, the interface for researchers to plugging in algorithms and datasets will be redesigned and implemented.

We also discussed about the motivation, methodology and implementation of the edge-attributed community search algorithm EACQ. The algorithm has been implemented and tested. By experiments, we show that EACQ can retrieve more desirable communities by considering the relationships. Meanwhile, we also discover that the time cost for constructing the CL-tree is too high for EACQ and we ought to continue to improve the preprocessing algorithm in the future.

In addition to the edge-attributed community search problem, we hope to provide one possible research direction. One goal for community search algorithms is to increase the cohesiveness of the community retrieved. It is common that the number of vertices retrieved can be large. In most cases, the number of common attributes is less than 3. This means that for the given query name and attribute set, the vertex in the community retrieved is “not that common”. To make the community retrieved more cohesive, introducing additional criteria might be helpful. In the program design, the use of edge importance is independent of the algorithm used for retrieving communities. However, it can be observed that by increasing the expected value of edge importance, the size of the community can be reduced. Since a higher value of edge importance means that the relationship between two vertices is more active, communities whose edges

have higher edge importance should be considered more cohesive. Hence, it might be worth trying to consider the edge importance when designing the new community search algorithm. The new algorithm may be called weighted edge-attributed community search algorithm if the definition of edge importance is also related to attributes. For example, the current edge importance represents the number of cooperation between two game players or the number of co-authorship between two authors. Though this value can represent the closeness of the two vertices, it is not related to the theme of the community retrieved. If edge importance represents how many times two game players have played a certain game together or the number of co-authorship of papers in a certain research area between two researchers, it might be intuitive to think that the vertices with higher edge importance are more likely in the same community with the corresponding theme.

8 Reference

[1] F. Zhang, Y. Zhang, L. Qin, W. Zhang, X. Lin. When engagement meets similarity: efficient (k, r)-core computation on social networks. *Proceedings of the VLDB Endowment*, *10*(10), 998-1009, 2017.

[2] M. Sozio, A. Gionis. The community-search problem and how to plan a successful cocktail party. *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2010.

[3] Y. Fang, R. Cheng, S. Luo, J. Hu, K. Huang. C-Explorer: browsing communities in large graphs. Proceedings of the VLDB Endowment, 10(12), 2017.

[4] P. Yi, B. Choi, S. S. Bhowmick, and J. Xu. Autog: A visual query auto completion framework for graph databases. *PVLDB*, 9(13):1505–1508, 2016.

[5] N. Jayaram, S. Goyal, and C. Li. Viiq: auto-suggestion enabled visual interface for interactive graph query formulation. *PVLDB*, 8(12):1940–1943, 2015.

[6] Y. Fang, R. Cheng, S. Luo, and J. Hu. Effective community search for large attributed graphs. PVLDB, 9(12):1233–1244, Aug. 2016.

[7] G. Qi, C.C. Aggarwal, and T. Huang, Community detection with edge content in social media networks. In Data Engineering (ICDE), 2012 IEEE 28th International Conference on (pp. 534-545). IEEE. April, 2012.

[8] Cui. W., Xiao. Y., Wang. H. and Wang. W., Local search of communities in large graphs. In Proceedings of the 2014 ACM SIGMOD international conference on Management of data (pp. 991-1002). ACM. June, 2014.