DYNAMIC MODULARITY BASED COMMUNITY DETECTION FOR LARGE SCALE NETWORKS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

RIZA AKTUNÇ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JANUARY 2015

Approval of the thesis:

DYNAMIC MODULARITY BASED COMMUNITY DETECTION FOR LARGE SCALE NETWORKS

submitted by **RIZA AKTUNÇ** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Gülbin Dural Ünver ______________
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Adnan Yazıcı ______________
Head of Department, **Computer Engineering**

Prof. Dr. İsmail Hakkı Toroslu ______________
Supervisor, **Computer Engineering Department, METU**

Examining Committee Members:

Prof. Dr. Faruk Polat
Computer Engineering Department, METU ______________

Prof. Dr. İsmail Hakkı Toroslu
Computer Engineering Department, METU ______________

Assoc. Prof. Dr. Pınar Karagöz
Computer Engineering Department, METU ______________

Assist. Prof. Dr. İsmail Sengör Altıngövde
Computer Engineering Department, METU ______________

Dr. Güven Fidan
CEO, ARGEDOR Information Tech. ______________

Date: ______________

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: RIZA AKTUNÇ

Signature :

ABSTRACT

DYNAMIC MODULARITY BASED COMMUNITY DETECTION FOR LARGE SCALE NETWORKS

Aktunç, Rıza
M.S., Department of Computer Engineering
Supervisor : Prof. Dr. İsmail Hakkı Toroslu

January 2015, 67 pages

In this work, a new fast dynamic community detection framework for large scale networks is presented. Most of the previous community detection algorithms are designed for static networks. Static modularity optimizer framework (SMO), which is introduced by Waltman & Van Eck, consists of such community detection algorithms. However, large scale social networks are dynamic and evolve frequently over time. To quickly detect communities in dynamic large scale networks, we proposed dynamic modularity optimizer framework (DMO) that is constructed by making the modularity based community detection algorithms placed in SMO dynamic. The proposed framework is tested on the mobile communication networks which are extracted from the raw call detail records (CDR) data of a GSM operator in Turkey. According to the results, community detection algorithms in the proposed framework perform better than algorithms in SMO when large scale dynamic networks are considered.

Keywords: dynamic, modularity optimization, community detection, large scale networks

ÖZ

BÜYÜK ÖLÇEKLİ AĞLARDA DİNAMİK TOPLULUK ALGILAMA ALGORİTMASI

Aktunç, Rıza
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. İsmail Hakkı Toroslu

Ocak 2015 , 67 sayfa

Bu çalışmada büyük ölçekli ağlar için yeni ve hızlı bir dinamik topluluk algılama algoritma çatısı sunulmuştur. Geçmişteki topluluk algılama algoritmalarının çoğu statik ağlar için tasarlanmıştır. Waltman & Van Eck bu tarzda topluluk algılama algoritmalarından oluşan statik modülerlik optimizasyonu (SMO) adlı algoritma çatısını literatüre katmıştır. Fakat büyük ölçekli sosyal ağlar dinamiktir ve zaman içinde çok çabuk gelişirler. Bu tip dinamik ağlarda toplulukları hızlı bir şekilde algılamak için SMO içerisinde yer alan modülerlik tabanlı topluluk algılama algoritmalarını dinamik olarak çalışacak hale getirerek dinamik modülerlik optimizasyonu (DMO) adlı algoritma çatısını tasarladık ve ileri sürdük. Tasarladığımız bu algoritma çatısını mobil iletişim ağları üzerinde test ettik. Bu iletişim ağlarını Türkiye'deki bir GSM operatöründen aldığımız bilgilerle oluşturduk. Testlerimizin sonuçlarına göre, büyük ölçekli dinamik ağlar düşünüldüğünde, tasarladığımız algoritma çatısı içindeki topluluk algılama algoritmaları SMO içindeki algoritmalardan daha iyi performans göstermişlerdir.

Anahtar Kelimeler: dinamik, modülerlik optimizasyonu, topluluk algılama, büyük ölçekli ağlar

To my dearest wife and family

ACKNOWLEDGMENTS

I would like to express my sincere gratitude and appreciation to my supervisor, Prof. Dr. İsmail Hakkı Toroslu for his encouragement and support throughout this study.

I would also like to convey thanks to jury members for their valuable comments on this thesis.

I owe my deepest gratitude to my dearest wife and family for their love and encouragement. Without them, this work could not have been completed.

I would also like to thank my current and former colleagues in TÜBİTAK BİLGEM, especially Emre Gürbüz, for their support during my M.Sc.

TABLE OF CONTENTS

ABSTRACT v

ÖZ vi

ACKNOWLEDGMENTS viii

TABLE OF CONTENTS ix

LIST OF FIGURES xii

LIST OF TABLES xiii

LIST OF ABBREVIATIONS xvi

CHAPTERS

1 INTRODUCTION 1

2 BACKGROUND 5

2.1 Community Definitions 5

2.1.1 Local Definitions 6

2.1.2 Global Definitions 7

2.2 Modularity 8

2.3 Community Detection Algorithms 11

2.3.1 Modularity Based 12

2.3.2 Compression Based 12

2.3.3 Significance Based 13

2.3.4 Diffusion Based 14

3 RELATED WORK 17

3.1 Dynamic Community Detection Algorithms 17

4 DYNAMIC MODULARITY OPTIMIZER FOR LARGE SCALE NETWORKS 21

4.1 Static Modularity Optimization Approach 21

4.2 Dynamic Modularity Optimization Approach 24

5 PREPROCESS 31

6 EXPERIMENTS & RESULTS 37

6.1 Experiments on Louvain Algorithm in SMO & DMO 37

6.1.1 The size of the base network: 10,000,000 Edges . . 37

6.1.1.1 Additions 37

6.1.1.2 Deletions 38

6.1.2 The size of the base network: 20,000,000 Edges . . 39

6.1.2.1 Additions 40

6.1.2.2 Deletions 41

6.2 Experiments on LMR Algorithm in SMO & DMO 42

6.2.1 The size of the base network: 10,000,000 Edges . . 42

6.2.1.1 Additions 42

6.2.1.2 Deletions 43

6.2.2 The size of the base network: 20,000,000 Edges . . 44
6.2.2.1 Additions 44
6.2.2.2 Deletions 45
6.3 Experiments on SLM Algorithm in SMO & DMO 46
6.3.1 The size of the base network: 10,000,000 Edges . . 46
6.3.1.1 Additions 47
6.3.1.2 Deletions 48
6.3.2 The size of the base network: 20,000,000 Edges . . 50
6.3.2.1 Additions 50
6.3.2.2 Deletions 52
6.4 Implementation Notes 54
7 CONCLUSION 57
7.1 Summary of Conducted Approach 57
7.2 Discussion 59
7.3 Future Work 62
REFERENCES 63

LIST OF FIGURES

FIGURES

Figure 2.1 Networks with high and low modularity values [50] 11

Figure 4.1 Pseudocode of Louvain Algorithm [48] 28

Figure 4.2 Pseudocode of LMR Algorithm [48] 29

Figure 4.3 Pseudocode of SLM Algorithm [48] 30

LIST OF TABLES

TABLES

Table 6.1 Louvain Algorithm in SMO (Base: 10,000,000 Edges, Addition) . . 38

Table 6.2 Louvain Algorithm in DMO (Base: 10,000,000 Edges, Addition) . . 38

Table 6.3 DMO Effects to Louvain Algorithm (Base: 10,000,000 Edges, Addition) 38

Table 6.4 Louvain Algorithm in SMO (Base: 10,000,000 Edges, Deletion) . . 39

Table 6.5 Louvain Algorithm in DMO (Base: 10,000,000 Edges, Deletion) . . 39

Table 6.6 DMO Effects to Louvain Algorithm (Base: 10,000,000 Edges, Deletion) 39

Table 6.7 Louvain Algorithm in SMO (Base: 20,000,000 Edges, Addition) . . 40

Table 6.8 Louvain Algorithm in DMO (Base: 20,000,000 Edges, Addition) . . 40

Table 6.9 DMO Effects to Louvain Algorithm (Base: 20,000,000 Edges, Addition) 40

Table 6.10 Louvain Algorithm in SMO (Base: 20,000,000 Edges, Deletion) . . 41

Table 6.11 Louvain Algorithm in DMO (Base: 20,000,000 Edges, Deletion) . . 41

Table 6.12 DMO Effects to Louvain Algorithm (Base: 20,000,000 Edges, Deletion) 41

Table 6.13 LMR Algorithm in SMO (Base: 10,000,000 Edges, Addition) 42

Table 6.14 LMR Algorithm in DMO (Base: 10,000,000 Edges, Addition) . . . 43

Table 6.15 DMO Effects to LMR Algorithm (Base: 10,000,000 Edges, Addition) 43

Table 6.16 LMR Algorithm in SMO (Base: 10,000,000 Edges, Deletion) 43

Table 6.17 LMR Algorithm in DMO (Base: 10,000,000 Edges, Deletion) 44

Table 6.18 DMO Effects to LMR Algorithm (Base: 10,000,000 Edges, Deletion) 44

Table 6.19 LMR Algorithm in SMO (Base: 20,000,000 Edges, Addition) 44

Table 6.20 LMR Algorithm in DMO (Base: 20,000,000 Edges, Addition) . . . 45

Table 6.21 DMO Effects to LMR Algorithm (Base: 20,000,000 Edges, Addition) 45

Table 6.22 LMR Algorithm in SMO (Base: 20,000,000 Edges, Deletion) 45

Table 6.23 LMR Algorithm in DMO (Base: 20,000,000 Edges, Deletion) 46

Table 6.24 DMO Effects to LMR Algorithm (Base: 20,000,000 Edges, Deletion) 46

Table 6.25 SLM Algorithm in SMO (Base: 10,000,000 Edges, Addition) 47

Table 6.26 SLM Algorithm in DMO (Base: 10,000,000 Edges, Addition) 47

Table 6.27 DMO Effects to SLM Algorithm (Base: 10,000,000 Edges, Addition) 47

Table 6.28 SLM Algorithm in DMO with Expected Modularity Value Set (Base: 10,000,000 Edges, Addition) 48

Table 6.29 Effects of DMO with Expected Modularity Value to the SLM Algorithm (Base: 10,000,000 Edges, Addition) 48

Table 6.30 SLM Algorithm in SMO (Base: 10,000,000 Edges, Deletion) 49

Table 6.31 SLM Algorithm in DMO (Base: 10,000,000 Edges, Deletion) 49

Table 6.32 DMO Effects to SLM Algorithm (Base: 10,000,000 Edges, Deletion) 49

Table 6.33 SLM Algorithm in DMO with Expected Modularity Value Set (Base: 10,000,000 Edges, Deletion) 50

Table 6.34 Effects of DMO with Expected Modularity Value to the SLM Algorithm (Base: 10,000,000 Edges, Deletion) 50

Table 6.35 SLM Algorithm in SMO (Base: 20,000,000 Edges, Addition) 51

Table 6.36 SLM Algorithm in DMO (Base: 20,000,000 Edges, Addition) 51

Table 6.37 DMO Effects to SLM Algorithm (Base: 20,000,000 Edges, Addition) 51

Table 6.38 SLM Algorithm in DMO with Expected Modularity Value Set (Base: 20,000,000 Edges, Addition) 52

Table 6.39 Effects of DMO with Expected Modularity Value to the SLM Algorithm (Base: 20,000,000 Edges, Addition) 52

Table 6.40 SLM Algorithm in SMO (Base: 20,000,000 Edges, Deletion) 52

Table 6.41 SLM Algorithm in DMO (Base: 20,000,000 Edges, Deletion) 53

Table 6.42 DMO Effects to SLM Algorithm (Base: 20,000,000 Edges, Deletion) 53

Table 6.43 SLM Algorithm in DMO with Expected Modularity Value Set (Base: 20,000,000 Edges, Deletion) 53

Table 6.44 Effects of DMO with Expected Modularity Value to the SLM Algorithm (Base: 20,000,000 Edges, Deletion) 54

Table 6.45 Overall contribution of DMO with respect to additions 55

Table 6.46 Overall contribution of DMO with respect to deletions 56

LIST OF ABBREVIATIONS

SMO	Static Modularity Optimizer
DMO	Dynamic Modularity Optimizer
LMR	Louvain with Multilevel Refinement Algorithm
SLM	Smart Local Moving Algorithm
SLMEVS	Smart Local Moving Algorithm with Expected Value Set
OSLOM	Order Statistics Local Optimization Method
COPRA	Community Overlap Propagation Algorithm
FacetNet	Framework for Analyzing Communities and EvoluTions in dynamic NETworks
iLCD	Intrinsic Longitudinal Community Detection
LPA	Label Propagation Algorithm
SLPA	Speaker-listener Label Propagation Algorithm
MCL	Markov Cluster Algorithm
SSD	Solid State Disk
RAM	Random Access Memory
Ghz	Gigahertz
GSM	Global System for Mobile Communications
CDR	Call Detail Records

CHAPTER 1

INTRODUCTION

In the last decade, the notion of social networking is emerged and produced very large graphs that consist of the information of their users. These graphs generally consist of the nodes that represent the users and edges that represent the relations among users. The nodes in these graphs generally tend to get together and construct communities of their own. Thus, it can be stated that social networks commonly have a community structure. These networks can be divided into groups of nodes that have denser connections inside the group; but fewer connections to the outside of the group. For example, in a GSM network, a group of users who call each other more densely than they call other users may construct their own community. In this case, the nodes represent the users and the edges represent the calls that users made. The detection of communities in these large networks is a problem in this area; therefore a lot of community detection algorithms such as [13, 21, 14, 33, 27, 26, 5, 43, 48] proposed in the literature. Almost all these community detection algorithms are static and designed for static networks.

However, most of the social networks are not static because they evolve in many ways. They may gain or lose users that are represented as nodes in the graphs over time. The users of these social networks may lose contact from each other or there can be new connections among users. In other words, some edges in the graphs may be removed or new edges may be added to the graph over time. All these processes may happen in a very small amount of time in a social network if it has a lot of active users. This kind of a social network may be called as highly dynamic. For example, popular social sites such as Facebook, Twitter, LinkedIn and so on have highly dynamic social

networks. Moreover, most GSM networks have millions of users and hundreds of calls made in seconds; therefore, they can also be labeled as highly dynamic networks. Addition or deletion of an edge or a node from a network which has millions of edges might seem insignificant; but when this additions or deletions of an edge or a node happen very frequently, they begin to change the community structure of the whole network and become very important. This change in the community structure raises the need of re-identification of communities in the network. This need arises frequently and creates a new problem in the community detection research area. This new problem requires somehow fast detection of communities in dynamic networks.

Community detection in large dynamic social networks brings considerable advantages in practice. For example, it can be used to optimize the performance of routing algorithms in communication networks. Nodes and edges represent people and mobile communications respectively in these networks. Routing the messages and calls among the people in these networks is very challenging since people moves around frequently and their calling behavior is not stable. In order to prevent forwarding unnecessary messages through nodes in different communities, dynamic community structure algorithm can be used to analyze the community structure of the communication network at the given timestamp and then messages can be forwarded directly to nodes in the same or related communities. Thus, the performance of routing messages and calls can be improved via decreasing the number of duplicate messages and overhead information in the routing process.

The first solution that comes into mind for community detection in large dynamic networks problem is the execution of static community detection algorithms already defined in the literature all over again to detect the new community structure whenever the network is modified. Nevertheless, this solution takes too much time in every modification of the large networks since it runs the community detection algorithm from scratch each time. A much efficient and less time consuming solution is to run the community detection algorithms not from scratch but from a point in the history of the network by storing and using the historical results of executions of the algorithms whenever network is evolved. In other words, updating previously discovered community structure instead of trying to find communities from scratch each time the network evolves consumes much less time and thus much efficient. This

solution method for the problem of detecting communities in large dynamic networks is the main focus of our study in this thesis work.

In this thesis, we modified the static modularity optimizer defined by Waltman & Van Eck [48] so that it would detect the communities in rapidly growing large networks dynamically and efficiently. As a result, we propose the dynamic modularity optimizer that dynamically detects communities in large networks by optimizing modularity and using its own historical results. We tested our proposed approach on the communication network that is constructed by preprocessing the CDR data taken from a GSM operator in Turkey. We demonstrated the effects of our contribution to the static modularity optimizer in two ways. One of them is the change in modularity value which determines the quality of the community structure of the network. The other one is the change in running time that determines the pace of the algorithm. The latter is more significant than the former because the community structure of the network must be quickly identified at the given timestamp before the next timestamp is reached. We realized that DMO improved SMO by decreasing its running time incredibly. In some experiments, the modularity value decreases, but it can be negligible. Moreover, there are some experiments where modularity value increases while running time decreases.

The rest of the thesis is organized as follows. Chapter 2 gives some basic information about static community detection algorithms and modularity. Chapter 3 gives most of the academic works previously done on the dynamic community detection concept. The proposed solution for dynamic community detection in large networks called as DMO is described in Chapter 4. In Chapter 5, the preprocess of the CDR data of the GSM network and construction steps of the graphs that DMO is tested on is explained. In Chapter 6, the results of the experiments of DMO and SMO are demonstrated. Moreover, these results of experiments of SMO and DMO are compared in comparison tables in Chapter 6. Finally, the thesis is concluded and the future works that can be done on the problem are mentioned in Chapter 7.

CHAPTER 2

BACKGROUND

2.1 Community Definitions

Defining the meaning of the community is one of the crucial factors that affect both the quality and the measurement of community detection algorithms. However, there is no definition of the community which is accepted universally. [16] It is partially because of the fact that not all networks have the same community structure. This fact leads to defining the meaning of the communities specific to the network which is going to be analyzed. Although there is no universal definition of community, there is a widespread informal definition of community concept. This definition states that a community is formed by nodes which interact with each other more frequently than with other nodes in the network. In other words, if a group of nodes have internal edges that link each of them in the group more than external edges which link a node in the group to a node outside of the group, this group can be labeled as a community. [16, 36, 44] This definition can be seen as the reference guideline for the basis of most community definitions. Finding all the structures that obey a community definition which is provided prior to running the algorithm would be very hard and time consuming. Therefore, in most cases, communities are heuristically constructed as the result of the algorithm without any precise priori definition provided. Apart from the ratio of internal and external edges, the connectedness of the nodes in the community is a factor which is taken into consideration in most community definitions. To be precise, connectedness means that for each pair of vertices of a community, there must be at least a path which resides in the boundaries of the community and connects the vertices. [16]

2.1.1 Local Definitions

Communities are parts of the network with a few ties with the rest of the network. From a perspective, they can be considered as separate entities with their own autonomy. Therefore, it might make sense to evaluate them individually without the information of the whole graph. While deciding whether a subgraph may be a community or not, local definitions take only that subgraph and its immediate neighbors into consideration and ignores rest of the network.

One of these local definitions searches cliques in networks and assigns them as communities. Clique is a set that there is an edge between each pair of its vertices, so it is a fully connected set of vertices and edges. Vertices in the graph may belong to more than one clique simultaneously, and this represents the overlapping communities. This is a property which is at the basis of the Clique Percolation Method that is introduced by Palla et al. [37] Simplest cliques are the ones that have 3 vertices. They are frequently encountered in real networks. But larger cliques are less frequently seen in real networks. Furthermore, the condition to be fully connected in order to be labeled as a community is really too strict. For instance, a subgraph with all possible internal edges except only one edge would be an extremely cohesive subgraph; nevertheless, it would not be labeled as a community under this recipe. In addition to being strict, finding cliques in a network is an NP-complete problem. [6] The Bron-Kerbosch method is a method that tries to find maximal cliques in a running time which grows exponentially with the size of the given network. [9] In order to overcome these restrictiveness and running time problems, the notion of clique is relaxed and clique-like objects are defined in the literature. [16, 1, 37]

Being clique-like checks the internal cohesion of the given subgraph and ignores external cohesion. However, in real life, a subgraph should not be labeled as community if there is a strong external cohesion between the subgraph and the rest of the graph. Therefore, external cohesion should not be ignored and taken into consideration while evaluating a subgraph. It can be included into the equation as it gets stronger the effect of the internal cohesion gets smaller. In fact, most recent definitions of community apply and use this methodology. For instance, Radicchi et al. introduced strong and weak community definitions in the literature. In a strong community, internal degree

of each vertex is greater than its external degree. However, in a weak community, it is enough that the internal degree of the whole subgraph is greater than its external degree. [38] These definitions of community do not look for clique or clique-like objects and take both internal and external cohesion into consideration. Therefore, they are more relaxed than searching clique or clique-like objects.

Using a fitness measure is another method to identify communities. A fitness measure can be any property such that it can be checked that a subgraph satisfies it by evaluating its cohesion. Obviously, if the fitness value is large, the community is definite in the same ratio as the fitness value. The simplest fitness measure would be internal density of edges. There could be a threshold of internal density and if it is passed by a subgraph, it could be labeled as community. If this threshold is 1, it means subgraph must be fully connected (clique) in order to be labeled as community. This leads to the conclusion that applying this fitness measure is an NP-complete task in the worst case. [18] Variants of this fitness measure focus on not the density but the number of internal edges of the subgraph. [4, 15, 23] Another measure is the relative density which is formulated as the ratio between the internal and the total degree of the given subgraph. Fitness measures can also be associated to the connectivity of the given subgraph to the rest of the graph. Briefly, this methodology evaluates the subgraphs of a graph by checking the given condition called as fitness measure and label those subgraphs that pass this check.

2.1.2 Global Definitions

Communities can also be defined by taking not only the subgraph but whole graph into consideration. There are many global criteria that can be used to identify communities in the literature. Nonetheless, in most cases, they are indirect definitions such that some global property of the graph is used in a community detection algorithm that labels communities at the end. However, there is a class of proper definitions which are based on the idea that a graph has community structure if it is different from a randomly constructed graph. This idea is based on the assumption that a random graph is not expected to have community structure, because any two vertices have the same probability to be adjacent. Therefore, a null model is defined as a graph that matches

the original graph only in some of its structural features, but other than those features just a random graph. The most popular null model is that proposed by Newman and Girvan. It consists of a randomized copy of the original graph, where edges are cut and rewired randomly, under the condition that the expected degree of each vertex matches the degree of the vertex in the original graph. [16, 32] This null model is the basic concept behind the definition of modularity, which is explained in the next section in detail.

2.2 Modularity

Modularity is a function that is used for measuring the quality of the results of community detection algorithms. If the modularity value of a partitioned network is high, it means that the network is partitioned well. Apart from quality measurement, modularity is used as the basis of some community detection algorithms. These algorithms try to detect communities (partitions) in a network by trying to maximize the modularity value of the network. Thus, modularity is a function that is used for both quality measurement and community detection.

Modularity is based on the idea that a randomly created graph is not expected to have community structure, so comparing the graph at hand with a randomly created graph would reveal the possible community structures in the graph at hand. This comparison is done through comparing the actual density of edges in a subgraph and the expected edge density in the subgraph if the edges in the subgraph were created randomly. This expected edge density depends on how random the edges created. This dependency is tied to a rule that defines how to create the randomness and called as null model. A null model is a copy of an original graph and it keeps some of this original graphs structural properties but not reflects its community structure. There can be multiple null models for a graph such that each of them keeps different structural properties of the original graph. Using different null models for the calculation of the modularity leads to different modularity calculation methods and values. The most common null model that is used for modularity calculation is the one that preserves the degree of each vertex of the original graph. With this null model, modularity is calculated as the fraction of edges that fall in the given communities minus such fraction in the null

model. [34, 16] The formula of modularity can be written as in 2.1

$$Q = \frac{1}{2m}\sum_{ij}(A_{ij} - P_{ij})\delta(C_i, C_j) \tag{2.1}$$

m represents the total number of edges of the graph. Sum iterates over all vertices denoted as i and j. A_{ij} is the number of edges between vertex i and vertex j in the original graph. P_{ij} is the expected number of edges between vertex i and vertex j in the null model. The δ function results as 1 if the vertex i and vertex j are in the same community ($C_i = C_j$), 0 otherwise. The null model can be created by cutting the edges between vertices; thus, creating stubs (half edges) and rewiring them to random vertices. Thus, it obeys the rule of keeping degrees of vertices unchanged. Cutting edges into half, creates $m * 2 = 2m$ stubs. In the null model, a vertex could be attached to any other vertex of the graph and the probability that vertices i and j, with degrees k_i and k_j , are connected, can be calculated. The probability p_i to pick a ramdom stub connection for vertex i is $\frac{k_i}{2m}$, as there are k_i stubs of i out of a total of $2m$ stubs. The probability of vertex i and vertex j being connected is $p_i p_j$, since stubs are connected independently of each other. Since there are $2m$ stubs, there are $2mp_ip_j$ expected number of edges between vertex i and vertex j. [16] This yields to equation 2.2

$$P_{ij} = 2mp_ip_j = 2m\frac{k_ik_j}{4m^2} = \frac{k_ik_j}{2m} \tag{2.2}$$

By placing equation 2.2 into equation 2.1, modularity function is presented as in equation 2.3.

$$Q = \frac{1}{2m}\sum_{ij}(A_{ij} - \frac{k_ik_j}{2m})\delta(C_i, C_j) \tag{2.3}$$

The resulting values of this modularity function lie in the range $[\frac{-1}{2}, 1)$. It would be positive if the number of edges within subgraphs is more than the number of expected edges in the subgraphs of null model. Higher values of the modularity function mean better community structures. [48]

This modularity function also applies to weighted networks. [48, 30] The modularity function for the weighted graphs can be calculated as in equation 2.4.

$$Q_w = \frac{1}{2W} \sum_{ij} (W_{ij} - \frac{s_i s_j}{2W}) \delta(C_i, C_j) \tag{2.4}$$

There are three differences. The first difference is that in the case of a weighted network W_{ij}, instead of A_{ij}, may take not just 0 or 1 but any non-negative value that represents the weight of the edge. The second one is that instead of m, which is total number of edges, W, which is the sum of the weights of all edges is used in the equation. The last one is that s_i and s_j which represents the sum of the weights of edges adjacent to vertex i and vertex j respectively is used in the equation instead of k_i and k_j which means the degree of vertex i and vertex j respectively. [16]

Apart from weighted networks, the modularity function defined in 2.3 has been extended in order to be also applicable to directed networks. [3, 28] When the edges are directed, stubs will also be directed and it changes the possibility of rewiring stubs and connecting edges. The calculation of this possibility in the directed case depends on the in- and out-degrees of the end vertices. For instance, there are two vertices A and B. A has a high in-degree and low out-degree. B has a low in-degree and high out-degree. Thus, in the null model of modularity, an edge will be much more likely to point from B to A than from A to B. [16] Therefore, the expression of modularity for directed graphs can be written as in equation 2.5

$$Q_d = \frac{1}{m} \sum_{ij} (A_{ij} - \frac{k_i^{out} k_j^{in}}{m}) \delta(C_i, C_j) \tag{2.5}$$

The sum of the in-degrees (out-degrees) equals m not $2m$ as in the case of undirected graph. Therefore, the factor 2 in the denominator of the first and second summand has been dropped. In order to get the modularity function to be applicable to directed weighted networks, the equations 2.4 and 2.5 can be merged; thus, equation 2.6 can be constructed as the most general expression of modularity. [16]

$$Q_{dw} = \frac{1}{W} \sum_{ij} (W_{ij} - \frac{s_i^{out} s_j^{in}}{W}) \delta(C_i, C_j) \tag{2.6}$$

As an example of networks which have high and low modularity values, figure 2.1 can be seen. In this figure, network A is an example of good modularity, whereas network B is an example of bad modularity.

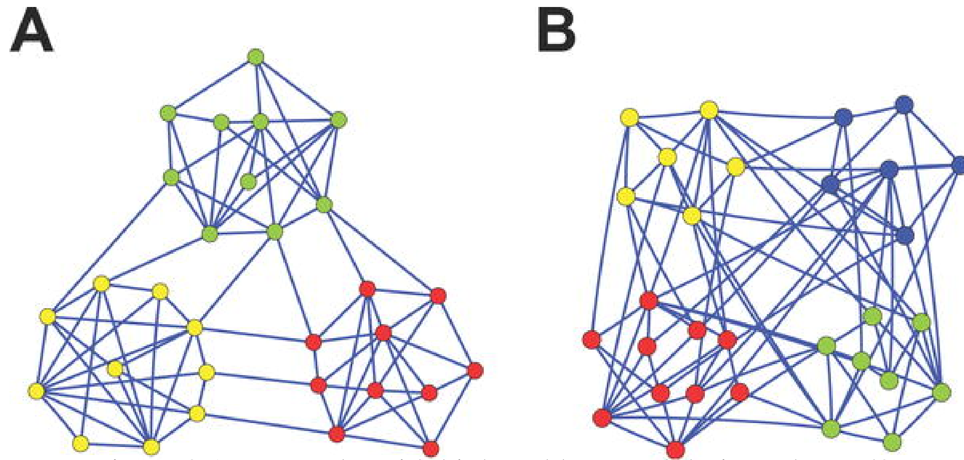

Figure 2.1: Networks with high and low modularity values [50]

There have been a few proposals of modified version of the modularity functions defined above as alternative modularity functions. These modified, extended versions for instance offer a resolution parameter that makes it possible to customize the granularity level at which communities are detected and to mitigate the resolution limit problem defined by Fortunato and Barthélemy [17]. [40] Moreover, there are modularity functions with a somewhat modified mathematical structure in the literature such as Reichardt & Bornholdt, 2006; Traag, Van Dooren, & Nesterov, 2011; Waltman, Van Eck, & Noyons, 2010. [48, 17, 46, 49]

2.3 Community Detection Algorithms

There are numerous community detection algorithms in the literature. They are implementing the notion of community structure differently. We selected a representative set of algorithms that we are going to explain in the following sections. We categorized them according to the method they apply to identify communities and explain them under these categories. [36]

2.3.1 Modularity Based

The idea of modularity-based community detection is to try to assign each vertex of the given network to a community such that it maximizes the modularity value of the network. The resulting assignments construct a community structure that is considered as the optimal community structure for the given network. Optimizing modularity is an NP-hard problem. [7] Exact algorithms that maximize modularity such as [2], [7], [53] can be used only for small networks.

There are many different heuristic algorithms proposed for modularity optimization in the literature. [16] They can be categorized as based on agglomerative hierarchical clustering ([13], [31]), simulated annealing ([21], [40]), extremal optimization ([14]), spectral optimization ([34], [33]), mean field annealing ([27]), and conformational space annealing ([26]). Nevertheless, most of these algorithms are suitable only for small and medium-sized networks.

For large-scale modularity optimization, heuristic algorithms are proposed. We are going to explain and discuss three of them currently in the literature. One of them is referred as Louvain algorithm and proposed by Blondel et al. in 2008. [5] The other one is an extension of the Louvain algorithm with a so-called multilevel refinement procedure. It is introduced by Rotta and Noack in 2011. [43] The last one is called as Smart Local Moving (SLM) algorithm that is proposed by Waltman and Jan van Eck in 2013. [48] All of these three algorithms are explained in detail in chapter 4.

2.3.2 Compression Based

There are community detection algorithms based on data compression proposed in the literature. They do not use the cohesion and separation concepts. They consider the community structure as a set of regularities in the network topology. This topology represents the whole network in a more compact way than the whole adjacency matrix. According to these algorithms, the best community structure is supposed to be the one that maximizes compactness and minimizes information loss. The algorithms mainly differ in the way they represent the community structure. They also differ in how they assess the quality of this representation. [36] InfoMod and InfoMap algo-

rithms can be categorized as compression based community detection algorithms.

InfoMod uses a simplified representation of the network that focuses on the community structure. It is formed by a community matrix and a membership vector. [41] Community matrix is an adjacency matrix whose cells contain community labels. Membership vector is a vector that associates each vertex to a community. Mutual information measure is used to decide how much information from the original network is preserved in the simplified representation. The one that has the maximal mutual information among all possible assignments of vertices to communities is selected as the best community structure detected. Simulated annealing is used for optimization. Minimum description length principle is used in order to select the optimal number of communities. [36]

InfoMap represents the community structure by a two level classification based on Huffman coding. One level detects communities in the network and the other level detects vertices in a community. [42] A random walk on the network using this classification requires some information. Minimizing the amount of this required information leads to finding the best community structure. A good community structure would have just a few inter community edges. With such a community structure, the random walker will probably stay longer inside communities. Therefore, only the second level will be needed to describe its path. Thus, amount of required information for this random walk will be minimized and this leads to a good community structure which has been known at the beginning. Simulated annealing is used for optimization of the amount of this required information. [36]

2.3.3 Significance Based

In a randomly created network, no community structure is expected; however, clusters of nodes that construct communities can be found due to random fluctuations. Statistical significance distinguishes such communities from actual communities. The algorithms in this class try to maximize statistical significance measure in order to detect communities. [36]

The order statistics local optimization method (OSLOM) is a local optimization method.

This method measures the statistical significance of individual communities and assigns the resulting values to the communities as their statistical significance values. Then, it tries to optimize these values in order to get a better community structure. [25] This statistical significance concept is defined as the probability of finding a similar community in a null model that has no community structure. Similarity is measured by comparing the size, degree sequence and internal connections of communities. The OSLOM, initially, tries to obtain a collection of significant communities by combining the vertices with their adjacent ones. Then, it tries to optimize the significance values of these communities by moving around vertices from one community to another possible community. This optimization process is repeated until it gets stable and the significance values are maximized. This community detection method can detect mutually exclusive, overlapped and hierarchical communities. [36]

2.3.4 Diffusion Based

This kind of approaches uses the information diffusion amount among the nodes in order to detect communities. It can be assumed that information is more efficiently propagated among vertices of the same community than vertices of different communities. The diffusion-based algorithms rely on this assumption and detect communities by considering how information is propagated in the parts of the network. [36]

A simple label propagation algorithm is proposed by Raghavan, Albert, and Kumara in 2007. This algorithm does not require either prior information about the communities or a predefined objective function to optimize on. It uses the network structure alone as its guide. It initializes every node with a unique label. At every iteration of this algorithm, each node adopts the label that most of its neighbors currently have. This iterative process resulted as labeling densely connected groups with unique different labels that represent communities. This label propagation algorithm runs in almost linear time; thus, it is efficient than most of the community detection algorithms in the literature with respect to their running time. [39]

As an extension to the label propagation algorithm, the community overlap propagation algorithm (COPRA) is proposed by Gregory in 2010. It is contribution to label

propagation algorithm is the ability to detect overlapping communities. Like the label propagation algorithm, nodes have labels that propagate between adjacent nodes so that members of a community reach a consensus on their community membership. The main contribution of the COPRA is to modify the label and propagation step to include information about more than one community. With this algorithm each node can have more than one label and this means that each node can belong to more than one community. Just like label propagation algorithm, the COPRA is also very fast. Therefore, it can process very large networks and detect communities including the overlapping ones in a short time. [20]

MarkovCluster algorithm simulates a diffusion process in the network to detect communities. [47] There is a transfer matrix defined in this algorithm. It describes the transition probabilities for a random walker evolving in the given network. This transfer matrix is modified by two transformation operations, expansion and inflation, iteratively until convergence. The result of the expansion operation would be a matrix showing the probability that a random walker starts from node i and reaches node j in p steps. In order to favor the higher probability values, the inflation operation raises each element in the matrix to some specified power. The value of this power affects the granularity of the final communities. After applying these expansion and inflation operations, the algorithm normalizes the resulting matrix to get a new transfer matrix, and repeats whole process until convergence. After convergence, the algorithm outputs the final matrix. This matrix can be considered as the adjacency matrix of a network with disconnected components, which correspond to communities in the original network. [36]

CHAPTER 3

RELATED WORK

3.1 Dynamic Community Detection Algorithms

Due to the dynamic features of many social networks [22], the need for detecting communities dynamically in the large networks is emerged in the latest years. There have been many community detection algorithms proposed in the literature to fulfill this need. Xu et al. divides the current research on community evolution into the following categories. Parameter estimation methods and probabilistic models have been proposed in the literature. [55, 45] A methodology that tries to find an optimal cluster sequence by detecting a cluster structure at each timestamp that optimizes the incremental quality can be classified as evolutionary clustering. [11, 24] Furthermore, tracking algorithms based on similarity comparison have also been studied in order to be able to describe the change of communities on the time axis. [19, 8] Apart from these algorithms that are focused on the evolution procedures of communities, community detection in dynamic social networks aims to detect the optimal community structure at each timestamp. [10, 35, 12] [54] Our algorithm can be classified as the last mentioned category which aims to detect optimal community structure at each timestamp with minimum running time. We are going to mention the properties of some of the proposed dynamic community detection algorithms in this section.

A unified framework that can be used for the analysis of communities and their evolution in dynamic networks is presented in 2009. It is called as FacetNet by its authors. It stands for "a Framework for Analyzing Communities and EvoluTions in dynamic NETworks". [29] This framework determines the community structure at a given

timestamp t by using community distribution information of the historic community structures before timestamp t and the observed network data at t. This framework can detect overlapping communities by having the ability of assigning a node to multiple communities. It has this ability because it applies a soft community membership concept instead of partitioning the network with hard boundaries. The authors argue that this soft community membership concept provides more details of the individuals' community participation behaviors as in the real networks individuals may tend to participate in more than one community. A stochastic block model is used for generating communities. In order to capture the community evolutions, a probabilistic model based on the Dirichlet distribution is adopted. By applying these probabilistic models, soft community memberships are assigned naturally to nodes; thus, it overcomes the problems faced by the parameters which exist in many dynamic algorithms. An iterative EM algorithm is used in order to guarantee the convergence to optimal solutions. The authors state that correctness and convergence of the algorithm is proven and it is shown that it has low time complexity when the network data is sparse.

An algorithm that detects both static and temporal communities is proposed by Cazabet et al. in 2010. It is named as Intrinsic Longitudinal Community Detection (iLCD). [10] It takes the dynamics of the networks and the overlapping communities into account. This algorithm updates the communities by adding them a new node if its number of second neighbors and robust second neighbors are greater than expected values. If the similarity between two communities is high, then the communities are merged. The similarity between two communities is measured as the ratio of nodes that are placed in both of these two communities. If the minimum patter is detected by the edges, a new community is created. This algorithm is based on two parameters. These parameters are the size of the minimal clique which effects the creation of new communities and the threshold for community merging.

An algorithm which performs well in detecting both overlapping nodes and overlapping communities with different degrees of diversity is proposed by Xie et al. in 2011. It is called as Speaker-listener Label Propagation Algorithm (SLPA). [52] Moreover, in this algorithm, labels are propagated among nodes according to dynamic interaction rules. Label distributions are maintained in the memory of each node. Each node

can be a speaker or a listener depending whether it serves as a information provider or consumer in SLPA. A node has the ability to have as many labels as it likes. The number of labels a node can have depends on the propagation experience in the stochastic processes which are driven by the underlying network structure. SLPA does not require any prior knowledge about the number of communities that must be detected. It is suitable for weighted, unweighted, directed and undirected networks. However, it produces different partitions in different runs due to random tie breaking strategy. This behavior of SLPA is not desirable in applications like tracking the evolution of communities in a dynamic network. Therefore, it is a disadvantage of SLPA for tracking community evolution in dynamic networks. The time complexity is O(tm), where t is a predefined maximum number of iterations, it is linear in the number of edges m.

Some of the authors of SLPA proposed LabelRank algorithm which stabilizes Label Propagation Algorithm (LPA) and extends Markov Cluster Algorithm (MCL) approach. [51] It stores, propagates and ranks labels in each node. In order to stabilize the propagation dynamics, it relies on four operators namely propagation, inflation, cutoff and conditional update. Eliminating the need of tie breaking, each node keeps multiple labels received from its neighbors. Nodes with same highest probability get together and form a community. The number of labels in each node monotonically decreases and drops to a small constant within few steps thanks to cutoff and inflation operators. Since there is no randomness in the simulation, the output is deterministic. Initialization of nodes takes O(m), adding self-loop takes O(n), each of the four operators takes O(m) on average. Thus, the running time of LabelRank is O(m), linear with the number of edges m.

In 2013, LabelRank is extended to LabelRankT with one extra conditional update rule by the authors of LabelRank. Only the nodes that are changed between two consecutive snapshots are updated in the algorithm. The previous snapshot is used for determining the dynamics in the current time step. It maintains the previous community structure and dynamically updates nodes that are involved in changes. This algorithm suits best to dynamic networks in which changes arrive as a stream thanks to this new conditional update rule. When a new edge is added in the stream, LabelRankT updates only the nodes that are attached to this edge. It performs as well as other static algorithms like MCL and Infomap, yet with lower running time. It

is faster than other dynamic detection algorithms like facetNet and iLCD. It can be applied to wireless sensor networks and mobile ad hoc networks. The overall running time of the algorithm for detecting evolving communities between two consecutive snapshots is O(Tm), implying O(m) in general where T is the number of iterations. LabelRankT can be integrated with SLPA in order to be able to detect overlapping communities.

CHAPTER 4

DYNAMIC MODULARITY OPTIMIZER FOR LARGE SCALE NETWORKS

4.1 Static Modularity Optimization Approach

SLM is a community detection algorithm that is evolved from Louvain algorithm. Therefore, Louvain algorithm must be explained first in order to explain SLM algorithm. Louvain algorithm is a large scale modularity based community detection algorithm that is proposed by Blondel et al in 2008. [5] The quality of detected communities by Louvain algorithm is measured by the method called modularity. The modularity of a network is a value that is between -1 and 1. This value presents the density of links inside communities over the density of links between communities. [34] When this value is close to 1, then the measured network can be called as modular network. In the case of weighted networks, modularity function can take weights into consideration and measure the quality of detected communities. Louvain algorithm uses modularity function as not only a measurement function but also an objective function to optimize.

Louvain algorithm is a recursive algorithm which has two steps running in each recursive call. Before the recursion starts, the algorithm assigns a different community to each node of the network whose communities are going to be detected. Therefore, in the initial case each node has its own community. In each recursive call the following steps are run:

1. It runs a local moving heuristic in order to obtain an improved community

structure. This heuristic basically moves each node from its own community to its neighbors' community and run the modularity function. If the result of the modularity function, which means quality, increased, the node would be kept in the new community; else, the node would be moved back to its previous community. This process is applied to each node for its each neighbor in random order and thereby heuristically the quality is tried to be increased.

2. The algorithm constructs a reduced network whose nodes are the communities that are evolved in the first step. Moreover, the weights of the edges in this reduced network are given by the sum of weights of the links between the nodes which reside in the corresponding two communities. Links between nodes of the same community in the old network are presented as self-links for the node that represents that community in the new reduced network. When this reduced network is fully constructed, then algorithm calls itself recursively and first step is applied to this reduced network.

The algorithm keeps recursing until no further improvement in modularity is measured and thereby there are no changes in the community structure. [5] The pseudo code of Louvain Algorithm provided by Ludo Waltman and Nees Jan van Eck can be seen in Figure 4.1.

Rotta and Noack proposed an algorithm that is a multilevel refinement procedure applied version of Louvain algorithm in 2011. [43] Louvain algorithm is moving hierarchical and after each recursive call the network is reduced. Therefore, after first recursive call, assigned communities become nodes and they are fed to local moving heuristic algorithm. Thus, for instance, in the third recursive call, the algorithm searches for the increase in modularity by only moving the communities found in the second recursive call. However, there is a possibility that the modularity is increased by moving the nodes of initial network at the end of each recursive call. The original Louvain algorithm does not take this possibility into consideration and run local moving heuristic algorithm only on reduced network. Louvain with multilevel refinement (LMR) algorithm takes this possibility into consideration and runs local moving heuristic on the constructed network at the end of each recursive call in order to search for increase in modularity by moving individual nodes apart from previously

formed communities. Since, this way the algorithm runs local moving heuristic with individual nodes level at the final recursive call, it is guaranteed that modularity cannot be increased further by changing individual nodes communities. [48] The original Louvain algorithm cannot guarantee this and this is the main difference between these algorithms. The main steps of LMR algorithm is described in Figure 4.2.

Louvain algorithm detects community structures whose modularity values are locally optimal with respect to community merging, but not necessarily locally optimal with respect to individual node movements. On the other hand, solutions found by the LMR algorithm are locally optimal with respect to individual node movements, but not necessarily locally optimal with respect to community merging. However, when these algorithms iteratively called by the way that each iteration assigns the previous iterations community structure to the nodes instead of assigning singleton communities, it becomes possible to find solutions that are locally optimal with respect to both community merging and individual node movements. [48] Since the Louvain algorithm applies local moving heuristic in the beginning of its recursive block and merges communities by reducing network in the end of its recursive block, calling it iteratively ensures that the resulting community structure cannot be improved further either by merging communities or by moving individual nodes from one community to another. Like the iterative variant of these algorithms SLM algorithm constructs community structures that are locally optimal with respect to both individual node movements and community merging. Besides these capabilities, SLM also tries to optimize modularity by splitting up communities and moving sets of nodes between communities. This is done by changing the way that local moving heuristic and network reduction runs.[48]

Both Louvain and LMR algorithms run local moving heuristic algorithm on the present network as the first step, and then construct the reduced network as the second step. However, the SLM algorithm changes the reduced network construction step by applying following processes:

1. It iterates over all communities that are formed by the first step. It copies each community and constructs a subnetwork that contains only the specific community's nodes.

2. It then runs the local moving heuristic algorithm on each subnetwork after assigning each node in the subnetwork to its own singleton community.

3. After local moving heuristic constructs a community structure for each subnetwork, the SLM algorithm creates the reduced network whose nodes are the communities detected in subnetworks. The SLM algorithm initially defines a community for each subnetwork. Then, it assigns each node to the community that is defined for the node's subnetwork. Thus, there is a community defined for each subnetwork and detected communities in subnetworks are placed under these defined communities as nodes in the reduced network.

This is the way that the SLM algorithm constructs the reduced network. After these processes, the SLM algorithm gives the reduced network to the recursive call as input and all the processes starts again for the reduced network. The recursion continues until a network is constructed that cannot be reduced further. To sum up, the SLM algorithm has more freedom in trying to optimize the modularity by having the ability to move sets of nodes between communities which cannot be done by either of other two algorithms. [48] Figure 4.3 demonstrates the main steps of the SLM algorithm in the shape of a pseudo code.

4.2 Dynamic Modularity Optimization Approach

We worked on the source code of a modularity optimizer that is provided by Waltman and Van Eck in 2013. While we are trying to run this source code of the modularity optimizer on our GSM network that we have produced earlier, we saw the possibility and opportunity of making the algorithms in the source code dynamic. The CDR data that we used to construct the mobile phone user network is very huge. Moreover, it keeps getting larger and larger very quickly. Therefore, whenever a new record is added to this CDR data, constructing the network from this CDR data and running the modularity optimizer to find the new community structure is a very time consuming task. At this very moment, we realized that making the modularity optimizer dynamic might reduce the overall running time. Thus, it might make analyzing the CDR data and detecting communities while it is growing possible. Therefore, we searched for

a way to make this optimizer run dynamically while new nodes and edges are being added to the network that is being analyzed. We propose the dynamic modularity optimizer (DMO) for the purpose of analyzing quickly growing large networks dynamically and detecting communities in these networks quicker.

The modularity optimizer that is provided by Waltman and Van Eck takes following items as parameters:

1. **Input file**

   A line in the input file must contain 2 node numbers and optionally the weight of the edge that links these two nodes. These 2 or 3 information must be delimited by a tab character. The numbering of nodes in the input file has to start at 0. For each pair of nodes, the node with the lower number must be listed first, followed by the node with the higher number. The lines in an input file must be sorted based on the node numbers (first based on the numbers in the first column, then based on the numbers in the second column).

2. **Output file**

   The algorithm prints the community ids to the output file. The id in the first line of the output file is the node number 0's community id. In the second line there is node number 1's community id. Like these, all the community ids that are assigned to the nodes are listed in the output folder at the corresponding lines.

3. **Modularity function**

   There are 2 kinds of modularity functions offered by the algorithm. One of them is the standard modularity function that is proposed by Newman and Girvan (2004) and Newman (2004). [31] [32] The other one is an alternative modularity function that is defined by Traag, Van Dooren, and Nesterov (2011). [46] The user can choose which one to be used in the algorithm to optimize on.

4. **Resolution parameter**

   It determines the granularity level at which communities are detected.

5. **Optimization algorithm**

Louvain, Louvain with multilevel refinement and SLM algorithms are offered for the user.

6. **Number of random starts**

   The local moving heuristic method deals with the nodes in random order. So, each time it runs, it can give different output. This parameter specifies how many random starts are going to be tried.

7. **Number of iterations per random start**

   This parameter determines how many iterations of the chosen optimization algorithm are going to be run per random start.

8. **Seed of the random number generator**

All three optimization algorithms initially assign each node to a different community, so each node has its own singleton community. In order to make running the modularity optimizer dynamically possible, we thought that we should change the way of assigning initial communities to nodes. For this purpose, we defined a procedure called "initialize communities" that does following operations:

1. Take a file as a parameter that contains community ids of nodes that has been analyzed.
2. Read the file line by line and for each line do following tasks:
   (a) Declare current community id as the number read in the current line.
   (b) Find the node number to be assigned to the current community by counting the line numbers in the file.
   (c) Assign the found node to the current community.
3. Assign remaining nodes to singleton communities so that each of them has its own community.

Let's consider that we detected the communities in a network which has one million edges. After that, a new edge is added to this network. Now we want to detect the communities in this new network. If we run the optimization algorithms, they would

assign the nodes to singleton communities at the beginning of the algorithms, so they would start the optimization process from the beginning. However, we have the output of the community detection of the old network, so we actually do not need to start the optimization from scratch. We should feed the detected communities of old network to the "initialize communities" procedure instead of assigning the nodes to singleton communities at the beginning of the algorithm. Thus, we can begin the optimization process almost at the end of it and minimize the running time of the optimization algorithm. In addition, by calling our procedure, we do not generally need to iterate as much as other algorithms iterate in order to reach their modularity level. In other words, with our approach, less number of iterations is generally enough. In order to make sure that the resulting modularity values of the algorithms with and without our approach are equal, we made the iterations stop when the modularity value is reached to a specified value. We modified the modularity optimizer and added 3 parameters in order to give it the ability to be dynamic. These 3 parameters are described below:

1. **Is dynamic**

   This parameter determines whether the modularity optimizer is going to be run in the dynamic mode or not.

2. **Initial community ids**

   This is meaningful if the dynamic mode is on. It is the file that is going to be fed to "initialize communities" procedure.

3. **Expected modularity value**

   The algorithm stops the iterations, whenever the modularity value is reached to this parameter.

Briefly, we modified the modularity optimizer that is offered by Waltman and Van Eck to run dynamically if the above parameters are set. Thus, large networks can be analyzed by the modified modularity optimizer dynamically, since the running time is decreased by the modification we described above.

```
LouvainAlgorithm

input:
    A: Adjacency matrix of a network
    c: Initial assignment of nodes to communities
output:
    c: Final assignment of nodes to communities

// Run the local moving heuristic.
c ← LocalMovingHeuristic(A, c)

if NumberOfCommunities(c) < NumberOfNodes(A) then
    // Construct a reduced network.
    A_reduced ← ReducedNetwork(A, c)
    c_reduced ← [1...NumberOfNodes(A_reduced)]

    // Perform a recursive call to identify the community structure of the reduced network.
    c_reduced ← LouvainAlgorithm(A_reduced, c_reduced)

    // Merge communities based on the community structure of the reduced network.
    c_old ← c
    for i ← 1 to NumberOfCommunities(c_old) do
        c(c_old = i) ← c_reduced(i)
    end for
end if
```

Figure 4.1: Pseudocode of Louvain Algorithm [48]

LouvainAlgorithmWithMultilevelRefinement

```
input:
    A: Adjacency matrix of a network
    c: Initial assignment of nodes to communities
output:
    c: Final assignment of nodes to communities

// Run the local moving heuristic.
c ← LocalMovingHeuristic(A, c)

if NumberOfCommunities(c) < NumberOfNodes(A) then
    // Construct a reduced network.
    A_reduced ← ReducedNetwork(A, c)
    c_reduced ← [1...NumberOfNodes(A_reduced)]

    // Perform a recursive call to identify the community structure of the reduced network.
    c_reduced ← LouvainAlgorithmWithMultilevelRefinement(A_reduced, c_reduced)

    // Merge communities based on the community structure of the reduced network.
    c_old ← c
    for i ← 1 to NumberOfCommunities(c_old) do
        c(c_old = i) ← c_reduced(i)
    end for

    // Run the local moving heuristic.
    c ← LocalMovingHeuristic(A, c)
end if
```

Figure 4.2: Pseudocode of LMR Algorithm [48]

```
SmartLocalMovingAlgorithm

input:
    A: Adjacency matrix of a network
    c: Initial assignment of nodes to communities
output:
    c: Final assignment of nodes to communities

// Run the local moving heuristic.
c ← LocalMovingHeuristic(A, c)

if NumberOfCommunities(c) < NumberOfNodes(A) then
    // For each community, construct a subnetwork and run the local moving heuristic.
    // Construct a reduced network based on the community structure of the subnetworks.
    c_old ← c
    j ← 0
    for i ← 1 to NumberOfCommunities(c_old) do
        A_sub ← Subnetwork(A, c_old, i)
        c_sub ← [1…NumberOfNodes(A_sub)]
        c_sub ← LocalMovingHeuristic(A_sub, c_sub)
        c(c_old = i) ← c_sub + j
        c_reduced([j + 1]…[j + NumberOfCommunities(c_sub)]) ← i
        j ← j + NumberOfCommunities(c_sub)
    end for
    A_reduced ← ReducedNetwork(A, c)

    // Perform a recursive call to identify the community structure of the reduced network.
    c_reduced ← SmartLocalMovingAlgorithm(A_reduced, c_reduced)

    // Merge communities based on the community structure of the reduced network.
    c_old ← c
    for i ← 1 to NumberOfCommunities(c_old) do
        c(c_old = i) ← c_reduced(i)
    end for
end if
```

Figure 4.3: Pseudocode of SLM Algorithm [48]

CHAPTER 5

PREPROCESS

This report is conducted on the call detail record (CDR) data which includes the records of GPRS, SMS and calls made in Ankara within a month. This data is given to Middle Eastern Technical University for academic research purposes by a GSM operator. Let us explain the format of the CDR data over an example as shown below.

Example record:

5971,d821f2fd2efb1ad369bcd41413cda3d5,42,23543,

26e5a5b2f1e8b5425b5b075b873737c5,06,20120929,191852,mmo,,33

Every piece of data in the record is delimited by a comma as seen in the example.

5971 is the cell base station id of our GSM operator's customer.

d821f2fd2efb1ad369bcd41413cda3d5 is the hashed version of the phone number of our GSM operator's customer.

42 is the plate code of the city number of our GSM operator's customer.

23543 is the cell base station id of other GSM operator's customer.

26e5a5b2f1e8b5425b5b075b873737c5 is the hashed version of the phone number of other GSM operator's customer.

06 is the plate code of the city number of other GSM operator's customer.

20120929 is the call date which is translated as 29th September 2012.

191852 is the call time that is translated as 19:18:52.

mmo is the CDR type which means that d821f2fd2efb1ad369bcd41413cda3d5 calls 26e5a5b2f1e8b5425b5b075b873737c5.

mmt is also a CDR type. If it was mmt instead of mmo in this record, it would mean that 26e5a5b2f1e8b5425b5b075b873737c5 calls d821f2fd2efb1ad369bcd41413cda3d5. So, mmo and mmt determine the direction of the call.

33 is the duration of the call in seconds.

There are 297,009,183 such call records in the CDR data. We constructed a graph from these records. In the constructed graph, the vertices represent the hashed numbers of the members; in a way they represent the members of the GSM network. The edges in the graph represent the relations between the members. The weight of an edge represents the number of calls made between the members which place at the end points of the edge. For instance, if number x calls number y z times in the analyzed data, then the number of calls graph of analyzed data includes an edge between x and y with weight z. The resulting graph has 12,521,352 nodes, 56,316,192 edges, and approximately 3.8 GB space on disk. In other words, we see that there are 12,521,352 people talks to each other by phone and there are 56,316,192 phone call relations in Ankara. At this point an example record in the constructed graph is shown below:

00000003412c765ebe443f278e3efde1 61a3cd5af19cbd27562b2bcc82a7ee0d 3

00000003412c765ebe443f278e3efde1 **calls** 61a3cd5af19cbd27562b2bcc82a7ee0d 3 times. There are 56 million 316 thousand 191 more records like this in the network file.

After having the graph constructed as described above, we decided to assign an integer to the hashed numbers in order to decrease the size of the file on disk. We assign an integer value starting from 0 to every distinct hashed number and change the values on the graph. Moreover, we produced a new file in order to store the information that which integer value is assigned to which hashed number. After this process, we now have a directed and weighted graph file whose size is approximately 1 GB, and an integer value hashed number match information file whose size is 514 MB.

We do not actually need to know the direction of the call in order to detect the communities in the network. Therefore, we converted the network to an undirected network in two steps described as follows:

1. We read all the records in the graph file and construct a map which contains number pairs such as "number1 number2" as key and weights (number of times number1 calls number2) as value.
2. For each entry in the map do following steps:
   (a) We get the key such as "number1 number2" and reverse it to be "number2 number1".
   (b) Ask for if there is any entry whose key is this reversed version.
   (c) If there is an entry, we get the value of that entry and sum it with the value of the entry which we are currently on. Then, write the key of the entry which we are currently on and produced sum to the undirected graph file.
   (d) If there is not an entry with the reversed version, we just write the key and value pair of the entry which we are currently on to the undirected graph file.
   (e) We remove the processed map entries.

For instance, if number1 calls number2 3 times and number2 calls number1 6 times, the resulting undirected graph file contains a record like number1 number2 9 which means that number1 and number2 is related with weight 9. The resulting undirected graph has 12,521,352 nodes, 44,768,912 edges, and approximately 0.8 GB space on disk.

The base community detection algorithms that we are going to work on this thesis are implemented in the modularity optimizer that is proposed by Waltman and Van Eck. The modularity optimizer asks you to provide the name of an input file which contains the graph. The input file is a simple tab-delimited text file listing all pairs of nodes in a network that are connected by an edge. The numbering of nodes starts at 0. For each pair of nodes, the node with the lower index is listed first, followed by the node with the higher index. The lines in an input file are sorted based on the node

indices (first based on the indices in the first column, then based on the indices in the second column). [48] In the case of a weighted network, edge weights are provided in a third column.

In order to convert our undirected and weighted graph file to an input file that can be fed to the modularity optimizer, we sorted our graph file by first constructing a map, then writing the entries of the constructed map to a file that is going to be fed to the modularity optimizer. This was a rather challenging task than the previous preprocessing steps because we needed to keep the weight information while sorting the numbers in a way that the modularity optimizer wants as described above. We found the solution in constructing a map which contains first number as key and an inner map as value. Inner map contains second number as key and weight as value. The type of the maps we used is TreeMap which is an implementation of a map structure that keeps its entries sorted on their keys. Thus, our constructed map would be sorted firstly based on first number (outer TreeMap key sort), then based on second number (inner TreeMap key sort) just as wanted by the modularity optimizer. We read our graph file and construct the map we defined above. Afterwards, we traverse the entries in this constructed map and write firstly the key, then inner maps key and value as a record. Thus, we now have the sorted undirected and weighted graph file that can be fed to the modularity optimizer. This file has 12,521,352 nodes and 44,768,912 edges(lines).

We created following files and run the related experiments on them:

network 9,000,000: contains first 9,000,000 edges(lines) of whole network.

network 9,900,000: contains first 9,900,000 edges(lines) of whole network.

network 9,990,000: contains first 9,990,000 edges(lines) of whole network.

network 9,999,000: contains first 9,999,000 edges(lines) of whole network.

network 10,000,000: contains first 10,000,000 edges(lines) of whole network.

network 10,001,000: contains first 10,001,000 edges(lines) of whole network.

network 10,010,000: contains first 10,010,000 edges(lines) of whole network.

network 10,100,000: contains first 10,100,000 edges(lines) of whole network.

network 11,000,000: contains first 11,000,000 edges(lines) of whole network.

network 19,000,000: contains first 19,000,000 edges(lines) of whole network.

network 19,900,000: contains first 19,900,000 edges(lines) of whole network.

network 19,990,000: contains first 19,990,000 edges(lines) of whole network.

network 19,999,000: contains first 19,999,000 edges(lines) of whole network.

network 20,000,000: contains first 20,000,000 edges(lines) of whole network.

network 20,001,000: contains first 20,001,000 edges(lines) of whole network.

network 20,010,000: contains first 20,010,000 edges(lines) of whole network.

network 20,100,000: contains first 20,100,000 edges(lines) of whole network.

network 21,000,000: contains first 21,000,000 edges(lines) of whole network.

CHAPTER 6

EXPERIMENTS & RESULTS

6.1 Experiments on Louvain Algorithm in SMO & DMO

6.1.1 The size of the base network: 10,000,000 Edges

We run the Louvain algorithm on network 10,000,000 with one random start and ten iterations parameters. The algorithm converged in 4 iterations and produced a community structure which has 0.8887 modularity value in 40 seconds and 969 milliseconds. We called the output file that contains communities as communities 10,000,000 for the future use of it. We need this file in order to feed the dynamic Louvain algorithm. In order to measure the effects of our contribution to the Louvain algorithm implemented in the static modularity optimizer, we run both static and dynamic Louvain algorithms on same networks and compare results.

6.1.1.1 Additions

We run the static Louvain algorithm on network 10,001,000, network 10,010,000, network 10,100,000, network 11,000,000 with one random start and ten iterations parameters. The results are shown in Table 6.1.

We run the dynamic Louvain algorithm on network 10,001,000, network 10,010,000, network 10,100,000, network 11,000,000 with one random start and ten iterations parameters by using communities 10,000,000 file as the starting point. The results are shown in Table 6.2.

Table 6.1: Louvain Algorithm in SMO (Base: 10,000,000 Edges, Addition)

# of Edges Added	# of Iterations to Converge	Modularity Value	Running Time
1,000	4	0.8889	49 secs 23 millisecs
10,000	4	0.8888	48 secs 638 millisecs
100,000	4	0.8881	41 secs 70 millisecs
1,000,000	4	0.8830	54 secs 727 millisecs

Table 6.2: Louvain Algorithm in DMO (Base: 10,000,000 Edges, Addition)

# of Edges Added	# of Iterations to Converge	Modularity Value	Running Time
1,000	3	0.8887	14 secs 510 millisecs
10,000	3	0.8886	16 secs 781 millisecs
100,000	3	0.8873	18 secs 476 millisecs
1,000,000	3	0.8758	23 secs 26 millisecs

The results indicate that there is almost no loss in the quality of the community detection with respect to modularity values; however, the dynamic implementation reduces the running time by at least over than 56 % as shown in Table 6.3.

Table 6.3: DMO Effects to Louvain Algorithm (Base: 10,000,000 Edges, Addition)

# of Edges Added	Change in Modularity Value	Change in Running Time
1,000	0.02% decreased	71% decreased
10,000	0.02% decreased	66% decreased
100,000	0.09% decreased	56% decreased
1,000,000	0.81% decreased	57% decreased

6.1.1.2 Deletions

We run the static Louvain algorithm on network 9,999,000, network 9,990,000, network 9,900,000, network 9,000,000 with one random start and ten iterations parameters. The results are shown in Table 6.4.

We run the dynamic Louvain algorithm on network 9,999,000, network 9,990,000, network 9,900,000, network 9,000,000 with one random start and ten iterations parameters by using communities 10,000,000 file as the starting point. The results are shown in Table 6.5.

Table 6.4: Louvain Algorithm in SMO (Base: 10,000,000 Edges, Deletion)

# of Edges Deleted	# of Iterations to Converge	Modularity Value	Running Time
1,000	4	0.8888	52 secs 225 millisecs
10,000	4	0.8889	43 secs 624 millisecs
100,000	4	0.8895	50 secs 187 millisecs
1,000,000	4	0.8953	41 secs 300 millisecs

Table 6.5: Louvain Algorithm in DMO (Base: 10,000,000 Edges, Deletion)

# of Edges Deleted	# of Iterations to Converge	Modularity Value	Running Time
1,000	2	0.8887	8 secs 77 millisecs
10,000	3	0.8888	15 secs 163 millisecs
100,000	3	0.8892	16 secs 968 millisecs
1,000,000	3	0.8931	16 secs 455 millisecs

The results indicate that there is almost no loss in the quality of the community detection with respect to modularity values; however, the dynamic implementation reduces the running time by at least over than 61 % as shown in Table 6.6.

Table 6.6: DMO Effects to Louvain Algorithm (Base: 10,000,000 Edges, Deletion)

# of Edges Deleted	Change in Modularity Value	Change in Running Time
1,000	0.01% decreased	84% decreased
10,000	0.01% decreased	65% decreased
100,000	0.03% decreased	66% decreased
1,000,000	0.24% decreased	61% decreased

6.1.2 The size of the base network: 20,000,000 Edges

We run the Louvain algorithm on network 20,000,000 with one random start and ten iterations parameters. The algorithm converged in 4 iterations and produced a community structure which has 0.8465 modularity value in 1 minute 58 seconds and 127 milliseconds. We called the output file that contains communities as communities 20,000,000 for the future use of it.

6.1.2.1 Additions

We run the static Louvain algorithm on network 20,001,000, network 20,010,000, network 20,100,000, network 21,000,000 with one random start and ten iterations parameters. The results are shown in Table 6.7.

Table 6.7: Louvain Algorithm in SMO (Base: 20,000,000 Edges, Addition)

# of Edges Added	# of Iterations to Converge	Modularity Value	Running Time
1,000	4	0.8467	2 mins 50 millisecs
10,000	4	0.8466	1 min 45 secs 244 millisecs
100,000	4	0.8463	2 mins 19 secs 552 millisecs
1,000,000	4	0.8439	2 mins 26 secs 651 millisecs

We run the dynamic Louvain algorithm on network 20,001,000, network 20,010,000, network 20,100,000, network 21,000,000 with one random start and ten iterations parameters by using communities 20,000,000 file as the starting point. The results are shown in Table 6.8.

Table 6.8: Louvain Algorithm in DMO (Base: 20,000,000 Edges, Addition)

# of Edges Added	# of Iterations to Converge	Modularity Value	Running Time
1,000	3	0.8465	24 secs 248 millisecs
10,000	3	0.8465	28 secs 409 millisecs
100,000	3	0.8459	35 secs 630 millisecs
1,000,000	3	0.8405	41 secs 97 millisecs

The results indicate that there is almost no loss in the quality of the community detection with respect to modularity values; however, the dynamic implementation reduces the running time by at least over than 71 % as shown in Table 6.9.

Table 6.9: DMO Effects to Louvain Algorithm (Base: 20,000,000 Edges, Addition)

# of Edges Added	Change in Modularity Value	Change in Running Time
1,000	0.02% decreased	80% decreased
10,000	0.01% decreased	73% decreased
100,000	0.04% decreased	75% decreased
1,000,000	0.40% decreased	71% decreased

6.1.2.2 Deletions

We run the static Louvain algorithm on network 19,999,000, network 19,990,000, network 19,900,000, network 19,000,000 with one random start and ten iterations parameters. The results are shown in Table 6.10.

Table 6.10: Louvain Algorithm in SMO (Base: 20,000,000 Edges, Deletion)

# of Edges Deleted	# of Iterations to Converge	Modularity Value	Running Time
1,000	4	0.8466	1 min 52 secs 461 millisecs
10,000	4	0.8466	2 min 5 secs 345 millisecs
100,000	4	0.8468	1 min 52 secs 619 millisecs
1,000,000	4	0.8498	1 min 47 secs 752 millisecs

We run the dynamic Louvain algorithm on network 19,999,000, network 19,990,000, network 19,900,000, network 19,000,000 with one random start and ten iterations parameters by using communities 20,000,000 file as the starting point. The results are shown in Table 6.11.

Table 6.11: Louvain Algorithm in DMO (Base: 20,000,000 Edges, Deletion)

# of Edges Deleted	# of Iterations to Converge	Modularity Value	Running Time
1,000	2	0.8465	15 secs 303 millisecs
10,000	2	0.8465	20 secs 161 millisecs
100,000	3	0.8467	31 secs 800 millisecs
1,000,000	3	0.8483	31 secs 420 millisecs

The results indicate that there is almost no loss in the quality of the community detection with respect to modularity values; however, the dynamic implementation reduces the running time by at least over than 71 % as shown in Table 6.12.

Table 6.12: DMO Effects to Louvain Algorithm (Base: 20,000,000 Edges, Deletion)

# of Edges Deleted	Change in Modularity Value	Change in Running Time
1,000	0.01% decreased	86% decreased
10,000	0.01% decreased	84% decreased
100,000	0.01% decreased	71% decreased
1,000,000	0.17% decreased	71% decreased

6.2 Experiments on LMR Algorithm in SMO & DMO

6.2.1 The size of the base network: 10,000,000 Edges

We run the LMR algorithm on network 10,000,000 with one random start and ten iterations parameters. The algorithm converged in 3 iterations and produced a community structure which has 0.8887 modularity value in 42 seconds and 325 milliseconds. We called the output file that contains communities as communities 10,000,000 for the future use of it. We need this file in order to feed the dynamic LMR algorithm. In order to measure the effects of our contribution to the LMR algorithm implemented in the static modularity optimizer, we run both static and dynamic LMR algorithms on same networks and compare results.

6.2.1.1 Additions

We run the static LMR algorithm on network 10,001,000, network 10,010,000, network 10,100,000, network 11,000,000 with one random start and ten iterations parameters. The results are shown in Table 6.13.

Table 6.13: LMR Algorithm in SMO (Base: 10,000,000 Edges, Addition)

# of Edges Added	# of Iterations to Converge	Modularity Value	Running Time
1,000	3	0.8889	49 secs 331 millisecs
10,000	3	0.8888	51 secs 231 millisecs
100,000	3	0.8881	42 secs 499 millisecs
1,000,000	3	0.8830	56 secs 973 millisecs

We run the dynamic LMR algorithm on network 10,001,000, network 10,010,000, network 10,100,000, network 11,000,000 with one random start and ten iterations parameters by using communities 10,000,000 file as the starting point. The results are shown in Table 6.14.

The results indicate that there is almost no loss in the quality of the community detection with respect to modularity values; however, the dynamic implementation reduces the running time by at least over than 50 % as shown in Table 6.15.

Table 6.14: LMR Algorithm in DMO (Base: 10,000,000 Edges, Addition)

# of Edges Added	# of Iterations to Converge	Modularity Value	Running Time
1,000	2	0.8887	13 secs 871 millisecs
10,000	2	0.8886	16 secs 146 millisecs
100,000	2	0.8873	17 secs 814 millisecs
1,000,000	3	0.8758	27 secs 397 millisecs

Table 6.15: DMO Effects to LMR Algorithm (Base: 10,000,000 Edges, Addition)

# of Edges Added	Change in Modularity Value	Change in Running Time
1,000	0.02% decreased	73% decreased
10,000	0.02% decreased	68% decreased
100,000	0.09% decreased	59% decreased
1,000,000	0.81% decreased	51% decreased

6.2.1.2 Deletions

We run the static LMR algorithm on network 9,999,000, network 9,990,000, network 9,900,000, network 9,000,000 with one random start and ten iterations parameters. The results are shown in Table 6.16.

Table 6.16: LMR Algorithm in SMO (Base: 10,000,000 Edges, Deletion)

# of Edges Deleted	# of Iterations to Converge	Modularity Value	Running Time
1,000	3	0.8888	58 secs 325 millisecs
10,000	3	0.8889	48 secs 572 millisecs
100,000	3	0.8895	54 secs 552 millisecs
1,000,000	3	0.8953	44 secs 98 millisecs

We run the dynamic LMR algorithm on network 9,999,000, network 9,990,000, network 9,900,000, network 9,000,000 with one random start and ten iterations parameters by using communities 10,000,000 file as the starting point. The results are shown in Table 6.17.

The results indicate that there is almost no loss in the quality of the community detection with respect to modularity values; however, the dynamic implementation reduces the running time by at least over than 63 % as shown in Table 6.18.

Table 6.17: LMR Algorithm in DMO (Base: 10,000,000 Edges, Deletion)

# of Edges Deleted	# of Iterations to Converge	Modularity Value	Running Time
1,000	2	0.8887	8 secs 225 millisecs
10,000	2	0.8888	14 secs 522 millisecs
100,000	2	0.8892	16 secs 554 millisecs
1,000,000	2	0.8931	16 secs 7 millisecs

Table 6.18: DMO Effects to LMR Algorithm (Base: 10,000,000 Edges, Deletion)

# of Edges Deleted	Change in Modularity Value	Change in Running Time
1,000	0.01% decreased	86% decreased
10,000	0.01% decreased	70% decreased
100,000	0.03% decreased	70% decreased
1,000,000	0.24% decreased	63% decreased

6.2.2 The size of the base network: 20,000,000 Edges

We run the LMR algorithm on network 20,000,000 with one random start and ten iterations parameters. The algorithm converged in 3 iterations and produced a community structure which has 0.8465 modularity value in 2 minute 453 milliseconds. We called the output file that contains communities as communities 20,000,000 for the future use of it.

6.2.2.1 Additions

We run the static LMR algorithm on network 20,001,000, network 20,010,000, network 20,100,000, network 21,000,000 with one random start and ten iterations parameters. The results are shown in Table 6.19.

Table 6.19: LMR Algorithm in SMO (Base: 20,000,000 Edges, Addition)

# of Edges Added	# of Iterations to Converge	Modularity Value	Running Time
1,000	3	0.8467	2 mins 2 secs 301 millisecs
10,000	3	0.8466	1 min 46 secs 853 millisecs
100,000	3	0.8463	2 mins 21 secs 973 millisecs
1,000,000	3	0.8439	2 mins 30 secs 160 millisecs

We run the dynamic LMR algorithm on network 20,001,000, network 20,010,000,

network 20,100,000, network 21,000,000 with one random start and ten iterations parameters by using communities 20,000,000 file as the starting point. The results are shown in Table 6.20.

Table 6.20: LMR Algorithm in DMO (Base: 20,000,000 Edges, Addition)

# of Edges Added	# of Iterations to Converge	Modularity Value	Running Time
1,000	2	0.8465	23 secs 427 millisecs
10,000	2	0.8465	27 secs 561 millisecs
100,000	2	0.8459	34 secs 937 millisecs
1,000,000	2	0.8405	40 secs 352 millisecs

The results indicate that there is almost no loss in the quality of the community detection with respect to modularity values; however, the dynamic implementation reduces the running time by at least over than 72 % as shown in Table 6.21.

Table 6.21: DMO Effects to LMR Algorithm (Base: 20,000,000 Edges, Addition)

# of Edges Added	Change in Modularity Value	Change in Running Time
1,000	0.02% decreased	81% decreased
10,000	0.01% decreased	75% decreased
100,000	0.04% decreased	75% decreased
1,000,000	0.40% decreased	73% decreased

6.2.2.2 Deletions

We run the static LMR algorithm on network 19,999,000, network 19,990,000, network 19,900,000, network 19,000,000 with one random start and ten iterations parameters. The results are shown in Table 6.22.

Table 6.22: LMR Algorithm in SMO (Base: 20,000,000 Edges, Deletion)

# of Edges Deleted	# of Iterations to Converge	Modularity Value	Running Time
1,000	3	0.8466	1 min 47 secs 609 millisecs
10,000	3	0.8466	2 mins 4 secs 317 millisecs
100,000	3	0.8468	1 min 47 secs 733 millisecs
1,000,000	3	0.8498	1 min 40 secs 574 millisecs

We run the dynamic LMR algorithm on network 19,999,000, network 19,990,000, network 19,900,000, network 19,000,000 with one random start and ten iterations

parameters by using communities 20,000,000 file as the starting point. The results are shown in Table 6.23.

Table 6.23: LMR Algorithm in DMO (Base: 20,000,000 Edges, Deletion)

# of Edges Deleted	# of Iterations to Converge	Modularity Value	Running Time
1,000	2	0.8465	14 secs 721 millisecs
10,000	2	0.8465	20 secs
100,000	2	0.8467	30 secs 389 millisecs
1,000,000	2	0.8483	30 secs

The results indicate that there is almost no loss in the quality of the community detection with respect to modularity values; however, the dynamic implementation reduces the running time by at least over than 70 % as shown in Table 6.24.

Table 6.24: DMO Effects to LMR Algorithm (Base: 20,000,000 Edges, Deletion)

# of Edges Deleted	Change in Modularity Value	Change in Running Time
1,000	0.01% decreased	87% decreased
10,000	0.01% decreased	84% decreased
100,000	0.01% decreased	72% decreased
1,000,000	0.17% decreased	70% decreased

6.3 Experiments on SLM Algorithm in SMO & DMO

6.3.1 The size of the base network: 10,000,000 Edges

We run the SLM algorithm on network 10,000,000 with one random start and ten iterations parameters. The algorithm converged in 10 iterations and produced a community structure which has 0.8906 modularity value in 1 minute 56 seconds and 949 milliseconds. We called the output file that contains communities as communities 10,000,000 for the future use of it. We need this file in order to feed the dynamic SLM algorithm. In order to measure the effects of our contribution to the SLM algorithm implemented in the static modularity optimizer, we run both static and dynamic SLM algorithms on same networks and compare results.

6.3.1.1 Additions

We run the static SLM algorithm on network 10,001,000, network 10,010,000, network 10,100,000, network 11,000,000 with one random start and ten iterations parameters. The results are shown in Table 6.25.

Table 6.25: SLM Algorithm in SMO (Base: 10,000,000 Edges, Addition)

# of Edges Added	# of Iterations to Converge	Modularity Value	Running Time
1,000	10	0.8906	2 mins 2 secs 444 millisecs
10,000	10	0.8906	2 mins 4 secs 576 millisecs
100,000	10	0.8900	2 mins 200 millisecs
1,000,000	10	0.8848	2 mins 15 secs 454 millisecs

We run the dynamic SLM algorithm on network 10,001,000, network 10,010,000, network 10,100,000, network 11,000,000 with one random start and ten iterations parameters by using communities 10,000,000 file as the starting point. The results are shown in Table 6.26.

Table 6.26: SLM Algorithm in DMO (Base: 10,000,000 Edges, Addition)

# of Edges Added	# of Iterations to Converge	Modularity Value	Running Time
1,000	10	0.8906	1 min 29 secs 234 millisecs
10,000	10	0.8906	1 min 27 secs 678 millisecs
100,000	10	0.8900	1 min 36 secs 692 millisecs
1,000,000	10	0.8850	1 min 51 secs 228 millisecs

The results indicate that there is no loss, in one case a little gain, in the quality of the community detection with respect to modularity values; however, the dynamic implementation reduces the running time by at least over than 17 % as shown in Table 6.27.

Table 6.27: DMO Effects to SLM Algorithm (Base: 10,000,000 Edges, Addition)

# of Edges Added	Change in Modularity Value	Change in Running Time
1,000	no change	27% decreased
10,000	no change	29% decreased
100,000	no change	20% decreased
1,000,000	0.02% increased	17% decreased

At these runs, we see that the dynamic algorithm may reach the modularity values

that are resulted in static SLM algorithm runs in less than 10 iterations. Therefore, we run the dynamic SLM algorithm on network 10,001,000, network 10,010,000, network 10,100,000, network 11,000,000 with one random start and expected modularity value parameters dynamically by using communities 10,000,000 file as the starting point. The results are shown in Table 6.28.

Table 6.28: SLM Algorithm in DMO with Expected Modularity Value Set (Base: 10,000,000 Edges, Addition)

# of Edges Added	# of Iterations to Converge	Modularity Value	Running Time
1,000	1	0.8906	10 secs 764 millisecs
10,000	5	0.8906	46 secs 455 millisecs
100,000	4	0.8900	42 secs 883 millisecs
1,000,000	2	0.8848	26 secs 988 millisecs

The results indicate that there is no loss in the quality of the community detection with respect to modularity values; however, the dynamic implementation with the expected modularity value set to the result of static SLM algorithm reduces the running time by at least over than 63 % as shown in Table 6.29.

Table 6.29: Effects of DMO with Expected Modularity Value to the SLM Algorithm (Base: 10,000,000 Edges, Addition)

# of Edges Added	Change in Modularity Value	Change in Running Time
1,000	no change	91% decreased
10,000	no change	63% decreased
100,000	no change	64% decreased
1,000,000	no change	80% decreased

6.3.1.2 Deletions

We run the static SLM algorithm on network 9,999,000, network 9,990,000, network 9,900,000, network 9,000,000 with one random start and ten iterations parameters. The results are shown in Table 6.30.

We run the dynamic SLM algorithm on network 9,999,000, network 9,990,000, network 9,900,000, network 9,000,000 with one random start and ten iterations parameters by using communities 10,000,000 file as the starting point. The results are shown in Table 6.31.

Table 6.30: SLM Algorithm in SMO (Base: 10,000,000 Edges, Deletion)

# of Edges Deleted	# of Iterations to Converge	Modularity Value	Running Time
1,000	10	0.8906	2 mins 19 secs 923 millisecs
10,000	10	0.8907	1 min 54 secs 157 millisecs
100,000	10	0.8912	1 min 57 secs 905 millisecs
1,000,000	10	0.8969	1 min 41 secs 557 millisecs

Table 6.31: SLM Algorithm in DMO (Base: 10,000,000 Edges, Deletion)

# of Edges Deleted	# of Iterations to Converge	Modularity Value	Running Time
1,000	10	0.8906	1 min 25 secs 925 millisecs
10,000	10	0.8907	1 min 23 secs 40 millisecs
100,000	10	0.8913	1 min 29 secs 420 millisecs
1,000,000	10	0.8970	1 min 25 secs 130 millisecs

The results indicate that there is no loss, in two cases a little gain, in the quality of the community detection with respect to modularity values; however, the dynamic implementation reduces the running time by at least over than 16 % as shown in Table 6.32.

Table 6.32: DMO Effects to SLM Algorithm (Base: 10,000,000 Edges, Deletion)

# of Edges Deleted	Change in Modularity Value	Change in Running Time
1,000	no change	38% decreased
10,000	no change	27% decreased
100,000	0.01% increased	24% decreased
1,000,000	0.01% increased	16% decreased

At these runs, we see that the dynamic algorithm may reach the modularity values that are resulted in static SLM algorithm runs in less than 10 iterations. Therefore, we run the dynamic SLM algorithm on network 9,999,000, network 9,990,000, network 9,900,000, network 9,000,000 with one random start and expected modularity value parameters dynamically by using communities 10,000,000 file as the starting point. The results are shown in Table 6.33.

The results indicate that there is no loss in the quality of the community detection with respect to modularity values; however, the dynamic implementation with the expected modularity value set to the result of static SLM algorithm reduces the running time

Table 6.33: SLM Algorithm in DMO with Expected Modularity Value Set (Base: 10,000,000 Edges, Deletion)

# of Edges Deleted	# of Iterations to Converge	Modularity Value	Running Time
1,000	1	0.8906	11 secs 120 millisecs
10,000	5	0.8907	43 secs 968 millisecs
100,000	1	0.8912	10 secs 774 millisecs
1,000,000	2	0.8969	19 secs 996 millisecs

by at least over than 61 % as shown in Table 6.34.

Table 6.34: Effects of DMO with Expected Modularity Value to the SLM Algorithm (Base: 10,000,000 Edges, Deletion)

# of Edges Deleted	Change in Modularity Value	Change in Running Time
1,000	no change	92% decreased
10,000	no change	61% decreased
100,000	no change	91% decreased
1,000,000	no change	80% decreased

6.3.2 The size of the base network: 20,000,000 Edges

We run the SLM algorithm on network 20,000,000 with one random start and ten iterations parameters. The algorithm converged in 10 iterations and produced a community structure which has 0.8494 modularity value in 4 minutes 28 seconds and 183 milliseconds. We called the output file that contains communities as communities 20,000,000 for the future use of it.

6.3.2.1 Additions

We run the static SLM algorithm on network 20,001,000, network 20,010,000, network 20,100,000, network 21,000,000 with one random start and ten iterations parameters. The results are shown in Table 6.35.

We run the dynamic SLM algorithm on network 20,001,000, network 20,010,000, network 20,100,000, network 21,000,000 with one random start and ten iterations parameters by using communities 20,000,000 file as the starting point. The results are shown in Table 6.36.

Table 6.35: SLM Algorithm in SMO (Base: 20,000,000 Edges, Addition)

# of Edges Added	# of Iterations to Converge	Modularity Value	Running Time
1,000	10	0.8494	5 mins 27 secs 435 millisecs
10,000	10	0.8494	4 mins 43 secs 312 millisecs
100,000	10	0.8492	5 mins 38 secs 982 millisecs
1,000,000	10	0.8467	5 mins 18 secs 539 millisecs

Table 6.36: SLM Algorithm in DMO (Base: 20,000,000 Edges, Addition)

# of Edges Added	# of Iterations to Converge	Modularity Value	Running Time
1,000	10	0.8495	3 mins 18 secs 319 millisecs
10,000	10	0.8495	3 mins 11 secs 539 millisecs
100,000	10	0.8493	3 mins 27 secs 473 millisecs
1,000,000	10	0.8469	3 mins 38 secs 299 millisecs

The results indicate that there is no loss and a little gain in all cases in the quality of the community detection with respect to modularity values; however, the dynamic implementation reduces the running time by at least over than 31 % as shown in Table 6.37.

Table 6.37: DMO Effects to SLM Algorithm (Base: 20,000,000 Edges, Addition)

# of Edges Added	Change in Modularity Value	Change in Running Time
1,000	0.01% increased	39% decreased
10,000	0.01% increased	32% decreased
100,000	0.01% increased	38% decreased
1,000,000	0.02% increased	31% decreased

At these runs, we see that the dynamic algorithm may reach the modularity values that are resulted in static SLM algorithm runs in less than 10 iterations. Therefore, we run the dynamic SLM algorithm on network 20,001,000, network 20,010,000, network 20,100,000, network 21,000,000 with one random start and expected modularity value parameters dynamically by using communities 20,000,000 file as the starting point. The results are shown in Table 6.38.

The results indicate that there is no loss, in one case a little gain, in the quality of the community detection with respect to modularity values; however, the dynamic implementation with the expected modularity value set to the result of static SLM

Table 6.38: SLM Algorithm in DMO with Expected Modularity Value Set (Base: 20,000,000 Edges, Addition)

# of Edges Added	# of Iterations to Converge	Modularity Value	Running Time
1,000	1	0.8494	18 secs 676 millisecs
10,000	1	0.8494	22 secs 609 millisecs
100,000	1	0.8492	22 secs 365 millisecs
1,000,000	2	0.8468	50 secs 783 millisecs

algorithm reduces the running time by at least over than 84 % as shown in Table 6.39.

Table 6.39: Effects of DMO with Expected Modularity Value to the SLM Algorithm (Base: 20,000,000 Edges, Addition)

# of Edges Added	Change in Modularity Value	Change in Running Time
1,000	no change	94% decreased
10,000	no change	92% decreased
100,000	no change	93% decreased
1,000,000	0.01% increased	84% decreased

6.3.2.2 Deletions

We run the static SLM algorithm on network 19,999,000, network 19,990,000, network 19,900,000, network 19,000,000 with one random start and ten iterations parameters. The results are shown in Table 6.40.

Table 6.40: SLM Algorithm in SMO (Base: 20,000,000 Edges, Deletion)

# of Edges Deleted	# of Iterations to Converge	Modularity Value	Running Time
1,000	10	0.8494	4 mins 26 secs 590 millisecs
10,000	10	0.8494	4 mins 35 secs 30 millisecs
100,000	10	0.8497	4 mins 38 secs 470 millisecs
1,000,000	10	0.8524	4 mins 29 secs 429 millisecs

We run the dynamic SLM algorithm on network 19,999,000, network 19,990,000, network 19,900,000, network 19,000,000 with one random start and ten iterations parameters by using communities 20,000,000 file as the starting point. The results are shown in Table 6.41.

The results indicate that there is no loss and a little gain in all cases in the quality of the community detection with respect to modularity values; however, the dynamic

Table 6.41: SLM Algorithm in DMO (Base: 20,000,000 Edges, Deletion)

# of Edges Deleted	# of Iterations to Converge	Modularity Value	Running Time
1,000	10	0.8496	3 mins 9 secs 434 millisecs
10,000	10	0.8496	3 mins 8 secs 998 millisecs
100,000	10	0.8498	3 mins 10 secs 988 millisecs
1,000,000	10	0.8527	3 mins 9 secs 989 millisecs

implementation reduces the running time by at least over than 29 % as shown in Table 6.42.

Table 6.42: DMO Effects to SLM Algorithm (Base: 20,000,000 Edges, Deletion)

# of Edges Deleted	Change in Modularity Value	Change in Running Time
1,000	0.02% increased	29% decreased
10,000	0.02% increased	36% decreased
100,000	0.01% increased	31% decreased
1,000,000	0.03% increased	29% decreased

At these runs, we see that the dynamic algorithm may reach the modularity values that are resulted in static SLM algorithm runs in less than 10 iterations. Therefore, we run the dynamic SLM algorithm on network 19,999,000, network 19,990,000, network 19,900,000, network 19,000,000 with one random start and expected modularity value parameters dynamically by using communities 20,000,000 file as the starting point. The results are shown in Table 6.43.

Table 6.43: SLM Algorithm in DMO with Expected Modularity Value Set (Base: 20,000,000 Edges, Deletion)

# of Edges Deleted	# of Iterations to Converge	Modularity Value	Running Time
1,000	1	0.8495	18 secs 876 millisecs
10,000	1	0.8495	19 secs 625 millisecs
100,000	1	0.8497	20 secs 967 millisecs
1,000,000	1	0.8524	21 secs 138 millisecs

The results indicate that there is no loss, in two cases a little gain, in the quality of the community detection with respect to modularity values; however, the dynamic implementation with the expected modularity value set to the result of static SLM algorithm reduces the running time by at least over than 92 % as shown in Table 6.44.

Our overall contribution to SMO is demonstrated in Table 6.45 and Table 6.46.

Table 6.44: Effects of DMO with Expected Modularity Value to the SLM Algorithm (Base: 20,000,000 Edges, Deletion)

# of Edges Deleted	Change in Modularity Value	Change in Running Time
1,000	0.01% increased	93% decreased
10,000	0.01% increased	93% decreased
100,000	no change	92% decreased
1,000,000	no change	92% decreased

6.4 Implementation Notes

We conducted our experiments on a machine whose processor is Intel(R) Core(TM) i5-2400 running Windows 7 Professional 64-bit Operating System. The machine is clocked at 3.10 GHz, has 4 cores, 8 GB of RAM and 256 GB of SSD. Our algorithms and measures are implemented in Java 1.7.0 25.

Table 6.45: Overall contribution of DMO with respect to additions

Algorithm	Base (# of Edges)	# of Edges Added	Change in Modularity Value	Change in Running Time
Louvain	10,000,000	1,000	0.02% decreased	71% decreased
Louvain	10,000,000	10,000	0.02% decreased	66% decreased
Louvain	10,000,000	100,000	0.09% decreased	56% decreased
Louvain	10,000,000	1,000,000	0.81% decreased	57% decreased
Louvain	20,000,000	1,000	0.02% decreased	80% decreased
Louvain	20,000,000	10,000	0.01% decreased	73% decreased
Louvain	20,000,000	100,000	0.04% decreased	75% decreased
Louvain	20,000,000	1,000,000	0.40% decreased	71% decreased
LMR	10,000,000	1,000	0.02% decreased	73% decreased
LMR	10,000,000	10,000	0.02% decreased	68% decreased
LMR	10,000,000	100,000	0.09% decreased	59% decreased
LMR	10,000,000	1,000,000	0.81% decreased	51% decreased
LMR	20,000,000	1,000	0.02% decreased	81% decreased
LMR	20,000,000	10,000	0.01% decreased	75% decreased
LMR	20,000,000	100,000	0.04% decreased	75% decreased
LMR	20,000,000	1,000,000	0.40% decreased	73% decreased
SLM	10,000,000	1,000	no change	27% decreased
SLM	10,000,000	10,000	no change	29% decreased
SLM	10,000,000	100,000	no change	20% decreased
SLM	10,000,000	1,000,000	0.02% increased	17% decreased
SLM	20,000,000	1,000	0.01% increased	39% decreased
SLM	20,000,000	10,000	0.01% increased	32% decreased
SLM	20,000,000	100,000	0.01% increased	38% decreased
SLM	20,000,000	1,000,000	0.02% increased	31% decreased
SLMEVS	10,000,000	1,000	no change	91% decreased
SLMEVS	10,000,000	10,000	no change	63% decreased
SLMEVS	10,000,000	100,000	no change	64% decreased
SLMEVS	10,000,000	1,000,000	no change	80% decreased
SLMEVS	20,000,000	1,000	no change	94% decreased
SLMEVS	20,000,000	10,000	no change	92% decreased
SLMEVS	20,000,000	100,000	no change	93% decreased
SLMEVS	20,000,000	1,000,000	0.01% increased	84% decreased

Table 6.46: Overall contribution of DMO with respect to deletions

Algorithm	Base (# of Edges)	# of Edges Deleted	Change in Modularity Value	Change in Running Time
Louvain	10,000,000	1,000	0.01% decreased	84% decreased
Louvain	10,000,000	10,000	0.01% decreased	65% decreased
Louvain	10,000,000	100,000	0.03% decreased	66% decreased
Louvain	10,000,000	1,000,000	0.24% decreased	61% decreased
Louvain	20,000,000	1,000	0.01% decreased	86% decreased
Louvain	20,000,000	10,000	0.01% decreased	84% decreased
Louvain	20,000,000	100,000	0.01% decreased	71% decreased
Louvain	20,000,000	1,000,000	0.17% decreased	71% decreased
LMR	10,000,000	1,000	0.01% decreased	86% decreased
LMR	10,000,000	10,000	0.01% decreased	70% decreased
LMR	10,000,000	100,000	0.03% decreased	70% decreased
LMR	10,000,000	1,000,000	0.24% decreased	63% decreased
LMR	20,000,000	1,000	0.01% decreased	87% decreased
LMR	20,000,000	10,000	0.01% decreased	84% decreased
LMR	20,000,000	100,000	0.01% decreased	72% decreased
LMR	20,000,000	1,000,000	0.17% decreased	70% decreased
SLM	10,000,000	1,000	no change	38% decreased
SLM	10,000,000	10,000	no change	27% decreased
SLM	10,000,000	100,000	0.01% increased	24% decreased
SLM	10,000,000	1,000,000	0.01% increased	16% decreased
SLM	20,000,000	1,000	0.02% increased	29% decreased
SLM	20,000,000	10,000	0.02% increased	36% decreased
SLM	20,000,000	100,000	0.01% increased	31% decreased
SLM	20,000,000	1,000,000	0.03% increased	29% decreased
SLMEVS	10,000,000	1,000	no change	92% decreased
SLMEVS	10,000,000	10,000	no change	61% decreased
SLMEVS	10,000,000	100,000	no change	91% decreased
SLMEVS	10,000,000	1,000,000	no change	80% decreased
SLMEVS	20,000,000	1,000	0.01% increased	93% decreased
SLMEVS	20,000,000	10,000	0.01% increased	93% decreased
SLMEVS	20,000,000	100,000	no change	92% decreased
SLMEVS	20,000,000	1,000,000	no change	92% decreased

CHAPTER 7

CONCLUSION

In this chapter, a summary of our approach is explained in the first section. Then, the advantages and disadvantages of the results of our experiments are given in Discussion section. Finally, what can be done as future work is described in the last section.

7.1 Summary of Conducted Approach

Waltman & Van Eck proposed and implemented a modularity optimizer in order to detect communities in large networks in 2013. We extended their implementation to define the community structure in a dynamic rather than static way. We made use of the past calculation results of the static modularity optimizer in order to calculate the current networks community structure. This usage is the main extension and contribution to the static modularity optimizer. In the basics, it is what extends the static modularity optimizer to be dynamic modularity optimizer.

Each node is initialized to be a singleton community in the static modularity optimizer. We changed this initialization step so that each node is assigned to its community that is computed in the past results of the algorithm. If the node is new to the particular network, its community is not computed yet; therefore, we initialize the newly added nodes as they have their own singleton communities. This is how we use the past results of community detection algorithms in order to extend them to be incremental and dynamic.

SMO runs the community detection algorithms iteratively until they converge. It takes the number of iterations as a parameter before running the specified community detection algorithm. Thus, it can stop running at the given iteration even if the algorithm has not converged yet. In our experiments, we chose to compare the running time and resulting networks modularity values of SMO and DMO. Both SMO and DMO take modularity function, resolution parameter, number of random starts, number of iterations per random start and seed of the random number generator as parameters. We decided to run both SMO and DMO with the same values assigned to these parameters in order to have a fair comparison.

We chose the original modularity function that is defined by Newman and Girvan in 2004 instead of alternative modularity function that is introduced by Traag, Van Dooren, and Nesterov in 2011. We used 1.0 as the resolution parameter because the value of 1.0 is for standard modularity-based community detection. The algorithm results a larger number of communities, if resolution parameter value is set to be above 1.0. Otherwise, it results a smaller number of communities. We set number of random starts to be 1 and number of iterations per random start to be 10. We set seed of the random number generator, which is used for determining the random order of nodes to walk on, as 0.

In the experiments, Louvain and LMR algorithms converged before 10 iterations. However, SLM algorithm did not converge and kept on increasing the modularity value of the given network until 10 iterations. Then algorithm stopped running anymore iterations. Some of the experiments of SLM algorithm resulted in modularity increase. This leads us to think about how much running time would DMO need in order to reach a modularity value same as SMO. In order to figure this out, we decided to stop DMO whenever it reaches the modularity value of the network that is partitioned by SMO. Therefore, we added a parameter that enables the algorithm to stop when the given network has reached the expected modularity value, in addition to number of iterations parameter. These two parameters enable the user to specify an expected community structure in two different ways. The running time of DMO is drastically decreased by setting the expected modularity value to the resulting modularity value of SLM algorithm. Thus, the modularity value does not change; however, the required time to analyze the network is decreased around 90 %. It can be inferred

from these experiments of SLM that the expected value of modularity parameter can be set if decrease in running time is more significant than the increase in modularity value.

To sum up, we extended SMO to be incremental and dynamic by using the historical results of community detection algorithms for the initial community assignments of the nodes. Thus, the number of node movement actions tried to maximize the modularity value is decreased. This led to decrease in running time of the algorithms. Moreover, it can lead to decrease in number of iterations to converge. Thus, if the algorithms run with a constant number of iterations parameter, the modularity value may result as increased.

7.2 Discussion

SMO and DMO both include 3 modularity based community detection algorithms which are called as Louvain, LMR and SLM. We extended SLM to take expected modularity values as parameter and called the new version as SLM with expected modularity value set (SLMEVS). We made experiments on all these 4 algorithms. Before conducting the experiments, we constructed the networks that are going to be partitioned by the algorithms. These networks are extracted from the call data records taken from a GSM company. These records contain the calls made by the users who live in Ankara within a month. The experiments are firstly categorized by the size of the base network that is going to be analyzed. We chose the sizes of the base networks to analyze as 10,000,000 and 20,000,000. After this classification, as the second step, we classify experiments on the network evolution method such as edge addition and edge deletion. In each experiment, the number of the edges deleted or added can be 1,000 and 10,000 and 100,000 and 1,000,000. Thus, there are $4 * 2 * 2 * 4 = 64$ experiments conducted in total.

In SMO, in each iteration, nodes are moved among communities in order to construct a partitioned network whose modularity value is optimum. The number of the node movements in SMO can be quite high because it starts the first iteration assuming each node has its own community which is very unlikely in the real network. How-

ever, in DMO, reaching the optimum modularity value with the current version of the network requires much less node movements in each iteration since the nodes of previous version of the network are already assigned to communities such that the community structure has the maximum modularity value and way more realistic. Therefore, each iteration of Louvain and LMR algorithms requires much less number of node movements and time in DMO than they are in SMO. Moreover, the experiments that we made on Louvain and LMR algoirthms demonstrate that the algorithms in DMO generally converges to the maximum modularity value in less iterations than the algorithms in SMO. This can be explained by the fact that the algorithms in DMO start the iterations with the advantage of historical data usage whereas the algorithms in SMO start the iterations from scratch every time. Briefly, thanks to our contribution, Louvain and LMR algorithms have 2 performance boosts. One of them is that they require much less number of node movements in each iteration; thus the time of each iteration is dramatically decreased. The second one is that they need less number of iterations to converge the optimum modularity value as shown in the experiments. Combining these 2 outcomes of our contribution to Louvain and LMR algorithms, their overall running time, which is effected by the time of each iteration and the total number of iterations, is dramatically decreased. This dramatic decrease in running time of the algorithms can be seen all together in Table 6.45 and Table 6.46. In addition to the change in running times of the algorithms, these tables also show the change in modularity values of the networks analyzed by the algorithms. There are examples of the modularity values increased, not changed and decreased in the experiments as seen in Table 6.45 and Table 6.46. The decreases in the modularity values never cross the 1% boundary and usually much less than 1%. Considering the amount of running time improvement, this very little amount of modularity value decrease, only in some experiments, may easily be negligible.

It can be inferred from the experiments that the modularity values of the networks that are analyzed by iterative Louvain and LMR algorithms are same under same conditions. Furthermore, the overall running times of iterative Louvain and LMR algorithms is very close to each other under same conditions. These inferences can be understand by comparing for example Table 6.1 and Table 6.13 or Table 6.2 and Table 6.14. This situation is valid for both SMO and DMO and it can be seen as abnormal

because it is known that LMR is the improved version of Louvain algorithm. An important factor to remind at this point is that SMO and DMO runs all its community detection algorithms multiple times iteratively and each time is called as an iteration. The number of iterations is given as parameter to both SMO and DMO. LMR algorithm improves the Louvain algorithm by running extra local moving heuristic method at the last stage of each recursion step. Thus, LMR algorithm partitions networks such that they have greater modularity values compared to ones analyzed by Louvain algorithm; nevertheless, its running time is also greater than Louvain algorithms running time. In other words, LMR algorithm increases the quality of partitions whereas it takes more time to run. When we run these algorithms iteratively by using SMO and DMO, we realized that LMR algorithm converges in less iterations than Louvain algorithm by comparing for example Table 6.1 and Table 6.13 or Table 6.2 and Table 6.14. It is because LMR gains more modularity value in each iteration than Louvain and cannot survive from converging to the same modularity value. LMR converges in less iteration; but, each iteration of LMR takes more time than each iteration of Louvain. Therefore, these advantages and drawbacks of LMR algorithm against Louvan algorithm compensate each other and make their results similar when they are run iteratively.

SLM is not like Louvain and LMR algorithms with respect to the convergence. It takes more number of iterations to converge for SLM algorithm, since SLM algorithm seeks the modularity increase in more number of ways. Thus, each iteration of SLM algorithm in the experiments resulted as modularity increase and did not converge at least in 10 iterations in both SMO and DMO. However, both LMR and Louvain algorithms do not have any run that lasts all 10 iterations. DMO cannot decrease the number iterations of convergence of SLM, however, it decreases the number of node movements needed in each iteration of SLM. This indicates that each iteration of SLM runs faster in DMO than SMO. Therefore, overall running time of SLM decreases in DMO compared to SMO; nonetheless, this decrease is not as much as it was in the experiments of Louvain and LMR algorithms. It is because of the fact that the number of iterations to converge remains same in DMO and SMO for SLM whereas it decrease for Louvain and LMR. In addition to little running time improvement, DMO contributes to SLM algorithm in SMO by using historical data for community initial-

ization and increasing the modularity values of the resulted networks. The overall results can be seen in Table 6.45 and Table 6.46.

In order to be able to decrease the overall running time of SLM algorithm even more, we added another parameter called expected modularity value that enables the algorithm stop when it is reached. We named this kind of new algorithm as SLMEVS and made same experiments on it with this new parameter set to the modularity value resulted from SLM algorithm in SMO. By this new algorithm and parameter, we aimed to decrease running time as much as possible while keeping the modularity value unchanged or increased. We reached our aim and decreased running time drastically and keep modularity value unchanged or increased as seen in Table 6.45 and Table 6.46.

7.3 Future Work

SMO and DMO are designed in such a way that they can be applied to all community detection algorithms which can give better results when they are run iteratively. However, we have only implemented three of these kinds of algorithms which are Louvain, LMR and SLM algorithms. As one of the future works, new community detection algorithms can be implemented in both SMO and DMO; thus, both static and dynamic performances of these new algorithms when they are run iteratively can be analyzed. Apart from extending the number of community detection algorithms implemented, the data sets that the algorithms are experimented on can be extended in order to get a better understanding on the performances of the implemented algorithms. In this thesis work, we chose to experiment the algorithms on the graphs with various sizes that are constructed by preprocessing a GSM networks CDR data. In addition to GSM networks, the networks that are used in the paper [48] that introduces SLM can be used in order to compare the results with them. They run Louvain, LMR and SLM algorithms iteratively and statically on the networks such as Amazon, DBLP, IMDB, LiveJournal and so on. DMO can also be experimented on all of these networks in the future.

REFERENCES

[1] Richard D. Alba. A graph-theoretic definition of a sociometric clique. *Journal of Mathematical Sciology*, 3:113–126, 1973.

[2] Daniel Aloise, Sonia Cafieri, Gilles Caporossi, Pierre Hansen, Leo Liberti, and Sylvain Perron. Column generation algorithms for exact modularity maximization in networks. *Physical Review E*, 82(4, article):–, jan 2010.

[3] Alex Arenas, J. Duch, A. Fernandez, and Sergio Gómez. Size reduction of complex networks preserving modularity. *CoRR*, abs/physics/0702015, 2007.

[4] Yuichi Asahiro, Refael Hassin, and Kazuo Iwama. Complexity of finding dense subgraphs. *Discrete Applied Mathematics*, 121(1-3):15–26, 2002.

[5] V.D. Blondel, J.L. Guillaume, R. Lambiotte, and E.L.J.S. Mech. Fast unfolding of communities in large networks. *J. Stat. Mech*, page P10008, 2008.

[6] Immanuel M Bomze, Marco Budinich, Panos M Pardalos, and Marcello Pelillo. The maximum clique problem. In *Handb. Comb. Optim.*, volume 4, pages 1–74, 1999.

[7] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Goerke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. On modularity clustering. *IEEE Transactions on Knowledge and Data Engineering*, 20(2):172–188, 2008.

[8] P. Brodka, S. Saganowski, and P. Kazienko. Group evolution discovery in social networks. In *Advances in Social Networks Analysis and Mining (ASONAM), 2011 International Conference on*, pages 247–253, 2011.

[9] Coenraad Bron and Joep Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.

[10] Remy Cazabet, Frédéric Amblard, and Chihab Hanachi. Detection of overlapping communities in dynamical social networks. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *SocialCom/PASSAT*, pages 309–314. IEEE Computer Society, 2010.

[11] Deepayan Chakrabarti, Ravi Kumar, and Andrew Tomkins. Evolutionary clustering. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD ’06, pages 554–560, New York, NY, USA, 2006. ACM.

[12] S.Y. Chan, P. Hui, and K. Xu. Community detection of time-varying mobile social networks. *Complex Sciences*, pages 1154–1159, 2009.

[13] Aaron Clauset, M.E.J. Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical Review E*, 70:066111, 2004.

[14] J. Duch and A. Arenas. Community detection in complex networks using extremal optimization. *Physical Review E*, 72:027104, 2005.

[15] Uriel Feige, Guy Kortsarz, and David Peleg. The dense k-subgraph problem. *Algorithmica*, 29(3):410–421, 2001.

[16] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486:75–174, 2010.

[17] Santo Fortunato and Marc Barthélemy. Resolution limit in community detection. *Proceedings of the National Acadamy of Sciences of the United States of America (PNAS)*, 104(1):36–41, 2007.

[18] M. Garey and D. Johnson. *Computers and Intractability - A guide to the Theory of NP-Completeness*. Freeman, San Fransisco, 1979.

[19] Derek Greene, Dónal Doyle, and Padraig Cunningham. Tracking the evolution of communities in dynamic social networks. In Nasrullah Memon and Reda Alhajj, editors, *ASONAM*, pages 176–183. IEEE Computer Society, 2010.

[20] S Gregory. Finding overlapping communities in networks by label propagation. *New Journal of Physics*, 12:1–26, 2010.

[21] R. Guimera, M. Sales-Pardo, and L.A.N. Amaral. Modularity from fluctuations in random graphs and complex networks. *Physical Review E*, 70(2):025101, 2004.

[22] Petter Holme and Jari Saramäki. Temporal networks. *Physics Reports*, 519(3):97–125, 2012.

[23] Klaus Holzapfel, Sven Kosub, Moritz G. Maaß, and Hanjo Täubig. The complexity of detecting fixed-density clusters. In Rossella Petreschi, Giuseppe Persiano, and Riccardo Silvestri, editors, *CIAC*, volume 2653 of *Lecture Notes in Computer Science*, pages 201–212. Springer, 2003.

[24] Min-Soo Kim and Jiawei Han. A particle-and-density based evolutionary clustering method for dynamic networks. *PVLDB*, 2(1):622–633, 2009.

[25] Andrea Lancichinetti, Filippo Radicchi, Jose J. Ramasco, and Santo Fortunato. Finding statistically significant communities in networks. *CoRR*, abs/1012.2363, 2010.

[26] Juyong Lee, Steven P. Gross, and Jooyoung Lee. Mod-csa: Modularity optimization by conformational space annealing. *CoRR*, abs/1202.5398, 2012.

[27] S. Lehmann and L. K. Hansen. Deterministic modularity optimization. *The European Physical Journal B*, 60(1):83–88, 2007.

[28] E. A. Leicht and M. E. J. Newman. Community structure in directed networks. *Phys. Rev. Lett.*, 100(11):118703, March 2008.

[29] Yu-Ru Lin, Yun Chi, Shenghuo Zhu, Hari Sundaram, and Belle L. Tseng. Analyzing communities and their evolutions in dynamic social networks. *TKDD*, 3(2), 2009.

[30] M. E. J. Newman. Analysis of weighted networks. *Phys. Rev. E*, 70(5):056131, November 2004.

[31] M. E. J. Newman. Fast algorithm for detecting community structure in networks. *Physical Review*, E 69(066133), 2004.

[32] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Phys. Rev. E*, 69(2):026113, February 2004.

[33] MEJ Newman. Finding community structure in networks using the eigenvectors of matrices. *Physical Review E*, 74(3):36104, 2006.

[34] MEJ Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, 2006.

[35] Nam P. Nguyen, Thang N. Dinh, Sindhura Tokala, and My T. Thai. Overlapping communities in dynamic networks: their detection and mobile applications. In Parmesh Ramanathan, Thyaga Nandagopal, and Brian Neil Levine, editors, *MOBICOM*, pages 85–96. ACM, 2011.

[36] Günce Keziban Orman, Vincent Labatut, and Hocine Cherifi. Comparative evaluation of community detection algorithms: A topological approach. *CoRR*, abs/1206.4987, 2012.

[37] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435:814–818, 2005.

[38] Filippo Radicchi, Claudio Castellano, Federico Cecconi, Vittorio Loreto, and Domenico Parisi. Defining and identifying communities in networks. *PNAS*, 101(9):2658–2663, March 2004.

[39] Usha N. Raghavan, Reka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks, September 2007.

[40] J. Reichardt and S. Bornholdt. Statistical mechanics of community detection. *Arxiv preprint cond-mat/0603718*, 2006.

[41] M. Rosvall and C.T. Bergstrom. An information-theoretic framework for resolving community structure in complex networks. *Proceedings of the National Academy of Sciences*, 104(18):7327, 2007.

[42] Martin Rosvall and Carl T. Bergstrom. Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences*, 105(4):1118–1123, 2008.

[43] Randolf Rotta and Andreas Noack. Multilevel local search algorithms for modularity clustering. *ACM Journal of Experimental Algorithmics*, 16, 2011.

[44] Lei Tang and Huan Liu. *Community Detection and Mining in Social Media*. Synthesis Lectures on Data Mining and Knowledge Discovery. Morgan and Claypool Publishers, 2010.

[45] Xuning Tang and Christopher C. Yang. Dynamic community detection with temporal dirichlet process. In *SocialCom/PASSAT*, pages 603–608. IEEE, 2011.

[46] Vincent A Traag, Paul Van Dooren, and Yurii Nesterov. Narrow scope for resolution-limit-free community detection. *Physical Review E*, 84(1):016114, 2011.

[47] Stijn Van Dongen. Graph clustering via a discrete uncoupling process. *SIAM J. Matrix Anal. Appl.*, 30(1):121–141, February 2008.

[48] Ludo Waltman and Nees Jan van Eck. A smart local moving algorithm for large-scale modularity-based community detection. *CoRR*, abs/1308.6604, 2013.

[49] Ludo Waltman, Nees Jan van Eck, and Ed C.M. Noyons. A unified approach to mapping and clustering of bibliometric networks. *Journal of Informetrics*, 4(4):629–635, 2010.

[50] Zhi Wang and Jianzhi Zhang. In search of the biological significance of modular structures in protein networks. *PLoS Computational Biology*, 3(6), 2007.

[51] Jierui Xie and Boleslaw K. Szymanski. Labelrank: A stabilized label propagation algorithm for community detection in networks. *CoRR*, abs/1303.0868, 2013.

[52] Jierui Xie, Boleslaw K. Szymanski, and Xiaoming Liu. Slpa: Uncovering overlapping communities in social networks via a speaker-listener interaction dynamic process. *CoRR*, abs/1109.5720, 2011.

[53] G. Xu, S. Tsoka, and L. G. Papageorgiou. Finding community structures in complex networks using mixed integer optimisation. *The European Physical Journal B*, 60(2):231–239, 2007.

[54] Hao Xu, Yan-Li Hu, Zhenwen Wang, Jianwei Ma, and Weidong Xiao. Core-based dynamic community detection in mobile social networks. *Entropy*, 15(12):5419–5438, 2013.

[55] Tianbao Yang, Yun Chi, Shenghuo Zhu, Yihong Gong, and Rong Jin. Detecting communities and their evolutions in dynamic social networks - a bayesian approach. *Machine Learning*, 82(2):157–189, 2011.