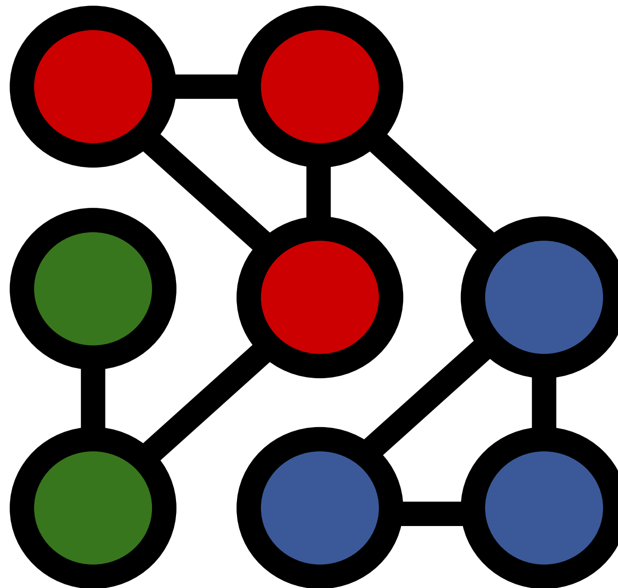
Utrecht University

Department of Information and Computing Science



Identifying and Characterizing Communities in Social Networks

Author:
Simon Pool
ICA-3032035

Supervisors:
Matthijs van Leeuwen
Francesco Bonchi
Arno Siebes

July 24, 2012

Abstract

Methods for detecting community structures in graphs already exist for many years. This subject is studied by physicists, sociologists and also computer scientists. Traditional methods consider only the vertices and edges, we call this graph data. In social networks much more information, in addition to the graph data is available. This can be demographical information, hobbies or any other interest people put online; we refer to this kind of data as description data. Traditional methods try to partition the vertices in groups according to a quality measure like modularity. These methods do not allow overlap; all vertices are member of exactly one group.

In real social networks, communities have overlap, for example your friends and your family. Thus, a good method for finding communities in social networks should allow communities to overlap. Other interesting information can be obtained by exploiting the description data. A very useful application is identifying which elements of the description data characterize a community.

In this thesis we study this problem, with the goal of finding the top-k communities in a certain data set. We introduce an algorithm which alternates between two steps. The first is finding closely linked vertices on the graph side with a fast and effective hill climbing algorithm. The other is reducing the description complexity of this community. The algorithm starts with a candidate set, and the algorithm is applied on each community one by one. This allows communities to overlap with the communities found before.

To evaluate our methodology, we performed experiments on real world data obtained from a number of online social networks, i.e. LastFM, Delicious, and Flickr. The results show that the proposed method identifies interesting and overlapping communities, characterized by detailed descriptions. Visualizations of both the subgraphs and the descriptions contribute to an easier interpretation and thus better understanding of the communities.

At the end we are able to find cohesive communities with concise descriptions, in large data sets, within a relatively short amount of time.

Contents

1 Introduction **3**
1.1 Motivation . . . 3
1.2 Goal . . . 3

2 Preliminaries **5**
2.1 Subgroup Discovery . . . 5
2.2 Exceptional Model Mining . . . 7

3 The Problem **8**
3.1 Problem statement . . . 9

4 Quality Measures **11**
4.1 Graph Space . . . 11
4.1.1 Community Quality . . . 12
4.2 Description Space . . . 14
4.3 Community Score . . . 15

5 Algorithms **16**
5.1 Exception Maximization Description Minimization . . . 16
5.1.1 Stop Criterion . . . 17
Saving History . . . 17
Maximum Iterations . . . 17
5.2 Generating Candidates . . . 17
5.2.1 Singletons . . . 18
5.2.2 Using Tiles . . . 18
5.2.3 Minimum Shortest Path . . . 18
5.2.4 Based on Description . . . 19
5.3 Exception Maximization . . . 20
5.3.1 Conditions and Consequences . . . 20
5.3.2 Add Operator . . . 20
5.3.3 Remove Operator . . . 21
5.3.4 Hill Climbing . . . 22
5.4 Description Minimization . . . 23
5.4.1 ReMine . . . 24
BestSplit . . . 25

6 Data **27**
6.1 Real World Data . . . 27
6.1.1 Last FM . . . 27
6.1.2 Delicious . . . 27
6.1.3 Flickr . . . 28
6.2 Synthetic Data . . . 28

7 Experiments **29**
7.1 Setup . . . 29
7.1.1 Hardware . . . 29
7.1.2 Software Implementation . . . 29
7.2 Graph Space . . . 29
7.2.1 EM Step (GreCO) . . . 29
7.2.2 Community Gain . . . 30
Inverse Conductance . . . 30
Intra Cluster Density . . . 31
Modularity . . . 32
7.3 Candidate Selection . . . 33
7.3.1 Singletons . . . 33
7.3.2 Tiling . . . 33
7.3.3 Based on Description . . . 33
7.3.4 Minimum Shortest Path . . . 33
7.4 Classifiers . . . 34
7.4.1 Decision Trees . . . 35
7.4.2 Nearest Neighbor . . . 35
7.4.3 Emerging Patterns . . . 36
7.4.4 ReMine . . . 36
7.5 Overall Results . . . 36
7.5.1 Stop Criterion EMDM . . . 36
7.5.2 Overlap . . . 37
Preventing too Much Overlap . . . 38
7.5.3 Communities Found . . . 39
Real World Data . . . 39
CFinder Data . . . 40
7.5.4 Runtime . . . 41
7.5.5 Communities found . . . 41

8 Related work **42**

9 Conclusions **44**
9.1 Community gain . . . 44
9.2 EMDM Algorithm . . . 44
9.2.1 Local search . . . 45
9.2.2 Candidate Set . . . 45
9.2.3 EM step . . . 45
9.2.4 DM step . . . 45
9.3 Future research . . . 45
9.3.1 Description space . . . 45
9.3.2 Graph space . . . 45
9.3.3 Post processing . . . 46

Acknowledgements **47**

Bibliography **48**

A Summary Top-50 Communities **50**

B Summary Found Communties **51**

C Top-50 Communities Listed **52**

Chapter 1

Introduction

1.1 Motivation

Online social networks are booming, Facebook started in 2004 and now has almost a billion users, Twitter has 500 million users and Google+ has already 170 million users after only 9 months. Users are putting more and more information about their lives online. These social networks have abundant information about their user's interests. Collecting this information and storing it is one thing, however analyzing the information and drawing interesting conclusions about it is the most appealing.

Finding cohesive groups with concise descriptions could be useful in advertising. A big beer brand wants to organize a big rock music festival. It is a new, one time only festival so nobody knows it. They want to earn money by selling all available tickets. They could use social media to advertise for the event. This would be much easier when having groups of densely connected users, and knowing what they, as a group, like to do. Someone who likes rock music but who has no friends with the same interest probably won't buy a ticket because he doesn't like to go alone. On the other hand, a group of friends that likes to go to dance festivals probably do not buy tickets because they don't like the music. If we had groups of densely connected users, and know their interest it would be much easier to select the right users.

There are is much research to be done on this rapidly growing subject. One important area of research and the topic we will be focusing on is *Detecting Communities and Characterizing them.* This is about finding the groups we wanted to find in the previous example. As of yet, there has not been an extensive amount of research done on this topic.

1.2 Goal

Let us introduce the subject with an informal description of the research goal of this project. A social network can be thought of as a set of individuals, users with each user containing some information. Each user entry in the network stores two types of data, their description data and their connections to other users.

Each user has connections to zero or more other users, known as friendship links. This network component of social networks is called the graph space, where each user is a vertex and each friendship link is an edge between two users. The description data consist of information which is not part of the network, for example gender, favorite color, name or age. When thinking of Facebook, description data also includes "likes" user gives to musicians, books, companies, pictures or any other type of information a user puts online. Essentially, most information in social networks associated with individual users which is not part of the graph structure can be transformed into description attributes. Every user consists of values for one or more of these attributes.

Within social networks there are groups of users that are more densely connected with each other than on average in the database. Examples where on average the density of edges is much higher can be observed in a group of friends, users that are fan of the same band, users that do the same study in the same city, users who live in the same village, and so on. These groups are called communities.

The goal of this research project is to find these communities, and characterize them. By characterizing the communities, we mean to find which properties in the description space are typical for those specific communities. The algorithm to find these communities and their description needs to be fast and efficient and should be suited to run on large data sets.

We build on top of the EMM framework, which is a generalization of the topic of datamining called subgroup discovery. Chapter 2 explains the details about this framework. In Chapter 3 we formally introduce the *Community Characterization Problem* on a formal way. We also explain how to transform this problem into an instance of EMM. In physics many methods of detecting communities in a graph are described, they use quality measures like modularity, inverse conductance or intra community density as quality measures. Most of them do not allow overlap, and most of them are not suited to run on large data sets. We introduce a new quality measure, *Community Gain*, in Chapter 4. This quality measure makes it possible to allow overlap, and is fast enough to run on large data sets. After that we introduce the algorithms necessary to complete our task. Finally we describe the data sets we use in Chapter 6 and present our results in Chapter 7.

Chapter 2

Preliminaries

In this chapter we will introduce the framework that will be used as a base for the research to detection and characterisation of communities. The framework is known as exceptional model mining (EMM), which is a topic from the data ming research area. Before we introduce EMM we start with explaining subgroup discovery, as EMM is a generalization of subgroup discovery.

2.1 Subgroup Discovery

Subgroup discovery is concerned with finding regions in the input space where the distribution of a single target attribute is substantially different from its distribution in the whole database [9]. Given are database DB with description attributes A and target attribute y. Each individual $r_i \in DB$ is a set of values for all attributes $a^j \in A$ and a value y_i for target attribute y. For each description attribute $a^j \in A$ the domain is specified by $dom(a^j)$, for target attribute y the domain is specified by $dom(y)$. Now an individual $r_i \in DB$ is a tuple $(x_i^1, \dots, x_i^{|A|}, y_i), x_i^j \in dom(a^j), y_i \in dom(Y)$.

Subgroup discovery focuses on finding subgroups $s \subset DB$ that have a deviating distribution on the value of the target attribute y compared to the whole database DB. A subgroup s is a subset of database $s \subset DB$. Each subgroup s has a description query Q_s, this is a query over attributes A. Each query Q_s is a conjunction of m selectors $Q_s = e^1 \wedge \cdots \wedge e^m$. Each selector $e^i \in Q_s$ is a tuple $(a^k, op_k, x_k), a^k \in A, op_k \in \{<, \geq, =, \neq\}, x_k \in dom(a^k)$.

Example 1. An example of a subgroup description query Q_s could be:

$$Q_s = \{a_1, =, 1\} \wedge \{a_2, =, 0\} \wedge \{\{a_4, \geq, 1\} \wedge \{a_5, <, 0\} \wedge \{a_6, \neq, 0\}$$

We denote the universe of all possible queries over A as $\mathbb{Q}_A$. Given database DB we define a function $g : \mathbb{Q}_A \to 2^{DB}$ associating to each query $Q \in \mathbb{Q}_A$ the set of records that satisfy Q.

Definition 1 (Subgroup)**.** We define a subgroup s associated with description query Q_s as the set of records $g(Q_s) \subset DB$ that satisfy Q_s. That is $g(Q_s) = \{r_i \in DB : A(r_i) \models Q_s\}$.

$$DB = \begin{array}{c} r_1 \\ r_2 \\ \vdots \\ r_{|DB|} \end{array} \left(\begin{array}{cccc|c} & & A & & y \\ \hline x_1^1 & x_1^2 & \dots & x_1^{|A|} & y_1 \\ x_2^1 & x_2^2 & \dots & x_2^{|A|} & y_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{|DB|}^1 & x_{|DB|}^2 & \dots & x_{|DB|}^{|A|} & y_{|DB|} \end{array} \right)$$

Figure 2.1: Subgroup discovery

A quality measure $\varphi : 2^{DB} \rightarrow \mathbb{R}$ is used to quantify the quality of a subgroup, i.e. how much a subgroup deviates from the whole database. The higher this value, the better the subgroup is. The goal of subgroup discovery is finding subgroups and their corresponding description queries for the top-k (with respect to the quality function $\varphi(s)$).

Example 2. An example could be a database with clients of a bank that have a loan. The description attributes are attributes $A = \{age, married, own\,house, income, gender\}$ that describe a person. The target attribute y is a boolean that says whether the client is able to pay back loan to the bank or not. An example of this data set is shown in Figure 2.2

DB =	A					Y	Subgroup	
	age	*married*	*own house*	*income*	*gender*	*pay back*	$g(Q_1)$	$g(Q_2)$
r_1	22	*no*	*no*	28,000	*male*	*no*		✓
r_2	46	*no*	*yes*	32,000	*female*	*no*		
r_3	24	*yes*	*yes*	24,000	*male*	*no*		✓
r_4	25	*no*	*no*	27,000	*male*	*no*		✓
r_5	29	*yes*	*yes*	32,000	*female*	*no*		✓
r_6	45	*yes*	*yes*	30,000	*female*	*yes*		
r_7	63	*yes*	*yes*	58,000	*male*	*yes*	✓	
r_8	36	*yes*	*no*	52,000	*male*	*yes*	✓	
r_9	23	*no*	*yes*	40,000	*female*	*yes*	✓	
r_{10}	50	*yes*	*yes*	28,000	*female*	*yes*		

Figure 2.2: Subgroup discovery example

Now the goal is to distinguish groups that have a deviating distribution on the value for the target attribute y compared to the whole database. In this case that means that we would like to find subgroups of records that consist of a large majority of records that have the same value for y. Two of these subgroup description queries can be:

$$Q_1 = \{\{income, \leq, 36000\} \wedge \{age, \geq, 37\} \wedge \{married, =, yes\}\}$$
$$Q_2 = \{\{income, \leq, 36000\} \wedge \{age, \leq, 37\}\}$$

Applying queries Q_1 and Q_2 on the database given in Figure 2.2 gives the following subset. The records selected by Q_1 and Q_2 are also marked in Figure 2.2:

$$g(Q_1) = \{r_7, r_8, r_9\}$$
$$g(Q_2) = \{r_1, r_3, r_4, r_5\}$$

Both subgroups selected by Q_1 and Q_2 are pure, i.e. all records within a subgroups $g(Q_1)$ and $g(Q_2)$ have the same target value $y_i \in dom(y)$. Records $g(Q_1)$ have target value $y = yes$ and records $g(Q_2)$ have target value $y = no$. The data set in Figure 2.2 is very small and has only few attributes, that makes it easy to find pure subgroups. In other cases, e.g. large data sets, it might not be possible to find subgroups that are pure within a reasonable time. In general you could say that the more pure a subgroup s is, the higher the quality $\varphi(s)$ is.

Subgroup discovery problem

Given a database DB the task of the Subgroup discovery is to find the top-k subgroups that have a deviating distribution of target value y, compared to the distribution of y in the whole database. The value $\varphi(s)$ is used to rank the subgroups found.

2.2 Exceptional Model Mining

Exceptional model mining can be thought of as a generalization of subgroup discovery [9]. The goal in subgroup discovery is finding subgroups with a deviating value of a *single* target attribute y. In exceptional model mining there are *multiple* target attributes $y \in Y$.

Again we have database DB with description attributes A, however, the target Y in exceptional model mining is a set Y of model attributes. Now each record $r_i \in DB$ consists of values for each description attribute $a^j \in A$ and each model attribute $b^j \in Y$. An individual $r_i \in DB$ is a tuple $(x_i^1, \dots, x_i^{|A|}, y_i^1, \dots, y_i^{|Y|})$ where values $x_i^j \in dom(a^j), a^j \in A, y_i^j \in dom(b^j), b^j \in Y$. A formal scheme of a data set used for exceptional model mining is shown in Figure 2.3.

$$DB = \begin{array}{c} \\ r_1 \\ r_2 \\ \vdots \\ r_{|DB|} \end{array} \left(\begin{array}{cccc|cccc} & A & & & & Y & & \\ \hline x_1^1 & x_1^2 & \dots & x_1^{|A|} & y_1^1 & y_1^2 & \dots & y_1^{|Y|} \\ x_2^1 & x_2^2 & \dots & x_2^{|A|} & y_2^1 & y_2^2 & \dots & y_2^{|Y|} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{|DB|}^1 & x_{|DB|}^2 & \dots & x_{|DB|}^{|A|} & y_{|DB|}^1 & y_{|DB|}^2 & \dots & y_{|DB|}^{|Y|} \end{array} \right)$$

Figure 2.3: Exceptional Model Mining

In subgroup discovery it is easy to calculate whether a value is deviating or not, for example by looking at the average value of the y in the whole database, and the average value of the y in a subgroup. With EMM this is much more difficult because Y consists of more than one attribute. To decide whether a set of records s is exceptional or not, the values of the target attributes Y of records s are used as input to build a model. A model could be a Bayesian network, a classifier, of something else. Given a quality measure $\varphi : dom(model) \rightarrow \mathbb{R}$ the quality of the model is calculated. If the quality is higher than a certain threshold ϵ the model is exceptional. With EMM the goal is to find exceptional models and their corresponding model queries $Q_m \in \mathbb{Q}_A$ for models m. The descriptions queries have the same structure as the description queries used in subgroup discovery. The query Q_m selects a set of records $g(Q_m)$.

Exceptional model mining problem

The goal of Exceptional model mining problem is to find the top-k models that are deviating compared to the model of the whole database. The value $\varphi(m)$ is used to rank the models found according to their quality. In our research we will use a specific instance of EMM, where the data used as values for attributes Y is graph data. A more detailed example is given in the next section.

Chapter 3

The Problem

In a social network, there are users and there is information associated with those users. We therefor represent a social network as an attributed graph, in which each user is a vertex in this graph. Users have a list of friends, these friendship links are the edges of the graph. Users also put other information online. For example a user likes his favorite musician, or says he is born in New York, we call this information description data. The data we consider in our research have both the graph data and description data. We want to find groups of users that have strong connections with each other in the social graph. We also want to find a description query as we saw in Section 2.1 to characterize them. We call this the *Community identification and characterization problem.*

Community identification and characterization can be treated as an instance of EMM, with a database DB, description attributes A and model attributes Y. But now attributes $y^j \in Y$ are the users them themselves, and the values of these attributes are the edges E between the users V. This gives us the attributed graph $G = (V, E, A)$ where V is the set of vertices (users), $E \subseteq V \times V$ is the set of undirected edges (friendship links) and $A = \{a^1, \dots, a^{|A|}\}$ is the set of description attributes. Each user $v_i \in V$ is associated with one or more attributes $A[i] \subseteq A$. The values of target attributes Y are the edges in the social graph. This means value y_i^j of user v_i has value $v_i^j = 1 \iff (v_i, v_j) \in E$ and $v_i^j = 0$ otherwise. The values of attributes $y^j \in Y$ can be thought of as the adjacency matrix of the social graph G. From now on we refer to the database as $DB = G(V, E, A)$. We use both r_i and v_i to refer to user i, dependent on whether we talk about the graph vertex or the whole user record.

For individual $r_i \in DB$, row vector A_i holds all description information of user v_i. In our research we only use binary attributes for values in $a^j \in A$, i.e. $\forall a^j \in A : dom(a^j) = \{0, 1\}$. For example, when attribute $a^1 = New\,York$ value a_i^1 of user v_i has value $a_i^1 = 1$ iff user v_i lives in New York and 0 otherwise. For readability we change the notation of the pattern language as defined in Section 2.1 to an easier notation. Also, to make the language more expressive we allow disjunction of patterns.

From now on, we denote the tuple $(a^j, =, 1)$ as $\{a^j\}$, and the tuple $(a^j, =, 0)$ as $\overline{\{a^j\}}$. Thus, each description query Q_C is a query over attributes A. A query Q_C consists of a disjunction of patterns p, e.g. $Q_C = p_1 \vee \ldots \vee p_n$. Each pattern p_i consists of a conjunction of positive and negative conjunctions, for example $p_i = \{a^1\} \wedge \{a^3, a^5\} \wedge \ldots \wedge \overline{\{a^2, a^1\}}$. In this context $\{a^1, \dots, a^n\}$ means a conjunction between all the attributes between the brackets, and $\overline{\{a^1, a^2\}}$ means $NOT(a^1 \wedge a^2)$. A description query could thus be:

$$Q_C = (\{a_1, a_3, a_6\} \wedge \overline{\{a_4\}} \wedge \{a_5, a_2\}) \vee (\overline{\{a_1, a_3, a_6\}} \wedge \{a_4\} \wedge \overline{\{a_5, a_2\}}) \vee (\overline{\{a_1, a_3, a_6\}} \wedge \overline{\{a_4\}} \wedge \{a_5, a_2\})$$

From now on we refer to the description attributes as tags or the *description space* and to information related to the graph as the *graph space.*

Example 3. An example of the data that could be used for the problem we are focusing on could be the following set of tags $A = \{$New York, 1986, Student, Coffee, Lady Gaga$\}$. In the example

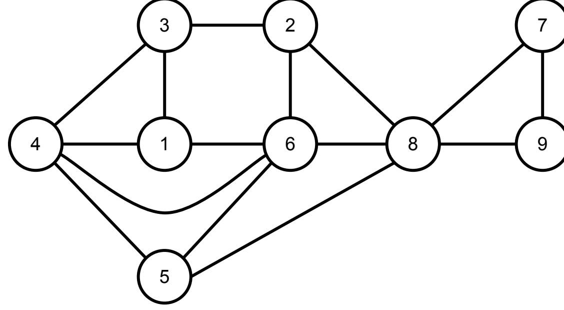


Figure 3.1: Example graph

$DB =$

	A					E								
	a^1	a^2	a^3	a^4	a^5	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
v_1	0	0	1	0	0	0	0	1	1	0	1	0	0	0
v_2	0	1	1	0	0	0	0	1	0	0	1	0	1	0
v_3	0	0	1	1	0	1	1	0	1	0	0	0	0	0
v_4	0	0	1	1	0	1	0	1	0	1	1	0	0	0
v_5	0	1	0	1	0	0	0	0	1	0	1	0	1	0
v_6	1	1	0	1	0	1	1	0	1	1	0	0	0	0
v_7	1	0	0	0	1	0	0	0	0	0	0	0	1	1
v_8	1	1	1	0	1	0	1	0	0	1	0	1	0	1
v_9	1	0	0	0	1	0	0	0	0	0	0	1	1	0

Figure 3.2: Example database, verify that data under E corresponds with Figure 3.1

there is a set of 9 users, $v_1, \ldots, v_9$. The graph used in this example is shown in Figure 3.1. The database corresponding with this data is given in Figure 3.2. The values of the description data are chosen at random.

Each user has used some of the tags and is connected to other users. This gives us the database DB as shown in Figure 3.2. The values a_i^j under A are 1 if and only if user v_i contains attribute a^j, for example value $a_9^1 = 1$ means user v_9 has flagged attribute a^1 (New York). The values under E is the adjacency matrix. As you can see value v_i^j has value $v_i^j = 1$ if and only if user v_i is connected to user v_j in the graph shown in Figure 3.1, value $v_i^j = 0$ otherwise.

A few community descriptions we could find in this data could be:

$$Q_1 = (\{Lady\ Gaga\})$$
$$Q_2 = (\overline{\{Lady\ Gaga\}} \wedge \{Student\}) \vee (\{Coffee\})$$

Applying queries Q_1 and Q_2 on the database given in Figure 3.2 gives the following communities:

$$g(Q_1) = \{r_7, r_8, r_9\}$$
$$g(Q_2) = (\{r_1, r_2, r_3, r_4, r_5, r_6\} \cap \{r_1, r_2, r_3, r_4\}) \cup \{r_3, r_4, r_5, r_6\}$$

3.1 Problem statement

Now that we have defined what the input data set looks like, it is time to focus on the goal of the research: identifying and characterizing communities. A community $C \subseteq DB$ is defined as a set

of individuals. The word *identifying* in this context means that we want to find communities that internally have a significantly higher value of connectedness, compared to the average of the whole database. The word *characterizing* means that we want to find description queries that represent the characteristic features of the individuals in $r_i \in C$.

A community C is described by a community description query $Q_C = \{p_1, \ldots, p_l\}$, where patterns $p_i \in Q_C$ have the same properties as the EMM patterns explained in Section 2.2, with the addition of allowing disjunctions and negations. The conjunction of these patterns determine which individuals from DB belong to community C. We denote the universe of all possible queries over A as $\mathbb{Q}_A$. Given database $DB = (V, E, A)$ we define a function $g : \mathbb{Q}_A \to 2^{DB}$ associating to each query $Q \in \mathbb{Q}_A$ the set of records that satisfy Q.

Definition 2. Community C corresponding to a community description query Q_C is the set of individuals $g(Q_C) \subset DB$ that satisfy Q_C. That is $g(Q_C) = \{r_i \in DB : A[i] \models Q_C\}$.

Definition 3. The complement of a community C is the set of individuals $\overline{C} \subset DB$ that are not member of C, i.e. $\overline{C} = DB \setminus C$.

In order to compare the quality of communities a quality measure needs to be defined. This quality measure should quantify the connectedness of a community. The more connected the community is, the higher this value should be.

Definition 4 (Graph quality measure)**.** A quality measure $\varphi : 2^V \to \mathbb{R}$ is defined to quantify the connectedness of a community C given a database $DB = (V, E, A)$. A high value $\varphi(C)$ means that the community is strongly connected compared to the whole database. For computing the value of φ only the graph space should be used, not the description space.

Another important part of this research is characterizing the community. It is important to obtain powerful descriptions in the description space for describing the communities. A good description is as short as possible, i.e. should have a low complexity. A description complexity measure is used to quantify the complexity of a description:

Definition 5 (Description complexity)**.** A measure to quantify the complexity of a description query Q_C for community C given a database $DB = (V, E, A)$. This is a function $\rho : \mathbb{Q}_A \to \mathbb{R}$. This function ρ computes a numeric complexity value for a description query Q_C. A lower value for this measure means a better description. For computing the value of ρ only the description space should be used, not the graph space.

Example In Figure 3.2 and Figure 3.1 an example data set is shown. As is clearly visible, community $C_1 = \{v_7, v_8, v_9\}$ is strongly connected within the community itself, in fact it has the maximum number of internal edges because it is a clique. There are also relatively few links to individuals outside the community. This community C should have a good score $\varphi(C)$.

When we look at the description data we see that it is possible to select all the individuals $r_i \in C$ by selecting on attribute a^5. A very short description query $Q_{C_1} = \{a^5\}$ results in selecting the right vertices, i.e. $g(Q_{C_1}) = \{v_7, v_8, v_9\}$. Another option for selecting these individuals could be $Q'_{C_1} = \{\{a^1\} \wedge \{a^5\}\}$. Description queries Q_{C_1} and Q'_{C_1} select the same individuals, but Q_{C_1} is shorter, thus $\rho(Q_{C_1}) < \rho(Q'_{C_1})$, and hance we prefer Q_{C_1}.

Community Characterization Problem

Given a database DB with the properties as specified in Section 3.1 the task is to find the top-k communities in database DB with respect to some objective function. As can be derived from Definition 4 and Definition 5 a good community C is well balanced between good properties in the connectedness sense, and the complexity of the description query. Some additional requirements may be added to resulting communities, like a minimum size $|C|$. It is important to note that communities may be overlapping.

Chapter 4

Quality Measures

In Definition 4 and 5 in Section 3.1 two formal quality and complexity measures are defined. In this section we will explain more about them and give an implementation for them. In the *Community Characterization Problem* three quality measures are important:

- Define a way to quantify the connectedness of a community. This quality measure is only about the graph space. The formal definition of this quality measure is given in Definition 4.
- Define how complex a description of a community is. The formal definition of this description complexity measure is given in Definition 5.
- Define how good a community and it is description are together, this is a combination of the first two measures.

4.1 Graph Space

The quality measure we will use for quantifying the quality of the graph part of the community is based on the intuition that the best communities are cliques, with no edges to vertices outside the clique. A clique is a set of vertices S with an edge between every pair of vertices in set S. An example of a graph with three communities that are all cliques is shown in Figure 4.1.

Our quality measure uses this idea to calculate the quality of a given community C. The idea is to compare a given partitioning into communities, with the situation in which the same communities would be cliques. We call the differences between these two situations *errors*. The more errors there are, the less is the quality. For each vertex v_i in community C are two types of errors, i.e. one concerned with the missing links withing the community, and one with the links that reach outside the community.

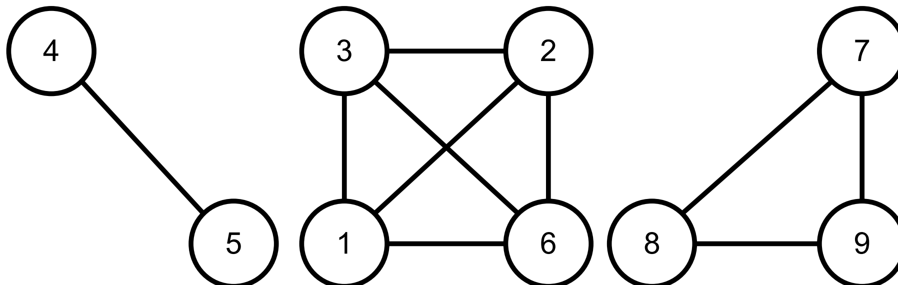


Figure 4.1: Three optimal communities: $\{v_1, v_2, v_3, v_6\}$, $\{v_4, v_5\}$ and $\{v_7, v_8, v_9\}$

1. **Between errors**: There are two vertices, v_i and v_j, with $v_i \in C$ and $v_j \in \overline{C}$. There is an error if edge $(v_i, v_j) \in E$ because that means there is an edge to a vertex v_j which is not a member of community C. The set of errors of this type related to vertex v_i is $Err_B(v_i)$. When clear from the context we will use $Err_B(v_i)$ for both the set of errors, and the size of this set.

2. **Within errors**: There are two vertices, v_i and v_j with $i < j$ and but now with both $v_i, v_j \in C$. There is an error if edge $(v_i, v_j) \notin E$ because there is an edge missing in between vertices in C. The set of errors of this type related to vertex v_i is $Err_W(v_i)$. When clear from the context we will use $Err_W(v_i)$ for both the set of errors, and the size of this set. We count within errors only in the vertex with the lowest index.

The next step is to define a model for counting the errors given a certain partitioning of individuals into communities. First we define what a model is.

Definition 6 (Model)**.** A model M is a set of communities in which every vertex $v_i \in V$ is part of exactly one community, i.e.:

$$\begin{aligned} \forall_{C_i, C_j \in M | i \neq j} : C_i \cap C_j &= \emptyset \\ \sum_{C_i \in M} |C_i| &= |V| \end{aligned}$$

We refer to the set of all possible models as $\mathbb{M}$

The enumeration of the errors of the graph G with model M is an enumeration of all the vertices $v_i \in V$ and the errors related to these vertices v_i. We use undirected graphs, that is why *within errors* between vertices v_i and v_j with $i < j$ are only counted in vertex v_i.

Definition 7 (Enumerate errors)**.** The enumeration of errors of graph $G = (V, E, A)$ encoded with model M is written down as.

$$encode(G|M) = \bigcup_{v_i \in V} \{v_i, \{Err_B(v_i)|Err_W(v_i)\}\}$$

Definition 8 (Counting errors)**.** The number of errors $L(G|M)$ of a graph $G = (V, E, A)$ encoded with model M is defined as the sum of the number of vertices and the total number of errors given a model M.

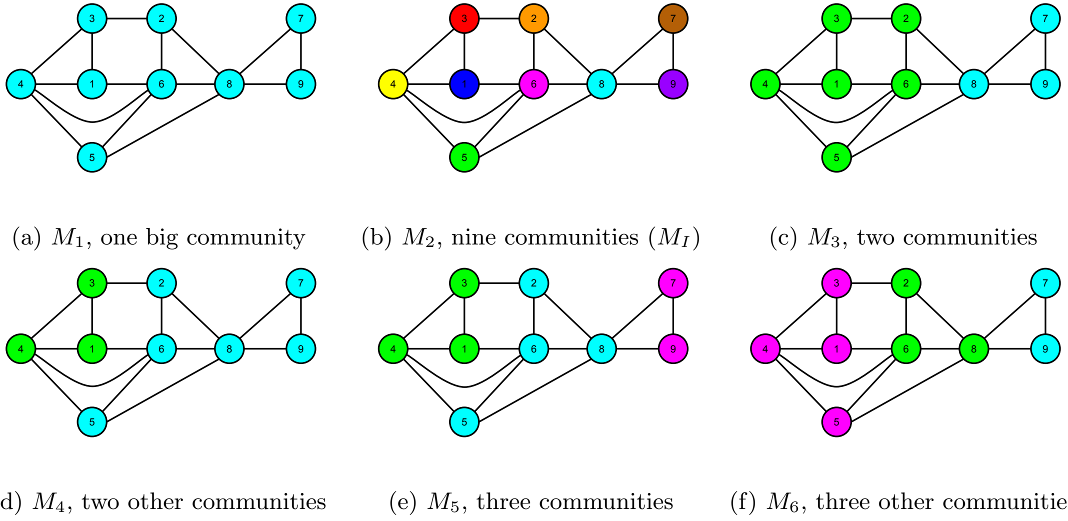
$$L(G|M) = \sum_{v_i \in V} (1 + |Err_B(v_i)| + |Err_W(v_i)|)$$

The optimal model $M \in \mathbb{M}$ for encoding graph G is the model with the least number of errors. Two get more familiar with the encoding we will start with some examples. In Figure 4.2 six possible models for the given graph are shown. Each model has a different list of errors which is given in Table 4.2g. The colors in the different figures represent different communities. Each figure is a different model.

When we order the models given in Figure 4.2 according to the number of errors given in Table 4.2g we will find that models M_3 and M_5, shown in Figures 4.2c and 4.2e are the best. These models contain the least number of errors, followed by model M_4 and M_6 with Figures 4.2d and 4.2f.

4.1.1 Community Quality

Now that we have defined the encoding scheme for the graph space we define the quality measure for a community. A problem with the error counting scheme as we have defined it, is that it need a model of the *whole graph* before we can say something about the quality of the set of communities. Another problem is that it is impossible to quantify the quality of an *individual community.* It is



(a) M_1, one big community (b) M_2, nine communities (M_I) (c) M_3, two communities

(d) M_4, two other communities (e) M_5, three communities (f) M_6, three other communities

M	$encode(G\|M)$	$L(G\|M)$
M_1	{1{|2,5,7,8,9},2{|4,5,7,9},3{|5,6,7,8,9},4{|7,8,9},5{|7,9},6{|7,9},7,8,9}	30
M_2	{1{3,4,6|},2{3,6,8|},3{1,2,4|},4{1,3,5,6|},5{4,6,8|},6{1,2,4,5,8|},7{8,9|}, 8{2,5,6,9|},9{7,8|}}	38
M_3	{1{|2,5},2{8|4,5},3{|5,6},4{|2},5{8|},6{8|},7,8{2,5,6|},9}	22
M_4	{1{6|},2{3|5,7,9},3{2|},4{5,6|},5{4|7,9},6{1,4|7,9},7,8,9}	24
M_5	{1{6|},2{3|5},3{2|},4{5,6|},5{4|},6{1,4|},7{8|},8{7,9|},9{8|}}	22
M_6	{1{6|5},2{3|},3{2|5},4{6|},5{6,8|},6{1,4,5|},7{8|},8{5,7,9|},9{8|}}	25

(g) Encoded versions

Figure 4.2: Six different models for graph G

very useful to quantify the quality of a single community without having the need to know how the whole graph is divided into communities. We will introduce an approximation method that quantifies the quality of a community by looking at the community and its neighbors only.

We call the measure we define *community gain*. The gain for a given community C is the decrease or increase of the number of errors when using community C. The community gain is defined as the difference in encoded size between encoding vertices $v_i \in C$ with two different models, model M_C and model M_I. Model M_C contains the set of vertices C and model M_I is the independent model. In the independent model M_I each vertex $v_i \in V$ is a separate community. $M_I = \bigcup_{v_i \in V} \{\{v_i\}\}$. The independent model can be thought of as a model that considers each user as a seperate, independent community. By comparing the model M_C with the independent model, it is possible to quantify the additional value of community C. The size of each community $C \in M_I$ is exactly one. Because of this, all the errors related to a vertex $v_i \in C$ with $C \in M_I$ are *between errors*, the size of $Err_B(v_i)$ is equal to the number of neighbors of v_i.

Definition 9 (Community gain)**.** The community gain $CG(C)$ of community C is defined as

$$CG(C) = \sum_{v_i \in C} (L(\{v_i\}|M_I)) - L(C|M_C)$$

.

Example As an example we look at the cyan community $C = \{v_2, v_5, v_6, v_8\}$ in model M_5, shown in Figure 4.2e. In this example the model M_5 contains at least community C. In model M_2, shown in Figure 4.2b we see the independent model.

$$\begin{aligned}
M_I &= \{\{v_1\}, \{v_2\}, \{v_3\}, \{v_4\}, \{v_5\}, \{v_6\}, \{v_7\}, \{v_8\}, \{v_9\}\} \\
M_5 &= \{\{v_1, v_3, v_4\}, \{v_2, v_5, v_6, v_8\}, \{v_7, v_9\}\} \\
C &= \{v_2, v_5, v_6, v_8\} \\
CG(C) &= \sum_{v_i \in C} (L(\{v_i\}|M_I)) - L(C|M_5) \\
encode(C|M_I) &= \bigcup_{v_i \in C} \{v_i, \{neighbors(v_i)\}\} \\
&= \{2\{|3, 6, 8\}, 5\{|4, 6, 8\}, 6\{|1, 2, 4, 5, 8\}, 8\{|2, 5, 6, 7, 9\}\} \\
L(C|M_I) &= 20 \\
encode(C|M_5) &= \{2\{5|3\}, 5\{4|\}, 6\{1, 4|\}, 8\{7, 9|\}\} \\
L(C|M_5) &= 11 \\
CG(C) &= L(C|M_I) - L(C|M_5) \\
CG(C) &= 9
\end{aligned}$$

We will use the *community gain* as quality measure φ as specified in Definition 4, that is $\varphi(C) = CG(C)$. $\varphi(C)$ is a function $\varphi : 2^V \to \mathbb{Z}$ as community gain can have both positive and negative values.

4.2 Description Space

For the description space we should have a measure to give a penalty based on the complexity of the description as defined in Definition 5. Before we start with the quality measures themselves we recall the description query language given in Section 2.1. A query Q is a disjunction of conjunctions of attributes. For example:

$$Q_C = (\{a^1, a^2\} \wedge \overline{\{a^3\}}) \vee (\{a^1\} \wedge \overline{\{a^4\}} \wedge \overline{\{a^5\}})$$

In this query Q we can see that there are 5 different attributes involved, attributes a^1 till a^5 and that it consists of two patterns: $p_1 = \{a^1, a^2\} \wedge \overline{\{a^3\}}$ and $p_2 = \overline{\{a^4\}} \wedge \overline{\{a^5\}}$. Three possible ways of measuring the complexity of description query Q could be:

- Count the number of patterns. That is $|Q|$, that would be 2.
- Count the total length of the query. That is $\sum_{p_i \in Q} |p_i|$. That would be 6.
- Count the number of attributes involved in the query. That is $|\bigcup_{p_i \in Q}|$, that would be 5.

Number of patterns It could be useful to look at the number of patterns in a query. Each pattern is a conjunction, which means that all used attributes are present in the records selected by a pattern. The other way around, there is no requirement for different patterns in one query to be similar. Because of that, a query with less patterns is much easier to interpret by humans.

Length of the query Another possibility is to look at the total length of the query. A longer query is more difficult to interpret is most cases. But it really depends on why the query is long. It could be because there are many different attributes used in the query, or it could be the case that there are many very similar records with only few different attributes.

Number of used attributes Both of the methods mentioned before have a big disadvantage in certain situations. How useful they are is very dependent of the situation where they are used. A solution to get the best of both methods is to look at the different number of attributes used in the query. This is does not depend on the number of patterns, or on the total query length. Each attribute used in the query contains information about users being or not being present in the corresponding community. The more attributes are needed to describe the community, the more complex the description is. That is why we use $|\bigcup_{p_i \in Q} p_i|$ as a measure for the complexity of description query Q.

$$\rho(Q) = |\bigcup_{p_i \in Q} p_i|$$

$\rho(Q)$ is the number of attributes involved in the query. That means $\rho : \mathbb{Q}_A \to [1, |A|]$, since we do not allow empty descriptions.

4.3 Community Score

The problem stated in Section 3.1 we mention the objective function $\frac{\varphi(C)}{\rho(Q_C)}$. Our goal is to find a score that is a combination between the connectedness of community C in the graph, and the ability to describe this community with a compact query Q_C. For the connectedness we defined the quality measure $\varphi(C)$, a better score means better properties for this community in the graph. For the description query we defined a penalty $\rho(C)$, a higher score means a more complex query. We defined the total community score for community C with description query Q_C as:

$$score(C) = \frac{\varphi(C)}{\rho(Q_C)}$$

Chapter 5

Algorithms

The algorithm to solve our problem we will use is a strategy that is based on alternating between finding exceptional models and minimizing the description complexity. We modeled our problem as an instance of exceptional model mining [9]. One strategy to solve this problem is an algorithm called *Exception Maximization Description Minimization*, as proposed in [14]. This algorithm does exactly what we want to do.

5.1 Exception Maximization Description Minimization

The Exception Maximization Description Minimization (EMDM) is an iterative algorithm which alternates between maximizing exceptionality and minimizing the description complexity. The algorithm starts with a candidate set of communities. Each candidate is processed *one by one*. When a candidate is processed, the next one starts. In each iteration it tries to improve exceptionality and minimize description complexity. Improving exceptionality in this context is getting a better score $\varphi(C)$, and minimizing the description complexity means reducing the complexity score $\rho(Q_C)$. After a while the community can stabilize or walk to a situation that is explored before. When the iterative part is finished, some post processing is done. The algorithm is given in Algorithm 1. In each iteration the steps are:

- Exception Maximization (EM): try to improve the exceptionality of the community.
- Description Minimization (DM): try to reduce the complexity of the description for this community.

These EMDM steps are repeated until a certain stop criterion is reached. EMDM terminates when the community is back in the same state as in an earlier iteration. That prevents the algorithm from doing the same steps again and again. We will go into more detail about generating candidates, and about both the EM and DM steps of the algorithm in the next sections of this chapter.

We have chosen to use a local search algorithm to find the communities, because the data sets can be very large. Finding the optimal solution is not possible with these amounts of data. To demonstrate what sizes the data sets have you can have a quick look on Table 6.1 on 27. Another advantage of this algorithm is that is possible to start with a given community C (a set of users) or a community description query Q_C as input of the algorithm. When the latter is done, $g(Q_C)$ is used as input candidate for the algorithm. We will go into more details about this in the next sections.

The algorithm always ends with an DM step, the community gain $CG(C)$ of the final community C is the CG value after applying the last description query Q_C found in the DM step.

Algorithm 1 Exception Maximization Description Minimization

Input: A database DB, a set of candidate communities $\mathcal{C}$
Output: Output: A set R containing the exceptional communities
1: $R \leftarrow \emptyset$
2: **for all** $C \in \mathcal{C}$ **do**
3: **while** stop criterion not reached **do**
4: $C \leftarrow$ ExceptionMaximization(C)
5: $C \leftarrow$ DescriptionMinimization(C)
6: **end while**
7: $R \leftarrow R \cup \{C\}$
8: **end for**
9: **return** R

5.1.1 Stop Criterion

The EMDM algorithm iterates between the EM and DM step. In most communities there is no description query that exactly selected the same vertices as the EM step did. Without a stop criterion the community would alternate between the selected vertices by the description query in the DM step, and the result of the hill climbing done in the EM step. Also situations where there is a never ending cycle every $n > 1$ EMDM iterations are possible. At least, there is no guarantee on convergence. That is why we introduced two things to guarantee termination. The stop criterion consists of two conditions

- Saving history: When a candidate is back in a state seen before there is no reason to go on, as the algorithm is deterministic.
- Maximum iterations: Stop after a fixed, predefined number of iterations.

Saving History

A community consists of a set of vertices C and a description query Q_C describing them. Before the EM step starts, and after every hill climbing step a very simple representation C_S of the community C is stored in to a set of history states H, but only if it is not already present in H. If C_S is already in H it means this state was seen before in an earlier iteration, or another candidate, and there is no reason to go on with improving this candidate. When the candidate is seen in an previous candate it will be rejected and will not be present in the set of final communities.

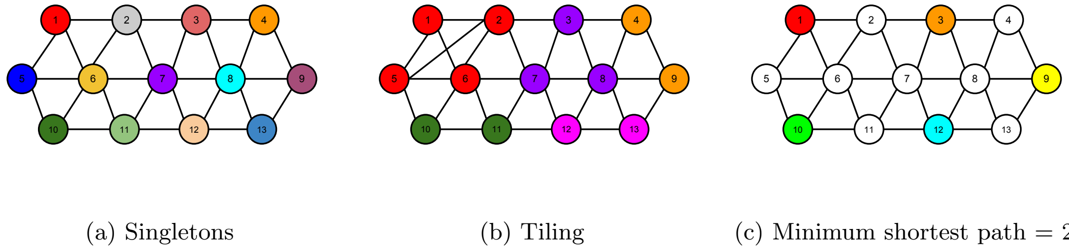
Maximum Iterations

After a predefined maximum number of iterations $maxIterations$ the EMDM algorithm always stops. The result is the community as created after the $maxIterations^{th}$ DM step. The disadvantage of this criterion is that it is a very unbalanced way to stop the communities. We only want to use this criterion if nothing else works within a reasonable number of iterations.

5.2 Generating Candidates

The EMDM algorithm tries to improve a given set of candidate communities $\mathcal{C}$. There are different ways of generating candidates, a good algorithm for generating candidates satisfies the following four conditions:

Number Each candidate is used as input for the EMDM algorithm. Running this algorithm on a candidate takes a while. If possible we want to reduce the number of candidates, without worsening the result.



(a) Singletons (b) Tiling (c) Minimum shortest path = 2

Figure 5.1: Three different ways of generating candidates

Speed The algorithm needs to be fast. It is only used for finding candidates, not for finding perfect communities. The real part of finding good communities is done by the hill climbing algorithm, after the candidates are generated. We don't want to waste too much time in finding candidates.

Distribution The generated candidates must be evenly distributed over the graph. The given quality measures in the previous section result in strong preference for certain communities. When candidates are close to each other they are very likely to end up in the same community. To prevent this, and reducing the risk of missing communities we want the candidates to be distributed evenly over the graph. When the graph consists of more than one component, we want at least one candidate on each component of the graph.

Connectedness The conditions specified in Section 4 show that the number of Err_W within a community C is very important for the community gain score $CG(C)$. In a good community C there should be at least a path from $v_1 \sim^* v_2 \forall_{v_1,v_2} \in C$. At least we do not want to have candidates with a negative value $CG(C)$.

We did experiments with four different ways of generating candidates.

5.2.1 Singletons

A very trivial way of generating candidates is creating a community for each vertex $v \in V$. This is very fast and the candidates are evenly spread over the graph. A disadvantage is that it generates many candidates. An example of a candidate set generated with this algorithm is given in Figure 5.1a. Each colored vertex in Figure 5.1a is a different community.

5.2.2 Using Tiles

Another way of generating candidates is to use tiling. A tile in the adjacency matrix is a clique in the graph. The tiling algorithm is applied on the adjacency matrix of graph $G(V, E, A)$. A clique has optimal connectedness within the community, but it could have many outgoing edges. Another disadvantage of using tiles as input is that finding them is an expensive operation. There is also no guarantee that all candidates are evenly spread over the graph, unless we allow tiles of size 1, and thus implicit insert all, or some, leftovers as singletons. The algorithm is given in Algorithm 2 and the result of running the tiling algorithm is shown in Figure 5.1b.

5.2.3 Minimum Shortest Path

The third method for generating candidates tries to reduce the number of candidates without losing the guarantee that all candidates are evenly distributed over the graph. It will create communities $C \in \mathcal{C}$ with size $|C| = 1$, just like the singletons algorithm. As an extra constraint to the singletons, there is a minimum distance $minDist$ of the shortest path in the graph $G(V, E, A)$ between each candidate. A minimum distance $minDist = n$ means the shortest path between two

Algorithm 2 Based on Tiling

Input: A set of vertices V in graph $G(V, E)$
Output: Output: S, a set of candidate communities
1: $S \leftarrow \emptyset$
2: **while** $V \neq \emptyset$ **do**
3: $A \leftarrow$ generateAdjacencyMatrix($G(V, E)$)
4: $C \leftarrow$ findVerticesInBiggestTile(A)
5: $V \leftarrow V \setminus C$
6: $S \leftarrow S \cup \{C\}$
7: **end while**
8: **return** S

candidates $C \in \mathcal{C}$ consists of at least n edges. An example of a candidate set generated with this algorithm and $minDist = 2$ is shown in Figure 5.1c.

Algorithm 3 Minimum Shortest Path

Input: A set of vertices $v_i \in V$ in graph $G(V, E)$
Input: A minimum shortest path length $minDist$ between each candidate
Output: Output: C, a set of candidate communities
1: $C \leftarrow \emptyset$
2: $S \leftarrow \emptyset$
3: **for** $i = 0 \rightarrow |V|$ **do**
4: **if** $v_i \notin S$ **then**
5: $C \leftarrow \{v_i\} \cup C$
6: $N \leftarrow \{v_i\}$
7: **for** $d = 0 \rightarrow minDist$ **do**
8: **for** $\forall u \in N$ **do**
9: $N \leftarrow N \cup neighbours(u, V)$
10: **end for**
11: **end for**
12: $S \leftarrow S \cup N$
13: **end if**
14: **end for**
15: **return** C

When the graph is not a connected graph, the distance between two vertices in different components is defined as ∞. This algorithm has the advantage of having at least one candidate on each component.

5.2.4 Based on Description

We introduced three algorithms to generate candidate communities based on the graph $G(V, E)$ as input for the EMDM algorithm. A very nice property of the EMDM algorithm is that it is both possible to start with the EM step and with the DM step. This means that it is also possible to start with one or more descriptions $\mathcal{Q} \subset \mathbb{Q}_A$ of a community as given in Definition 2. These community descriptions are used to select candidates.

Based on these descriptions $\mathcal{Q} \subset \mathbb{Q}_A$ each candidate $C \in \mathcal{C}$ is generated as:

$$\mathcal{C} = \bigcup_{Q_C \in \mathcal{Q}} \{\{g(Q)\}\}$$

5.3 Exception Maximization

In the EM step in the algorithm the set of vertices $v_i \in C$ should be adjusted in such a way that the exceptionality of community C increases, i.e. the resulting community has a better $CG(C)$ value. The EM algorithm we will use is a hill climbing algorithm. We chose for a local search algorithm because the graphs used in social networks are quite large. There is also no advantage of using a slower, global search algorithm. Our goal is to find good communities, with overlap. The fact that we judge each community individually makes also that it is possible to find communities in parallel processes. The EM algorithm we use starts with a candidate community C, and only modifies it with operator $\oplus(C)$ when that operator modifies C on such a way that it will have a higher $CG(C)$ score afterwards.

Definition 10 (Add)**.** Vertex $v \in \overline{C}$ can be added to community C with the *add*(C, v) operator.

$$ADD(C, v) = C \cup \{v\}$$

Definition 11 (Remove)**.** Vertex $v \in C$ can be removed from community C with the *remove*(C, v) operator.

$$REMOVE(C, v) = C \setminus \{v\}$$

5.3.1 Conditions and Consequences

In this section we will discuss the consequences of using the operators and find out when it is useful to apply each operator. In general applying an operator $\oplus$ to community C is useful if this causes an increase in community gain, i.e. $CG(\oplus(C)) > CG(C)$. By looking at the properties of M_I and M_C we will find out the influence of applying $\oplus$ on the community gain. The community gain $CG(C)$ is based in the difference on the number of errors under the independent model and the community model: $CG(C) = L(C|M_I) - L(C|M_C)$.

Before explaining when we have to choose which operator we recall the two types of errors used in the models given in Section 4.1. There are *within errors* Err_W for edges missing within the community, and *between errors* for edges between different communities. The choice for the operator heavily depends on the type of errors present in the community.

5.3.2 Add Operator

If vertex $u \in \overline{C}$ is added to community C the following things will happen:

Encoding $C \cup \{u\}$ with M_I For the encoded size $L(C \cup \{u\}|M_I)$ nothing will change the number of errors for vertices $v_i \in C$ because every vertex already has is own community. For vertex u new errors are introduced. This number of errors is equal to the number of edges associated with vertex u. This is easy to verify with Definition 8.

$$L(C \cup \{u\}|M_I) = L(C|M_I) + L(\{u\}|M_I) \tag{5.1}$$

Encoding $C \cup \{u\}$ with M_C For the encoded size $L(C \cup \{u\}|M_C)$ more things will happen with the encoded version of community $C \cup \{u\}$:

- New Err_B errors will occur for errors between vertex u and vertices $v \in \overline{C}$. This has size $|Err_B(u)|$.
- New Err_W errors between vertices $v \in C$ and vertex u with $(u, v) \notin E$. These errors are only counted in one of the vertices v and u. This has size $|Err_W(u)|$. In this context $|Err_W(u)|$ is the number of Err_W errors if u were in community C, this is the total number of edges between u and vertices $v \in C$.

- Type$_B$ errors between vertices $v \in C$ and vertex u will disappear. These errors are the opposite of the type$_W$ errors (if u was a community member) of vertex u. This set of errors has size $|C| - |Err_W(u)|$.
- An extra integer is needed for describing vertex u itself. This has size 1.

This will give us the following equation:

$$L(C \cup \{u\}|M_C) = L(C|M_C) + |Err_B(u)| + |Err_W(u)| - (|C| - |Err_W(u)|) + 1 \quad (5.2)$$
$$L(C \cup \{u\}|M_C) = L(C|M_C) + |Err_B(u)| + |Err_W(u)| - |C| + |Err_W(u)| + 1 \quad (5.3)$$
$$L(C \cup \{u\}|M_C) = L(C|M_C) + |Err_B(u)| + 2 \times |Err_W(u)| - |C| + 1 \quad (5.4)$$

Condition Adding a vertex u to community C is useful if $CG(C \cup \{u\}) > CG(C)$, and because $CG(C) = L(C|M_I) - L(C|M_C)$ that means $L(C|M_I)$ should increase with a higher value then $L(C|M_I)$ does. $L(\{u\}|M_I)$ is $degree(u) + 1$, we can calculate the degree as $degree(u) = |C| - |Err_W(u)| + |Err_B(u)|)$. From equation 5.1 and 5.4 can be read that this gives us:

$$L(\{u\}|M_I) > |Err_B(u)| + 2 \times |Err_W(u)| - |C| + 1 \quad (5.5)$$
$$|C| - |Err_W(u)| + |Err_B(u)| + 1 > |Err_B(u)| + 2 \times |Err_W(u)| - |C| + 1 \quad (5.6)$$
$$2 \times |C| - |Err_W(u)| + |Err_B(u)| > |Err_B(u)| + 2 \times |Err_W(u)| \quad (5.7)$$
$$2 \times |C| + > 3 \times |Err_W(u)| \quad (5.8)$$

Now there is a formal condition that states when it is useful to add a vertex u to community C. The increase of the community gain of adding u to C is $\Delta CG(add(C, u)) = 2 \times |C| - 3 \times |Err_W(u)|$.

5.3.3 Remove Operator

In some cases it is useful to remove a vertex v from a community C because the community gain $CG(C)$ improves. The formal definition for the remove operator is given in Definition 11.

Encoding $C \setminus \{u\}$ with M_I For the encoded size $L(C \setminus \{u\}|M_I)$ nothing will change the number of errors for vertices $v_i \in C \setminus \{u\}$ because every vertex already had his own community. Errors related to vertex u will disappear from $encode(C \setminus \{u\}|M_I)$. This is easy to verify with Definition 8.

$$L(C \setminus \{u\}|M_I) = L(C|M_I) - L(\{u\}|M_I) \quad (5.9)$$

Encoding $C \setminus \{u\}$ with M_C For the encoded size $L(C \setminus \{u\}|M_C)$ this will happen with the encoded version of community C. In this situation $|C|$ and $Err_X(u)$ refer to the situation where u is not yet removed from C.

- Some Err_B errors between vertex u and $v \in \overline{C}$ will disappear, because both u and v are not in C. This has size $|Err_B(u)|$
- Some Err_W errors between vertices $v \in C$ and vertex u with $(u, v) \notin E$ will disappear. These errors are only in one of the vertices v and u. This has size $|Err_W(u)|$.
- New Err_B errors will occur for errors between vertex u and vertices $v \in C$. This has size $|C| - |Err_W(u)| - 1$.
- One that was needed for describing vertex u itself is not needed anymore. This has size 1.

This will give us the following equation:

$$L(C \setminus \{u\}|M_C) = L(C|M_C) - |Err_B(u)| - |Err_W(u)| + (|C| - |Err_W(u)| - 1) - 1 \tag{5.10}$$
$$L(C \setminus \{u\}|M_C) = L(C|M_C) - |Err_B(u)| - |Err_W(u)| + |C| - |Err_W(u)| - 2 \tag{5.11}$$
$$L(C \setminus \{u\}|M_C) = L(C|M_C) - |Err_B(u)| - 2 \times |Err_W(u)| + |C| - 2 \tag{5.12}$$

Condition Removing a vertex u from community C is useful if $CG(C \setminus \{u\}) > CG(C)$, and because $CG(C) = L(C|M_I) - L(C|M_C)$ that means $L(C|M_I)$ should decrease with a lower value then $L(C|M_I)$ does. From equation 5.9 and 5.12 can be read that this gives:

$$-L(\{u\}|M_I) > -|Err_B(u)| - 2 \times |Err_W(u)| + |C| - 2 \tag{5.13}$$
$$L(\{u\}|M_I) < |Err_B(u)| + 2 \times |Err_W(u)| - |C| + 2 \tag{5.14}$$
$$|C| - |Err_W(u)| + |Err_B(u)| < |Err_B(u)| + 2 \times |Err_W(u)| - |C| + 2 \tag{5.15}$$
$$|C| - |Err_W(u)| < 2 \times |Err_W(u)| - |C| + 2 \tag{5.16}$$
$$2 \times |C| < 3 \times |Err_W(u)| + 2 \tag{5.17}$$

Now there is a formal condition that states when it is useful to remove a vertex u from community C. The increase of the community gain of removing u from C is $\Delta CG(REMOVE(C, u)) = 3 \times |Err_W(u)| + 2 - 2 \times |C|$.

5.3.4 Hill Climbing

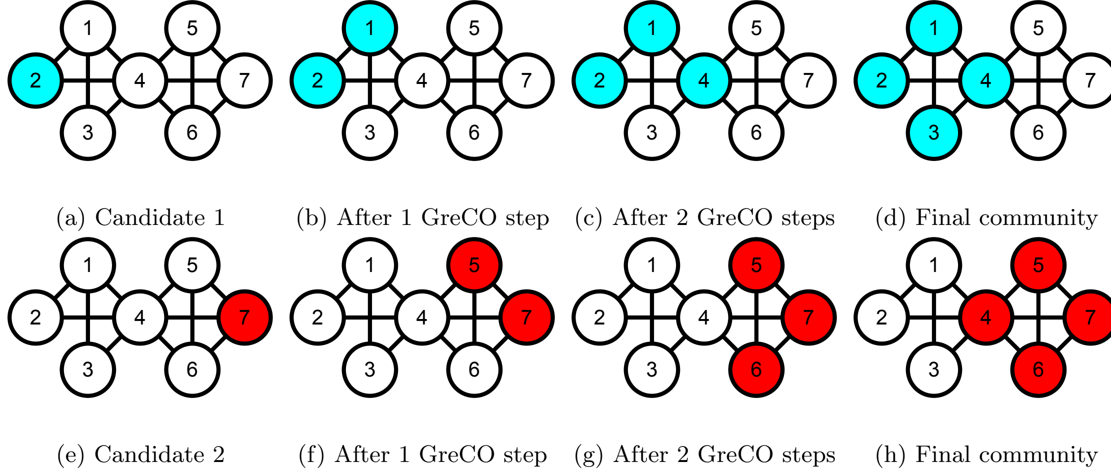
With the operations defined in the preceding section it is possible to execute a hill climbing algorithm, which optimizes a community in each step. The *Greedy Community Optimizer* (GreCO) algorithm selects a set of possible operations each GreCO step, applies the best one, and goes to the next iteration. The set of possible operations OP_C that are potentially useful is limited to $OP_C = \{\forall_{v_i \in C} REMOVE(C, v) : |C| > 1\} \cup \{\forall_{v \in \overline{C}} ADD(C, v) : \exists w \in C \wedge (v, w) \in E\}$. This is a $REMOVE$ operation for each vertex $v_i \in C$, and an ADD operation for each vertex $v_i \in neighbors(C)$. The best operation is the one with the highest increase in community gain: $\arg\max_{op \in OP_C} \Delta CG(op)$. An operation is only useful when the change in community gain is nonnegative for the add operator, and positive for the remove operator. If two communities have the same community gain, the add operator is preferred to the remove operator. We only allow the ADD operator to be applied when the community gain does not change when applying it. This gives us the guarantee that the algorithm terminates, because it always reaches a point where it is not possible anymore to grow without getting a lower community gain score.

Algorithm 4 Greedy Community Optimizer (GreCO)

Input: A candidate community C and a graph $G(V, E)$
Output: The result after applying of the hill climbing algorithm.

1: **loop**
2: $\quad \mathcal{O} \leftarrow \{\forall_{v_i \in C} REMOVE(C, v) : |C| > 0\}$
3: $\quad \mathcal{O} \leftarrow \mathcal{O} \cup \{\forall_{v \in \overline{C}} ADD(C, v) : \exists w \in C \wedge (v, w) \in E\}$
4: $\quad \oplus \leftarrow \arg\max_{op \in \mathcal{O}} \Delta CG(op)$
5: $\quad$ **if** $\Delta CG(\oplus) > 0 \vee (\Delta CG(\oplus) = 0 \wedge \oplus_{type=add})$ **then**
6: $\quad\quad C \leftarrow \oplus(C)$
7: $\quad$ **else**
8: $\quad\quad$ **return** C
9: $\quad$ **end if**
10: **end loop**

Example 4. We start with showing that the EM phase of the algorithm works really well on a toy example. In Figure 5.2 a small example with a graph of 7 vertices and a set of 2 candidates consisting of only one vertex are given. In Table 5.2i some information about the set of possible operations, the community gain score, and the density of the community is given.



(a) Candidate 1 (b) After 1 GreCO step (c) After 2 GreCO steps (d) Final community

(e) Candidate 2 (f) After 1 GreCO step (g) After 2 GreCO steps (h) Final community

Fig	C	$\|C\|$	ICD	CG	*Possible operations (A = ADD, R = REMOVE)*
5.2a	$\{2\}$	1	-	0	A(1):2, A(4):2
5.2b	$\{2,1\}$	2	1	2	A(3):3, A(4):6, R(1):0, R(2):0
5.2c	$\{2,1,4\}$	3	1	6	A(3):9, A(5):6, A(6):6, A(7):6, R(2):2, R(1):2, R(4):2
5.2d	$\{2,1,4,3\}$	4	0.83	9	A(5):8, A(6):8, A(7):8, R(2):6, R(1):3, R(4):3, R(3):6
5.2e	$\{7\}$	1	-	0	A(4):2, A(5):2, A(6):2
5.2f	$\{7,5\}$	2	1	2	A(4):6, A(6):6, R(5):0, R(7):0
5.2g	$\{7,5,6\}$	3	1	6	A(4):12, R(7):2, R(5):2, R(6):2
5.2h	$\{7,5,6,4\}$	4	1	12	A(1):11, A(2):11, A(3):11, R(5):6, R(6):6, R(7):6

(i) Greedy community optimizer steps for candidates 1 and 2

Figure 5.2: Two candidates and the GreCO steps resulting in the final communities

It is easy to verify that the GreCO algorithm always selects the operation that results in the community with the highest information gain. When there is more than one operation possible resulting in the same information gain the *ADD* operation has a higher priority than the *REMOVE* operator. When choosing between two possible *ADD* operations resulting in communities with the same information gain, the vertex with the lowest index is picked.

5.4 Description Minimization

For minimization of the description for community C we create a description query Q_C. Building a query Q_C in fact is like building a classifier. A classifier is an object that is able to assign a class label y to a record, given a set of values for the description attributes. In our project we tried many different classifiers (see also Section 7.4 on page 34), but we chose for descriptive queries.

In Section 2.2 we introduced exceptional models. We explained that our target attributes Y are the users themselves, and the values of attributes Y are the adjacency matrix. The model we build is based on the community found by the GreCO algorithm. This model reduces target attributes Y (adjacency matrix) to a single binary value y. The value of y depends on being, or not being, a member of C. Class label $y_i = 1$ means individual $v_i \in C$, class label $y_i = 0$ means

$v_i \in \overline{C}$. To get this description we use the ReMine [17] algorithm. We most accurate description query in terms of precision and recall. When two queries perform equally accurate, the shortest one is chosen. It is important to note that the values of y are different for different communities.

After we found the description query community C is updated to $g(Q_C)$.

Algorithm 5 DM Step

Input: A database DB with attributes A, and a community $C \subseteq V$
Output: An updated community C and a description query Q_C describing C
1: $Q_C \leftarrow findBestDescriptionQuery(DB, C)$
2: $C \leftarrow g(Q_C)$
3: **return** (C, Q_C)

5.4.1 ReMine

For finding the description query we use an algorithm inspired on the ReMine algorithm[17]. Let us start with an informal introduction to ReMine. ReMine is an iterative algorithm with two steps in each iteration. The algorithm starts with all records $v_i \in DB$ in one big partition $\mathcal{P} = \{DB\}$. In the first step it tries to find the best split patterns $f_i \in F_{new}$ in each partition $P_i \in \mathcal{P}$. Information gain is used as quality measure to find out which one is the best. After finding a single pattern in each partition, it merges all the records back together and goes to the second step. The second step is to split the database on all combinations of having, and not having each pattern $f_i \in F \cup F_{new}$ found in this and earlier iterations. A new, larger set of partitions is obtained by doing this and then it starts with the next iteration. In each next iteration it finds one good pattern in each partition, and splits the database on all combinations of having and not having each pattern.

The original ReMine algorithm stops when the new partitioning found is the same as the partitioning in the previous iteration. This can take very long on the large data sets of our project. That is why we stop when a certain number of iterations *maxIterations* is reached.

The *bestSplit* method tries to split a partition on a split with the highest information. That means splitting in such a way that one of the partitions P_i have a high percentage of $\frac{|v \in C \wedge v \in P_i|}{P_i}$ and the other partition P_j has high percentage of $\frac{|v \in \overline{C} \wedge v \in P_j|}{|P_j|}$. I.e., the class label used for building the classifier is $y_i = 1 \iff v_i \in C$ and $y_i = 0$ otherwise.

The algorithm is shown in 6

Algorithm 6 ReMine

Input: A database DB with attributes A, and a community $C \subseteq V$
Input: The maximum number of iterations *maxIterations*
Output: A set of partition $\mathcal{P}$ with the set of features describing it
1: $F \leftarrow \emptyset$
2: $count \leftarrow 0$
3: **while** $count < maxIterations$ **do**
4: $\quad \mathcal{P} \leftarrow split(DB, F)$
5: $\quad F_{new} \leftarrow \emptyset$
6: $\quad$ **for all** partition $P_i \in \mathcal{P}$ **do**
7: $\quad\quad F_{new} \leftarrow F_{new} \cup bestSplit(P_i, C)$
8: $\quad$ **end for**
9: $\quad F \leftarrow F_{new} \cup F_{new}$
10: $\quad \mathcal{P} \leftarrow split(DB, F)$
11: $\quad count \leftarrow count + 1$
12: **end while**
13: **return** $(\mathcal{P}, F)$;

The ReMine algorithm finds a set of partitions $\mathcal{P}$ is found. Each partition $P \in \mathcal{P}$ is described

by a conjunction of having or not having attributes F_P. The class label of the majority class is assigned to each partition. All partitions $P \in \mathcal{P} | P_{classlabel} = 1$ are partitions that consist of a majority of records being a member of community C.

Example 5. An example of partitions found by the ReMine algorithm on a hypothetical database is given in Table 5.1. The description query Q_C to describe this community would be

$$Q_C = (\{a_1, a_3, a_6\} \wedge \overline{\{a_4\}} \wedge \{a_5, a_2\}) \vee (\overline{\{a_1, a_3, a_6\}} \wedge \{a_4\} \wedge \overline{\{a_5, a_2\}}) \vee (\overline{\{a_1, a_3, a_6\}} \wedge \overline{\{a_4\}} \wedge \{a_5, a_2\})$$

$P \in \mathcal{P}$	$f_1 = a_1 \wedge a_3 \wedge a_6$	$f_2 = a_4$	$f_3 = a_5 \wedge a_2$	$\|P\|$	$\|r_i \in C\|$	$\|r_i \in \overline{C}\|$	$P_{classlabel}$
P_1	1	1	1	21	4	17	0
P_2	1	1	0	0	0	0	N/A
P_3	1	0	1	5	4	1	1
P_4	1	0	0	17	4	13	0
P_5	0	1	1	18	18	0	0
P_6	0	1	0	31	29	2	1
P_7	0	0	1	21	17	4	1
P_8	0	0	0	49	2	47	0

Table 5.1: Result of a run of the ReMine algorithm

The final classifier tests whether a record $v_i \in DB$ has all features F_P of one or more of the partitions with class label 1. If so, than record v_i is selected by the algorithm as a member of community C. If it does not contain all of the features $f \in F_P$ then it is not a member of C. The final classifier has the form as desribed in Chapter 3.

BestSplit

The original ReMine uses FP-trees [7] to find the best split pattern. Our data sets are much larger in both terms of number of attributes and number of records than the data sets they use. That is why we used a beam search to find the best split pattern in each partition. This approach doesn't give us necessarily the optimal pattern, but with setting the parameters *beamLength* and the maximum pattern length *maxPatternLength* we can balance it between accuracy and calculation time. The *bestSplit* procedure is shown in Algorithm 7.

Algorithm 7 bestSplit

Input: A set of records $P \subseteq DB$ and a community $C \subseteq V$
Input: The configuration parameters $maxPatternLength$ and $beamLength$
Output: A pattern f_{split} to split on and two new partitions P_1, P_2.

```
1: B ← {(∅, 0)} list (beam) of tuples (patterns, information gain) starting with empty pattern
2: length ← 0
3: while length < maxPatternLength do
4:    for all patterns f ∈ B do
5:       B_new ← ∅
6:       for all attribute a_j ∈ A \ f do
7:          f_new = f ∪ {a_j}
8:          ig ← informationGain(P, DB, C, f_new)  − information gain when splitting on f_new
9:          ADD(B_new, (f_new, ig));
10:      end for
11:      B ← B ∪ B_new
12:      sort(B)  − according to information gain when splitting on pattern
13:      B ← TAKE(B, beamLength)
14:   end for
15: end while
16: return  (B[0], v_i ∈ DB ⊨ B[0], v_i ∈ DB ⊭ B[0]);
```

Chapter 6

Data

Data used in the experiments consist of two parts, the description part and the graph part. For the graph part we use data from real applications in all experiments. For the description part we use both data retrieved from real world social networks and data constructed with CFinder[15]. We will give some details about the data sets and the way we constructed the data.

Data set	*Source tag data*	$\|Tags\|$	*Tag density*	$\|V\|$	$\|E\|$	$\frac{\|E\|}{\|V\|^2}$
Delicious	Real data set	1350	0,0389	1861	7664	0,00221
Flickr S	Real data set	2791	0,0298	100267	3781947	0,00037
Flickr M	Real data set	7565	0,0136	100267	3781947	0,00037
Flickr L	Real data set	16215	0,0072	100267	3781947	0,00037
Flixster	CFinder	5844	0,0252	25000	221625	0,00035
Friendster	CFinder	8088	0,0252	32768	267528	0,00024
Last FM	Real data set	11946	0,0015	1892	12717	0,00355

Table 6.1: Data sets used in experiments

6.1 Real World Data

Data sets Flickr, Last FM and Delicious are data sets where both the graph part and the description part are retrieved from online social networks. In all three sets the vertices in the graph represent users that have an account on the corresponding websites. The description part consists of tags used by the corresponding users.

6.1.1 Last FM

On www.last.fm users can connect themselves with other users by adding them as friend. The graph defined by this friendship links is used as the graph data in this data set. Users can play music and a list of played songs and artist is kept in the database. Each artist is associated with one or more specific music styles, like song for peace, piano, death metal or pop. The description data is a matrix with music styles as attributes, $A = \{$song for peace, piano, death metal, pop, $\ldots\}$. The description matrix has attribute $x_i^j = 1$ when user v_i has played a song of an artist associated with music style $a_j \in A$ and $x_i^j = 0$ otherwise.

6.1.2 Delicious

Delicious is a website where users can share their bookmarks. Each user has a contact list, in this list they can save the users from which they like the bookmarks. We used the contact lists of a set

of users as the input for constructing the graph. Users can categorize their bookmarks by adding tags to a bookmark. The description data is a matrix with these tags as attributes, $A = \{$search engines, travel, bookstores, $\ldots\}$. The description matrix has attribute $x_i^j = 1$ when user r_i has tagged a bookmark with tag $a_j \in A$ and $x_i^j = 0$ otherwise.

6.1.3 Flickr

The Flickr data set is preprocessed into three different data sets. For each data set the graph is the same, it is constructed from the explicit social network Flickr is. For the description space the tags users assigned to photos are used. The set of description attributes could be $A = \{$waterfall, building, Nikon, pretty woman, $\ldots\}$. The problem with this data set is that there are so many tags that it is not useful to use them all. Tags that are used very often probably not tell something specific about a set of users, e.g. *Nikon*. On the other hand, tags that are use only few times, like *me on the moon* are also not very useful. We ordered the tags by the frequency that they occur in the database. We first removed the top 5% tags.

After that we wanted to have three different sets of tags, to see the difference in performance with different sets of tags. That is why created three different data sets. We allowed tags with a frequency of at least: 100, 250 and 500. This resulted in three sets of attributes of different sizes: $attributes(Flickr\ S) \subset attributes(Flickr\ M) \subset attributes(Flickr\ L)$. The description matrix has attribute $x_i^j = 1$ when user v_i has tagged a photo with tag $a_j \in A$ and $x_i^j = 0$ otherwise.

6.2 Synthetic Data

To compare the results of our quality measure with other methods of finding communities in graphs we also created description data with CFinder. CFinder is a tool to find communities using the Clique Percolation Method [2]. This is a well-known method for finding communities. Creating description data based on the communities found with CFinder should result in compact descriptions, because the description data is based on community stuctures.

As input networks in CFinder we used several graphs from real social networks. We used CFinder to find a set of communities $\mathcal{C}$ in these graph. We filtered the set of communities found by CFinder by requiring a minimum size for each community. The minimum size we used is different for each set. Our goal was to have the number of attributes $\frac{|V|}{8} < |A| < \frac{|V|}{2}$.

We used each community $C \in \mathcal{C}$ as an attribute a_i in the attribute space A. Each community $C^j \in \mathcal{C}$ found by CFinder is seen as an attribute in the generated description matrix A. The description matrix has attribute $x_i^j = 1$ when user v_i is a member of community C^j found by CFinder, and $x_i^j = 0$ otherwise.

After generating the attributes based on the communities found by CFinder $4 \times |\mathcal{C}|$ extra attributes are added to the description matrix A as noise. The values of these attributes are initially 0 for all records. Now we have the total set of attributes, we only need to add some noise to make the data more realistic. That is done by flipping each value x_i^j in the description matrix from 0 to 1, or from 1 tot 0 with a chance of 0.025. We chose for 0.025 because that is the average density of the real data sets.

Chapter 7

Experiments

In previous sections we defined the problem, discussed related topics, listed down the algorithms and described the data. These are all the tools we need to have before start running experiments. In this section we show step by step that the algorithm works and what results we can get.

7.1 Setup

7.1.1 Hardware

All experiments are done with a Windows 7 64 bit Professional operating system. The system contains an Intel i7 Q740 processor. This processor has 4 cores and supports hyper threading. The system has 12 GB internal memory. For all the experiments we used 8 threads at the same time.

7.1.2 Software Implementation

We created our own software package and data format to run all the experiments. The software we created for doing the experiments is written in C# with .NET 4.0. Both for the graph data, and the description data we used a matrix implementation. Each row vector in the matrices is a bit array. These are arrays of integers with each bit representing an entry in the matrix. After the generation of candidates the EMDM iterations are done in multiple threads. A shared HashMap used as object to store the previous states.

7.2 Graph Space

In this subsection we present experiments in the graph space only. That means we only consider the EM step in only one operation of the EMDM algorithm. The EM algorithm given in Algorithm 4 is a hill climbing algorithm. Given a candidate, i.e. a set of vertices, it starts improving it till the community is in a local optimum. This section is about the steps from the candidate to the local optimum.

7.2.1 EM Step (GreCO)

In Section 5.3 we already showed that the GreCO algorithm we use will always terminate. We start with an example, on a small subset of the graph part of the Delicious data set. We took the first 250 vertices from the original Delicious graph, and all 1426 edges that are between these vertices. The only reason to reduce the number of vertices is just because it is not possible to visualize all the 1861 nodes and distinguish the colors of the resulting communities.

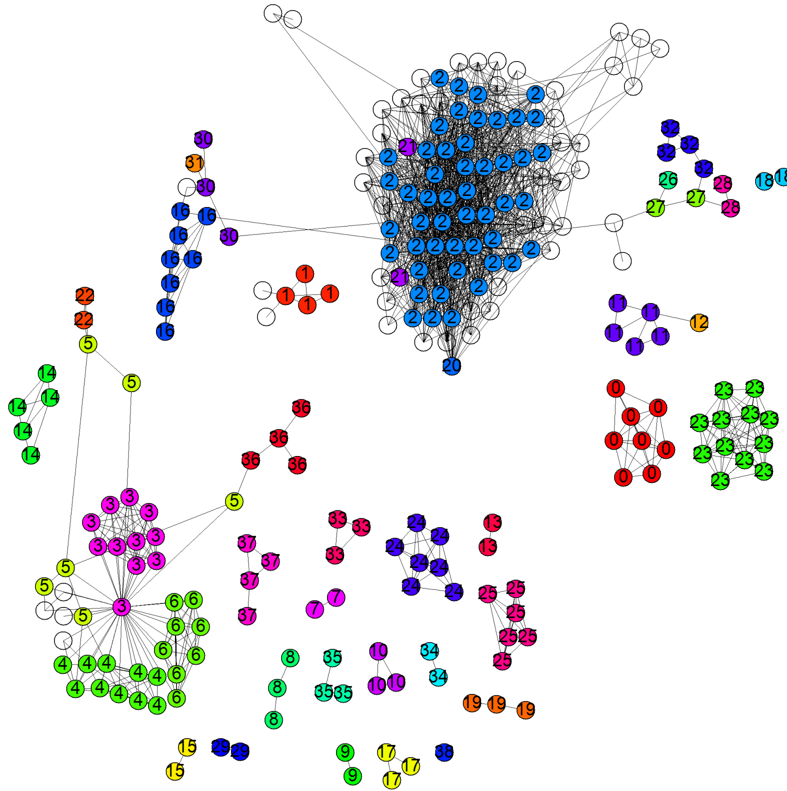


Figure 7.1: Resulting communities in the first 250 vertices from the Delicious data set

In Figure 7.1 we show the resulting communities of applying the EM steps on the reduced Delicious data set. Each community found has different color and a different number. It is easy to verify that the communities found by the algorithm make sense. Some vertices were part of more than one community. In that case the vertex is colored with the color of the community with the highest community gain. The white vertices are not part of any community.

In the next subsection we go more in to detail to show that our quality measure makes sense. We will do this by comparing them with other quality measures used in the field of community detection.

7.2.2 Community Gain

In the previous subsection we gave an indication that the EM algorithm works by just looking at one image. Now we will proof that it works by comparing it with other common quality measures. The measures we compare ours with are *inverse conductance*, *intra cluster density*, and *modularity*. For each data set we used the community with the highest community gain as example.

Inverse Conductance

The first quality measure we compare the community gain with is *inverse conductance*. This quality measure is a very simple network community score. It can be thought of as the ratio between the number of edges inside the community and the number of edges leaving the community [10]. That is:

$$Inverse\ conductance(C) = 1 - \frac{|\{(u,v) : u \in C, v \in \overline{C}\}|}{\sum_{u \in C} degree(u)}$$

A higher value for the inverse conductance means a better community. In Figure 7.2 we show that an increase of the community gain also implies an increase of the inverse conductance of the same community.

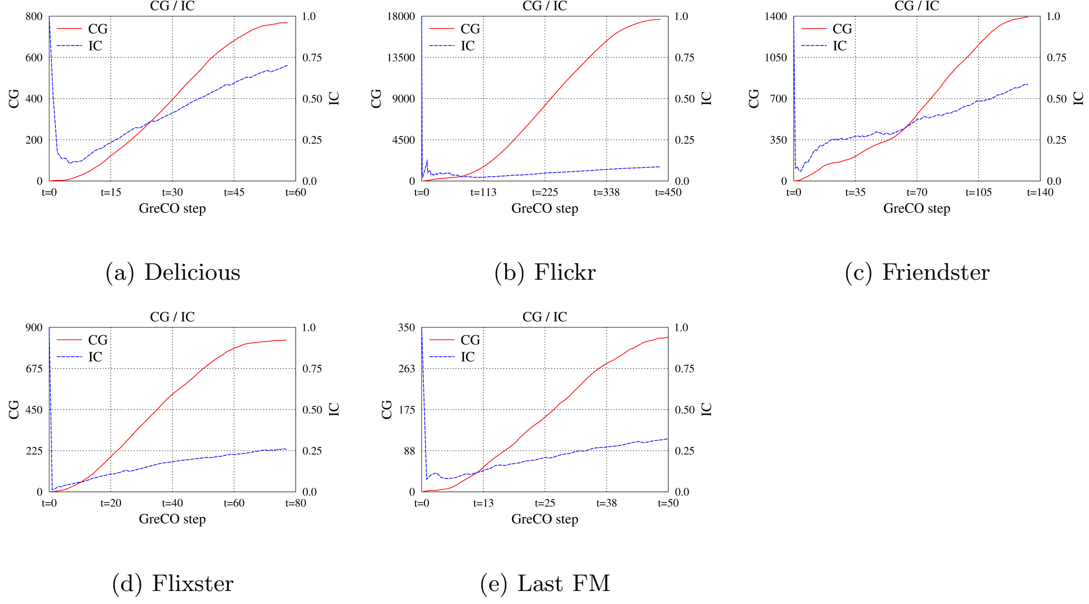


Figure 7.2: Community gain and inverse conductance in each GreCO step

Intra Cluster Density

The intra cluster density is the ratio of the number of edges in a community C, and the number of edges if the vertices $v_i \in C$ if the community were a clique. That is:

$$Intra\ cluster\ density(C) = \frac{|\{(u,v) : u \in C, v \in C, v \neq u\}|}{|C|(|C|-1)/2}$$

Figure 7.3 represents the graphs of the community gain and the intra cluster density. The intra cluster density tends to stabilize around 0.5. Compared to the density $\frac{|E|}{|V|^2}$ of the whole graphs, as shown in Table 6.1 on page 27 the intra cluster density of the communities found is much higher.

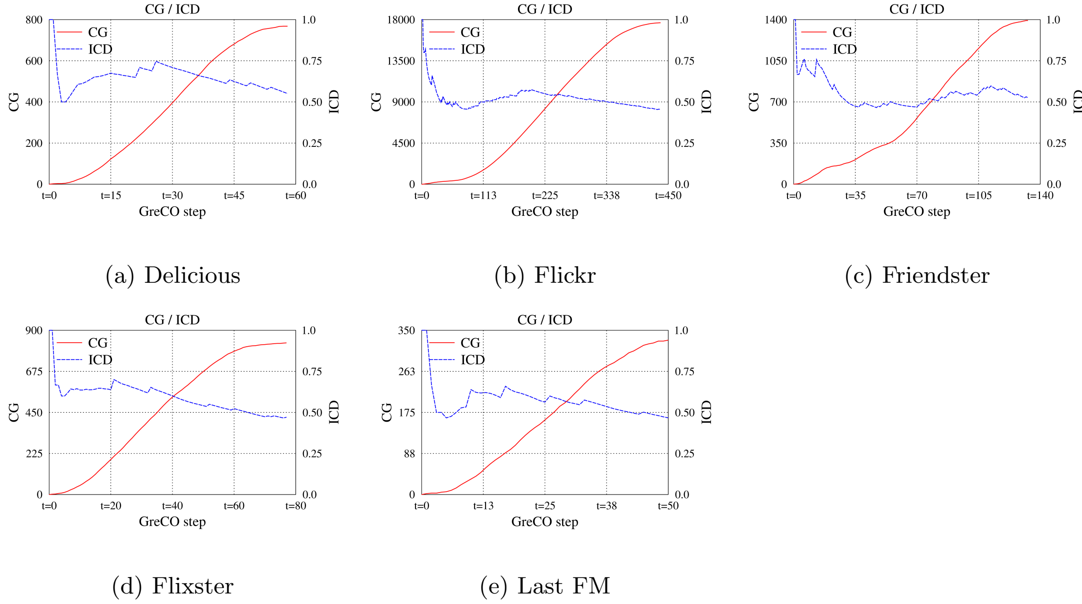


Figure 7.3: Community gain and intra community density in each GreCO step

Modularity

One of the most used measures to quantify the quality of a community is modularity. Modularity is the ration between the number of edges between the vertices of a community, and the expected number of vertices in the null model [10].

A good null model of a social network can be generated by cutting all the edges into two stubs. To generate the null model $G'(V, E')$, two random stubs are drawn from all remaining stubs and connected. This is repeated till all the stubs are connected to another one. The new graph is the null model. All vertex degrees have the same as in the original graph, but the edges are different. The modularity of a community is the ratio between the real number of edges and the number of edges in the null model. That is:

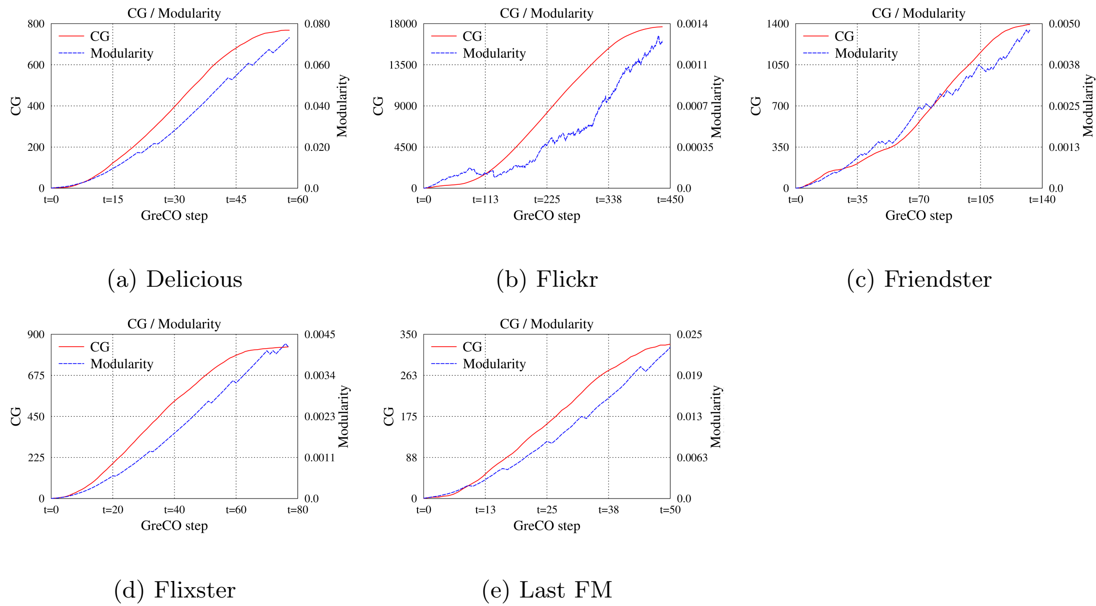
$$Modularity(C) = \frac{1}{4m}(|\{(u,v) : u, v \in C, (u,v) \in E\}| - |\{(u,v) : u, v \in C, (u,v) \in E'\}|)$$

An easier way then generating null models on this way is by using the vertex degrees and number of edges in the graph for generating the null model. That is what we used for calculating the modularity:

$$Modularity(C) = \sum_{u,v \in C \times C} (|\{(u,v)\} \cap E| - \frac{degree(u) \times degree(v)}{2 \times |E|})$$

In this formula $|\{(u,v)\} \cap E|$ is 1 if there is an edge between vertex u and vertex v, and 0 otherwise. $\frac{degree(u) \times degree(v)}{2 \times |E|}$ is the chance of the existence of a an edge between u and vertex v in the null model.

The resulting experiments displayed in Figure 7.4 show that an increase in community gain also resulted in a higher value for the modularity.



(a) Delicious (b) Flickr (c) Friendster

(d) Flixster (e) Last FM

Figure 7.4: Community gain and modularity in each GreCO step

7.3 Candidate Selection

The EMDM algorithm begins with a candidate set of communities $\mathcal{C}$. Finding a community and finding the corresponding description is a complex operation that takes much time from the CPU and memory. That is why it is important to reduce the number of candidates that we use as input of the EMDM algorithm. In Section 5.2 we proposed four algorithms.

7.3.1 Singletons

This algorithm works very well in terms of results, but generates many candidates. We investigated this algorithm but treated it as a special instance of the minimum shortest path algorithm with a minimum distance $minDist = 0$.

7.3.2 Tiling

This algorithm worked very well on the small graphs, but finding tiles in larger graphs seemed impossible within a reasonable amount of time because of the large size of the data. One of the requirements of the candidate finding algorithms was that they are fast, the goal is to reduce the total time. The tiling algorithm clearly does not meet this requirement. That is why we did not spend any time on further research.

7.3.3 Based on Description

This algorithm is useful when you want to start with a description as input. In that case speed is not a requirement, it is just another way of generating input. In fact the algorithm starts with the DM step when using this algorithm to generate the candidates.

7.3.4 Minimum Shortest Path

This algorithm, introduced in Section 5.2.3 tried to find candidates that have a given minimum shortest path distance from each other. This way different candidates are generated from different

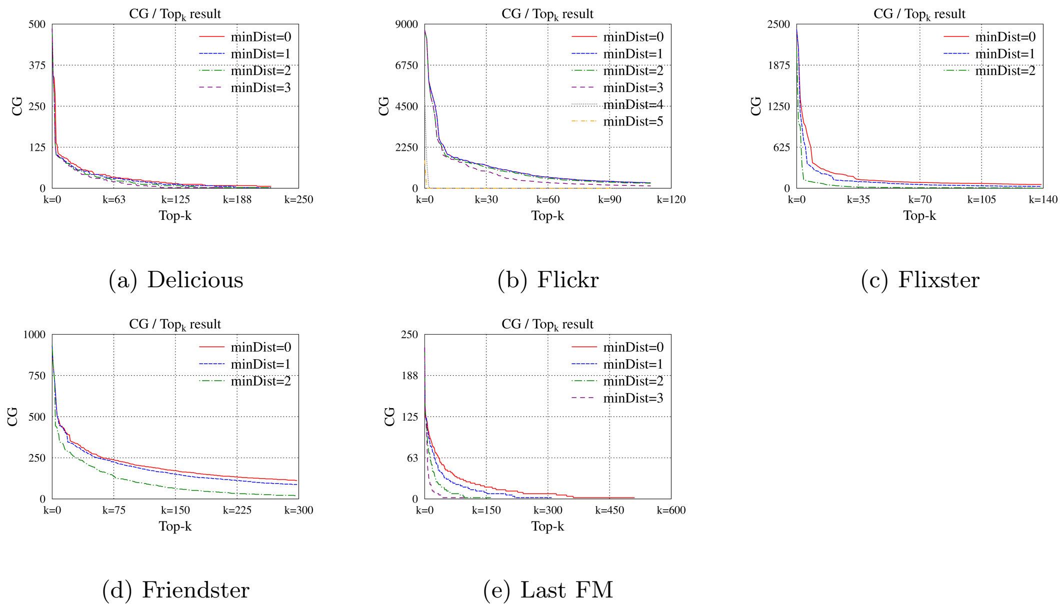
areas on the graph. We investigated the relation between the minimum distance *minDist*, the number of candidates generated, and the community gain in the top-k results.

Table 7.1 gives us the number of candidates generated with values $0 \leq minDist \leq 5$. In some data sets there was no shortest path longer than a certain *minDist* value. At that point only one candidate is left in a connected graph.

Data set	*minDist*					
	0 *(Singletons)*	1	2	3	4	5
Delicious	1861	508	267	180	1	1
Flickr	100267	31992	7214	2006	175	62
Flixster	25000	5684	208	1	1	1
Friendster	32768	6508	695	1	1	1
Last FM	1892	741	217	101	1	1

Table 7.1: Number of candidates with each *minDist* value.

Our goal was to reduce the number of candidates without reducing the community gain. We looked at the top-k communities found, ordered by community gain and then by size. The value for k is dependent on the data set. Figure 7.5 gives us the community gain at the top-k communities. It is easy to see that the community gain is going down at a higher *minDist* value. But at $minDist = 1$ the decrease is really small or almost nonexistent. In the Flickr data set the decrease in community starts at $minDist = 3$, compared to the $minDist = 0$ only 2% of the number of candidates is left. The experiments that *minDist* can often be increased without negatively affecting the community gain of the top-k, effectively making search more effective



(a) Delicious (b) Flickr (c) Flixster

(d) Friendster (e) Last FM

Figure 7.5: Number of candidates with each *minDist* value

7.4 Classifiers

The biggest problem we had was to find a suitable classifier for using in the DM step. A classifier needs to have these properties:

- It should not take too long to build them, since each candidate in each EMDM iteration a classifier needs to be build.
- It should have a good precision/recall score. Otherwise it does not describe the community found in the EM step.
- Part of the goal or our research is *characterizing communities*. So it should be possible to transform the classifier into something that is easy to interpret by humans.

We tried many different classifiers like decision trees, nearest neighbor, emerging patterns, ReMine, our own adaption of ReMine as explained in Section 5.4.1, and few others.

7.4.1 Decision Trees

Our first idea was to build a decision tree. For each attribute a^j the information gain when splitting on each attribute is calculated. This is done by first splitting the database into two partitions and then calculating the information gain.

$$
\begin{aligned}
P_{a^j=1} &= \{r_i \in DB : r_i^j = 1\} \\
P_{a^j=0} &= \{r_i \in DB : r_i^j = 0\} \\
IG(a^j, DB) &= impurity(DB) - \left(\frac{|P_{a^j=1}|}{|DB|} \times impurity(P_{a^j=1}) + \frac{|P_{a^j=0}|}{|DB|} \times impurity(P_{a^j=0})\right) \\
&\text{with} \\
impurity(P) &= \frac{|\{r_i \in P : r_i \in C\}|}{|P|} \times \frac{|\{r_i \in P : r_i \in \overline{C}\}|}{|P|}
\end{aligned}
$$

The attribute with the highest information gain is chosen as split node. All records r_i with label $r_i^j = 1$ ($P_{a^j=1}$) go to the left side of the split node, all other nodes ($P_{a^j=0}$) to the right side. After this is done this step is repeated in recursion on both partitions $P_{a^j=1}$ and $P_{a^j=0}$ until there is a three with leaf nodes with an impurity of 0 or no improvement is possible. The class label belonging to a leaf node is determined by the majority of class labels of the corresponding partition. So if the majority of the records belong to C, i.e. $|\{r_i \in P_{leafnode} : r_i \in C\}| > \frac{|P_{leafnode}|}{2}$ a leaf node has class label 1.

After the classifier is build a record r_i can be classified by entering the tree at the root and go left or right dependent of the values r_i for the split nodes. When arrived in the leaf node the class label of the leaf node is assigned to the record.

The problem was that the decision trees we either very large, for example > 2500 nodes for all of the top-50 communities from the 25000 records Flickr L subset, which is overfitting. We tried to balance this by introducing a maximum height of the decision tree, but that did not work out. We also tried changing the majority rule to determine the class labels of the leaf nodes, instead of just looking to the majority class we tried something dependent in of the ratio $\frac{|C|}{|DB|}$. When we did this the precision and recall dropped from almost 1 to 0.1. Overall was not possible to create a classifier with decision trees that satisfied all of the requirements.

7.4.2 Nearest Neighbor

For both partitions C and $\overline{C}$ a vector NN with length $|A|$ is created. This vector contains the average number of 1 values for each attribute in each partition. The vectors have the following values:

$$\forall a^j \in A: \; NN_C[j] = \frac{|\{r_i \in C : r_i^j = 1\}|}{|C|}$$
$$\forall a^j \in A: \; NN_{\overline{C}}[j] = \frac{|\{r_i \in \overline{C} : r_i^j = 1\}|}{|\overline{C}|}$$

Now a record r_i is classified by calculating the Euclidean distance between r_i and the vectors NN_C and $NN_{\overline{C}}$. If the distance to NN_C is the smallest, r_i is classified as belonging to the community, otherwise r_i is classified is not belonging to the community.

Unfortunately the precision and recall values obtained with this classifier where really bad, both below 0.25. That is not enough for this research.

7.4.3 Emerging Patterns

Another approach which looked promising was using emerging patterns. Finding a community C could be thought of as splitting the database into two partitions, C and $\overline{C}$. Emerging patterns are patterns that are frequent in one partition and not frequent in the other [3]. These properties are exactly what we want, they characterize the difference between C and $\overline{C}$.

Unfortunately finding emerging patterns is an expensive operation, and it did not work out on the large and sparse data sets we have.

7.4.4 ReMine

We started with implementing the original ReMine[17] algorithm which uses FPTrees[7]. This algorithm finds the optimal pattern to split on. Unfortunately finding optimal split patterns took too long. That is why we modified the algorithm on such a way that it does not find the optimal pattern, but a good one. This modification is explained in Section 5.4.1

With this adapted version of ReMine we were able to find a classifier that satisfies all of our requirements:

- It does not take too long to build them.
- It has a decent precision/recall value.
- By transforming it into a tag cloud it is easy to interpret them.

Dependent of the size and the density of the description data it is possible to tune the algorithm with the two parameters. A longer beam and a longer maximum pattern length increase the runtime, but also increase the precision/recall scores.

Examples of the results found with this classifier are demonstrated in the section with the final results, Section 7.5.3.

7.5 Overall Results

7.5.1 Stop Criterion EMDM

In Section 5.1.1 we discussed the stop criterion we introduced to guarantee the termination of the EMDM algorithm. We had two solutions to do this:

- Maximum iterations
- Saving history

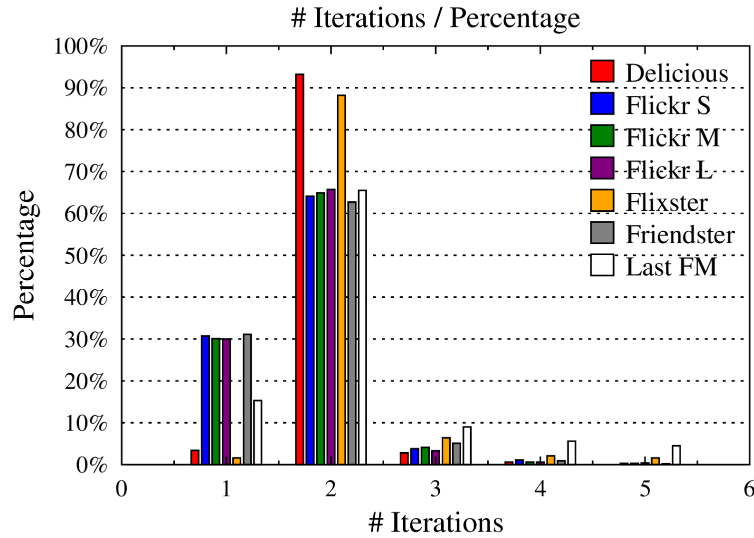


Figure 7.6: Number of iterations

The experiments showed that these methods worked really well. We chose a maximum number of iterations $maxIterations = 5$ and ran the experiments. During the experiments we counted how many iterations each candidate did before it was in a state that was seen before. It turned out that only few communities were stopped because $maxIterations$ was reached. Table 7.2 and Figure 7.6 show the number of communities that used each $1 \leq n \leq 5$ iteration before they were in a state as seen before. It is very easy to see that most candidates only need 2 EMDM iterations before they are the same as seen before, by themselves or by another candidate.

Data set	*1*	*2*	*3*	*4*	*5*	*Total*
Delicious	6	164	5	1	0	176
Flickr S	446	933	55	16	5	1455
Flickr M	437	943	60	8	5	1453
Flickr L	436	954	48	9	6	1453
Flixster	3	165	12	4	3	187
Friendster	877	1767	143	26	6	2819
Last FM	27	116	16	10	8	177

Table 7.2: Number of iterations before seeing earlier situation

7.5.2 Overlap

The EMDM algorithm allows overlap between communities. In fact, there is not a single measure to prevent overlap. In many cases this is an advantage. We start first with the same example as used in Section 7.2.1, now shown in Figure 7.7. The number in each vertex represents the number of communities that it is part of. The darker the color, the more communities contain those vertex.

If we look at the central vertex indicated by the arrow it is easy visible from communities it is part of (if not, look at Figure 7.1). This is a good example where overlap is useful. In social networks overlap between natural communities is a very frequent occurring thing.

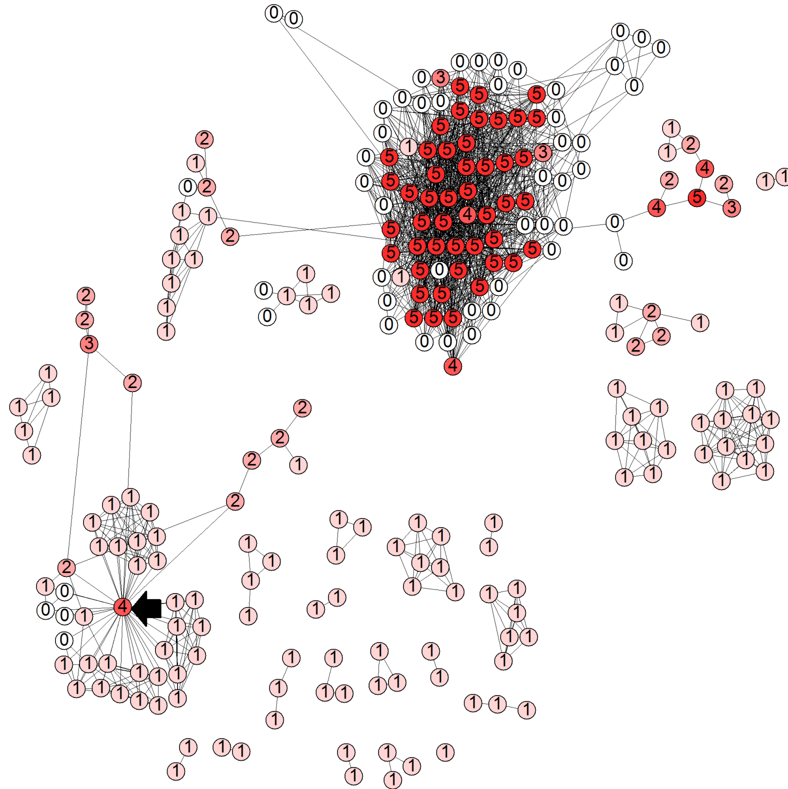


Figure 7.7: Resulting communities in first 250 vertices from Delicious data set

The example given in 7.7 is very small. A few other examples from the larger data sets show that allowing overlap is useful. For each data set there are many communities that have overlapping vertices, but a majority of vertices that are not overlapping. A real world example could be a person that is friends with both friends from the city where he is born, and the city where he studies. He himself is the overlap, the two groups of friends are the two different communities he is a member of. Table 7.3 gives an example from each data set we did experiments with.

Data set	$\|C_1\|$	$\|C_2\|$	$\|C_1 \cap C_2\|$	$\|C_1 \cup C_2\|$	*Overlap*
Delicious	14	13	2	25	8.0%
Flickr M	352	200	36	516	7.0%
Flixster	45	56	3	98	3.1%
Friendster	63	52	10	105	9.5%
Last FM	26	11	1	36	2.8%

Table 7.3: Overlap examples

Preventing too Much Overlap

Sometimes the communities found by EMDM are very similar. The method of saving history as explained in Section 5.1.1 and proved to be useful in Section 7.5.1 has a second advantage except speeding up the whole process and guaranteeing termination. It also filters duplicate communities from the resulting community set.

It turned out that the EM step has a very strong preference for certain communities. That is why reducing the number of candidates worked so well. But in many cases it turned out that

Figure	*Data set*	*ID*	*Patterns*	*Total length*	*Attributes*	$\|C\|$	$CG(C)$
7.8a	Flickr L	996	33	377	32	58	1188
7.8b	Flickr L	1361	25	406	30	125	3647
7.8c	Flickr L	1922	46	593	39	75	2011
7.8d	Last FM	5	3	10	5	9	27
7.8e	Last FM	217	4	14	4	7	18

Table 7.4: Pattern lengths for example communities

EMDM found a few large communities with only one vertex different from each other. The union of these communities had a lower community gain score than each individual, so it was not in the result set.

For those cases we did some post processing by requiring a minimum Jaccard distance. The Jaccard distance is defined as:

$$JD(C_1, C_2) = \frac{|C_1 \cup C_2| - |C_1 \cap C_2|}{|C_1 \cup C_2|}$$

A higher Jaccard distance between two communities means less overlap. In the post processing a minimum Jaccard distance of 0.25 turned out to work pretty well. But this can be different on different data sets, and it may be very useful to look at the similar communities and think about why they are not one community.

7.5.3 Communities Found

Our goal was to find communities, with compact descriptions. In this subsection we present some examples communities we found. In the examples we separate the results from the real world data sets and the data sets we generated with CFinder.

Real World Data

We have three real world data sets, Delicious, Flickr, and Last FM. In each of these data sets we were able to find good results on the graph part. It turned out that the classifiers we found were very complex and large to interpret. They consist out of many conjunctions. In Table 7.4 some of the communities we found are shown.

Each of the n patterns is a conjunction of one or more attributes. The total classifier is the disjunction of all patterns, it has the same form as specified in Section 3. The total length is the sum of the lengths of all patterns. The attributes column gives us the number of distinct attributes used in the classifier. Although the total length is quite high, the number of attributes is much lower. This means there is a lot of similarity between each of the patterns.

To interpret a classifier we looked at all the distinct attributes $a^j \in Q_C$ one by one. We looked at the information gain when splitting DB on a^j. The higher the information gain, the more important the attribute is for the community.

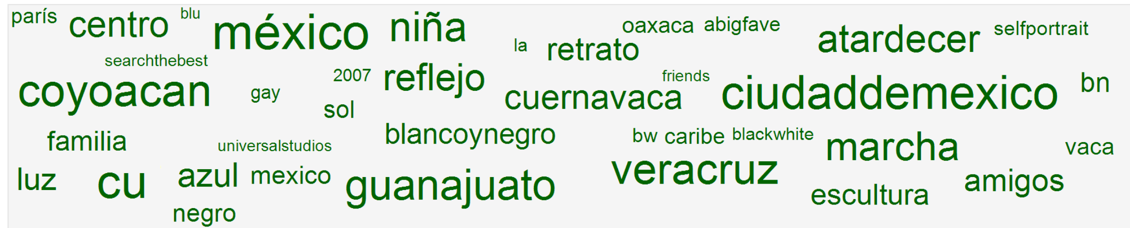
The information gain is calculated as:

$$IG(C, a^j) = impurity(DB) - \left(\frac{P_{c=0,a^j=1} + P_{c=1,a^j=1}}{|DB|} + \frac{P_{c=0,a^j=0} + P_{c=1,a^j=0}}{|DB|}\right)$$

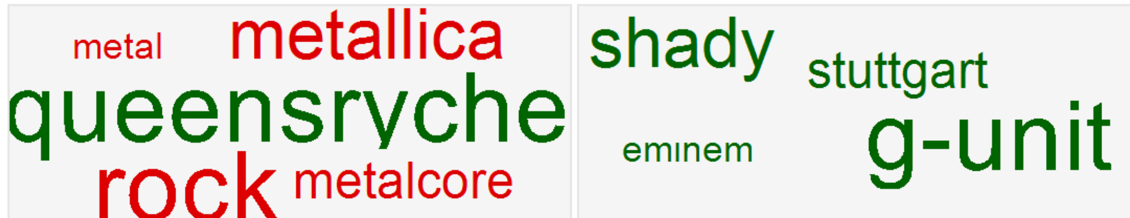
After splitting DB on a^j we now how important an attribute is to a community C. Attributes can contribute both in a positive or a negative way. If the ratio $\frac{inCom}{inCom+outCom}$ of records r_i containing attribute $a_i^j = 1$ is higher before the split, an attribute is contributing negative, i.e. a record r_i having $a_i^j = 1$ is less likely to be a member of the community than another record r_k having $a_k^j = 1$.

(a) Flickr L community 996

(b) Flickr L community 1361



(c) Flickr L community 1922



(d) Last FM community 5 (e) Last FM community 217

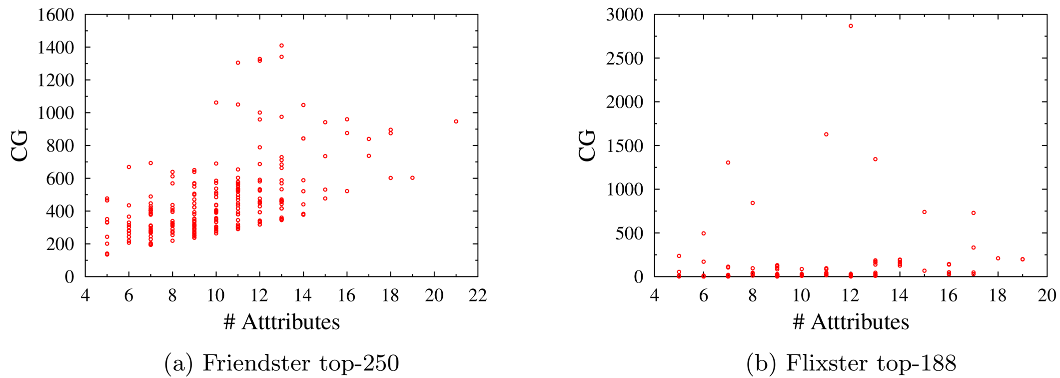
Figure 7.8: Tag clouds for different communities

To make it easier to interpret the complex classifiers we visualized them using tag clouds. More important tags are larger, and less important tags are smaller. Tags in green are contributing positive, tags in red are contributing negative. The tag clouds are from the same communities as mentioned in Table 7.4.

It is very clear that the tags are related to each other. Unfortunately we did not find any compact descriptions from the Delicious data set that can be easily presented here.

CFinder Data

The data sets generated with CFinder have description data that is created from community like structures. We expect shorter communities than with the real world data sets. In Section 4.3 we discussed the quality measure to quantify the quality of the communities. As quality measure $\varphi(C)$ for the graph part we used community gain, and as description complexity measure $\rho(Q_C)$ we use the number of attributes used in the description query. The total objection function is $\frac{\varphi(C)}{\rho(Q_C)}$. We ordered the results descending according to these score function and took the top-k communities found.



(a) Friendster top-250

(b) Flixster top-188

Figure 7.9: Top-k communities

Data set	*minDist*	$\vert Candidates \vert$	*EM only time (s)*	*EMDM time (s)*	*EM/Total*
Delicious	3	180	0.5	47	1.1
Flickr L	3	2006	1493	219373	0.7
Flixster	2	208	0.9	4585	0.02
Friendster	1	6508	28	69408	0.04
Last FM	2	217	0.7	82	0.9

Table 7.5: Runtime of different experiments

Figure 7.9 is a plot of the description complexity $\rho(Q_C)$ and the community gain of the two data sets we generated with CFinder. In the Friendster data set we found far more communities with a high community score. This is mainly caused due to the graph part of the community. This can be verified by looking at the differences between the two data sets in Figure 7.5 in Section 7.3.

In both data sets we found communities that are large, and have a high community gain score. All the communities found have queries consisting of less than 22 attributes, and the majority has even less than 12.

7.5.4 Runtime

The EM algorithm we developed is quite fast, unfortunately the DM part took much more time. With CFinder we were only able to find communities in the few social graphs we had with low density, and even then it was not possible to find communities in graphs with more than 25000 vertices. In our EMDM approach very much time was spent in building classifiers. We think the method used in the EM step can also be very useful in other applications than the EMDM algorithm. That is why we also illustrate the performance of the EM step without finding classifiers. For all data sets we used the lowest *minDist* settings that did not cause a significant drop down in the quality of the result. Table 7.5 gives us the runtimes of the complete experiment. This also includes the time of reading the description matrix and the graph from the disk, generate candidates, do the filtering at the and, and sort the result according to the community score.

7.5.5 Communities found

In appendices A, B and C we listed the communities found by our research, and gave a summary about it. Appendix A is a table of statistics about the top-50 communities found. Appendix B is a table of statistics about all the communities found. Appendix C lists the top-50 communities themselves, with scores of every community found.

Chapter 8

Related work

The problem of community detection is a topic on which much research is done, and with many different methods. See [10] for an empirical comparison of different algorithms and quality measures. Most of the methods only focus on the graph part of networks, not on the description data that we consider as well. The majority of the methods do graph partitioning, i.e. assigning each vertex to exactly one community.

Kahn, Yan and Wu introduce the concept of proximity patterns in [8]. Many traditional methods use the concept of frequent patterns, which are patterns occurring frequently in the database. Proximity patterns try to blur the boundary between the graph data and the description data. When mining proximity patterns not only patterns within one vertex are considered. Patterns can also exist of one attribute of a vertex u, and one attribute of a vertex v, as long as there is an edge $(u, v) \in E$. They developed a model called *Nearest Probabilistic Association* to define the frequency of a proximity pattern. This approach transforms the problem of finding communities in graph and attribute data into something similar as traditional frequent pattern mining; for which many methods exist. They also developed an adapted FP-tree algorithm which is quite fast.

In literature there is not a single accepted definition for a cluster in a graph. Silva, Meira and Zaki suggest using γ-*quasi-cliques* in [13]. An γ-quasi-cliques with $0 < \gamma \leq 1$ is a set of vertices V where every vertex $v \in V$ is connected to at least $\gamma(|V| - 1)$ vertices. This paper introduces the *SCORP* algorithm. It starts with mining all frequent item sets for all users. For each frequent item set it selects the set of vertices containing all items. Then it finds all quasi-cliques in these sets, and tries to adjust the set of items selecting these vertices. A big disadvantage of this approach is that it needs to find all frequent item sets, that can be a very large set with a low minimum support σ. Another disadvantage is that there are a many parameters to tune: $\gamma, \sigma_{min}, \epsilon_{min}, min_s ize$. There is no general good set of settings, it heavily depends on the data set.

Moser, Colak, Rafiey and Ester introduce a problem similar to our problem as *finding cohesive patterns* in [11]. A cohesive pattern is defined as a connected sub graph whose density exceeds a given threshold. Furthermore a cohesive pattern has, in a large enough subspace, homogeneous feature values. Integrating constraints on the frequent item sets reduces the number of patterns substantially. They developed an algorithm called *CoPaM* that efficiently finds the set of all maximal cohesive patterns. The results of this method set are reasonable, but the size of the data sets is many times smaller than the data sets we used (742 vs. 100267). With patterns longer than 5 it was not possible to generate them because of memory overflow.

Atzmueller and Mitzlaff use subgroup discovery as a framework to find communities in [1]. A subgroup discovery function $sd(s)$ is more or less what we call community queries. This description selects certain vertices of DB. They start with building an adapted version of FP-trees. Every

tree node of the FP-tree also store edge information about the community they represent. After generating these FP-tree upper bounds of the possible community qualities can be calculated. By calculating these upper bounds, an algorithm called COMODO can order, sort and prune this tree and efficiently find communities. They use traditional quality measures like modularity and inverse conductance to quantify the quality of a community.

There are many studies to detection of communities in networks, they are related to both social networks, and physics. In physics network analysis is a frequently studied topic, for example in. Many of these studies focus on the graph part only, not on the description atrributes. Du et al.[4] propose an algorithm *ComTector* in. They start with finding maximal cliques as first step in finding communities and use these to find communitues.

Chapter 9

Conclusions

During the experiments we were getting more and more convinced that our approach of finding communities is a very useful one. Our goal was to find cohesive communities with concise descriptions. This turned out to work really well with our approach. The algorithm was fast enough to work on large data sets, and we found many communities with descriptions that made sense. Allowing overlap turned out to be very useful.

In the introduction we mentioned an example of the beer brand that wants find the right public to sell their tickets to. When looking at the community shown in Figure 7.8d on page 40 this is exactly the group we wish to find in the introduction. When we change "beer brand" to "airline" and "rock festival tickets" to "tickets to Mexico" we would be very satisfied with the results displayed in Figure 7.8c.

We will explain the most important results of our research slightly more formal.

9.1 Community gain

The quality measure *community gain* we defined in this thesis seems to be very useful. A better community gain score implied a better modularity, inverse conductance, and a far higher intra cluster density than over the whole graph. This strong proves that the community gain score is a good score to define the quality of a community. When looking at the picture shown in Section 7.1 it is easy to verify that the communities make sense. Groups of vertices that are close together and have many internal edges belong to the same community. A large advantage of the community gain is that it is quite easy to optimize, as we have proven with the GreCO heuristic.

9.2 EMDM Algorithm

The EMDM algorithm did not work out on the way as we hoped it would. We hoped that both the EM and the DM step would improve the community each iteration. The idea was that the community would converge to a final state and that the EM or DM step would not change the algorithm. The experiment in Section 7.5.1 illustrates that in many cases the algorithm just alternates between one community found in the EM step, being adjusted in the DM step, and then goes back to the same result as found in the EM step. Om the other hand; it demonstrates that the EM step is very powerful, it has a strong preference for certain communities.

The EMDM algorithm with a candidate set as input also allows overlap. In our opinion allowing overlap is a very natural thing in social networks. People have certain circles of friends, which have small overlap. By prohibiting overlap it is never possible to find natural communities. Experiment 7.5.2 confirms our vision on this point.

9.2.1 Local search

Working with candidates that only look at vertices in the same area of the graph, and no need to look at the whole graph is a good property. It makes is possible to run different candidates in parallel. In an era where computers are getting more and more processors this is useful. Another advantage of local search is that there is no need to divide the whole graph into communities before knowing the quality of the partitioning.

9.2.2 Candidate Set

A big advantage of our approach is that it is possible to start with both the EM and the DM step. This makes it possible to generate candidates based on a description, which could be very useful in some cases, like advertising. The idea of reducing the number of candidates by requiring a minimum distance between each candidate worked really well. It reduced the number of candidates very much, without reducing the accuracy.

9.2.3 EM step

With CFinder it was possible to find the communities in graphs till about 25 thousand vertices, after that it took more than the 10 GB memory available in the computers we used. It also took more than 24 hours to find all the communities with graphs of this size. With our program finding all communities in the Flickr data set (100267 vertices), with a setting of $minDist = 3$ takes only 24 minutes. This makes it a very useful method for finding communities in large graphs.

9.2.4 DM step

We had the many problems with finding good descriptions for each community. It was possible to find a very good classifier, but then it was so large that it was not possible to interpret it by hand. With the adapted ReMine approach and our method of determining the importance of each tag in the classifier we were able to generate tag clouds. These tag clouds made it possible to characterize the communities. Unfortunately finding descriptions was a complex operation. For the Flickr L data set, with $minDist = 3$ the EM steps took only 1.16% of the total runtime. The rest was for the DM steps.

9.3 Future research

Although we are very satisfied after the research there are many points interesting for further investigation.

9.3.1 Description space

All the data we used for the description data was tag data. The EMDM algorithm does not restrict the data to be binary tag data. The most problems we had during the research, had to do with the classifiers. It would be very interesting to try to work with different types of data and algorithms on the description side. It could also result in a more useful application of the EMDM algorithm, where it slowly converges into good communities.

9.3.2 Graph space

We focused on undirected, unweighted graphs only. But it can be useful to work with weighted graphs and/or directed graphs. For example two persons, queen of pop *Madonna* and one of her million fans can have a friendship link. Probably the fan knows much more about her, than other way around. This could be modeled with a directed graph.

Using weights for edges could also be interesting. In our research we just say people are friends, or they are not. Instead of using this binary value we could make it an integer dependent on how close their relation is. This can be done by counting how often they chat with each other, watch each other's profiles, or check each other's photos.

9.3.3 Post processing

As mentioned before, the EM step has a strong choice for certain communities. It finds many communities that are more or less similar. We now solved it by filtering them on a minimum Jaccard distance. May be there are better ways, for example by combining similar communities in the result set to something better.

Acknowledgements

First and foremost I would like to thank my supervisor dr. Matthijs van Leeuwen for his guidance during the two years it took to do this research and write the thesis. I also want to thank dr. Francesco Bonchi from Yahoo for helping me getting data sets and participating in the many discussions Matthijs and I had on Skype. Without Matthijs and Francesco I would not have been able to do this research and write this thesis. Apart from Matthijs and Francesco I would also like to thank prof. dr. Arno Siebes for reading and judging my thesis.

Also, I would like to take this opportunity to thank my father, who works as a computer scientist himself, for his advice. Special thanks go to my Canadian friends Patrick and Mina who corrected the many mistakes I made in English writing.

Last but not least, I want to thank my girlfriend, friends and family helping me forget everything written down in this thesis during the evenings and weekends. Finally, I want to thank Utrecht University and my study friends for the fun we had in the seven years we spent together!

Bibliography

[1] M. Atzmueller and F. Mitzlaff. Efficient descriptive community mining. In *Proc. 24th Intl. FLAIRS Conference*. AAAI Press, 2011.

[2] I. Derényi, G. Palla, and T. Vicsek. Clique percolation in random networks. *Phys. Rev. Lett.*, 94:160202, Apr 2005.

[3] G. Dong and J. Li. Efficient mining of emerging patterns: Discovering trends and differences. In *Efficient Mining of Emerging Patterns: Discovering Trends and Differences*, pages 43–52, 1999.

[4] N. Du, B. Wu, X. Pei, B. Wang, and L. Xu. Community detection in large-scale social networks. In *Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis*, WebKDD/SNA-KDD '07, pages 16–25, New York, NY, USA, 2007. ACM.

[5] S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75 – 174, 2010.

[6] F. Geerts, B. Goethals, and T. Mielikäinen. Tiling databases, 2004.

[7] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Min. Knowl. Discov.*, 8(1):53–87, Jan. 2004.

[8] A. Khan, X. Yan, and K.-L. Wu. Towards proximity pattern mining in large graphs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 867–878, New York, NY, USA, 2010. ACM.

[9] D. Leman, A. Feelders, and A. Knobbe. Exceptional model mining. In W. Daelemans, B. Goethals, and K. Morik, editors, *Machine Learning and Knowledge Discovery in Databases*, volume 5212 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-87481-2_1.

[10] J. Leskovec, K. J. Lang, and M. Mahoney. Empirical comparison of algorithms for network community detection. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 631–640, New York, NY, USA, 2010. ACM.

[11] F. Moser, R. Colak, A. Rafiey, and M. Ester. Mining cohesive patterns from graphs with feature vectors. In *SDM*, pages 593–604. SIAM, 2009.

[12] J. Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465 – 471, 1978.

[13] A. Silva, W. Meira, Jr., and M. J. Zaki. Structural correlation pattern mining for large graphs. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, MLG '10, pages 119–126, New York, NY, USA, 2010. ACM.

[14] M. van Leeuwen. Maximal exceptions with minimal descriptions. *Data Mining and Knowledge Discovery*, 21:259–276, 2010. 10.1007/s10618-010-0187-5.

[15] T. Vicsek. Cfinder.

[16] H. Zhang, B. Qiu, C. L. Giles, H. C. Foley, and J. Yen. An lda-based community structure discovery approach for large-scale social networks. In *ISI'07*, pages 200–207, 2007.

[17] A. Zimmermann, B. Bringmann, and U. Ruckert. Fast, effective molecular feature mining by local optimization. In J. Balcazar, F. Bonchi, A. Gionis, and M. Sebag, editors, *Machine Learning and Knowledge Discovery in Databases*, volume 6323 of *Lecture Notes in Computer Science*, pages 563–578. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-15939-8-36.

Appendix A

Summary Top-50 Communities

This appendix gives the average scores of the top-50 communities found in the data sets. For readability and because of the number of columns the data is split into two tables.

Data set	$\|tags\|$	*Density*	$\|V\|$	$\|E\|$	*minDist*	*#Candidates*
Delicious	1350	0.0389	1861	7664	3	180
Flickr S	2791	0.0298	100267	3781947	3	1453
Flickr M	7565	0.0136	100267	3781947	3	1453
Flickr L	16215	0.0072	100267	3781947	3	1453
Flixster	5844	0.0252	25000	221625	2	208
Friendster	8088	0.0252	32768	267528	1	6508
Last FM	11946	0.0015	1892	12717	2	217

Table A.1: Summary of top-50 communities found

Data set	*Size*	*CG*	*ICD*	*#Patterns*	*#Attributes*	*Length*	*Score*	*Runtime*
Delicious	13.3	101.6	0.78	4.5	12.9	43.3	95.80	7.8
Flickr S	78.7	2584.4	0.55	16.3	26.9	297.4	829.60	167.9
Flickr M	78.1	2550.4	0.55	16.3	24.3	273.8	856.30	187.5
Flickr L	83.3	2744.4	0.54	23.3	25.9	405.9	858.40	246.2
Flixster	24.4	311.1	0.61	5.2	10.8	39.6	124.20	29.9
Friendster	41.1	606.8	0.58	4.2	9.0	29.3	3029.70	66.9
Last FM	17.2	97.2	0.55	5.6	9.4	39.1	86.90	10.9

Table A.2: Summary of top-50 communities found

Appendix B

Summary Found Communties

This appendix gives the average scores of all the communities found in the data sets. For readability and because of the number of columns the data is split into two tables.

Data set	$\|tags\|$	*Density*	$\|V\|$	$\|E\|$	*minDist*	*#Candidates*	*#Communities*
Delicious	1350	0.0389	1861	7664	3	180	167
Flickr S	2791	0.0298	100267	3781947	3	1453	835
Flickr M	7565	0.0136	100267	3781947	3	1453	843
Flickr L	16215	0.0072	100267	3781947	3	1453	848
Flixster	5844	0.0252	25000	221625	2	208	173
Rriendster	8088	0.0252	32768	267528	1	6508	1530
Last FM	11946	0.0015	1892	12717	2	217	134

Table B.1: Summary of communities found

Data set	*Size*	*CG*	*ICD*	*#Patterns*	*#Attributes*	*Length*	*Score*	*Runtime*
Delicious	7.3	39.0	0.77	3.1	9.6	25.1	88.10	3.2
Flickr S	10.7	181.1	0.61	4.4	13.5	51.2	965.90	11.4
Flickr M	10.6	177.6	0.61	4.25	11.3	43.2	967.90	12.7
Flickr L	11.0	190.2	0.62	4.7	10.4	49.8	966.50	16.3
Flixter	11.0	97.2	0.58	3.8	9.6	26.1	102.60	9.4
Friendster	14.5	101.1	0.60	4.3	10.7	29.7	2945.80	10.0
Last FM	8.9	39.5	0.61	4.2	6.5	21.3	107.40	4.7

Table B.2: Summary of communities found

Appendix C

Top-50 Communities Listed

This appendix lists the top-50 communities found in the data sets we used, with respect to the score function. Each data set takes one full page.

- Delicious, Table C.1 on page 53.
- Flickr S, Table C.2 on page 54.
- Flickr M, Table C.3 on page 55.
- Flickr L, Table C.4 on page 56.
- Flixster, Table C.5 on page 57.
- Friendster, Table C.6 on page 58.
- Last FM, Table C.7 on page 59.

Rank	*ID*	*Size*	*CG*	*ICD*	*#Patterns*	*#Attributes*	*Pattern length*	*Score*
1	49	20	353	0.95	7	13	56	27.1
2	59	49	768	0.55	15	29	246	26.4
3	77	18	291	0.96	7	14	49	20.7
4	44	12	108	0.87	3	6	12	1
5	165	10	90	1	2	5	7	1
6	124	14	149	0.87	3	10	23	14.9
7	84	12	102	0.84	3	9	16	11.3
8	1	13	111	0.80	3	10	17	11.1
9	82	18	168	0.69	5	18	58	9.3
10	153	14	113	0.74	5	13	39	8.6
11	106	9	57	0.86	3	7	14	8.1
12	69	8	56	1	3	7	15	8
13	137	16	87	0.57	3	11	23	7.9
14	177	19	102	0.53	5	13	51	7.8
15	16	9	66	0.94	4	9	22	7.3
16	18	12	102	0.84	4	14	32	7.2
17	123	14	122	0.78	5	17	45	7.1
18	107	22	99	0.47	5	14	43	7.0
19	143	7	42	1	2	6	9	7
20	127	12	90	0.78	4	13	30	6.9
21	55	25	150	0.5	9	23	146	6.5
22	158	10	84	0.95	4	13	37	6.4
23	4	11	74	0.78	4	12	30	6.1
24	176	6	30	1	1	5	5	6
25	160	11	83	0.83	3	14	34	5.9
26	14	19	102	0.53	8	18	101	5.6
27	142	19	102	0.53	8	18	100	5.6
28	140	7	39	0.95	2	7	10	5.5
29	161	22	96	0.47	6	18	64	5.3
30	57	7	42	1	3	8	18	5.2
31	146	7	42	1	2	8	12	5.2
32	48	19	99	0.52	6	19	64	5.2
33	118	6	30	1	2	6	8	5
34	113	12	60	0.63	4	12	34	5
35	150	9	60	0.88	4	12	32	5
36	37	22	99	0.47	6	21	78	4.7
37	154	11	89	0.87	5	19	53	4.6
38	92	7	42	1	3	9	15	4.6
39	95	10	60	0.77	3	13	28	4.6
40	173	15	96	0.63	6	22	88	4.3
41	174	10	39	0.62	4	9	21	4.3
42	175	8	56	1	5	13	34	4.3
43	40	19	99	0.52	9	23	124	4.3
44	79	8	38	0.78	3	9	20	4.2
45	42	8	41	0.82	3	10	20	4.1
46	58	9	57	0.86	5	14	50	4.0
47	33	13	48	0.53	4	12	31	4
48	32	13	63	0.60	4	16	44	3.9
49	62	8	53	0.96	4	14	36	3.7
50	39	6	30	1	4	8	21	3.7

Table C.1: Top-50 Communities Delicious

Rank	*ID*	*Size*	*CG*	*ICD*	*#Patterns*	*#Attributes*	*Pattern length*	*Score*
1	230	355	23154	0.45	2	8	13	2894.2
2	807	313	17661	0.45	2	8	13	2207.6
3	558	103	2970	0.52	1	5	5	5
4	891	193	13203	0.57	36	34	607	388.3
5	1892	184	8262	0.49	34	38	828	217.4
6	1624	196	9543	0.49	53	46	1288	207.4
7	19	121	3342	0.48	5	22	55	151.9
8	377	101	2945	0.52	10	27	141	109.0
9	361	101	3269	0.54	18	31	280	105.4
10	139	140	4952	0.50	61	50	1220	99.0
11	25	112	3069	0.49	19	33	349	9
12	894	71	1799	0.57	11	25	123	71.9
13	941	100	2949	0.53	32	42	653	70.2
14	603	58	1296	0.59	8	22	114	58.9
15	1117	73	1968	0.58	21	34	387	57.8
16	1833	82	2406	0.57	41	43	883	55.9
17	1526	44	776	0.60	6	15	59	51.7
18	311	100	2355	0.49	43	52	1071	45.2
19	996	58	1188	0.57	12	29	186	40.9
20	167	70	1335	0.51	22	33	351	40.4
21	1050	46	759	0.57	10	20	123	37.9
22	1073	70	1563	0.54	35	42	725	37.2
23	1496	43	591	0.55	5	16	56	36.9
24	134	52	822	0.53	8	23	111	35.7
25	1161	43	594	0.55	5	17	64	34.9
26	386	59	872	0.50	9	25	150	34.8
27	540	34	552	0.66	6	16	63	34.5
28	1002	40	651	0.61	5	19	57	34.2
29	957	58	1227	0.58	34	37	663	33.1
30	112	46	723	0.56	9	24	111	30.1
31	427	52	867	0.55	9	29	151	29.8
32	276	61	1002	0.51	30	34	639	29.4
33	439	42	468	0.51	4	16	45	29.2
34	914	49	672	0.52	16	24	231	2
35	988	55	1077	0.57	29	39	401	27.6
36	144	34	378	0.55	4	14	41	2
37	1157	46	591	0.52	10	22	167	26.8
38	1720	55	894	0.53	30	34	496	26.2
39	581	37	471	0.56	6	18	74	26.1
40	1298	40	600	0.58	9	23	126	26.0
41	457	56	956	0.54	13	39	256	24.5
42	1577	35	371	0.54	5	16	46	23.1
43	491	52	729	0.51	14	33	264	22.0
44	536	44	593	0.54	14	28	204	21.1
45	994	31	333	0.57	6	16	59	20.8
46	1931	52	696	0.50	19	35	380	19.8
47	829	42	639	0.58	11	33	209	19.3
48	756	20	191	0.66	3	10	20	19.1
49	1609	35	431	0.57	13	23	179	18.7
50	1137	31	465	0.66	9	25	132	18.6

Table C.2: Top-50 Communities Flickr S

Rank	*ID*	*Size*	*CG*	*ICD*	*#Patterns*	*#Attributes*	*Pattern length*	*Score*
1	1021	355	23157	0.45	3	6	12	3859.5
2	636	310	17667	0.45	5	14	39	1261.9
3	558	103	2970	0.52	1	4	4	742.5
4	251	121	3342	0.48	1	5	5	668.4
5	891	193	13203	0.57	15	23	166	574.0
6	194	184	8262	0.49	24	37	559	223.2
7	382	196	9531	0.49	127	60	3064	158.8
8	377	101	2945	0.52	23	28	382	105.1
9	361	101	3269	0.54	14	33	247	99.0
10	139	140	4952	0.50	57	50	1146	99.0
11	1056	112	3069	0.49	28	33	456	9
12	894	71	1799	0.57	9	20	76	89.9
13	1117	73	1968	0.58	16	25	187	78.7
14	603	58	1296	0.59	12	17	147	76.2
15	1833	82	2406	0.57	30	32	489	75.1
16	1413	97	2346	0.50	21	32	379	73.3
17	276	58	996	0.53	5	17	64	58.5
18	134	52	822	0.53	9	15	100	54.8
19	167	70	1335	0.51	10	27	134	49.4
20	1050	46	759	0.57	10	17	125	44.6
21	1526	44	776	0.60	12	18	138	43.1
22	540	34	552	0.66	4	13	41	42.4
23	1555	55	1080	0.57	16	27	204	4
24	1496	43	591	0.55	5	15	42	39.4
25	457	56	956	0.54	7	25	118	38.2
26	1073	70	1563	0.54	42	41	664	38.1
27	996	58	1188	0.57	32	32	360	37.1
28	427	52	867	0.55	14	24	193	36.1
29	1157	46	591	0.52	4	17	40	34.7
30	1931	52	696	0.50	9	21	97	33.1
31	1720	55	894	0.53	27	28	458	31.9
32	957	58	1227	0.58	31	39	645	31.4
33	112	46	723	0.56	9	23	105	31.4
34	574	61	1002	0.51	18	33	373	30.3
35	1298	40	600	0.58	10	22	133	27.2
36	386	59	872	0.50	13	32	251	27.2
37	1002	40	651	0.61	11	24	154	27.1
38	914	49	672	0.52	25	25	413	26.8
39	1577	34	369	0.55	3	14	23	26.3
40	581	37	468	0.56	5	18	56	2
41	675	43	594	0.55	15	23	162	25.8
42	342	49	669	0.52	13	30	203	22.3
43	439	42	468	0.51	8	21	122	22.2
44	1239	37	399	0.53	6	18	58	22.1
45	1137	31	465	0.66	6	21	92	22.1
46	1344	52	729	0.51	17	33	348	22.0
47	1789	28	321	0.61	6	15	45	21.4
48	628	35	425	0.57	9	20	104	21.2
49	759	34	378	0.55	7	18	87	2
50	829	42	639	0.58	9	31	181	20.6

Table C.3: Top-50 Communities Flickr M

Rank	*ID*	*Size*	*CG*	*ICD*	*#Patterns*	*#Attributes*	*Pattern length*	*Score*
1	1519	196	9546	0.49	1	2	2	4
2	1021	355	23157	0.45	3	6	12	3859.5
3	891	193	13203	0.57	28	17	279	776.6
4	566	307	17661	0.45	142	49	3920	360.4
5	1001	184	8256	0.49	32	28	537	294.8
6	155	101	2945	0.52	17	25	213	117.8
7	157	199	6561	0.44	119	62	3307	105.8
8	139	140	4952	0.50	61	47	1241	105.3
9	952	103	2970	0.52	19	29	258	102.4
10	1056	112	3069	0.49	32	30	491	102.3
11	894	71	1799	0.57	12	19	124	94.6
12	361	101	3269	0.54	42	39	667	83.8
13	791	122	3323	0.48	36	42	611	79.1
14	1403	100	2949	0.53	33	40	509	73.7
15	1117	73	1968	0.58	21	27	312	72.8
16	603	58	1296	0.59	13	18	151	7
17	1045	100	2355	0.49	26	38	423	61.9
18	1833	82	2406	0.57	46	39	593	61.6
19	134	52	822	0.53	8	15	76	54.8
20	167	70	1335	0.51	10	25	131	53.4
21	1526	44	776	0.60	6	16	70	48.5
22	1073	70	1563	0.54	27	33	495	47.3
23	132	81	1977	0.53	42	42	627	47.0
24	1676	37	471	0.56	5	11	37	42.8
25	540	34	552	0.66	4	13	39	42.4
26	988	55	1077	0.57	21	26	233	41.4
27	1050	46	759	0.57	9	19	111	39.9
28	1002	40	651	0.61	10	17	87	38.2
29	996	58	1188	0.57	33	32	377	37.1
30	574	61	1002	0.51	17	27	241	37.1
31	457	58	954	0.52	17	27	203	35.3
32	1157	43	591	0.55	5	17	45	34.7
33	386	59	872	0.50	8	26	125	33.5
34	1670	55	870	0.52	12	26	174	33.4
35	112	46	723	0.56	11	22	118	32.8
36	1789	28	321	0.61	7	10	37	32.1
37	505	62	848	0.48	34	28	557	30.2
38	1931	52	696	0.50	10	23	126	30.2
39	675	43	594	0.55	21	20	234	29.7
40	957	58	1227	0.58	32	42	666	29.2
41	144	34	378	0.55	6	13	60	29.0
42	439	42	468	0.51	7	17	69	27.5
43	450	49	666	0.52	17	25	210	26.6
44	628	35	425	0.57	7	16	65	26.5
45	1720	55	894	0.53	25	34	478	26.2
46	1298	40	600	0.58	17	23	218	26.0
47	1137	31	465	0.66	8	18	110	25.8
48	609	49	669	0.52	21	27	235	24.7
49	1415	52	729	0.51	16	31	311	23.5
50	81	30	372	0.61	8	16	82	23.2

Table C.4: Top-50 Communities Flickr L

Rank	*ID*	*Size*	*CG*	*ICD*	*#Patterns*	*#Attributes*	*Pattern length*	*Score*
1	64	79	2868	0.64	4	12	31	2
2	9	61	1614	0.62	2	7	11	230.5
3	141	56	1712	0.70	4	11	32	155.6
4	122	64	1515	0.58	10	13	104	116.5
5	167	43	843	0.64	2	8	13	105.3
6	95	34	495	0.62	2	6	11	82.5
7	163	63	852	0.47	10	16	105	53.2
8	162	64	840	0.47	12	16	122	52.5
9	207	22	237	0.67	2	5	9	47.4
10	181	19	171	0.66	3	6	16	28.5
11	154	41	326	0.46	8	13	75	25.0
12	127	31	204	0.47	5	9	30	22.6
13	153	31	333	0.57	5	17	51	19.5
14	23	16	114	0.65	3	7	16	16.2
15	169	37	291	0.47	9	18	91	16.1
16	198	21	144	0.56	4	9	30	1
17	97	15	105	0.66	3	7	16	1
18	170	22	132	0.52	5	9	28	14.6
19	20	28	198	0.50	7	14	65	14.1
20	149	24	195	0.56	6	14	46	13.9
21	81	30	207	0.49	11	16	101	12.9
22	46	19	102	0.53	4	8	21	12.7
23	197	25	114	0.46	4	9	29	12.6
24	183	24	174	0.54	7	14	52	12.4
25	172	12	96	0.81	4	8	19	1
26	5	35	212	0.45	8	19	89	11.1
27	94	31	207	0.48	10	19	99	10.8
28	103	10	54	0.73	2	5	9	10.8
29	45	16	84	0.56	5	9	31	9.3
30	159	13	96	0.74	4	11	27	8.7
31	61	13	87	0.70	3	10	19	8.7
32	106	7	39	0.95	2	5	8	7.8
33	123	19	96	0.52	5	13	37	7.3
34	69	22	105	0.48	7	16	68	6.5
35	137	13	54	0.56	5	9	27	6
36	205	22	102	0.48	7	17	60	6
37	176	11	44	0.6	6	8	38	5.5
38	165	13	57	0.57	5	11	25	5.1
39	173	16	69	0.52	8	15	82	4.6
40	114	8	35	0.75	4	8	18	4.3
41	178	9	48	0.77	5	11	28	4.3
42	92	13	42	0.51	4	10	21	4.2
43	180	7	33	0.85	4	8	18	4.1
44	196	10	27	0.53	3	7	15	3.8
45	96	7	30	0.80	3	8	14	3.7
46	145	10	45	0.66	6	13	42	3.4
47	2	10	30	0.55	4	9	21	3.3
48	105	9	27	0.58	4	9	17	3
49	158	9	30	0.61	5	10	28	3
50	72	5	20	1	3	7	15	2.8

Table C.5: Top-50 Communities Flixster

Rank	*ID*	*Size*	*CG*	*ICD*	*#Patterns*	*#Attributes*	*Pattern length*	*Score*
1	4726	69	1152	0.49	2	8	12	1
2	6378	70	1392	0.52	6	11	38	126.5
3	4328	46	705	0.56	2	6	10	117.5
4	407	63	1410	0.57	7	13	61	108.4
5	6370	52	1062	0.60	8	10	56	106.2
6	2366	40	507	0.55	3	6	13	84.5
7	5205	49	648	0.51	4	8	27	8
8	4825	54	1005	0.56	7	13	53	77.3
9	382	67	1137	0.50	7	15	60	75.8
10	4989	43	606	0.55	4	8	24	75.7
11	5156	38	650	0.64	4	9	24	72.2
12	10	41	650	0.59	4	9	24	72.2
13	5025	35	569	0.65	4	8	26	71.1
14	5927	70	1125	0.48	10	16	102	70.3
15	108	27	351	0.66	1	5	5	70.2
16	99	33	489	0.64	3	7	18	69.8
17	6055	61	888	0.49	5	13	36	68.3
18	1342	28	333	0.62	2	5	9	66.6
19	796	49	732	0.54	5	11	43	66.5
20	397	27	330	0.64	2	5	9	6
21	5284	50	851	0.56	5	13	39	65.4
22	5286	49	846	0.57	7	13	59	65.0
23	3468	31	447	0.65	3	7	16	63.8
24	4729	31	366	0.59	2	6	9	6
25	124	37	549	0.60	3	9	17	6
26	123	49	588	0.5	5	10	35	58.8
27	121	49	645	0.51	5	11	34	58.6
28	6496	45	585	0.53	5	10	32	58.5
29	5098	31	405	0.62	3	7	18	57.8
30	4797	34	507	0.63	4	9	24	56.3
31	4974	23	506	1	5	9	25	56.2
32	2246	31	393	0.61	3	7	18	56.1
33	972	36	498	0.59	5	9	42	55.3
34	6295	47	605	0.51	6	11	39	5
35	857	31	330	0.56	3	6	18	5
36	253	36	384	0.53	4	7	23	54.8
37	137	34	543	0.65	5	10	41	54.3
38	5360	35	488	0.60	4	9	22	54.2
39	5229	34	429	0.58	2	8	13	53.6
40	200	43	579	0.54	8	11	58	52.6
41	967	27	315	0.63	2	6	10	52.5
42	5213	35	419	0.56	2	8	14	52.3
43	2391	29	314	0.59	2	6	10	52.3
44	5186	31	417	0.63	4	8	28	52.1
45	1136	40	495	0.54	4	10	22	49.5
46	1899	40	543	0.56	4	11	37	49.3
47	2134	40	591	0.58	6	12	45	49.2
48	252	25	243	0.60	3	5	16	48.6
49	4915	26	242	0.58	2	5	9	48.4
50	451	43	474	0.50	5	10	41	47.4

Table C.6: Top-50 Friendster

Rank	*ID*	*Size*	*CG*	*ICD*	*#Patterns*	*#Attributes*	*Pattern length*	*Score*
1	108	43	330	0.45	3	3	6	1
2	43	40	330	0.47	4	7	17	47.1
3	195	43	348	0.46	7	12	45	2
4	110	42	366	0.47	10	13	98	28.1
5	49	25	297	0.66	7	12	44	24.7
6	165	41	347	0.47	16	18	176	19.2
7	148	28	309	0.60	10	17	110	18.1
8	185	43	354	0.46	20	22	241	16.0
9	42	42	348	0.46	15	22	160	15.8
10	14	16	54	0.48	4	4	10	13.5
11	28	16	81	0.55	4	7	17	11.5
12	158	13	45	0.52	4	4	10	11.2
13	193	13	45	0.52	4	4	10	11.2
14	2	11	38	0.56	2	4	5	9.5
15	100	16	81	0.55	4	9	25	9
16	61	10	30	0.55	4	4	10	7.5
17	8	15	51	0.49	4	7	21	7.2
18	15	19	84	0.49	6	12	36	7
19	84	16	48	0.46	4	7	19	6.8
20	83	16	54	0.48	4	8	19	6.7
21	191	19	81	0.49	7	12	49	6.7
22	1	16	66	0.51	5	10	35	6.6
23	71	15	51	0.49	4	8	21	6.3
24	45	7	42	1	4	7	19	6
25	44	15	60	0.52	5	10	28	6
26	145	8	29	0.67	4	5	14	5.8
27	5	13	51	0.55	4	9	25	5.6
28	88	16	45	0.45	5	8	27	5.6
29	17	9	36	0.66	4	7	17	5.1
30	75	10	36	0.6	4	7	18	5.1
31	47	14	50	0.51	5	10	35	5
32	162	14	53	0.52	6	11	35	4.8
33	149	14	53	0.52	6	12	37	4.4
34	105	13	48	0.53	5	11	36	4.3
35	86	13	42	0.51	6	10	38	4.2
36	137	13	45	0.52	4	11	25	4.0
37	26	10	24	0.51	4	6	16	4
38	51	13	51	0.55	7	13	58	3.9
39	79	7	30	0.80	3	8	16	3.7
40	78	7	15	0.57	4	4	10	3.7
41	35	12	36	0.51	6	10	33	3.6
42	58	13	42	0.51	6	12	41	3.5
43	123	10	27	0.53	4	8	24	3.3
44	132	13	36	0.48	4	11	31	3.2
45	10	16	54	0.48	7	17	75	3.1
46	73	10	36	0.6	6	12	48	3
47	106	6	21	0.8	4	7	14	3
48	112	13	36	0.48	5	12	39	3
49	134	4	6	0.66	2	2	3	3
50	167	7	15	0.57	4	5	11	3

Table C.7: Top-50 Communities Lastfm