RNR: REDUCE AND RAISE - BOTTOM-UP COMMUNITY DETECTION IN BIPARTITE AND TRIPARTITE GRAPHS

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

BY

KADIR EKMEKCI

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

FEBRUARY 2019

Approval of the thesis:

RNR: REDUCE AND RAISE - BOTTOM-UP COMMUNITY DETECTION IN BIPARTITE AND TRIPARTITE GRAPHS

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

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Halit Oğuztüzün
Head of Department, **Computer Engineering** _____

Prof. Dr. Göktürk Üçoluk
Supervisor, **Computer Engineering, METU** _____

Prof. Dr. İsmail Hakkı Toroslu
Co-supervisor, **Computer Engineering, METU** _____

Examining Committee Members:

Prof. Dr. Pınar Karagöz
Computer Engineering, METU _____

Prof. Dr. Göktürk Üçoluk
Computer Engineering, METU _____

Assoc. Prof. Dr. Tansel Dökeroğlu
Computer Engineering, Ted University _____

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, Surname: Kadir Ekmekci

Signature :

ABSTRACT

RNR: REDUCE AND RAISE - BOTTOM-UP COMMUNITY DETECTION IN BIPARTITE AND TRIPARTITE GRAPHS

Ekmekci, Kadir
M.S., Department of Computer Engineering
Supervisor: Prof. Dr. Göktürk Üçoluk
Co-Supervisor: Prof. Dr. İsmail Hakkı Toroslu

February 2019, 63 pages

In this thesis, we propose a novel approach to solve the near-biclique detection and near-triclique detection problems. Clique detection in graphs is proven to be $NP-hard$. There are various heuristics, and different approaches to deal with this problem. Our approach differs from the existing algorithms in a sense that it aims to find near-cliques that include nodes that are higher in degree not just in the clique but also in the whole graph. We make a trade-off between speed and quality, in favor of speed. In this thesis, we explain and compare seven novel algorithms, which are developed upon the existing algorithm, the popular Louvain method, which is a modularity-based community detection algorithm. The algorithms are tested using computer generated graphs. We observe that our algorithms perform better for the sparse networks with dense regions.

Keywords: tripartite graph, bipartite graph, bipartite matching, tripartite matching, graph theory, stable marriage problem, Louvain methond, community, near-clique,

modularity

ÖZ

İVY: İNDİRGE VE YÜKSELT - İKİ PARÇALI VE ÜÇ PARÇALI ÇİZGELERDE TABANDAN BAŞLAYARAK KOMÜNİTE BULMA

Ekmekci, Kadir
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Göktürk Üçoluk
Ortak Tez Yöneticisi: Prof. Dr. İsmail Hakkı Toroslu

Şubat 2019 , 63 sayfa

Bu tezde, iki parçalı yakın-klik bulma ve üç parçali yakın-klik bulma problemlerine özgün ve hızlı bir yaklaşım sunuyoruz. Klik bulma probleminin $NP - zor$ olduğu kanıtlandı. Bu problemi çözmek için birçok buluşsal algoritma ve degisik yaklasimlar sunuldu. Bizim yaklaşımımız diğer yöntemlerden şu şekilde ayrışmaktadır: Bizim algoritmamız yüksek dereceli düğümlerin bir arada toplandığı yakın-klikleri bulmayı amaçlıyor. Hız için kaliteden ödün verdigimiz bu tezde yedi tane özgün algoritmayı kıyaslayıp, açıklıyoruz. Bu algoritmalar, varolan bir algoritma olan Louvain metodu kullanılarak geliştirilmişlerdir. Louvain metodu modüler-bazlı bir komünite bulma algoritmasıdır. Algoritmalarımızı bilgisayar ile üretilmiş çizgelerle test ettik. Gözlemlerimize göre, algoritmalarımız yoğun bölgeleri olan ve seyrek çizgelerde daha iyi çalışıyor.

Anahtar Kelimeler: üç parçalı çizge, iki parçalı çizge, iki parçalı eşlestirme, üç parçalı eşleştirme, çizge teorisi,stabil evlilik problemi, Louvain metodu, komünite, yaklaşık

klik, modülerlik

I dedicate this thesis to my family and friends, who are always there for me through thick and thin.

ACKNOWLEDGMENTS

First of all, I would like to thank both of my supervisors, Prof. Dr. İsmail Hakkı Toroslu and Prof. Dr. Göktürk Üçoluk for their support and insights throughout this journey, giving me the time I needed to work, and for their direction and feedback. Their advices were the most valuable. I also thank my thesis committee members for taking time off for me, and their valuable commentaries and suggestions.

I am very thankful to my friends for their support for encouragement. I would like to thank Taylan Doğan particularly for his psychological support.

I would like to thank my family for their endless support and unconditional love. To my mom who taught me to be positive and open-minded during the toughest times.

Lastly, I would like to thank my dad who is always in my thoughts, and started the fire in me for music, computers and science. Even after your departure, I still feel your helping hand in every aspect of my life.

TABLE OF CONTENTS

ABSTRACT v

ÖZ vii

ACKNOWLEDGMENTS x

TABLE OF CONTENTS xi

LIST OF TABLES xiv

LIST OF FIGURES xv

LIST OF ABBREVIATIONS xvii

CHAPTERS

1 INTRODUCTION 1

2 RELATED WORK 5

2.1 Community Detection Primer 6

2.1.1 Community 6

2.1.2 Complexity 6

2.2 Community Detection 7

2.2.1 Graph Partitioning 7

2.2.2 Hierarchical Clustering 8

2.2.3 Partitional Clustering 8

2.2.4 Spectral Clustering 9

2.2.5 Divisive Algorithms 10
2.2.6 Modularity-Based Methods 10
3 METHODOLOGY 11
3.1 Proposed Algorithms 11
3.2 Stages of the Algorithms 17
3.2.1 Reduction 17
3.2.1.1 Calculating the Priorities 17
3.2.1.2 Matching 20
3.2.1.3 Reduce 26
3.2.2 Finding Near-Cliques 27
3.2.2.1 Modularity 27
3.2.2.2 Local Moving Heuristic 29
3.2.2.3 Louvain Algorithm 30
3.2.3 Raise Stage 31
3.3 Complexity and Constraints 32
3.3.1 Reduce 32
3.3.1.1 Calculating the Priority Lists 32
3.3.1.2 Matching Algorithms 33
3.3.1.3 Merge 33
3.3.1.4 Reduce 34
3.3.2 Louvain Algorithm 34
3.3.3 Raise 34
3.3.4 Near-Biclique Detection Algorithms 34

3.3.5 Near-Triclique Detection Algorithms 35
3.4 Advantages 36
4 EXPERIMENTS AND RESULTS 37
4.1 Experiment Data 37
4.2 Experiment Metrics 38
4.2.1 Modularity 38
4.3 Correctness 39
4.4 Results 41
4.4.1 Bipartite Graphs 41
4.4.2 Tripartite Graphs 46
5 CONCLUSION AND FUTURE WORK 57
REFERENCES 59

LIST OF TABLES

TABLES

Table 3.1 Near-Biclique Detection Complexities. 35

Table 3.2 Near-Triclique Detection Complexities. 35

Table 4.1 Near-clique partition sizes. Bipartite graph partition sizes: $200x250$. 48

Table 4.2 Near-clique partition sizes. Bipartite graph partition sizes: $1000x1350$. 50

Table 4.3 Near-clique partition sizes. Bipartite graph partition sizes: $1500x2050$. 52

Table 4.4 Near-clique partition sizes on sparse tripartite graphs. 52

Table 4.5 Near-clique partition sizes on sparse tripartite graph with dense regions. 55

Table 4.6 Densities of the near-cliques found by $BDMSCM$ on bipartite graphs. 55

Table 4.7 Densities of the near-cliques found by $TDMSCM$ on tripartite graphs. 56

LIST OF FIGURES

FIGURES

Figure 3.1 $7x8$ bipartite graph. 21

Figure 3.2 Maximum matching. 25

Figure 3.3 Highest degree matching. 27

Figure 3.4 Matching using convolution matrix. 28

Figure 3.5 Matching using sorted convolution matrix. 29

Figure 3.6 Reduced network. 30

Figure 4.1 $15x10$ bipartite graph. 40

Figure 4.2 $15x10$, matchings. 41

Figure 4.3 $15x10$, communities in the reduced network. 42

Figure 4.4 Near-clique $\#1$ found by $BDMSCM$. 43

Figure 4.5 Near-clique $\#2$ found by $BDMSCM$. 44

Figure 4.6 Remaining nodes. 45

Figure 4.7 Bipartite Graph, Partition Sizes: $40x40$. 46

Figure 4.8 Bipartite Graph, Partition Sizes: $200x250$. 47

Figure 4.9 Bipartite Graph, Partition Sizes: $1000x1350$. 49

Figure 4.10 Bipartite Graph, Partition Sizes: $1500x2050$. 51

Figure 4.11 Running time of near-biclique detection algorithms. 53

Figure 4.12 Sparse tripartite graphs. 53

Figure 4.13 Sparse tripartite graphs with dense regions. 54

Figure 4.14 Shuffled sparse tripartite graph with dense regions. 54

Figure 4.15 Running time of near-triclique detection algorithms. 56

LIST OF ABBREVIATIONS

BDMM	Near-Biclique Detection with Maximum Matching
BDHDM	Near-Biclique Detection with Highest Degree Matching
BDMCM	Near-Biclique Detection with Matching Using Convolution Matrix
BDMSCM	Near-Biclique Detection with Matching Using Sorted Convolution Matrix
TDHDM	Near-Triclique Detection with Highest Degree Matching
TDMCM	Near-Triclique Detection with Matching Using Convolution Matrix
TDMSCM	Near-Triclique Detection with Matching Using Sorted Convolution Matrix

CHAPTER 1

INTRODUCTION

The infamous Königsberg's bridges was the inception of the graph theory which goes back to 1726 [1]. Ever since, graph algorithms have been improved nicely, and they have helped us understand the complex structures in many fields. These days, graphs analysis algorithms are ubiquitous, and they are used in wide range of domains. Especially, finding dense components in networks has a very important place in variety of systems, such as social networks, bioinformatics, financial markets and world wide web. Dense components in such systems give us valuable information about the systems, and improves our comprehension of such networks [2].

Real world networks, such as social networks are typically sparse; however, they might include dense regions which have high connectivity between their nodes. Those regions indicate high interactivity, and similarity. Such regions may indicate the groups of friends in a social network, accurate search results in the results of a web browser query, or supply chains in supply chain economy and so more [3, 4, 5].

Community detection is a challenging and important problem whether in unipartite or in multipartite graphs. It necessitates dividing the network into clusters in which a node must be connected with many other nodes in the same cluster, and the connections between the nodes in a cluster and the other nodes in other clusters must be sparse. It is permissible that two clusters share some of the nodes between each other depending on the design of the method. Finding such dense regions is a useful tool for us to understand these large graphs more thoroughly [6].

Real networks are comprised of plethoric amount of nodes. The higher the number of nodes the graph has, the more intractable the problem gets. It is proved in Karp's paper that this problem is NP-hard [7]. There has been many heuristics to solve this problem. With the ubiquity of big data, researchers are getting more inclined to solve this problem to serve different needs. Many algorithms are fast, especially for small and medium graphs. There are different heuristics. These heuristics are explained in section 2.

In these thesis, we propose and compare seven novel near-clique detecting algorithms. A $near - clique$ is a graph structure that is a dense subgraph in a graph. These algorithms are quite similar to each other. Four of those methods detect near-bicliques, and the other three methods detect near-tricliques. These methods find the non-overlapping bicliques or tricliques, in which nodes and cliques have many-to-one relationship. Some nodes might be left out, and they might not be in any near-cliques. Using these methods we aim to find the near-cliques, which include nodes that have high degrees. We make a trade-off between quality of near-cliques and degree of the nodes in the cliques. A community that consists of people with many contacts can be given as an example. The methods proposed are agglomerative, modularity-based novel methods. Our algorithms have three stages. In first stage, the network, which is either bipartite or tripartite, is reduced to a unipartite network. Then near-cliques are found in the reduced network. Lastly the near-cliques found are raised to the near-bicliques or near-tricliques, depending on the original graph. Our algorithms are intended for unweighted, undirected graphs.

We introduce a different approach to detect near-tricliques. Our approach is fast and scalable for large networks. As the number of partitions in the network increases, the complexity increases parallel to the number of partitions.

Rest of this thesis is organized as follows: In chapter 2, we briefly summarized past work on community and near-clique detection. We present our algorithms in chapter 3. In chapter 4, we present and discuss the experiment results. And in chapter 5, we

discuss the future work.

CHAPTER 2

RELATED WORK

As stated in chapter 1, considering its applications and uses, community detection is a solid problem in the graph theory. Despite all the efforts towards this problem, discovering communities in large graphs, which contain millions of nodes, is still a challenging problem that calls for novel solutions.

In this chapter, we first introduce graph theory concepts that are related to the community and near-clique detection as a basis for the later sections and ideas.Then, we consider a widely known algorithms.

Before proceeding with the primer section, section 2.1, note that, there are many distinct approaches that are trying to deal with the problem of community detection. The methods such as, traditional techniques like graph partitioning, hierarchical clustering, partitional clustering and comparatively modern ones such as divisive algorithm of Girvan and Newman, methods based on statistical inference, dynamic algorithms that use random walks, clique based methods, etc. These algorithms are mentioned in section 2.2.

2.1 Community Detection Primer

2.1.1 Community

Despite the number of studies that are focused on detecting communities in graphs, the notion of $community$ is not strictly defined. That is, a $community$ is more of an intuitive concept rather than a mathematical one. However, it can be summarized as a group of nodes that are connected densely internally. Since there are no formal definitions, there are plenty of distinct recipes for the term $community$ in the literature.

Note that, the concept of $community$ in a graph only makes sense when the graph is actually sparse. In other words, number of edges and nodes should not differ considerably. On the other hand, if a given graph is densely connected, any $community$ that we discover is basically a group of data points that are near to each other. Hence, they do not give us insights of great significance in that case.

2.1.2 Complexity

Real world networks have excessive amount of nodes and edges, which makes the efficiency of community detection algorithms more important. For some algorithms computational complexity is hard or even impossible to detect. It is vital to have estimate of complexity for such algorithms. Louvain algorithm, that we use in our methods, is an example of such algorithms [8].

Many community detection or clique detection algorithms are **NP-hard**. Such solutions can only be applied to a small sized networks. For such cases, heuristics that do not give an exact solution are preferred with the advantage of lower complexity. Heuristics, such as Louvain algorithm, are generally non-deterministic, and give different results to the same input.

2.2 Community Detection

In this section, community detection techniques are discussed. These methods are not strict, meaning that an algorithm might belong more that one categories. Only the ones related to our method are discussed here.

2.2.1 Graph Partitioning

This involves dividing the graph into predefined number of clusters using notions such as *cut size*, and *minimum bisection*. *Cut size* is the number of edges connects two clusters, and *minimum bisection problem* is an $NP-hard$ problem that tries to partition the graph into two sets of equal sizes while minimizing the cut size between the clusters [9]. Thus most algorithms in this category are $NP-hard$. Although not optimal, there are some algorithms exist that produce good results [10, 11, 12].

One of the popular methods is The Kernighan-Lin algorithm [13]. This algorithm is used today with combination with other algorithms and techniques, because good guess is needed to assign input parameters such as number of clusters. Since this algorithm runs in $O(n^2 logn)$, it is used to improve the partitions that other methods find. However this algorithms is fast, it is slower than our method, which has a time complexity less than $O(n^2)$. Kernighan-Lin algorithm is extended in 1988 by Suaris and Kedem [14]. The aim of the extension was to find partitions in any number of parts, but complexity worsens at fast pace as the number of clusters increases. Because of the fact that it is needed to guess the input parameters such as number of clusters in the network, and sizes of the clusters, these algorithms are not great when used alone for the networks initially unknown. There are also some other heuristics [10, 12, 15].

These algorithms necessitates initial knowledge or good guessing about the network that will be partitioned. Because of that they are not good for community detection for the networks initially unknown.

2.2.2 Hierarchical Clustering

Unlike graph partitioning methods, these algorithms does not have to foresee the number of clusters to be find. In such cases, where the number of clusters is unknown, and the network has hierarchical structure such as social networks, hierarchical cluster may be used.

These algorithms are divided into two categories:

- **Agglomerative algorithms:** Nodes are merged into similar communities in a recursive fashion [16]. Algorithms start with the vertices as separate clusters, which are called *singletons* and ends up with unique cluster. These algorithms are *bottom-up*, since they start with small clusters and enlarge them.
- **Divisive algorithms:** These algorithms detect vertices with low connections and remove them from clusters iteratively [17, 18, 19, 20]. These algorithms are *top-down*, since they start with bigger pseudo clusters and trim them

This hierarchy depends on the stopping conditions such as cluster size, some quality function such as modularity, or number of clusters. We use Louvain algorithm in our methods, which is an agglomerative, bottom-up algorithm. Stopping condition for the Louvain algorithm is modularity [8]. One drawback of these algorithms is the hierarchical construction might be pseudo, because the original graph does not have to have a hierarchical structure. Also if node has one neighbour it will probably be clustered as cluster by itself. it is common to use this approach with social networks, biological data, marketing data, etc.

2.2.3 Partitional Clustering

The algorithms that fall into this category predetermines the number of clusters. Generally it is shown with k. k points are put on a $n-dimensional$ plane and algorithms work according to the distances between nodes and those k points in metric space.

Data set is decomposed into a set of predetermined number of disjoint clusters by optimizing a *cost function*. *Cost function* depends on your purposes of clustering. Cost functions try to minimize or maximize one of the followings:

- Diameter: Distance between the two most distant points in the cluster.
- Average distance: Average distance between the all points in the cluster.
- Distance to center: Distances of each point from the *centroid*, which does not necessarily collides with a point in a cluster. Aim is to minimize the largest value of distances.
- Average distance to center: Average distances of points from the centroid.

k-means clustering is a famous method in this category [21]. It tries to minimize the distances between the nodes and *centroids*. Extended versions of this algorithms exist. In 2007, it is extended to a graph-based data using geodesic distance, which is the length of the path between the nodes [22]. *Fuzzy k-means clustering* is another popular method. In this method clusters may overlap(*fuzzy*) [23].

As with the *graph partitioning* methods, the limitation is the fact that these methods must predetermine the number of clusters. Also some graphs are not naturally defined in metric space.

2.2.4 Spectral Clustering

These algorithms make use of the *eigenvectors* and *Laplacian matrix*. It includes transformation of original data into a set of points in space. These points then clustered using other techniques. The advantage of it is, it is able to separate points that can not be done by directly clustering original data. The first contribution is done by Donath and Hoffmann [24].

2.2.5 Divisive Algorithms

These algorithms falls into the category of *hierarchical clustering*. The aim is to remove *inter-cluster* edges that presumably connects dense sub-regions. This way clusters get disconnected.

Algorithm proposed by Newman and Girvan is popular [17, 18]. Algorithm can be summarized such as:

1. Calculate the *edge betweenness'* of edges.
2. Remove the edge with highest *betweenness* value.
3. Recalculate the *betweenness* values and centrality according to current edges.
4. Repeat step 2 and 3 until no edges remain.

Girvan and Newman used three alternative *edge betwenness* values: *geodesic edge betweenness*, *random-walk edge betweenness* and *current-flow edge betweenness*. This algorithm does not find overlapping communities. This algorithm runs in $O(m^2n)$ time, where m is the number of edges in the network, and n is the number of nodes on the network. This algorithm is slower than our method.

2.2.6 Modularity-Based Methods

Although *Newman-Girvan modularity* is originally defined as stopping criteria for *hierarchical clustering* methods, it is used in many other clustering methods [18]. Modularity is explained more detailed in section 3.2.2.1. There are several *modularity based clustering algorithms*. There are 3 categories: fast algorithms [8, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35], more accurate algorithms with high complexity [36, 37, 38], and the ones that stays in between the former and the latter [39, 40, 41, 42]. These algorithms are searching for the partitions with high modularity values. *Modularity optimization* is an $NP - complete$ problem [43]. Several methods approximates the *modularity* well in a reasonable time.

CHAPTER 3

METHODOLOGY

In this chapter, the algorithms we propose, cost functions and the metrics that we use to assess the algorithms are explained. We present seven algorithms, which differ only in the first and the last step of the pipeline. We propose four algorithms to solve the near-biclique detection problem, and three algorithms to solve the near-triclique detection problem.

General pipeline for the algorithms that are proposed is as follows:

1. Reduce the graph to unipartite graph. Original graph might be bipartite or tripartite.
2. Detect the near-cliques in unipartite graph.
3. Raise the near-cliques to near-bicliques or near-tricliques.

3.1 Proposed Algorithms

In this section, our algorithms are explained. All of the algorithms detect non-overlapping near-cliques in the network. The algorithms are agglomerative, and modularity-based.

All the algorithms first reduce the graph to a unipartite graph. Secondly, near-cliques in the unipartite graph are found. Lastly the near-cliques are raised according to the original graph, which is either bipartite or tripartite. Algorithms differ in the first

and final stages. In the first stage, reduction from multipartite graph to unipartite graph occurs using different matching algorithms and priorities. Matching algorithms matches the nodes from different partitions with each other in order to reduce the network. The priorities are used in matching algorithms to give the nodes precedence. These matching algorithms that we use are novel algorithms based on the algorithms from the literature [44, 45]. The matching algorithms are explained in section 3.2.1 in detail.

The algorithms are divided into 2 categories:

- **Near-biclique detection algorithms:** Four algorithms are intended to detect near-bicliques in networks. These algorithms follow the same steps. These steps can be seen in algorithm 1. The algorithms take a bipartite graph as an input.

  The procedures consist of three stages and they are defined as follows:

  1. **Reduce:** This stage consists of three steps.
     (a) Calculating Priorities: First step is calculating the priority lists of each node in the graph. The priority list of a node in the left partition holds the priorities given to the nodes from right partition that are connected with the list's owner. The priorities change according to the algorithm used, which are explained in detailed in the section 3.2.1.1. These priorities are used in matching algorithms, which is the next step.
     (b) Matching: After calculating priority lists, nodes from the left partition are matched with the nodes from right partition using matching algorithms.
     (c) Reduction: In this step, each matching is treated as a new node, and bipartite graph is reduced to a unipartite graph.
  2. **Louvain Method:** After unipartite graph is created, Louvain algorithm is used to find near-cliques in that graph.

3. **Raise:** The last step of the algorithms is the raise step, in which, the near-cliques found in the previous step with Louvain algorithm, are raised to a near-bicliques.

The near-biclique detection algorithms are as follows:

- **Near-Biclique Detection with Maximum Matching:** This algorithm is the only algorithm that does not calculate a priority list. This algorithm serves as a random case to compare other algorithms with, since it matches nodes regardless of their properties. Other algorithms are expected to outperform this algorithm. Without considering the graph properties, except for the fact that the nodes to be matched must have an edge between them, in the matching step it matches the nodes from left partition with the nodes from the right partition using the maximum bipartite matching algorithm (algorithm 5), which is explained in section 3.2.1.2.
- **Near-Biclique Detection with Highest Degree Matching:** This algorithm uses the N-M matching algorithm (algorithm 6), in a way that is explained in section 3.2.1.2. N-M matching algorithm is a novel algorithm inspired from the stable marriage algorithm. N-M matching algorithm tries to match the nodes of an unevenly partitioned bipartite graph. This algorithm uses the degrees of the nodes while calculating the priority lists of the nodes, and matches according to the priority lists of the nodes. This algorithm aims to match the nodes with higher degrees together so that the resultant near-cliques are dense with nodes that have higher degrees.
- **Near-Biclique Detection with Matching Using Convolution Matrix:** This algorithm uses the N-M matching algorithm, in a way that is explained in section 3.2.1.2. This algorithm aims to match the clustered regions in the adjacency matrix together, so that the near-cliques found by Louvain algorithm are dense, and nodes of the near-cliques have higher degrees. This algorithm is the plain version of the next algorithm.
- **Near-Biclique Detection with Matching Sorted Using Convolution Matrix:** This algorithm uses N-M matching algorithm, in a way that is explained in section 3.2.1.2. This algorithm is the augmented version of the

previous algorithm. While calculating the priorities, it sorts both the rows and columns of the graph according to the degrees of the nodes, so that the convolution matrix gives us better results on a random graph. By sorting the rows and columns of the matrix, this algorithm is expected to give better results than the other algorithms.

- **Tripartite Clustering:** The other three algorithms are crafted for near-triclique detection. These algorithms follow the same steps. These steps can be seen in algorithm 2. These algorithms take three adjacency matrix as an input. The first matrix represents the edges between the first and the second partition. The second matrix represents the edges between the first and the third partition. The last matrix represents the edges between the second and the third partition.

  The procedure consists of three stages and they defined as follows:

  1. **Reduce:** This stage consists of four steps.
     (a) Calculating Priorities: First step is calculating the priority lists for each node in the graph. The priority values change according to the algorithm used. These priorities are used in matching algorithms, which is the next step.
     (b) Matching: After calculating the priority lists, nodes from the first partition are matched with the nodes from the second partition using matching algorithms. Secondly nodes from the first partition are matched with the nodes from the third partition using matching algorithms.
     (c) Merge: The output of the matching stage is two sets of matchings (M_1, M_2), which are the matchings from first partition to second partition, and the matchings from first partition to third partition (lines 2 and 3 from the algorithm 2). M_1 and M_2 are merged so that nodes from all partitions are matched together. Merge operation traverses the M_1 and M_2. If there are common nodes in M_1 and M_2, it merges those entries together to get a tripartite matching.
     (d) Reduction: In this step, each matching is reduced to a new node, and graph is reduced to a unipartite graph.

2. **Louvain Method:** After unipartite graph is created, Louvain algorithm is used to find near-cliques in that graph.

3. **Raise:** The last step is the raise step. In this step, the near-cliques, which are found in the previous step, are raised to near-tricliques.

The near-triclique detection algorithms are as follows:

- **Near-Triclique Detection with Highest Degree Matching:** This algorithm uses the N-M matching algorithm (algorithm 6). It can be seen in the algorithm 2 that it uses the N-M matching algorithm twice to match the first and the second partitions together and the first and the third partitions together, in a way that is explained in section 3.1.2.2. This algorithm uses the degrees of the nodes while calculating the priority lists of the nodes. It aims to match the nodes with higher degrees together, so that resultant near-cliques are dense and the nodes it includes have higher degrees.
- **Near-Triclique Detection with Matching Using Convolution Matrix:** This algorithm uses N-M matching algorithm. This algorithm does the matching same with the $TDHDM$ algorithm with different priority values. The matching used here is explained in section 3.1.2.3. This algorithm aims to match the clustered regions in the adjacency matrix together, so that near-cliques that Louvain algorithm finds are dense and, nodes of the near-cliques have high degrees. This algorithm is the plain version of the $TDMSCM$ algorithm.
- **Near-Triclique Detection with Matching Using Sorted Convolution Matrix:** This algorithm uses N-M matching algorithm. This algorithm does the matching same with the $TDMCM$ algorithm with an extra sort operation. The matching used here is explained in section 3.1.2.4. This algorithm is the augmented version of the $TDMCM$ algorithm. While calculating the priorities, it sorts both the rows and columns of the graph according to degree of the nodes, trying to bring dense regions together as much as possible. This algorithm is expected to get better results than the other algorithms.

Algorithm 1 General Algorithm For Finding Near-Bicliques

Input: G : adjacency matrix

Output: C : near-bicliuqes

1: $L \leftarrow setPriorities(G)$

2: $M \leftarrow match(G, L)$

3: $G' \leftarrow reduce(G, M)$

4: $C \leftarrow louvain(G')$

5: $C \leftarrow raise(C, M)$

Algorithm 2 General Algorithm For Finding Near-Tricliques

Input: G_1 : adjacency matrix

Input: G_2 : adjacency matrix

Input: G_3 : adjacency matrix

Output: C : near-bicliuqes

1: $L \leftarrow setPriorities(G_1, G_2)$

2: $M_1 \leftarrow match(G_1, L)$

3: $M_2 \leftarrow match(G_2, L)$

4: $M \leftarrow merge(M_1, M_2)$

5: $G' \leftarrow reduce(G_1, G_2, G_3, M)$

6: $C \leftarrow louvain(G')$

7: $C \leftarrow raise(C, M)$

3.2 Stages of the Algorithms

3.2.1 Reduction

The reduction stage of the algorithms are explained here. This stage consists of three steps:

- Calculating the Priorities
- Bipartite/Tripartite Matching
- Reduction

These 3 steps of reduction is explained here.

3.2.1.1 Calculating the Priorities

The procedure of calculating the priorities explained in this section. N-M matching algorithm matches the nodes according to the priorities. Matching algorithms give precedence to the nodes that have higher priority values, except for the maximum matching algorithm, which does not use priority values. In other words, nodes with high priorities are matched first and inclined to stay matched.

Each node has a priority list, which indicates the precedence of the nodes that owner of the list is inclined to match with. In other words, the entry of the priority list of a node indicates how much the node is inclined to match with that entry. Lists do not involve the nodes that do not have an edge between them and the list's owner.

Here, we explain the two ways used to calculate the priority list for the nodes.

- **Using Node Degrees:** This method makes use of the degrees of the nodes. Higher the degree of a node, higher the priority. The priority list of node i in the left partition is calculated by using this formula:

$$L_{ij} := A_{ij} \frac{\sum_n A_{nj}}{m}, \qquad \forall j \in \{0, 1, ..., N\}$$

The priority list of node j in the right partition is calculated by using this formula:

$$L_{ji} := A_{ij} \frac{\sum_n A_{in}}{m}, \qquad \forall i \in \{0, 1, ..., M\}$$

where,

- A denotes the adjacency matrix of the given network.
- A_{ij} denotes the jth entry in ith row of A.
- L_{ij} denotes the priority of node j for node i.
- M denotes the number of rows.
- N denotes the number of columns.
- m denotes the total number of edges between the two partitions.

Here, priority list of a node is calculated. Each node's degree is calculated, and divided by m for normalization. Then that value is multiplied with 1, if there is an edge between N_i and N_j, 0 otherwise, because priority list of a node only includes a node if there is an edge between them.

- **Using Convolution Matrix:** Convolution matrix, in this context, is a matrix that is used to calculate the priority lists. Convolution matrix is calculated using the filter F,

$$F = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix},$$

because we want to match the nodes if they are in a cluster and close in the adjacency matrix. The filter F traverses every cell of the A, which is the adjacency matrix. For each cell, it multiplies the value of the cell and values of the eight surrounding cells by the corresponding value of F. Then it adds the

results, the value in the convolution matrix is set with this result. To formulate this process, each cell (i, j) of the convolution matrix is calculated using this formula,

$$C_{ij} = A_{i-1j-1}F_{00} + A_{i-1j}F_{01} + A_{i-1j+1}F_{02}+$$

$$A_{ij-1}F_{10} + A_{ij}F_{11} + A_{i-1j+1}F_{12}+$$

$$A_{i+1j-1}F_{20} + A_{i+1j}F_{21} + A_{i+1j+1}F_{22}$$

where,

- C denotes the convolution matrix.
- A denotes the adjacency matrix.

After calculating the convolution matrix. Priority list of node i in the left partition is calculated with this function,

$$L_{ij} := \frac{((C_{ij} + \sum_k C_{ik})C_{ij})}{m}, \qquad \forall j \in \{0, 1, ..., N\}$$

and priority list of node j in the right partition is calculated with this function,

$$L_{ji} := \frac{((C_{ij} + \sum_k C_{kj})C_{ij})}{m}, \qquad \forall i \in \{0, 1, ..., M\}$$

where,

- L_{ij} denotes the priority of node j for the node i.
- M denotes the number of rows in C
- N denotes the number of columns in C
- m denotes the total number of edges in the bipartite graph.

To illustrate this process, let's say,

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}, C = \begin{pmatrix} 4 & 6 & 5 & 3 & 2 & 3 & 5 & 4 \\ 5 & 7 & 5 & 3 & 2 & 3 & 5 & 4 \\ 4 & 5 & 2 & 1 & 0 & 1 & 2 & 2 \\ 2 & 2 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 2 & 2 & 3 & 2 \\ 0 & 0 & 0 & 1 & 2 & 2 & 3 & 2 \end{pmatrix},$$

$$L_{10} = ((4 + 5 + 4 + 2 + 1) + 5) * 5/18 = 5.83$$

$$L_{11} = ((6 + 7 + 5 + 2 + 1) + 7) * 7/18 = 10.9$$

$$L_{12} = ((5 + 5 + 2) + 5) * 5/18 = 4.7$$

$$L_{13} = 0$$

$$L_{14} = 0$$

$$L_{15} = 0$$

$$L_{16} = ((5 + 5 + 2 + 1 + 3 + 3) + 5) * 5/18 = 6.7$$

$$L_{17} = ((4 + 4 + 2 + 1 + 2 + 2) + 4) * 4/18 = 4.2$$

Priority list of node 1 in left partition is calculated as: $1, 6, 0, 2, 7$.

3.2.1.2 Matching

In this section, matching algorithms are explained. To illustrate these algorithms, outputs of the algorithms are presented in the corresponding sections. The algorithms run with the graph shown in figure 3.1. In this graph, nodes $3, 4, 5, 11, 12, 13$ form a clique. Also node 1 has 3 edges and node 8 has 4 edges.

The matching algorithms try to match the nodes from the left partition with the nodes from the right partition of the bipartite graph. 2 nodes can be matched if and only if

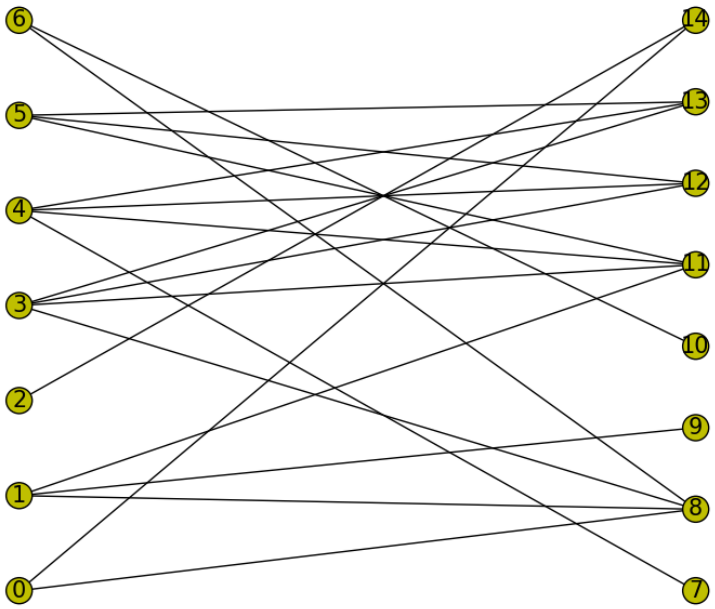

Figure 3.1: $7x8$ bipartite graph.

they have an edge between them. Both biclique detection and triclique detection algorithms use the same algorithms explained here. Even though all of the algorithms in this section are designed for bipartite graphs, we came up with a little trick to enable us to use these algorithms with tripartite algorithms. Tripartite algorithms first do the matching between the first and the second partitions, then do the matching between the first and the third partitions. Then, it searches for commonalities in the bipartite matchings. If it finds a common node, it merges them. Doing so tripartite matchings are complete.

The matching algorithms except the first algorithm (maximum matching algorithm) uses the N-M matching algorithm (algorithm 3). N-M matching algorithm is an extension of the stable marriage problem [45] that we propose. Stable marriage problem is a problem of matching N men and N women without leaving any person out. N-M matching however, tries to match N men with M women. Size of the resulting matchings is less than or equal to N, and less than or equal to M. Each node can only be matched with one node from the opposite partition. Each node has a priority list of neighbours, which is different for all nodes. These priorities specify the node's incli-

nation to be matched with the nodes from opposite partition. In algorithm 3, $rMatch$ is the array matchings, $lFree$ is the availability of the nodes in left partition, L_l is the node with id l in the left partition, $L_l.PL$ is the priority list of the L_l. The function used in line 16 is shown in figure 4. Algorithm works in the following way: In lines 4 to 8 it finds a free node from the left partition (l). Then in lines 9 to 23, for each node, r, in the priority list of l, it tries to match l with r. If r is free, r and l are matched (lines $10 - 13$). If r is currently matched, but prefers l over the current match, l and r are matched (lines $14 - 22$). After traversing the priority list of the l, if l is not matched, it dumps l (lines $24 - 26$), and lastly the matchings are returned.

The matching algorithms are as follows:

1. **Maximum Matching:**
   This algorithm [44] tries to match as many nodes as possible in the left partition of a bipartite graph with the nodes in the right partition of a bipartite graph. The size of the output of this algorithm has the size lesser than or equal to the size of the smaller partition of graph. The algorithm can be seen in algorithm 5. Algorithm works in the following way: In lines 1 to 6 it traverses all cells in adjacency matrix, for each cell it calls $depthFirstMatch$ function (line 5), which performs a depth first search and tries to match all the nodes that are not seen yet. In lines 2 to 4, it makes all the neighbours of the current node unseen. Lastly matchings are returned. The resulting matchings of this algorithm can be seen in figure 3.2. The input graph for this example is shown in figure 3.1. Since it tries to match as many nodes as possible. It matches all seven nodes from the left partition without considering the any other properties.

2. **Highest Degree Matching:**
   This algorithm uses N-M matching algorithm. The priority lists of the nodes are constructed using the degrees of the nodes. A node with a higher degree has higher priority. The resulting matchings of this algorithm can be seen in figure 3.3. The input graph for this example is shown in figure 3.1. It matches the nodes of cliques, except 13. It matches 3 with 8, which has 4 neighbours.

Algorithm 3 N-M Matching

Input: L : nodes of the left partition

Input: R : nodes of the right partition

Output: $rMatch$: $size(R)$ sized array, initially, all values are -1

1: $lFree$: $size(L)$ sized array, initially, all values are 0
2: $freeCount \leftarrow size(L)$
3: **while** $freeCount > 0$ **do**
4: **for** $l = 0$ to $size(L)$ **do**
5: **if not** $lFree[l]$ **then**
6: $break$
7: **end if**
8: **end for**
9: **for** $r = 0$ to $size(L_l.PL)$ **do**
10: **if** $rMatch[r] = -1$ **then**
11: $rMatch[r] \leftarrow l$
12: $lFree[l] \leftarrow 1$
13: $freeCount- = 1$
14: **else**
15: $l1 \leftarrow rMatch[r]$
16: **if not** $rFavorsL1OverL(R, r, l, l1)$ **then**
17: $rMatch[r] \leftarrow l$
18: $lFree[l] \leftarrow 1$
19: $lFree[l1] \leftarrow 0$
20: $break$
21: **end if**
22: **end if**
23: **end for**
24: **if** $i = size(L_l.PL)$ **then**
25: $lFree[l] \leftarrow 1$
26: $freeCount \leftarrow freeCount - 1$
27: **end if**
28: **end while**

Algorithm 4 rFavorsL1OverL

Input: R : priority list of the right partition

Input: l : index of a node in the left partition

Input: l_1 : index of a node in the right partition

for $r = 0$ to $size(R)$ **do**
  if $R[r] = l_1$ **then**
    return True
  end if
  if $R[r] = l$ **then**
    return False
  end if
end for
return True

Algorithm 5 Maximum Bipartite Matching Algorithm

Input: A : adjacency matrix

Input: M : number of rows in A

Input: N : number of columns in A

Output: $rMatch$: N sized array, initially, all values are -1

for $l = 0$ to M **do**
  for $r = 0$ to N **do**
    $seen[r] \leftarrow -1$
  end for
  $depthFirstMatch(A, l, seen, matchR)$
end for
return rMatch

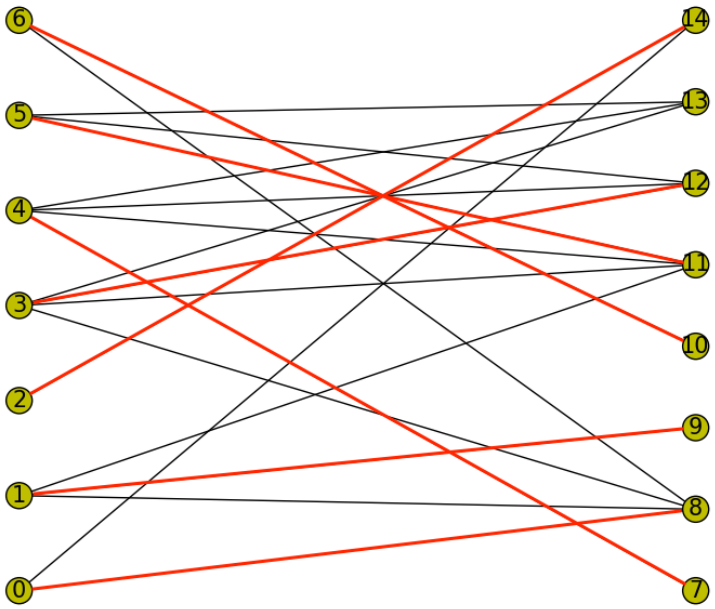

Figure 3.2: Maximum matching.

3. **Matching Using Convolution Matrix:**
   This algorithm uses N-M matching algorithm. The priority lists of the nodes constructed using convolution matrix. After convolution matrix is created, priority lists are calculated as explained in section 3.2.1.1. After creating the priority lists, N-M matching algorithm is used with the priority lists. The resulting matchings of this algorithm can be seen in figure 3.4. The input graph for this example is shown in figure 3.1. It matches the nodes of the clique, and matches 1 with 8, which are the nodes with most neighbours besides clique.

4. **Matching Using Sorted Convolution Matrix:**
   After sorting the input graph's adjacency matrix according to the degrees of nodes both row wise and column wise, previous algorithm is run on sorted matrix. This algorithm first sorts the rows of the matrix according to the degrees, then sorts the columns of the matrix according to the degrees. The resulting matchings of this algorithm can be seen in figure 3.5. The input graph for this example is shown in figure 3.1. It matches the nodes of the clique, and matches 1 with 8, which are the nodes with most neighbours besides clique.

Algorithm 6 Depth First Match

Input: A : adjacency matrix

Input: l : index of a node in left partition

Input: $seen$: array of already traversed nodes

Input: $matchR$: array of matchings

Output: $matchR$: array of matchings

```
1: for r = 0 to N do
2:   if A[l][r] and not seen[r] then
3:     seen[r] ← 1
4:     if matchR[r] or depthFirstMatch(A, matchR[r], seen, matchR) then
5:       matchR[r] ← l
6:       return 1
7:     end if
8:   end if
9: end for
10: return 0
```

3.2.1.3 Reduce

In this step, the matchings, found in the previous step, are reduced to a single node. The reduced network can be seen in figure 3.6. For this example matchings of the 4th method(matching using sorted convolution matrix) are chosen to be reduced. The reduced network is the input of the Louvain algorithm. The nodes in the reduced network and corresponding matchings of original network are as follows:

- Node 0: $1 - 8$
- Node 1: $3 - 13$
- Node 2: $4 - 12$
- Node 3: $5 - 11$

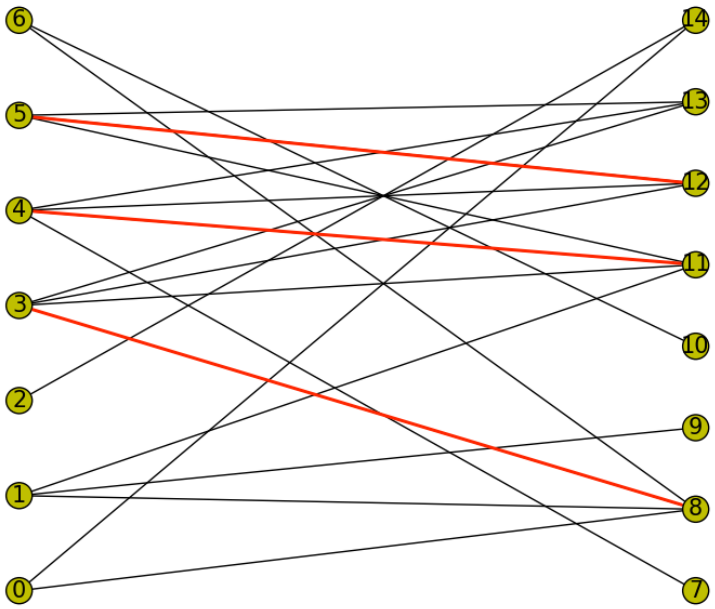

Figure 3.3: Highest degree matching.

3.2.2 Finding Near-Cliques

After reducing the original network, next step is the detecting communities in the reduced network. For this task, Louvain algorithm is used [8]. Louvain algorithm is a modularity-based, bottom-up algorithm. It uses the modularity described by Girvan and Newman [18].

3.2.2.1 Modularity

Modularity function can be written as,

$$Q = \frac{1}{2m} \sum_{i,j} (A_{ij} - \frac{k_i k_j}{2m}) \delta(c_i, c_j)$$

where,

- m denotes the total number of edges in the graph.
- A denotes the adjacency matrix. If there is an edge between node i and node j,

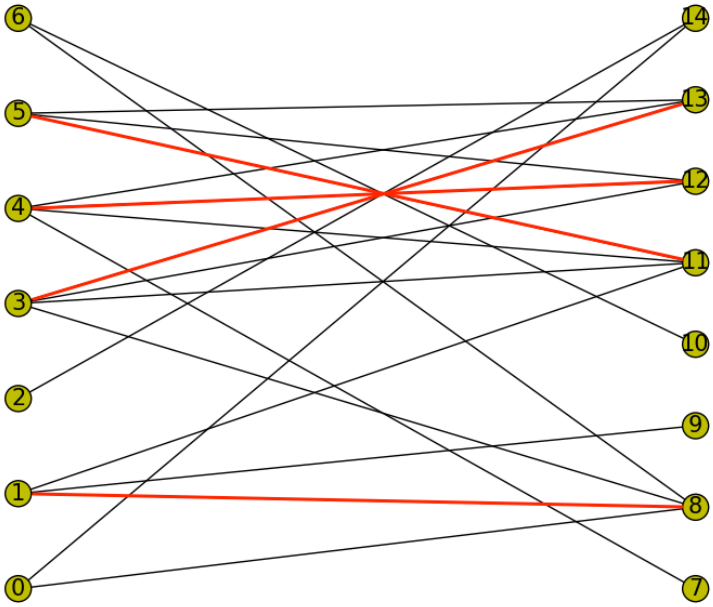

Figure 3.4: Matching using convolution matrix.

$A_{ij} = 1$, else $A_{ij} = 0$.

- k_i and k_j denotes the degrees of node i and node j respectively.
- c_i and c_j denotes the community that node i belongs to and community that node j belongs to respectively.
- The function $\delta(c_i, c_j)$ checks whether c_i is equal to c_j, in other words, if c_i and c_j is the same community. $\delta(c_i, c_j) = 1$, if node i and node j are in the same community, $\delta(c_i, c_j) = 0$ otherwise.

This function assesses the community structure of the network as a whole. Higher values of modularity means better community structure. Note that modularity is always in between the range [-1,1]. Intuitively, modularity calculates two values, and subtracts them. the first value is,

$$\sum_{i,j} A_{ij}\delta(c_i, c_j),$$

which is the number of edges between the nodes that are in the same community. As communities get bigger and denser this value get bigger. Second value is,

$$\sum_{i,j} \frac{k_i k_j}{2m}\delta(c_i, c_j),$$

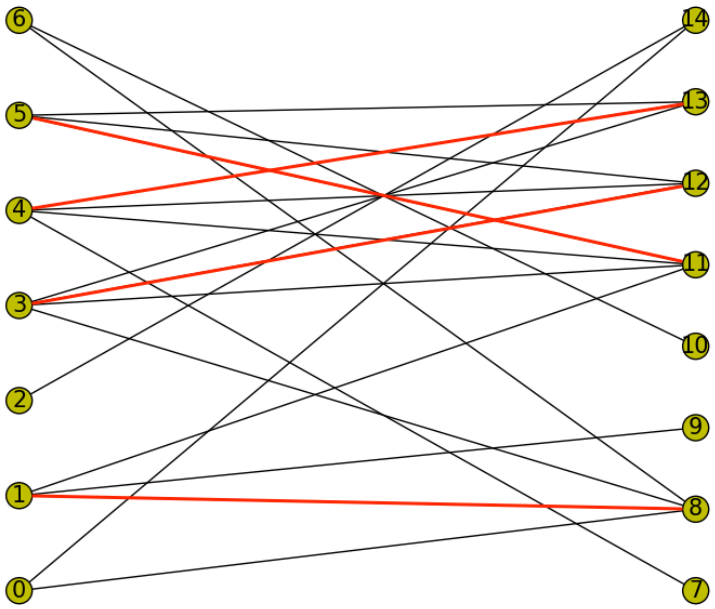

Figure 3.5: Matching using sorted convolution matrix.

which calculates the density of communities, if network was defined random. By subtracting these two value, it finds the value of wellness of the community structure in the sense of how much denser the communities in the graph in reference to the communities if the network would be random. Modularity is positive ($(0, 1]$), if there are more edges within the communities that you expect by chance.

3.2.2.2 Local Moving Heuristic

Louvain algorithm uses local moving heuristic algorithm as a first step. local moving heuristic algorithm is an algorithm that is used to increase modularity of a network. It moves the nodes from one community to another to increase the modularity. It iterates over the nodes randomly, for each node, it checks whether moving this node to one of the other communities increases the modularity. If so, that node is moved to the community that results in the highest modularity gain. The algorithm keeps going until moving nodes between communities has no modularity gain.

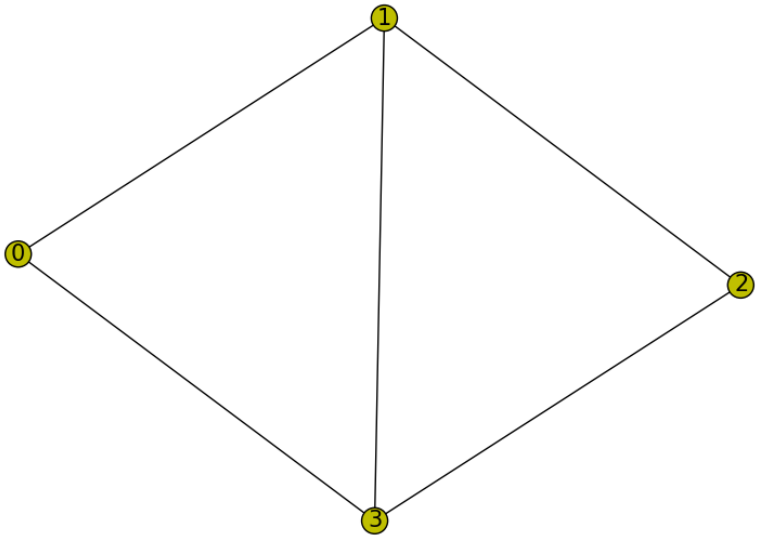

Figure 3.6: Reduced network.

3.2.2.3 Louvain Algorithm

Louvain algorithm is a recursive algorithm used to detect communities in a unipartite network. The algorithm can be seen in algorithm 7. It makes use of local moving heuristic algorithm which manipulates the graph according to the modularity of the graph. Louvain method takes the reduced network as input and initializes the network in a way that each node belong to a community that the only node in that community is itself. After assigning each node to separate communities, it uses local moving heuristic and moves the nodes between communities to get a network with higher modularity (line 1). After reaching a higher modularity than before, each community is reduced to a node and reduced network is constructed (line 3). Since each community is treated as one new node, the weight of an edge between two new nodes in the reduced network equals to the total number of edges between the nodes in the two corresponding communities in the original network. After reduced network is constructed, same procedure starts again for the reduced network (line 5), and it continues recursively following the same procedure until the network can not be reduced anymore. In lines 6 to 8 it replaces old cliques with the new cliques. Resultant network has lesser number of nodes than the original graph, where each node correspond

to a community which holds nodes from the original graph. This way, each node from the original is assigned to a community.

Algorithm 7 The Louvain Algorithm

Input: A : adjacency matrix of the reduced graph

Input: C : initial assignments of nodes to the cliques

Output: C : final assignments of nodes to the cliques

1: $C \leftarrow LMH(A, C)$
2: **if** c is not converged **then**
3: $A' \leftarrow Reduce(A)$
4: $C' \leftarrow [1...Nodes(A')]$
5: $C' \leftarrow Louvain(A', C')$
6: **for** $i = 1$ to Size(A') **do**
7: $C[i] \leftarrow C'[i]$
8: **end for**
9: **end if**

The output of this stage is unipartite near-cliques. Each node in those cliques is comprised of two nodes if the original network is bipartite, three nodes if the original network is tripartite. For the input graph shown in the 3.6, Louvain algorithm assigns all nodes to one community.

3.2.3 Raise Stage

Raise stage is the last stage of the algorithms. In this stage, the near-cliques found by the Louvain algorithm are raised to a near-bicliques or near-tricliques.

3.3 Complexity and Constraints

In this study, the algorithms represent graphs using adjacency matrices. In depth explanations of these stages are given in section 3.1. The average complexities of algorithms are presented here stage by stage.

In the subsections, M refers to the number of nodes in the first partition, N refers to the number of nodes in the second partition, and K refers to the number of nodes in the third partition. Note that the adjacency matrices of the initial network is used throughout the algorithm, which has a size of $O(MN)$ for bipartite networks, and $O(MN + MK + NK)$ for tripartite networks. While considering the space complexities of the stages in algorithms, adjacency matrix is not added. However it is reflected to the overall complexity.

3.3.1 Reduce

This stage consists of four steps. Note that first three steps happen twice for tripartite networks. They run for the first and the second partitions, and the first and the third partitions. Bipartite algorithms do not involve the last step.

3.3.1.1 Calculating the Priority Lists

The priority lists used in matching algorithms are created in two ways. The first one is using node degrees, and the second one is using the convolution matrix that is created in this step. the complexities of these two ways are as follows:

- **Node degrees:** Calculating degrees of all nodes is done by traversing the adjacency matrix. The time complexity of degree calculation is $O(MN)$. Using these degrees to calculate priority lists for all nodes is $O(MN)$. In overall, the time complexity of this step is $O(MN)$.
- **Convolution matrix:** Construction of convolution matrix is done by traversing the adjacency matrix once. The time complexity of convolution matrix

construction is $O(MN)$, calculating sum of entries in convolution matrix is $O(MN)$, calculating priorities for all nodes is $O(MN)$. In overall the time complexity of this step is $O(MN)$. sorting takes $O(logM + logN)$ time if sort is needed.

There is extra space to hold cumulative degree lists for each node, which is $O(M+N)$, and priority lists, which is $O(M+N)$. For convolution matrix extra matrix is needed, which is $O(MN)$. in overall the space complexity of first way is $O(2M + 2N) = O(M + N)$, and the time complexity of the second way is $O(2M + 2N + MN) = O(MN)$.

3.3.1.2 Matching Algorithms

The complexities of the two matching algorithms are as follows:

- **Maximum bipartite matching:** To fulfill this task, the adjacency matrix is traversed once. Time complexity of max bipartite matching algorithm is $O(MN)$.
- **N-M matching:** The time complexity of the stable marriage problem is $O(N^2)$ [46]. However this algorithm are extended to find matchings between uneven partitions. The time complexity of N-M matching is $O(2MN) = O(MN)$.

Space complexity for both matching algorithms is $O(M + N)$.

3.3.1.3 Merge

This step occurs only in the triclique algorithms. After we have the matchings between the first and the second partitions, and the first and the third partitions, which are M_1, M_2, each node on left partition of M_2, is traversed for each node on left partition of M_1 to find collisions. The time complexity of this step is $O(M_1M_2)$. Extra space is $O(MN)$.

3.3.1.4 Reduce

In this step reduced network is constructed, which uses matchings calculated in the previous steps. Each cell in the reduced graph is filled either 1 or 0. Reduced graph has a size of N'^2, where $N' <= M$ and $N" <= N$. The time complexity of this step is $O(N'^2)$. Extra space is needed for reduced graph, which is $O(N'^2)$.

3.3.2 Louvain Algorithm

Louvain algorithm runs until it converges. i.e. it stops when it finds the best local modularity. At each pass it traverses all of the nodes in the network. Each pass takes $O(N')$ time. The number of passes cannot be determined exactly. Modularity optimization is known to be $NP - hard$ [47]. It seems it runs in $O(N'log(N'))$ time, where N' is the number of nodes in the reduced network. Space complexity is $O(N'^2 + N')$. It uses extra space for initial matrix, and assignments of nodes to cliques.

3.3.3 Raise

The time complexity of this stage is $O(N')$, where N' is equal to the N' in the previous section. All nodes in the near-cliques found in previous step traversed once. Extra space needed for near-tricliques is $O(3N')$. Extra space needed for near-bicliques is $O(2N')$.

3.3.4 Near-Biclique Detection Algorithms

Each near-biclique detection algorithm proposed has the same complexities. The general procedure can be seen in algorithm 1. The complexities are listed on table 3.1:

The overall time complexity of near-biclique detection algorithms proposed is $O(MN)$. The overall space complexity is $O(MN)$.

	Time Complexity	Space Complexity
setPriorities	$O(MN)$	$O(MN)$
match	$O(MN)$	$O(MN)$
reduce	$O(N'^2)$	$O(MN)$
louvain	$O(N'logN')$	$O(MN)$
raise	$O(N')$	$O(MN)$
overall	$O(MN)$	$O(MN)$

Table 3.1: Near-Biclique Detection Complexities.

3.3.5 Near-Triclique Detection Algorithms

Each near-triclique detection algorithm proposed has the same complexity. The general procedure can be seen in algorithm 2. The complexities are listed on table 3.2.

	Time Complexity	Space Complexity
setPriorities	$O(M(N+K))$	$O(MN+MK+NK)$
match	$O(M(N+K))$	$O(MN+MK+NK)$
merge	$O(M'_1M'_2)$	$O(MN+MK+NK)$
reduce	$O(N'^2)$	$O(MN+MK+NK)$
louvain	$O(N'logN')$	$O(MN+MK+NK)$
raise	$O(N')$	$O(MN+MK+NK)$
overall	$O(M(N+K))$	$O(MN+MK+NK)$

Table 3.2: Near-Triclique Detection Complexities.

The overall time complexity of near-triclique detection algorithms proposed is $O(M(N+K))$. Extra space is needed for adjacency matrices is $O(MN+NK+NK)$. Overall space complexity is $O(MN+MK+NK)$.

3.4 Advantages

The main advantage of this method is that it is fast and runs in $O(MN)$ time for bipartite networks, and $O(M(N+K))$ for tripartite networks. where M is the number of nodes in the first partition, N is the number of nodes in the second partition, and K is the number of nodes in the third partition.

The fastest method in literature runs in linear time, but it requires priori information about initial network, and makes an assumption that the communities in the network are similar in size [48].

Newman and Girvan point out that the high computational demands is the main set-back of their algorithm [18]. It runs in worst-case time $O(E^2V)$ on dense graphs, and $O(V^3)$ on sparse graphs, where E is the number of edges in the network, and V is the number of vertices in the network. Which is much slower than our method.

Bipartite verison of the modularity that is coined by Newman and Girvan is defined in [49]. It can be used with other standard algorithms such as [35], which would have the worst-case running time of $O((E + V)V)$, or $O(V^2)$ on a sparse graph, which looks on par with our algorithm, however our algorithm runs in $O(MN)$ for bipartite graphs, where $M + N = V$. Also our algorithm runs in $O(M(N + K))$ for tripartite graphs, where $M + N + K = V$. That makes our algorithm scalable for large networks. Also, our algorithm is suitable to be extended for multipartite graphs.

CHAPTER 4

EXPERIMENTS AND RESULTS

All experiments are carried out with a Mid-2014 MacBook Pro Retina with 2.5 GHz Intel Core i7, and 16GB 1600 MHz DDR3 ram.

4.1 Experiment Data

We have used computer generated data sets in order to analyze the correctness and efficiency of our method. We have used three different data sets for our experiments. The data sets are generated randomly using "$int\ rand(void)$", and "$void\ srand(unsigned\ int)$" functions of C programming language, which are located in "$stdlib.h$" library.

- "$int\ rand(void)$" function returns a pseudorandom number in the range of 0 to RAND_MAX, which is also defined in "$stdlib.h$" library and guaranteed to be at least 32767, and has the value of 2147483647 in our environment.
- "$void\ srand(unsigned\ int)$" function seeds the random number generator used by rand function, also defined in "$stdlib.h$".

Both functions use linear congruential generators [50].

3 types of data sets used in our experiments are listed as follows:

- **Sparse graphs:**
  These graphs are randomly generated sparse graphs. The graphs have different densities. These graphs are shuffled using Knuth's shuffling algorithm after creating random edges [50].

- **Sparse graphs with dense regions:**
  These graphs have dense regions. The nodes of a dense region are closely located in the adjacency matrix in this data set.

- **Shuffled sparse graphs with dense regions:**
  These graphs have dense regions. The nodes of a dense region are not necessarily closely located in the adjacency matrix in this data set. These graphs are shuffled using Knuth's shuffling algorithm , which produces a uniformly random permutation of the input array in linear time, which is proved in 1938 by Fisher and Yates [51].

Note that all the nodes in the graphs have at least one neighbour.

4.2 Experiment Metrics

4.2.1 Modularity

To evaluate the near-cliques found, we define a new modularity. Our modularity evaluates the resultant cliques according to their density, and the ratio of their size to the size of the original graph. As a metric we do not use the Newman and Girvan's modularity [18]. Generally community detection algorithms assigns all nodes to some community. Since our algorithms might leave some nodes out, it would be biased to use Newman and Girvan's modularity. The modularity function used as a metric in our evaluations of bipartite near-cliques is as follows:

$$Q = (\sum_{n} \frac{(\sum_{i,j} A_{ij}\delta(c_i, c_j, n))C_n}{m_n G_{size}})$$

where,

- A denotes the adjacency matrix of the original graph.
- m_n denotes the maximum number of edges that a nth near-clique can have.
- c_i denotes the near-clique that node i belongs to.

- $\delta(c_i, c_j, n)$ function returns 1 if c_i and c_j are the same community, and c_i is the nth near-clique, returns 0 otherwise.
- C_n denotes the number of nodes in nth clique.
- G_{size} denotes the number of nodes in the graph.

This function, calculates the densities of each near-clique, for each clique density is multiplied with the number of nodes in the clique, and divided by the total number of nodes in the graph. Calculating the densities is done by calculating the number of edges between the two partitions, and dividing it to the maximum number of edges that that particular near-clique can have. The modularity value is between $[0, 1]$.

This function calculates the number of edges between the two partitions of near-cliques. In order to port it to the tripartite graphs, calculate the number of edges between the the three partitions of near-cliques.

4.3 Correctness

To show the correctness of our method we run the $BDMSCM$ on a small sized($15x10$) graph, which can be seen in figure 4.1. As a result, $BDMSCM$ gives us 2 near-cliques, which can be seen in figure 4.4 and 4.5. In the original graph, there are 2 near-cliques. The first near-clique consists of the nodes $1, 2, 4, 7, 9$ in the left partition, and nodes $16, 18, 19, 20, 22$ in the right partition. First near-clique has 25 edges. Second near-clique consists of the nodes $0, 3, 5, 10, 12$ in the left partition, and nodes $15, 23, 24$ in the right partition. Second near-clique has 15 edges. The matchings can be seen in figure 4.2. The reduced network that is constructed according to those matchings can be seen in figure 4.3. Figure 4.3 also shows the communities in the reduced network, found by Louvain algorithm. The nodes in the reduced network and corresponding matchings of original network are as follows:

- Node 0: $7 - 20$
- Node 1: $1 - 18$

- Node 2: $2 - 19$
- Node 3: $9 - 16$
- Node 4: $4 - 22$
- Node 5: $5 - 24$
- Node 6: $0 - 15$
- Node 7: $3 - 23$

Our algorithm finds 2 near-cliques which are sized as $3x3$, and $5x5$. First near-clique found consists of the nodes $0, 3, 5$ in the left partition and $15, 23, 24$ in the right partition. First near-clique has 9 edges. Second near-clique found consists of the nodes $1, 2, 4, 7, 9$ in the left partition, and $16, 18, 19, 20, 22$ in the right partition. Second near-clique has 25 edges. The results are fairly good. Our algorithm finds the second near-cliqe exactly, and finds the first near-clique with two less nodes in the left partition. Remaining nodes and edges can be seen in figure 4.6. The modularity of this example is 0.64.

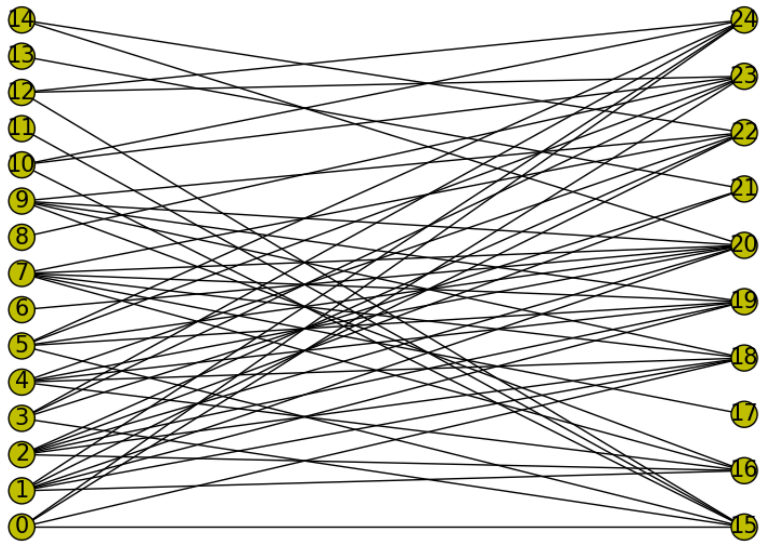

Figure 4.1: $15x10$ bipartite graph.

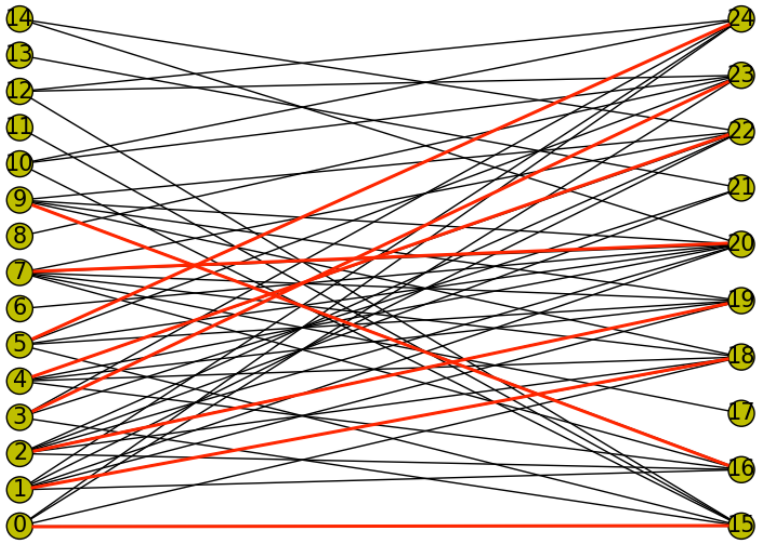

Figure 4.2: $15x10$, matchings.

4.4 Results

In this section, results of our experiments are presented. We tried our method with both bipartite and tripartite graphs. Since the Louvain algorithm chooses nodes randomly, the modularity values, and times, that are presented here, are the average of 50 different experiments. Experiment results are measured with the modularity formula explained in the section 4.2.1.

4.4.1 Bipartite Graphs

Results of the experiments carried out with the bipartite graphs are explained in this section. Figures 4.7, 4.8, 4.9, and 4.10 show the results of experiments on bipartite graphs. x-axis denotes the densities of the graphs, and y-axis shows the modularity values. If density is $1/200$, then there is one 1 in every other 200 cells. Each figure relates to a graph with a different size. First figure shows the results of an experiment carried out with a graph that has 40 nodes in the left partition, and 40 nodes in the right partition. Second figure shows the results of an experiment carried out with a graph that has 200 nodes in the left partition, and 250 nodes in the right partition.

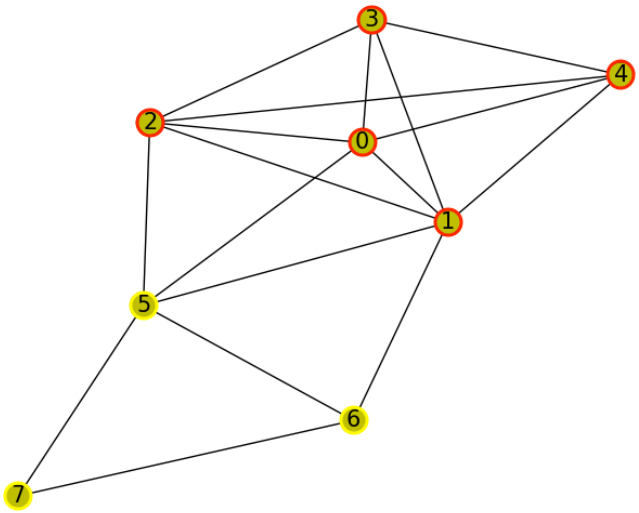

Figure 4.3: $15x10$, communities in the reduced network.

Third figure shows the results of an experiment carried out with a graph that has 1000 nodes in the left partition, and 1350 nodes in the right partition. Fourth figure shows the result of an experiment carried out with a graph that has 1500 nodes in the left partition, and 2050 nodes in the right partition. Lastly figure 4.11 shows the running times of the algorithms in average. Tables 4.1, 4.2, and 4.3 show the sizes of the cliques found by these algorithms. The sizes shown in these tables are the sizes of any one partition of the cliques. Table 4.6 shows the sizes and the densities of the cliques that are found by $BDMSCM$. These are the results of the experiments done with the most sparse graphs in the data set which are shown in figures 4.7, 4.8, 4.9, and 4.10. The modularity values of each set of cliques are shown in the figures 4.8, 4.9, and 4.10 respectively. Number of nodes in any partition of the graphs spans from 40 to 2050. Number of nodes in the graphs spans from 80 to 3550. The graphs have dense regions and different densities. For these experiments $BDMM$ are expected to perform worst, since it is the random case algorithm, and $BDMSCM$ is expected to perform best.

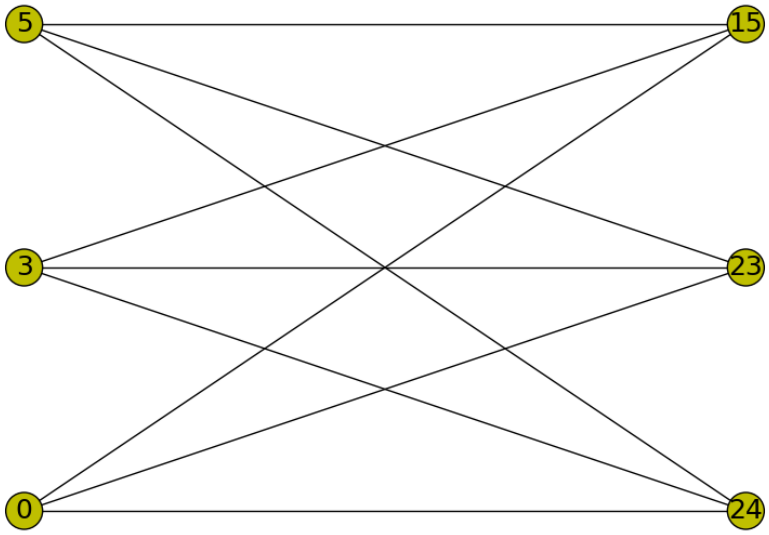

Figure 4.4: Near-clique $\#1$ found by $BDMSCM$.

The results are as follows:

- **Graph partition sizes:** $40x40$:

  It can be observed in the figure 4.7 that for a small graph, the modularities of the results of the four algorithms are intertwined, meaning that we can not say one is better than the other in terms of modularity. Modularity values are between 0.41 and 0.485, which are fair modularity values considering the example in the section 4.3, which has a size of $15x10$. The modularity of the cliques found with $BDMSCM$ in that graph is 0.64, and it finds the cliques almost perfect. This graph has a size of $40x40$, and note that modularity values drop with the increase in size of the graph.

- **Graph partition sizes:** $200x250$:

  It can be seen in the figure 4.8 that, for this set of inputs, $BDMSCM$ performs

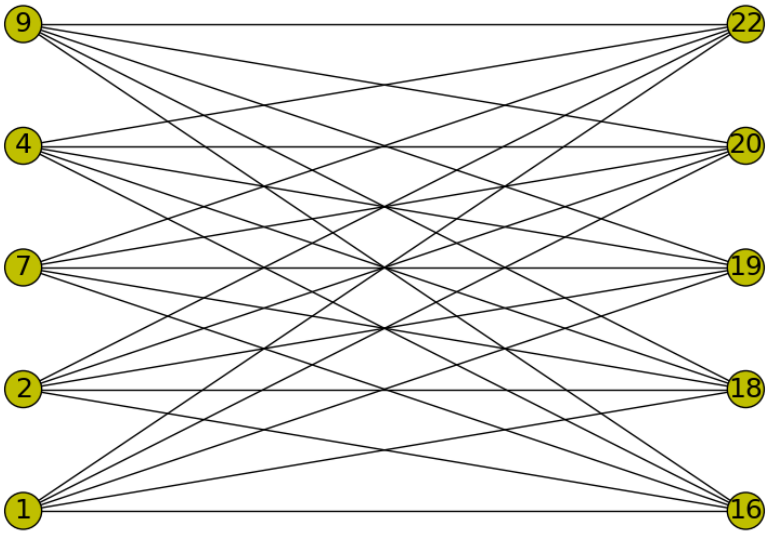

Figure 4.5: Near-clique $\#2$ found by $BDMSCM$.

the best. The modularity values of $BDMSCM$ increases with the decrease in the density of the graph. Table 4.1 shows that the size of cliques are decreasing with the decrease of density of the graph. However, modularities increase. It means that the cliques found on sparser networks are better in terms of modularity. This notion apply for all the algorithms for this data set. For small sized graphs, our algorithms performs better if graph is sparse and dense regions. Modularity values are between 0.23, and 0.33.

- **Graph partition sizes:** $1000x1350$:

It can be seen in the figure 4.9, modularity values decrease from the start until it reach the density of $1/75$, then values increase and continue to increase with the decrease in the density. $BDMM$ performs worst. $BDMSCM$ produces the best modularity values. Table 4.2 shows that with the decrease in density, number of cliques found by our algorithms decreases. Cliques are smaller but the modularity values are greater. This means that the cliques found in sparser graphs are better in terms of density and modularity. For this data set, our

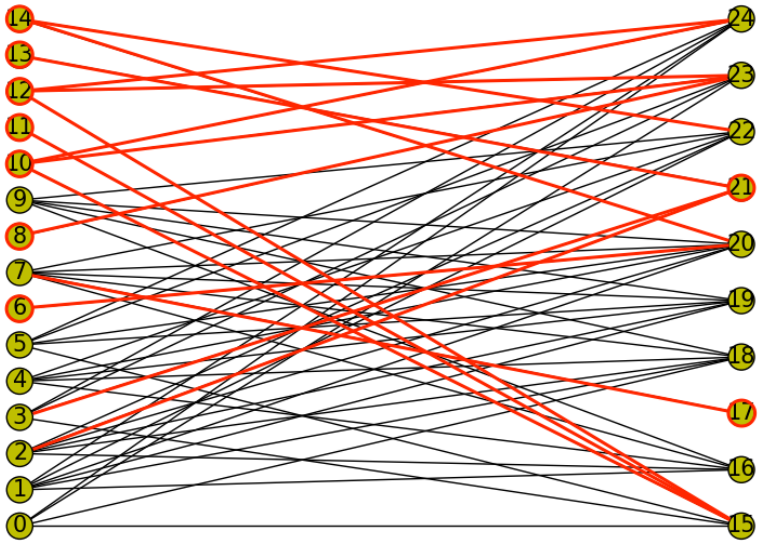

Figure 4.6: Remaining nodes.

algorithms perform better for sparser networks. Modularity values of cliques found by $BDMSCM$ are between 0.22, and 0.281. The difference between $BDMSCM$ and $BDMM$, the random case, starts to become noticeable in this data set. The difference in modularity is around 0.05 for sparse graphs.

- **Graph partition sizes:** $1500x2050$:

It can be seen in the figure 4.10 that similar to the previous experiment, modularity values decrease from the start until it reach the density $1/130$, then it increases and continues to increase with the decrease in the density. $BDMM$ performs worst. $BDMSCM$ produces the best modularity values. Similar to the previous experiments, table 4.3 shows that clique sizes decreases with the decrease in the density. Cliques are smaller but the modularity values are higher. It means that cliques found on sparser graphs are better in terms of modularity and density. Our algorithms perform better for the medium sized sparse graphs with dense regions. The difference between $BDMSCM$, and $BDMM$ can be seen here similar to the previous experiment. The difference is around 0.045.

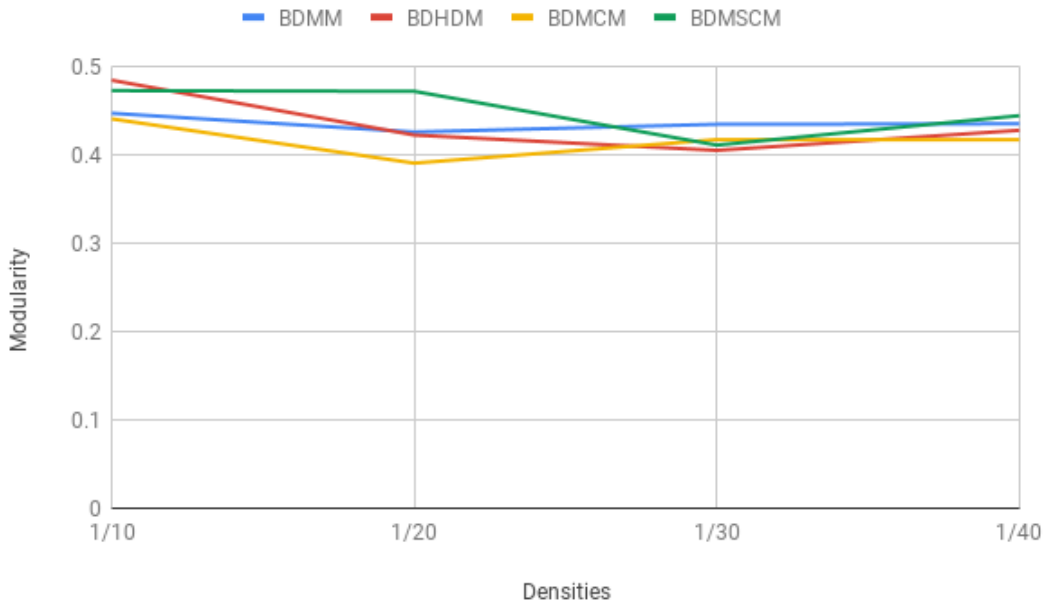

Figure 4.7: Bipartite Graph, Partition Sizes: $40x40$.

Results of experiments carried out with small and medium sized bipartite graphs are resulted as expected. $BDMM$ performs the worst, and $BDMSCM$ performs the best. All algorithms perform better on sparser graphs. In figure 4.11, we see the average time it takes to run the experiments for bipartite networks. This figure is plotted in log-scale. In this figure, x-axis shows the time in seconds, and y-axis shows the sizes of the bipartite graphs. Algorithm is fast and on a bipartite graph that has 1500 nodes in the first partition, and 2050 nodes in the second partition, $BDMSCM$ runs in 4.21 seconds. Slowest algorithm is the $BDMSCM$, but as expected, it has the best modularity values in the experiments. Since our method is fast, it is a small and sensible trade-off between speed and quality. Also quadratic time complexity can be seen clearly in figure 4.11.

4.4.2 Tripartite Graphs

Experiments carried out with the tripartite graphs are explained here. Figures 4.12, 4.13, and 4.14 show the results of experiments on tripartite graphs. x-axis denotes the size of the graphs, and y-axis denotes the modularities. Three sets of experiments

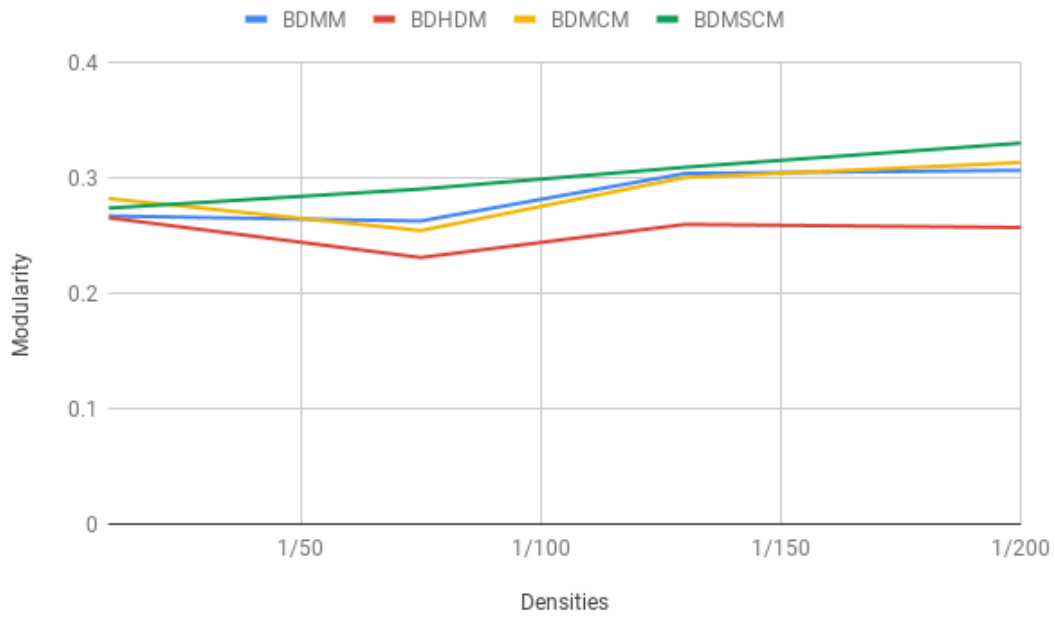

Figure 4.8: Bipartite Graph, Partition Sizes: $200x250$.

are carried out. Tables 4.4, and 4.5 show the sizes of the cliques found by these algorithms. The sizes shown in these tables are the sizes of any one partition of the cliques. Table 4.7 shows the sizes and the densities of the cliques that are found by $TDMSCM$. These are the results of the experiments done with the graphs shown in 4.14. The modularity values of each set of cliques are shown in the figures 4.12, 4.13 and 4.14 respectively. Each set differs on how the networks are constructed. How the networks are constructed are explained in the section 4.1. The sets include small graphs, medium graphs, and large graphs. Number of nodes in any partition of the graphs spans from 15 to 23000. Number of nodes in the graphs spans from 60 to 58500. First set consists of sparse matrices. Second set consists of sparse matrices with dense regions. The nodes of a dense region are closely located in the adjacency matrix in the second data set. Third set consists of graphs with dense regions with no constraints. Second set of experiments are carried out to set an upper limit for modularities with graphs with dense regions. Figure 4.12 shows the results of experiments carried out with sparse graphs. Figure 4.13 shows the results of experiments carried out with sparse graphs that have dense regions whose edges are close in the adjacency matrix. Figure 4.14 shows the results of experiments carried out with sparse graphs

200x250	**BDMM**	**BDHDM**	**BDMCM**	**BDMSCM**
10	26 27 28 41 51	29 30 33 37 43	19 24 31 37 57	14 32 35 35 55
75	24 24 26 26 26 28 31	7 21 23 24 27 28	19 20 22 27 47	8 17 17 19 20 25 35
130	1 1 1 1 2 20 24 24 26 27 27 35	16 18 20 21 23	16 17 17 22 36	16 16 17 19 36
200	1 1 1 1 1 1 1 23 23 26 26 28 29 31	14 20 29 33	12 16 16 24 38	14, 14, 14, 17, 19, 33

Table 4.1: Near-clique partition sizes. Bipartite graph partition sizes: $200x250$.

with dense regions. Lastly, figure 4.15 shows the running times of the algorithms in average. Since we saw that our algorithms works better for lower densities in the experiments carried out on bipartite graphs, all the graphs have high densities in these experiments.

The results are as follows:

- **Sparse Graphs:**
  In these experiments, the data sets include randomly created sparse graphs. Because of the fact that the graph is sparse, and graphs have no dense regions, the modularity values are low. Modularity values are between 0.041 and 0.0836 for small sized networks, and between 0.0209 and 0.0214 for medium sized networks. These modularities are low for both small and medium networks. The resultant modularities can be seen in figure 4.12. Since the graphs are random and sparse, the algorithms perform poorly. The number of cliques found can be seen in table 4.4.

- **Sparse Graphs with Dense Regions:**
  In these experiments, the data sets include randomly created sparse graphs that have dense regions. Nodes of dense regions are closely located in the adjacency matrix. $TDMCM$ is expected to perform best for these data sets. Figure 4.13

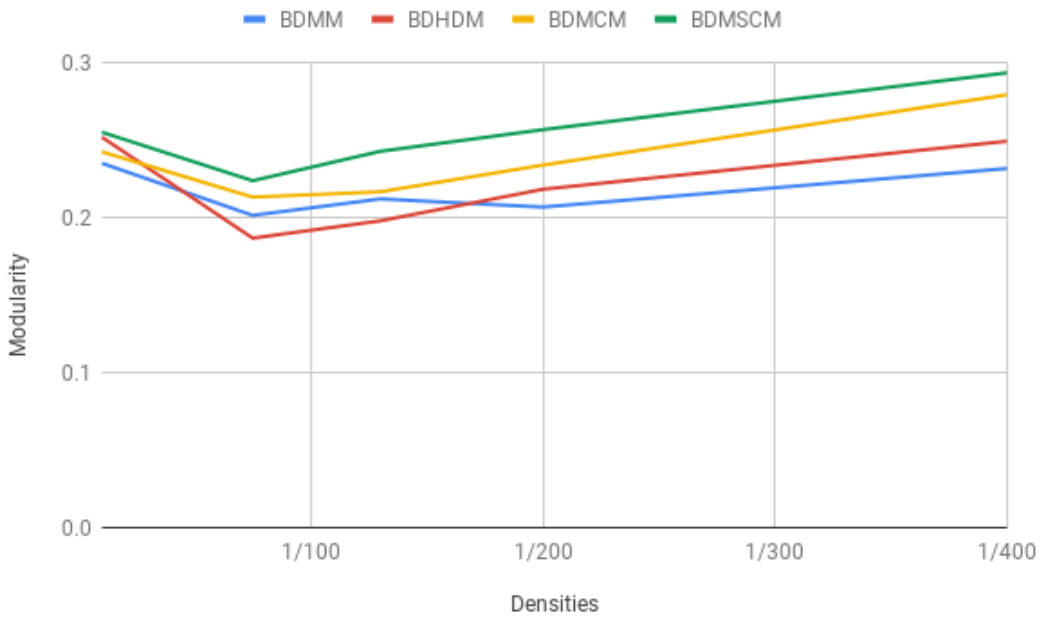

Figure 4.9: Bipartite Graph, Partition Sizes: $1000x1350$.

shows that $TDMCM$ performs better for the small sized graphs, but with the increase in size of the graph, $TDMSCM$ performs the best. The modularity values are between 0.26 and 0.86. Which are fairly good results for small and medium sized networks. Note that these experiments set an upper limit for modularities with graphs with dense regions.

- **Shuffled Sparse Matrices with Dense Regions:**
  In these experiments, the data set includes graphs with dense regions. This data set includes large graphs as well. It can be seen from figure 4.14 that the performance of $TDMSCM$ is the best. $TDMSCM$ has the highest values for small, medium, and large graphs. The modularity values of the resultant cliques are between 0.114, and 0.2 for small sized networks, between 0.074, and 1.02 for medium sized networks, between 0.064 and 0.1 for large networks. Number of nodes in the cliques found can be seen in the table 4.5. When we compare results on this table and 4.4, values in former are greater than the values in latter as expected. Also modularities of former is significantly greater than the latter. The modularity values of this data set is slightly lower than the previous one,

1000x1350	**BDMM**	**BDHDM**	**BDMCM**	**BDMSCM**
10	150, 151, 162, 209, 234	73, 146, 178, 208, 268	105, 106, 129, 238, 298	84, 128, 142, 199, 251
75	63, 90, 96, 110, 123, 240	111, 125, 315, 317	103, 105, 137, 141, 218	109, 112, 126, 139, 188
130	79, 94, 106, 171, 123, 255	3, 114, 120, 281, 289	86, 103, 121, 129, 214, 238	65, 96, 147, 256, 305
200	96, 100, 111, 236, 335	84, 87, 103, 265, 297	79, 91, 109, 167, 176, 242	67, 95, 103, 106, 187, 254
400	1, 1, 1, 99, 114, 120, 152, 187, 210	75, 92, 99, 100, 254	96, 106, 110, 142, 148, 196	61, 89, 99, 101, 128, 207

Table 4.2: Near-clique partition sizes. Bipartite graph partition sizes: $1000x1350$.

which sets an upper limit for the modularity.

In figure 4.15, we see the average time it takes to run the experiments in the third data set. This figure is plotted in log-log-scale. In this figure x-axis shows the time in seconds, and y-axis shows the sizes of the tripartite graphs. As expected, the bigger the graph, slower the algorithms runs. For a tripartite graph that has 2300 nodes in the first partition, 1500 nodes in the second partition, and 2050 nodes on the third partition running time of $TDMSCM$ is 11.36 seconds. For a tripartite graph that has 23000 nodes in the first partition, 15000 nodes in the second partition, and 20500 nodes on the third partition running time of $TDMSCM$ is 1519.99 seconds. Slowest algorithm is the $TDHDM$, which runs in 2262.54 seconds for the same graph. $TDMSCM$ has better modularity values in the experiments as expected. Table 4.7 shows that for medium sized graphs, densities of the near-cliques are higher. Quadratic time complexity can be seen clearly in figure 4.15. In overall the modularity values drops as the size increases because of the modularity function used. Since it divides the densities to the number of nodes in the graph, modularity is decreasing as the graphs get larger. In these experiments we saw that $TDMSCM$ performs better than $TDHDM$ and

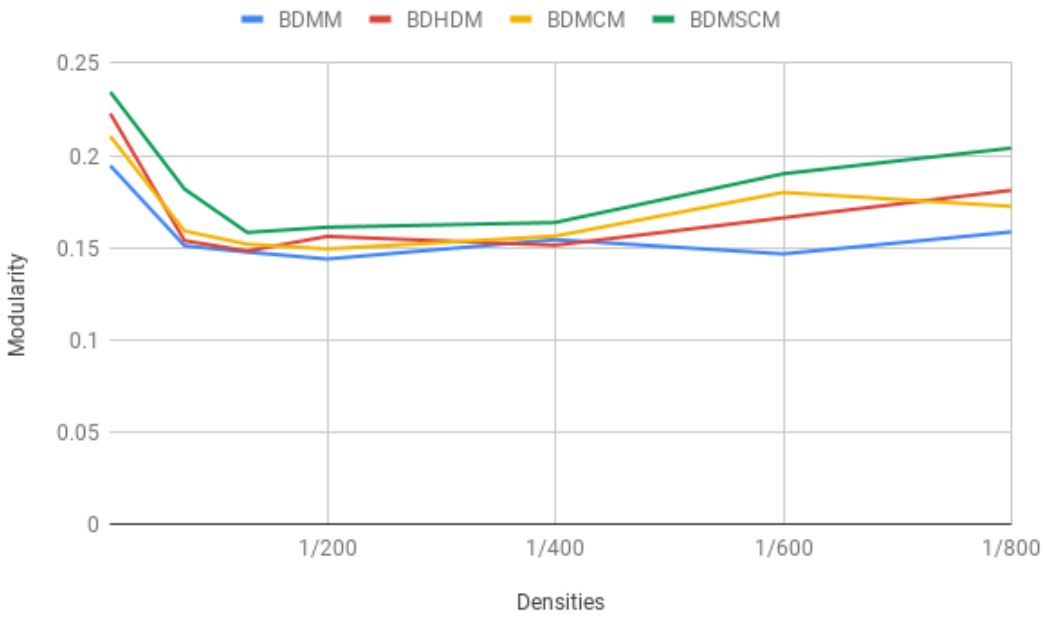

Figure 4.10: Bipartite Graph, Partition Sizes: $1500x2050$.

$TDMCM$, and produces fairly good modularity values.

1500x2050	**BDMM**	**BDHDM**	**BDMCM**	**BDMSCM**
10	210, 300, 311, 402	214, 226, 270, 577	185, 187, 361, 481	164, 177, 225, 242, 490
75	128, 326, 263, 335, 339	215, 246, 377, 581	171, 195, 304, 333, 385	187, 201, 243, 279, 431
130	187, 281, 301, 371	188, 318, 343, 361	171, 191, 302, 318, 380	192, 224, 324, 395
200	148, 191, 265, 287, 432	185, 196, 212, 277, 280	147, 154, 155, 214, 338	225, 246, 347, 397
400	6, 120, 189, 254, 278, 461	153, 204, 275, 351	169, 206, 345, 467	181, 207, 227, 329, 414
600	1, 2, 4, 215, 217, 268, 300, 485	73, 175, 316, 320	174, 224, 275, 428	159, 189, 264, 333
800	1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 175, 184, 236, 360, 523	2, 148, 166, 318, 319	1, 3, 282, 290, 422	2, 141, 178, 292, 374

Table 4.3: Near-clique partition sizes. Bipartite graph partition sizes: $1500x2050$.

	TDHDM	**TDMCM**	**TDMSCM**
60x60x60	5, 7, 7, 8	4, 5, 8, 10, 11	8, 8, 9, 9
120x150x150	8, 10, 10, 10, 11, 11, 12	5, 8, 9, 11, 12, 13, 18	9, 9, 10, 10, 13, 13
250x200x275	11, 11, 13, 15, 17, 17, 20	11, 11, 14, 15, 21, 22, 23, 26	12, 14, 18, 18, 20, 23, 27
1000x1350x500	29, 32, 35, 44, 46, 55, 67, 70, 92	21, 34, 36, 37, 49, 50, 54, 71, 74	24, 41, 41, 50, 51, 58, 61, 65, 66

Table 4.4: Near-clique partition sizes on sparse tripartite graphs.

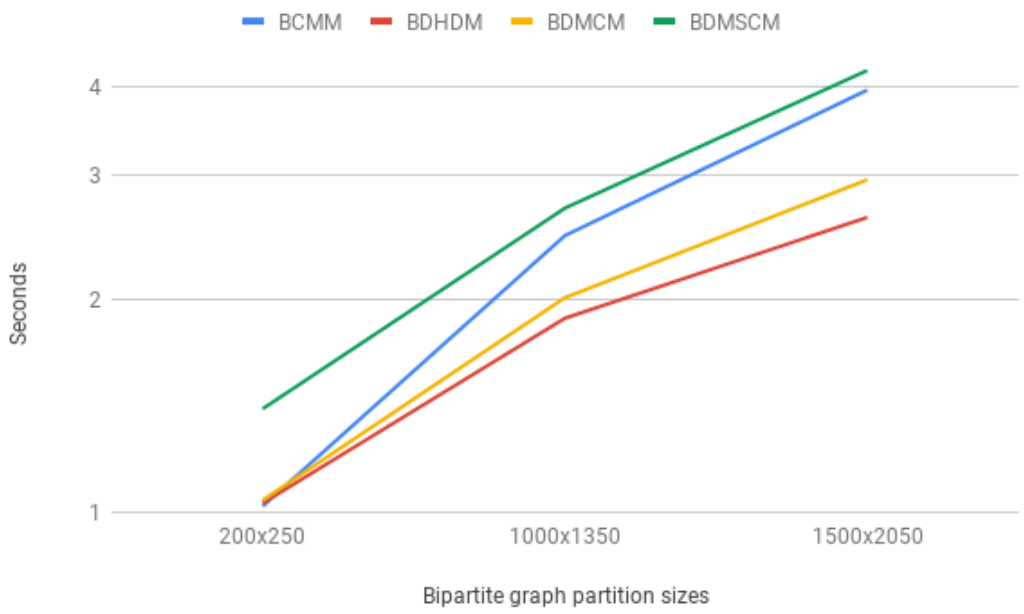

Figure 4.11: Running time of near-biclique detection algorithms.

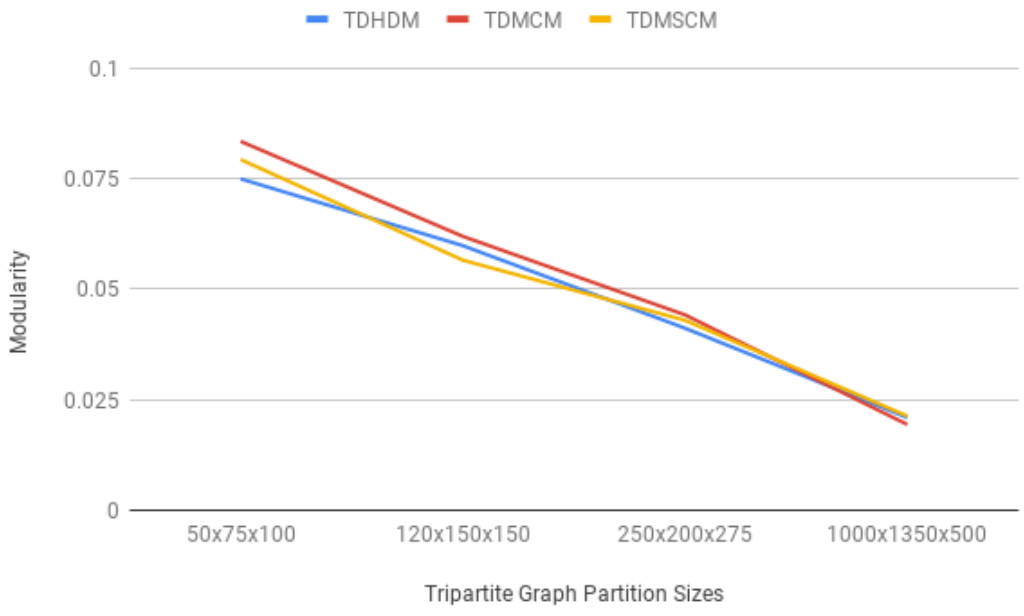

Figure 4.12: Sparse tripartite graphs.

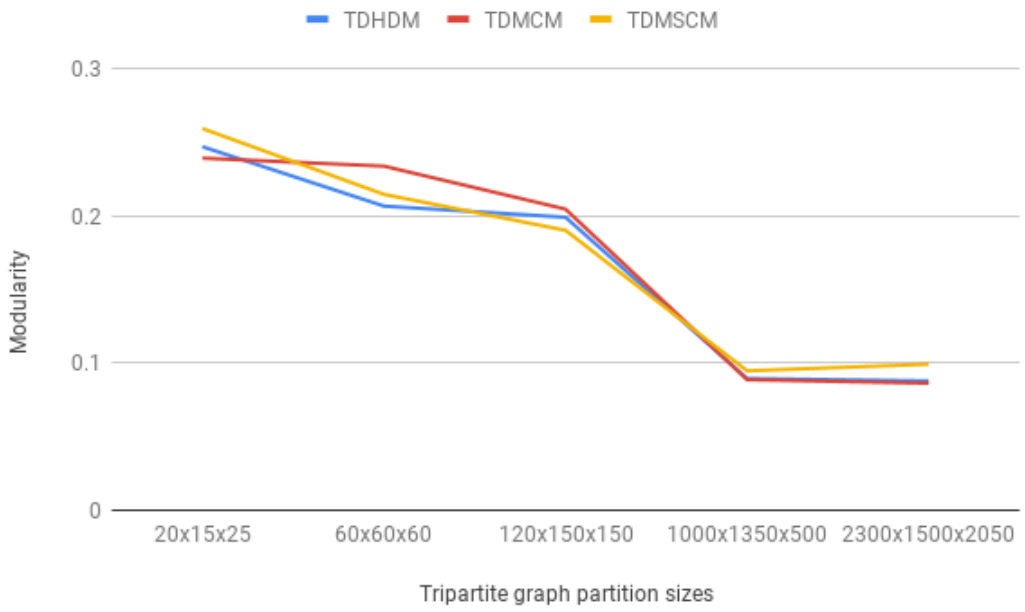

Figure 4.13: Sparse tripartite graphs with dense regions.

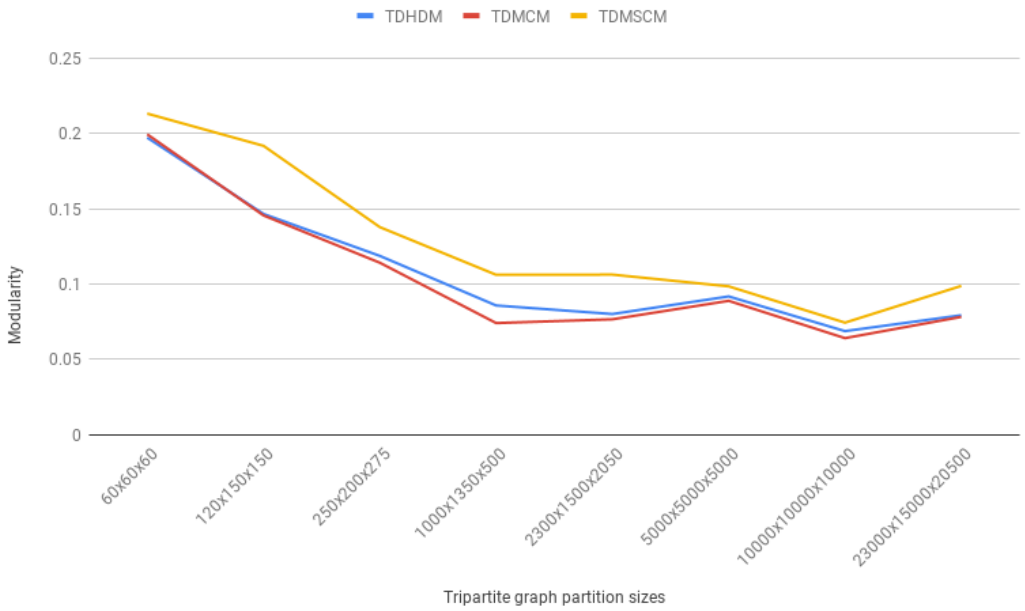

Figure 4.14: Shuffled sparse tripartite graph with dense regions.

	TDHDM	**TDMCM**	**TDMSCM**
60x60x60	10, 16	12, 18	9, 17
120x150x150	26, 30	35, 35	25, 44
250x200x275	35, 40, 47	27, 36, 62	50, 50
1000x1350x500	34, 54, 62, 127	54, 73, 74, 82, 124	32, 33, 37, 183
2300x1500x2050	156, 165, 218, 226, 281	87, 112, 144, 144, 270, 229	32, 37, 159, 183
5000x5000x5000	273, 355, 440, 455, 531, 542	436, 465, 517, 534, 567, 725	393, 609, 798, 780
10000x10000x10000	716, 795, 1118, 1520, 1572	412, 538, 1781, 1329, 1692	1, 4, 639, 1341, 1435, 1502, 1526
23000x15000x20500	1049, 1378, 1747, 1581, 2323, 3249	996, 1157, 1405, 1555, 1980, 2081, 2478	972, 2499, 2843, 3954

Table 4.5: Near-clique partition sizes on sparse tripartite graph with dense regions.

	Densities	**Sizes of Partitions in Cliques**
40x40	0.750, 0.906	2, 16
200x250	0.674, 0.719, 0.837, 0.568, 0.803, 0.451	14, 14, 14, 17, 19, 33
1000x1350	0.478, 0.556, 0.461, 0.508, 0.363, 0.331	61, 89, 99, 101, 128, 207
1500x2050	0.750, 0.478, 0.333, 0.230, 0.241	2, 141, 178, 292, 374

Table 4.6: Densities of the near-cliques found by $BDMSCM$ on bipartite graphs.

	Densities	**Sizes of Partitions in Cliques**
1000x1350x500	0.255, 0.246, 0.277, 0.204	32, 33, 37, 183
2300x1500x500	0.255, 0.278, 0.232, 0.208	32, 37, 159, 183
5000x5000x5000	0.196, 0.105, 0.240, 0.102	393, 609, 780, 798
10000x10000x10000	0.667, 0.229, 0.0458, 0.208, 0.076, 0.128, 0.0876	1, 4, 639, 1341, 1435, 1502, 1526

Table 4.7: Densities of the near-cliques found by $TDMSCM$ on tripartite graphs.

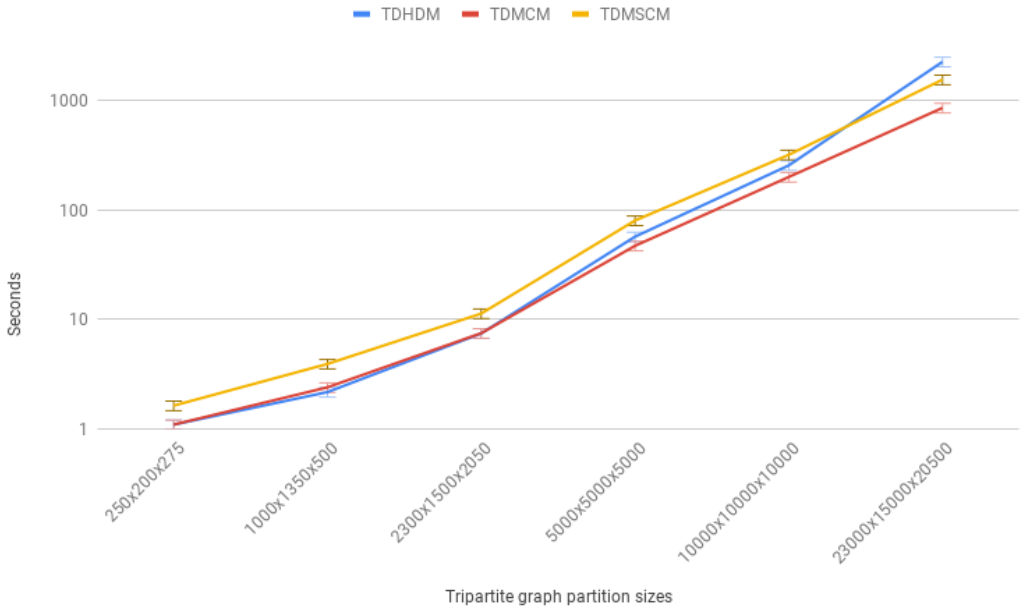

Figure 4.15: Running time of near-triclique detection algorithms.

CHAPTER 5

CONCLUSION AND FUTURE WORK

In this thesis, we investigate the near-biclique detection problem and near-triclique detection problem. We compared seven novel approaches to solve these problems. Four of the algorithms we propose are used for the former, and three of them are used for the latter. Since the problems are $NP - complete$, we do not look for the optimal solutions. The optimal solution is not scalable for large graphs. In this work, we make a trade-off between speed and quality in favor of speed. We use Louvain algorithm in our methods, because it is fast and produces fairly good results. The time complexity of Louvain algorithm is $O(nlogn)$, which is explained in detail in section 3.3.2.

In our document, we first briefly introduced the community detection problem and mentioned the historical background of community detection. In Chapter 3, we explained the methods proposed in this thesis. And lastly in chapter 4 we examined and explained the experiment datum, metrics, environment, and the results.

In our study, we present agglomerative, modularity-based, novel methods to solve the near-clique problem for bipartite and tripartite graphs. We believe these novel approaches have a bright future, and can be improved further. Our methods solve the problem for unweighted, undirected bipartite graphs and tripartite graphs, and produce near-cliques that have same number of nodes in it's partitions. Out work can be updated for weighted, directed graphs. Also we believe that this method can be improved so that it can be a general solution for community detection. In other words, solves the multi-partite community detection problem.

Through our experiments, we have found that $BDMSCM$ performs better than the other proposed methods for bipartite graphs, and $TDMSCM$ performs better than the other proposed algorithms for tripartite graphs. They are fast and produces fair results. We have also found out that these algorithms perform better for the sparse graphs that have dense regions.

REFERENCES

[1] L. Euler *Commentarii Academiae Petropolitanae*, 1726.

[2] V. E. Lee, N. Ruan, R. Jin, and C. Aggarwal, *A Survey of Algorithms for Dense Subgraph Discovery*, pp. 303–336. Boston, MA: Springer US, 2010.

[3] D. Gibson, R. Kumar, and A. Tomkins, "Discovering large dense subgraphs in massive graphs," VLDB '05, pp. 721–732, VLDB Endowment, 2005.

[4] G. W. Flake, S. Lawrence, and C. L. Giles, "Efficient identification of web communities," in *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, (New York, NY, USA), pp. 150–160, ACM, 2000.

[5] D. Zhang, J. Dong, A. Nagurney, A. Nagurney, and E. E. Publishing, "Innovations in financial and economic networks," 2003.

[6] A. K. Jain, "Data clustering: 50 years beyond k-means," *Pattern Recognition Letters*, vol. 31, no. 8, pp. 651 – 666, 2010. Award winning papers from the 19th International Conference on Pattern Recognition (ICPR).

[7] R. M. Karp, *Reducibility among Combinatorial Problems*, pp. 85–103. Boston, MA: Springer US, 1972.

[8] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, 2008.

[9] R. Boppana, "Eigenvalues and graph bisection: An average-case analysis," pp. 280–285, 11 1987.

[10] A. Pothen, *Graph Partitioning Algorithms with Applications to Scientific Computing*, pp. 323–368. Dordrecht: Springer Netherlands, 1997.

[11] G. K. P. R. Suair, "An algorithm for quadrisection and its application to standard cell placement," *IEEE Transactions on Circuits and Systems*, vol. 35, pp. 294 – 303, 3 1988.

[12] E. Barnes, "An algorithm for partitioning the nodes of a graph," *SIAM Journal on Algebraic Discrete Methods*, vol. 3, no. 4, pp. 541–550, 1982.

[13] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

[14] P. R. Suaris and G. Kedem, "An algorithm for quadrisection and its application to standard cell placement," *IEEE Transactions on Circuits and Systems*, vol. 35, pp. 294–303, March 1988.

[15] B. Bollobás, *Random Graphs*, pp. 215–252. New York, NY: Springer New York, 1998.

[16] P. Pons and M. Latapy, "Computing communities in large networks using random walks," in *Computer and Information Sciences - ISCIS 2005* (p. Yolum, T. Güngör, F. Gürgen, and C. Özturan, eds.), (Berlin, Heidelberg), pp. 284–293, Springer Berlin Heidelberg, 2005.

[17] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[18] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Phys. Rev. E*, vol. 69, p. 026113, Feb 2004.

[19] F. Radicchi, C. Castellano, F. Cecconi, V. Loreto, and D. Parisi, "Defining and identifying communities in networks," *Proceedings of the National Academy of Sciences*, vol. 101, no. 9, pp. 2658–2663, 2004.

[20] S. Şahin Koç, M. Özer, İsmail Hakkı Toroslu, H. Davulcu, and J. Jordan, "Triadic co-clustering of users, issues and sentiments in political tweets," *Expert Systems with Applications*, vol. 100, pp. 79 – 94, 2018.

[21] J. Macqueen, "Some methods for classification and analysis of multivariate

observations," in *In 5-th Berkeley Symposium on Mathematical Statistics and Probability*, pp. 281–297, 1967.

[22] M. J. Rattigan, M. Maier, and D. Jensen, "Graph clustering with network structure indices," in *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, (New York, NY, USA), pp. 783–790, ACM, 2007.

[23] J. C. Dunn†, "Well-separated clusters and optimal fuzzy partitions," *Journal of Cybernetics*, vol. 4, no. 1, pp. 95–104, 1974.

[24] W. Donath and A. J. Hoffman, "Lower bounds for the partitioning of graphs," *IBM Journal of Research and Development*, vol. 17, pp. 420–425, 09 1973.

[25] H. Du, M. W. Feldman, S. Li, and X. Jin, "An algorithm for detecting community structure of social networks based on prior knowledge and modularity," *Complexity*, vol. 12, no. 3, pp. 53–60, 2007.

[26] L. Danon, A. Díaz-Guilera, J. Duch, and A. Arenas, "Comparing community structure identification," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2005, no. 09, p. P09008, 2005.

[27] A. Noack and R. Rotta, "Multi-level algorithms for modularity clustering," in *Experimental Algorithms* (J. Vahrenhold, ed.), (Berlin, Heidelberg), pp. 257–268, Springer Berlin Heidelberg, 2009.

[28] J. Mei, S. He, G. Shi, Z. Wang, and W. Li, "Revealing network communities through modularity maximization by a contraction–dilation method," *New Journal of Physics*, vol. 11, no. 4, p. 043025, 2009.

[29] P. Schuetz and A. Caflisch, "Efficient modularity optimization by multistep greedy algorithm and vertex mover refinement," *Phys. Rev. E*, vol. 77, p. 046112, Apr 2008.

[30] P. Schuetz and A. Caflisch, "Multistep greedy algorithm identifies community structure in real-world and computer-generated networks," *Phys. Rev. E*, vol. 78, p. 026112, Aug 2008.

[31] K. Wakita and T. Tsurumi, "Finding community structure in mega-scale social networks: [extended abstract]," in *Proceedings of the 16th International Con-*

ference on World Wide Web, WWW '07, (New York, NY, USA), pp. 1275–1276, ACM, 2007.

[32] A. Clauset, M. E. J. Newman, and C. Moore, "Finding community structure in very large networks," *Phys. Rev. E*, vol. 70, p. 066111, Dec 2004.

[33] J. M. Pujol, J. Béjar, and J. Delgado, "Clustering algorithm for determining community structure in large networks," *Phys. Rev. E*, vol. 74, p. 016107, Jul 2006.

[34] B. Xiang, E.-H. Chen, and T. Zhou, *Finding Community Structure Based on Subgraph Similarity*, pp. 73–81. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[35] M. E. J. Newman, "Fast algorithm for detecting community structure in networks," *Phys. Rev. E*, vol. 69, p. 066133, Jun 2004.

[36] R. Guimerà, M. Sales-Pardo, and L. A. N. Amaral, "Modularity from fluctuations in random graphs and complex networks," *Phys. Rev. E*, vol. 70, p. 025101, Aug 2004.

[37] C. P. Massen and J. P. K. Doye, "Identifying communities within energy landscapes," *Phys. Rev. E*, vol. 71, p. 046101, Apr 2005.

[38] A. Medus, G. Acuña, and C. Dorso, "Detection of community structures in networks via global optimization," *Physica A: Statistical Mechanics and its Applications*, vol. 358, no. 2, pp. 593 – 604, 2005.

[39] M. E. J. Newman, "Modularity and community structure in networks," *Proceedings of the National Academy of Sciences*, vol. 103, no. 23, pp. 8577–8582, 2006.

[40] S. Lehmann and L. K. Hansen, "Deterministic modularity optimization," *The European Physical Journal B*, vol. 60, pp. 83–88, Nov 2007.

[41] S. White and P. Smyth, *A Spectral Clustering Approach To Finding Communities in Graphs*, pp. 274–285.

[42] L. Danon, A. Díaz-Guilera, J. Duch, and A. Arenas, "Comparing community structure identification," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2005, no. 09, p. P09008, 2005.

[43] U. Brandes, D. Delling, M. Gaertler, R. Gorke, M. Hoefer, Z. Nikoloski, and D. Wagner, "On modularity clustering," *IEEE Transactions on Knowledge and Data Engineering*, vol. 20, pp. 172–188, Feb 2008.

[44] D. B. West, *Introduction to Graph Theory*. Prentice Hall, 2 ed., September 2000.

[45] D. Gale and L. S. Shapley, "College admissions and the stability of marriage," *The American Mathematical Monthly*, vol. 69, no. 1, pp. 9–15, 1962.

[46] S. Miyazaki and K. Iwama, "A survey of the stable marriage problem and its variants," in *2008 International Conference on Informatics Education and Research for Development of Knowledge Society Infrastructure(ICKS)*, vol. 00, pp. 131–136, 01 2008.

[47] U. Brandes, D. Delling, M. Gaertler, R. Goerke, M. Hoefer, Z. Nikoloski, and D. Wagner, "Maximizing Modularity is hard," *arXiv e-prints*, p. physics/0608255, Aug. 2006.

[48] F. Wu and B. A. Huberman, "Finding communities in linear time: a physics approach," *The European Physical Journal B*, vol. 38, pp. 331–338, Mar 2004.

[49] M. Barber, "Modularity and community detection in bipartite networks," *Physical review. E, Statistical, nonlinear, and soft matter physics*, vol. 76, p. 066102, 01 2008.

[50] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[51] G. Goudswaard, R. A. Fisher, and F. Yates, "Statistical tables for biological, agricultural and medical research," *Revue de l'Institut International de Statistique / Review of the International Statistical Institute*, vol. 31, no. 3, p. 449, 1963.