MINING INTERESTING SUBNETWORKS FROM GRAPHS WITH NODE ATTRIBUTES

A Dissertation
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Aditya Praneeth Goparaju

In Partial Fulfillment of the Requirements
for the Degree of
DOCTOR OF PHILOSOPHY

Major Department:
Computer Science

April 2018

Fargo, North Dakota

NORTH DAKOTA STATE UNIVERSITY

Graduate School

Title

MINING INTERESTING SUBNETWORKS FROM GRAPHS WITH NODE ATTRIBUTES

By

Aditya Praneeth Goparaju

The supervisory committee certifies that this dissertation complies with North Dakota State University's regulations and meets the accepted standards for the degree of

DOCTOR OF PHILOSOPHY

SUPERVISORY COMMITTEE:

Dr. Saeed Salem
Chair

Dr. Kendall Nygard

Dr. Simone Ludwig

Dr. Mukhlesur Rahman

Approved:

April 20 2018 — Date

Dr. Saeed Salem — Department Chair

ABSTRACT

A lot of complex data in many scientific domains such as social networks, computational biology and internet of things (IoT) is represented using graphs. With the global expansion of internet, social networks had an explosive growth with billions of users in FaceBook. Similarly research in Bio-informatics generated massive amounts of genomic data (protein protein interaction networks) from several high throughput techniques. Due to the large amount of data involved, researchers have turned to data mining techniques to discover meaningful and relevant information from large graphs.

One of the most intriguing questions in graphs representing complex data is to find communities or clusters. The members in a clusters have high density of edges to other members within the cluster while very low edges to members outside of the cluster. Real world graphs often have additional attribute data characterizing either the nodes or edges of a graph, such as age or interests of a person in a social network. Recent research has combined the problem of community detection with subspace similarity over attribute data. For example, in the context of social networks, we might be interested in finding groups of friends who are of similar age and share common interests. The use of attribute data in finding clusters is shown to be effective in many application areas such as targeted advertising in social network or detecting protein complexes in protein protein interaction networks which might be indicative of diseases such as cancer.

In this dissertation, we propose multiple algorithms for mining communities with similarity in attributes from node-attributed graphs. Experiments on real world datasets show that the proposed approach is effective in mining meaningful clusters.

ACKNOWLEDGEMENTS

I would like to express my thanks to my advisor Dr. Saeed Salem for supporting me during all these years. He has been a tremendous mentor, his advice and help during my research has been invaluable. I would like to thank him for encouraging and helping me in my research, without which this research wouldn't have been possible. I would like to express my gratitude to all my committee members, Dr. Hyunsook Do, Dr. William Perrizo and Dr. Gokhan Egilmez, Dr. Kendall Nygard, Dr. Simone Ludwig and Dr. Mukhlesur Rahman for taking the time to evaluate my project and helping me to graduate from the University.

I also want to express my thanks for North Dakota State University and the Department of Computer Science for offering me the opportunity to study at such a great school. The people and the environment have made it a great place to study and learn.

I thank my fellow students, Kutub Syed and Tyler Brazier, Ming Chao and Bassam Qormosh for working with and beside me during my research. Lastly I also want to thank my wife for supporting and encouraging me through out this journey.

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGEMENTS iv

LIST OF TABLES ix

LIST OF FIGURES x

1. INTRODUCTION 1

1.1. Motivation examples 2

1.1.1. Biology 2

1.1.2. Social networks 2

1.1.3. Enron email data set 3

1.1.4. Developer networks in open source software 3

1.2. Goal of this thesis 4

1.3. Organization of the thesis 4

2. RELATED WORK 6

2.1. Communities 6

2.2. Community detection literature 7

2.2.1. Graph partitioning techniques 7

2.2.2. Hierarchical clustering 7

2.2.3. Spectral clustering 7

2.2.4. Markov clustering 8

2.2.5. Modularity 8

2.2.6. Enumeration tree based community detection 8

2.3. Cohesive community detection 10

2.3.1. Attributes in graphs 10

2.3.2. Node attributes 10

2.3.3. Edge attributes 11
2.4. Enumeration algorithms for cohesive community detection 11
2.4.1. GAMer 12
2.4.2. DME 12
2.4.3. DECOB 12
3. MINING DENSE COHESIVE SUBNETWORKS 13
3.1. Problem description 13
3.2. Algorithm 16
3.2.1. RedCone approach 16
3.2.2. Multithreaded RedCone 18
3.3. Representative Set 19
3.3.1. Finding Similarity Scores 20
3.3.2. Similarity Graph 22
3.3.3. Representative clusters - Set cover approach 22
3.3.4. Representative clusters - K-Medoids approach 22
3.4. Experiments 25
3.4.1. Cohesive clusters in YeastHC 25
3.4.2. Cohesive clusters in BioGRID 27
3.4.3. Running Time 27
3.4.4. Multithreaded Runtime 29
3.4.5. Representative set 30
4. MINING COHESIVE SUBNETWORKS 35
4.1. Problem description 36
4.2. Algorithm 36
4.2.1. Brute force approach 36
4.2.2. MinCone approach 37

4.2.3. Multithreaded MinCone approach 40
4.3. Experiments 43
4.3.1. Cohesive clusters in the YeastHC 43
4.3.2. Running Time 44
4.3.3. Multithreaded Runtime 45
5. SAMPLING DENSE AND COHESIVE NETWORKS 47
5.1. Sampling 47
5.2. Related work 48
5.2.1. Vertex based sampling 49
5.2.2. Edge based sampling 49
5.2.3. Traversal based sampling 49
5.3. Problem Description 50
5.4. Sampling algorithm 52
5.4.1. Metropolis-Hastings algorithm 53
5.4.2. Random walk 55
5.4.3. Uniform sampling 55
5.4.4. Targeted sampling - sampling dense and cohesive modules 56
5.5. Experiments 57
5.5.1. Uniform sampling 58
5.5.2. Targeted sampling 59
6. CONCLUSION AND FUTURE WORK 65
6.1. Conclusion 65
6.1.1. Mining dense and cohesive sub networks 65
6.1.2. Mining cohesive sub networks 66
6.1.3. Sampling dense and cohesive sub networks 66
6.2. Future work 67

6.2.1. Cohesive communities with noisy node attributes 67

6.2.2. Cohesive communities with edge attributes 67

6.2.3. Cohesive communities with ranked edge interaction data 68

6.2.4. Sampling cohesive communities with multi relational edge attribute data . . 70

6.2.5. Parallel approach to sampling 72

REFERENCES 75

LIST OF TABLES

Table | Page

3.1. Topological properties of cohesive dense clusters for Yeast dataset. 26

3.2. Topological properties of cohesive dense clusters for Biogrid dataset. 29

3.3. Similarities of medoids in Representative Set. 33

4.1. Topological properties of cohesive clusters for the YeastHC dataset. 43

5.1. Summary statistics of the visit counts of the samples in uniform sampling which includes minimum, maximum, mean and standard deviation of the visit counts. 59

5.2. Results of sampling algorithm on Yeast dataset. 60

5.3. Results of sampling algorithm on Human dataset. 62

5.4. Traversal of enumeration and sampling algorithm. 62

5.5. Biological enrichment analysis on modules generated by sampling algorithm in human dataset. 64

LIST OF FIGURES

Figure Page

2.1. Visualizing communities in a sample web site graph 6

2.2. Enumeration tree example. (a) A sample graph. (b) Enumeration tree for the sample graph; node order $1 < 2 < 3 < 4$ is followed while generating child nodes. 9

2.3. Graph with node attributes and two cohesive communities. 10

2.4. Graph with edge attributes and two cohesive communities. 11

3.1. Example node attribute graph. The graph shows two communities, vertex 4 belongs to both communities. 13

3.2. Example graph (a) and its enumeration tree (b). $\theta = 0.7$. Crosses show which branches are pruned. The discovered maximal clusters are in green. 16

3.3. The input graph and its corresponding enumeration tree. There are four threads which build the enumeration tree independently. There are 4 threads and each thread builds subtree for 2 first level children. 19

3.4. Finding representative clusters. (a) The maximal cohesive clusters. (b) The cluster similarity graph for $\alpha = 0.5$, edges show dissimilarity between clusters; (c) Applying k-medoids algorithm results in finding clusters and medoids (highlighted in blue), edges show distances between clusters. 21

3.5. Representative clusters. (a) Set of maximal cohesive dense clusters, (b) Projecting maximal cohesive dense clusters to points in space. (c) Finding partitions around the random medoids. (d) Detecting steady state medoids $m1 = P3$ and $m2 = P2$ and its partitions $c1$ and $c2$ respectively. The representative maximal cohesive dense clusters are the set of steady state medoids $m1, m2$ 23

3.6. Functional interpretation of patterns: GOTERMs and KEGG pathways enrichment for the YeastHC dataset for $t = 0.3, s_{min} = 30$. 27

3.7. Representation of a dense and cohesive patterns from the YeastHC dataset, showing their network structure and attribute similarity of nodes in each pattern (a) number of vertices = 8 (b) number of vertices = 9; Parameters: $\theta = 0.7$, $t = 0.5$ and $s_{min} = 20$ (Only 40 attributes are shown) 28

3.8. Runtime comparison of GAMer, DECOB and RedCone on the YeastHC dataset (a) parameters : $t = 0.4$ and $s_{min} = 20$ (b) parameters : $\theta = 0.6$ and $t = 0.4$ 28

3.9. Runtime comparison of DECOB and RedCone on the BioGRID dataset (a) parameters : $t = 0.4$ and $s_{min} = 60$ (b) parameters : $\theta = 0.7$ and $t = 0.4$ 29

3.10. Runtime comparison of multiple threads on the Yeast dataset with varying density, parameters : (a) $t = 0.4$ and $s_{min} = 60$ (b) $t = 0.3$ and $s_{min} = 65$ 31

3.11. Runtime comparison of multiple threads on the Yeast dataset with varying dimension, parameters : (a) $t = 0.4$ and $\theta = 0.9$ (b) $t = 0.4$ and $\theta = 0.7$ 31

3.12. Runtime comparison of multiple threads on the Human dataset with varying density, parameters : $s_{min} = 4$ 32

3.13. Runtime comparison of multiple threads on the Human dataset with varying dimension, parameters : $\theta = 0.6$ 32

3.14. Average intra partition similarity $AvgPartitionSim$ and average inter medoid similarity $AvgMedoidSim$ with varying alpha values. 33

3.15. Average intra partition similarity $AvgPartitionSim$ and average inter medoid similarity $AvgMedoidSim$ with varying K values. 34

4.1. Example of cohesive network. Graph containing two communities, (a) An example of dense cohesive subgraph and (b) an example of a cohesive subgraph 35

4.2. MinCone approach.(a) Sample enumeration tree where A, B, U, U^* are all cohesive clusters, cluster U is the parent of the cluster U^* (b) cohesive cluster represented by U (c) child cohesive cluster represented by U^* (d) MST of U^*. 39

4.3. The input graph and a portion of the corresponding enumeration tree built by MT_Mincone . There are four threads which build parts of the enumeration tree independently. Each thread builds two subtrees from the first level children. Crosses show which branches are pruned. The discovered maximal cluster is highlighted by a green box. 41

4.4. Representation of a cohesive pattern from the YeastHC dataset (a) Network structure of the pattern (b) Attributes of the nodes in the pattern; Parameters: $t = 0.4$ and $s_{min} = 40$ (Only 80 attributes are shown). Notice the pattern is not particularly dense . 44

4.5. Runtime comparison of brute force approach and MineCone on the YeastHC dataset (a) parameters : $t = 0.4$ (b) parameters : $s_{min} = 40$ 44

4.6. Speedup in runtime for multiple threads (a) on the Yeast dataset with varying dimension while tolerance is fixed at $t = 0.35$ (b) on the Yeast dataset with varying tolerance while dimension is fixed at $s_{min} = 30$ (c) on Human dataset with varying dimension 45

5.1. (a) Module enumeration process enumerates all modules from the input graph. Modules are shows as circles and representative modules are shown by filled circles. Summarization process then tries to reduce this exponential output set to a representative set. (b) Sampling technique outputs a reduced set directly from the input graph database without enumeration. 48

5.2. A input graph G and the corresponding POG represented by M, assuming every module of G is cohesive and dense. 51

5.3. Uniform sampling where $|N| = 107$, sampling algorithm ran for 10700 iterations (a) Visit counts of each sample (b) Histogram of visit counts 59

5.4. Targeted sampling where $|N| = 3819$, sampling algorithm ran for 75000 iterations (a) Visit counts of each sample (b) Scatter plot graph between score of the module and the visit count of each module. The blue line indicates the positive correlation between score of the module and visit count. 60

6.1. Graph with edge attributes and two communities. 68

6.2. Graph with ranked edge attributes. The dashed ellipse shows a community formed by including the topics from a broadcast message. 69

6.3. A graph containing multi relational edges, (a) An example of co-authorship network where two authors co-authored a paper and (b) an example of citation network where authors cited each other. In this example the community formed by $\{1, 2, 5, 6\}$ is dense in both co-authorship and citation network with a minimum density of 0.6, assuming they are cohesive in their edge attributes 70

6.4. DBLP graph. (a) An example DBLP graph with multiple relational edges, solid edges are co-authorship and dashed edges are citation edges. (b) Partial order graph for input graph in (a) with co-authorship topology. (c) Partial order graph for input graph in (a) with citation topology. Notice that subgraph $\{A, C\}$ does not exist in POG for citation topology. 72

6.5. Multi threaded approach to sampling in POG with two threads (Th1 and Th2) performing parallel random walk. Each thread is restricted to its respective area highlighted by color. 73

1. INTRODUCTION

Biological, social and technological networks have been modeled as graphs, and graph analysis has become crucial to understand these complex systems. In each of these areas, a vertex represents a gene, person or a node and their interaction or relationship is represented by an edge. Often these vertices or edges have properties associated with them which can be modeled as attributes. These properties for example, can represent the personal profile attributes such as age or interests of a person in a social network or gene expression data which encodes information that can determine the dysregulation of a gene in a disease [21].

One of the most intriguing questions in graphs representing complex data is to find communities or clusters [26]. Communities are groups of vertices of a graph that have a high concentration of edges within the group and very low concentration of edges between these groups. With the availability of attribute data it is highly desirable to find communities which also exhibit similarity over its attribute data. A community can be viewed as an independent region of a graph, where all the vertices or edges exhibit similar properties or behavior. Communities that have a dense network structure and maintain attribute similarity are called cohesive communities.

Cohesive community detection has received some attention recently [77, 36, 30], however, this concept is still fairly new and requires further study. Most of the recent research have some sort of limitations. Some approaches use a stricter representation for communities which might miss some interesting communities. While some other approaches are not very flexible in handling attributes. This research attempts to address the question of detecting cohesive communities while maintaining a subspace similarity over real (floats) attributes.

Detecting communities is very essential as communities have many practical applications [26]. A community in a protein protein interaction network can represent biological complexes which can be used to diagnose diseases [70]. A community of friends in a social network with similar interests can be targeted for advertisements or recommendations [79]. With such an ever increasing list of applications it is very critical to find novel ways to detect cohesive communities with attribute data over nodes or edges.

In this paper we discuss new approaches to find cohesive communities in rich graphs. We first define what cohesive communities mean and then show techniques to mine cohesive communities from a graph. We also compare our technique against the state of art algorithms and present our results.

1.1. Motivation examples

In this section we shall discuss some motivational examples for the application of communities in various fields of science and establish their importance. Furthermore we also see how integrating the definition of communities with similarity of attributes is proving to be further beneficial.

1.1.1. Biology

In bioinformatics, the interactions between proteins is generally represented as an interaction graph known as protein protein interaction (PPI) network, where nodes represent proteins and edges represent pairwise interactions between them. Application of network clustering methods had significant impact which have led to extraction of functional modules such as protein complexes [60] or regulatory pathways [64]. These complexes are a cornerstone of many biological processes and together they form various types of molecular machinery that perform a vast array of biological functions, such as finding targets for antimicrobial drugs [60]. These complexes or clusters are proving very useful in identifying potential biomarkers in a variety of diseases such as Tuberculosis, Pediatric Pneumonia and Pulmonary Sarcoidosis [6, 78, 53]. Recent research in bioinformatics shows that integrating gene expression profiles with the PPI network structure improves diagnosis and prognosis of cancer [15, 16]

1.1.2. Social networks

The development and analysis of social networks and the application of graph theory in sociology has been studied since the early 1900's [27]. Social network analysis produces an alternate view, where the individuals are less important than their relationships with other actors within the network. This approach has turned out to be useful for explaining many real-world phenomena [75]. As social media is gaining popularity [2], billions of online profiles (attributes) exist on popular websites like Facebook, Twitter, etc. Combining community detection with attribute data has given rise to multitude of applications in the recent times. Behavior and sentiment analysis during elections [68, 69], location-based interaction analysis [80, 14] and marketing and recommender

systems development like used in Facebook [13] are some of the applications that stem from social network analysis.

1.1.3. Enron email data set

Enron email data set is a large corpus of emails generated by employees of the Enron Corporation which was later used for investigation after the company's collapse. The original dataset contains 619,446 email messages [46]. A subset of the original data has been normalized and annotated with category labels by UC Berkeley [25]. This subset of data contains 1700 email messages and each email is categorized with three labels from a set of 53 labels. An email network was constructed from this email data set where nodes represent employees and edges represent an email communication between the employees. The category labels on the email were associated to the edge attributes. Finding dense communities who frequently exchange emails with "financial bankruptcy" or "fraud" topics can be extremely useful to investigators who can localize their search to people who participate in certain key topics [62].

1.1.4. Developer networks in open source software

In software engineering developers collaborate to work together and in doing so they form inherent developer networks [39]. Code review is a process in which the author of a specific code asks others relevant expert developers to review the code before submitting to the code repository. The code review process in open source software is difficult because of the distributed and voluntary participation of developers. Finding a cluster of relevant expert developers who can review code related to a specific area is one of the big challenges in this space. In a developer network, each node represents a developer and an edge is drawn between two developers when they co-comment on a code review. The class or modules that a developer has reviewed and commented are modeled as the node attributes for that developer, indicating their expertise. The problem of finding a relevant set of expert developers can be reduced to the problem of finding communities in the developer network who have similar attributes.

Community detection is very essential as is made evident by the preceding examples. Communities which have similarities in either node or edge attributes show more promise in their utility. Note that the similarity here is only in the subspace of attributes, i.e., only a relevant subset of attributes need to be similar in the set of attributes. We propose some novel approaches to address this problem of detecting cohesive communities with subspace similarity of attributes.

1.2. Goal of this thesis

Now that we have looked at some motivating examples and benefits provided by communities we would like to formally present the goal of this thesis.

Our goal is to devise efficient algorithms to mine cohesive communities from networks. We define cohesive communities which are similar in both network structure and attributes and confirm from our experiments that cohesive communities are more robust and promising.

We present multiple algorithms to mine cohesive communities and demonstrate our algorithm's efficiency against the state of the art algorithms. We also present the results from our experiments which showcase the effectiveness of cohesive communities.

1.3. Organization of the thesis

The remainder of this thesis is organized as follows. In chapter 2, we discus some background literature and related work in the area of community detection.

In chapter 3, we present an enumeration tree based pattern generation method to mine dense and cohesive communities. We first present all the preliminary definitions and concepts required to formulate the problem and discuss the algorithm. We also present a summarization technique to find representative communities. This research presented in section 3.2 is based on research published in the Network Modeling Analysis in Health Informatics and Bioinformatics journal in 2015 [34].

In addition, we also show a parallel approach to mining dense and cohesive clusters using multiple threads and discuss the algorithm. This research presented in section 3.3 was presented in Proceedings of the 7th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics in 2016 [32]. Finally we compare our approach to the state of the art algorithms and show our results for both single and multi-threaded algorithms in section 3.5.

Furthermore we started to find efficient ways to mine cohesive communities without density constraint. Chapter 4 presents a pattern generation method to find cohesive communities without density constraint. This research is based on the research paper published in Bioinformatics and Biomedicine (BIBM) IEEE International Conference in 2015 [35]. Once again, we implemented a parallel approach to mine cohesive only communities utilizing multiple threads. This research was

published in 9th International Conference on Bioinformatics and Computational Biology (BICOB 2017) [33]. We compare and show the results for both sigle and multi-threaded algorithms in 4.3.

In chapter 5, we present a sampling technique which significantly improves the performance of community detection. Unlike the enumeration techniques presented in chapter 3 and 4, this technique can output a reduced set of cohesive and dense modules without enumerating the entire output space. This research paper is in the process of getting published. Finally chapter 6 concludes this thesis and discusses the possible extensions for this research in future.

2. RELATED WORK

Detecting communities is of great importance in sociology, biology and computer science disciplines where systems are often represented as graphs. This problem is very hard and not yet satisfactorily solved, despite the huge effort of a large interdisciplinary community of scientists working on it over the past many years. The problem has had a long tradition and it has appeared in various forms in several disciplines. This section presents some background concepts and discusses recent work in this area.

2.1. Communities

Real world graphs often have a broad degree distribution, i.e., there exists many vertices with low degree while very few vertices have a high degree. This power law distribution [23] of vertex degree intuitively illustrates the high level of order and organization in a real world graph. One distinctive difference in real world graph is that they exhibit local and global inhomogeneities; high concentrations of edges within special groups of vertices, and low concentrations between these groups. This feature of real networks is called a community. Figure 2.1 shows a sample of the web graph consisting of the pages of a web site and their directed hyperlinks. Communities are indicated by similarly colored vertices.

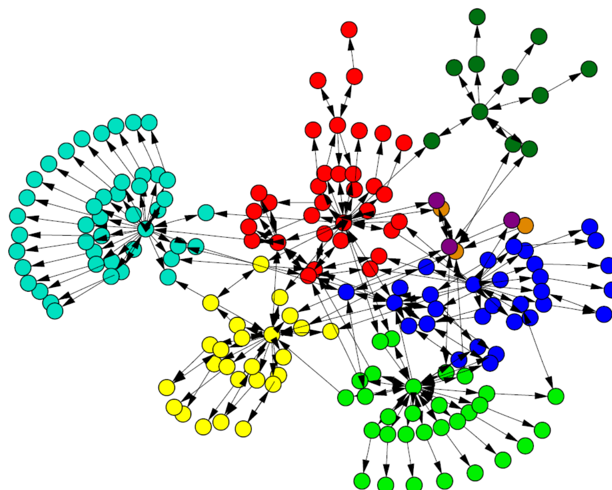

Figure 2.1. Visualizing communities in a sample web site graph

One of the major issue with community detection is that there is no universally accepted quantitative definition of a community. Often times the definition arises from the problem at hand or the application domain. Intuitively one can say that a community should have many edges among itself while having very few edges between the community members and rest of the graph. Another required property for a community

is connectedness, that is every member within a community should be reachable by any other community member.

A full membership community or a clique has edges between every pair of the vertex in the community. Cliques are very strict because every vertex is forced to have an edge to every other vertex. Finding cliques in a graph is a NP hard problem [8]. Quasi cliques are a relaxed version of cliques where each vertex needs to have a minimum number of edges to be a part of the community. Mining communities in a graph defined by quasi cliques was discussed in [81]. Yet another way to measure the quality of a community is to calculate the density, which is the total number of edges in a community over the total possible edges in that community. Mining communities defined by density was discussed in [71].

Unlike the clique definition both quasi clique and density definition are not anti-monotone [55]. This implies that mining communities using either quasi clique or density definition are harder problems when compared to mining cliques, and therefore are also a NP-complete problem [28].

2.2. Community detection literature

2.2.1. Graph partitioning techniques

Community detection has been widely researched in graph theory. Traditional community detection methods partition the graph into a predefined number of clusters such that the number of edges between these groups are minimal [45]. The number of clusters is an important input parameter, as it restricts all the vertices from ending up in the same cluster. The cluster size input parameter make sure that the algorithm does not output many small and uninteresting clusters. It is very difficult to anticipate the number and size of the clusters in a big graph, which is one of the main reasons that graph partition algorithms are not very well suited to cluster detection in large graphs.

2.2.2. Hierarchical clustering

Hierarchical clustering algorithms [38] can reveal the multilevel structure of a graph. Many social networks display several levels of grouping of the vertices, with small clusters included within large clusters, which are in turn included in larger clusters, and so on. Hierarchical clustering techniques start with defining a similarity measure, such as euclidean distance and compute a similarity matrix between all vertices of the graph. The algorithm then finds clusters of vertices with high similarity. Hierarchical clustering algorithms do not require the preliminary knowledge of the cluster size and count which makes them better than the traditional partitioning techniques.

2.2.3. Spectral clustering

Among the many community detection algorithms spectral clustering methods have dominated the literature. Spectral clustering consists of transforming the vertices of a graph into a set of points in space, whose coordinates are elements of eigenvectors. These set of points are then clustered via traditional clus-

tering algorithms. Typically these traditional clustering algorithms works on the data directly, however, spectral clustering works with the eigenvectors of the similarity matrix, which gives a more global encoding of the similarities between points. One of the early contributions of spectral algorithm utilized eigenvectors of the adjacency matrix [20]. A later and a more popular version of spectral algorithm utilized the eigenvector of the second smallest eigenvalue of the Laplacian matrix [24].

2.2.4. Markov clustering

The basic idea behind Markov clustering (MCL) is to simulate a flow within a graph, to promote flow where the current is strong, and to demote flow where the current is weak [72]. If clusters exist in a graph, then according to the paradigm current across the clusters will wither away, thus revealing cluster structure in the graph. The algorithm builds a column stochastic (square) matrix, which can be interpreted as the matrix of the transition probabilities of a random walk (or a Markov chain) defined on the graph. The MCL algorithm is an iterative process of applying two operators - expansion and inflation - on an initial stochastic matrix, in alternation, until convergence. The graph described by the final stable matrix is disconnected, and its connected components are the communities of the original graph. The MCL is one of the most used clustering algorithms in bioinformatics.

2.2.5. Modularity

In some approaches communities are viewed as an essential part of the entire graph, i.e., communities cannot be isolated without destroying the graph. In such cases a null model is first created. A null model is a graph that matches the original graph in some aspects but otherwise has totally random distribution of edges. The idea is that a null model being totally random doesn't have preferential edges to form a community. The null model gives a metric for each subgraph to measure a community structure. A subgraph is deemed as a community, if the number of internal edges exceeds the expected number of internal edges the same subgraph would have in a null model. Newman and Girvan [58] presented one such null model which is later used in partitioning the graph until communities are detected. A quality function *modularity* evaluates the goodness of the partitions of the subgraph.

2.2.6. Enumeration tree based community detection

Many graph mining algorithms create an enumeration tree to mine communities or clusters in a graph [81, 54, 71]. Figure 2.2 shows a sample graph and its enumeration tree. The enumeration tree starts with a null set at the root. The first level search nodes in the enumeration tree contain each individual vertex of the graph. Each of these first level search nodes are expanded to form children nodes by adding one vertex from the graph, which is not already present in that search node. An enumeration tree typically follows a strict ordering of vertices which makes sure that the same subgraph is not repeated twice in the tree. In Figure 2.2 we see that a child node always grows with a vertex that has a lower order than all

the other vertices already present in the search node. Each search node in the tree satisfies the community definition. The clique definition for a community is an anti-monotone property [55], i.e., as we go down the enumeration tree, if a search node doesn't satisfy the clique definition then no child node generating from that search node will ever satisfy the clique definition. Thus we can stop generating child nodes from that search node.

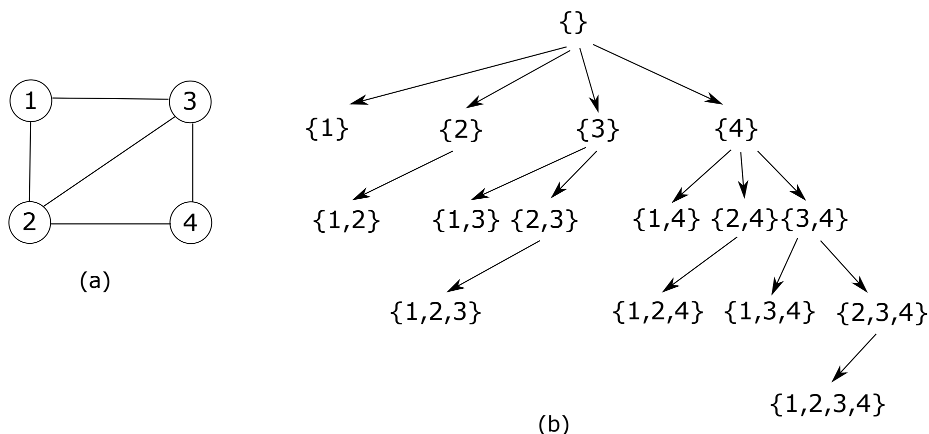

Figure 2.2. Enumeration tree example. (a) A sample graph. (b) Enumeration tree for the sample graph; node order $1 < 2 < 3 < 4$ is followed while generating child nodes.

2.2.6.1. Enumerating quasi cliques

Quick is an efficient algorithm to find maximal quasi-cliques from an undirected graph discussed in [54]. This algorithm builds an enumeration tree where each node in the tree has a candidate set of vertices which can be used to extend the current search node. Quick follows a strict ordering among its vertices to reduce the number of duplicate search node in its sample space. Quick applies several effective pruning techniques based on the degree of the vertices to prune unqualified vertices as early as possible, such as pruning on the vertex degree and graph diameter.

2.2.6.2. Enumerating dense clusters

The algorithm discussed by Uno et al. [71], traverses the enumeration tree in a depth first manner. This algorithm uses the reverse search technique [3] to generate child search nodes in the enumeration tree. Reverse search does not need to memorize the previously visited search nodes to avoid duplicates. The algorithm adapts the reverse search paradigm and only enumerates valid quasi clique child search nodes at each iteration.

2.3. Cohesive community detection

2.3.1. Attributes in graphs

Often additional data sources are available which can annotate the nodes or the edges of the graph with attribute data. In a PPI network an attribute value can represent gene expression data, which encodes the differential expression value of each gene when exposed to stimuli. In a social network, an attribute might correspond to the personal profile of a member such as age, interests, locale, etc. The concept of homophily suggests that cultural, behavorial, genetic or material information that flows through a network tends to get localized to people or entities with similar attributes [56]. So in addition to observing interactions in a network, it is also important to consider the attributes of entities [36, 44].

2.3.2. Node attributes

Node attributes represent properties of vertices in a graph. As noted above the profile data in a social network such as age and interests are examples of node attributes. Usually they are modeled as a vector of attributes corresponding to each vertex in the graph. Figure 2.3 shows a sample social network where each node is a person and the attributes shown are properties of the person such as age, city and interests. The figure shows two cohesive communities which are not only dense but also have similarity in their attributes. Notice that each vertex is similar on a subset of attributes with the attributes of other vertices in its community, for e.g., vertices (0,1,2,3) are similar in age and city.

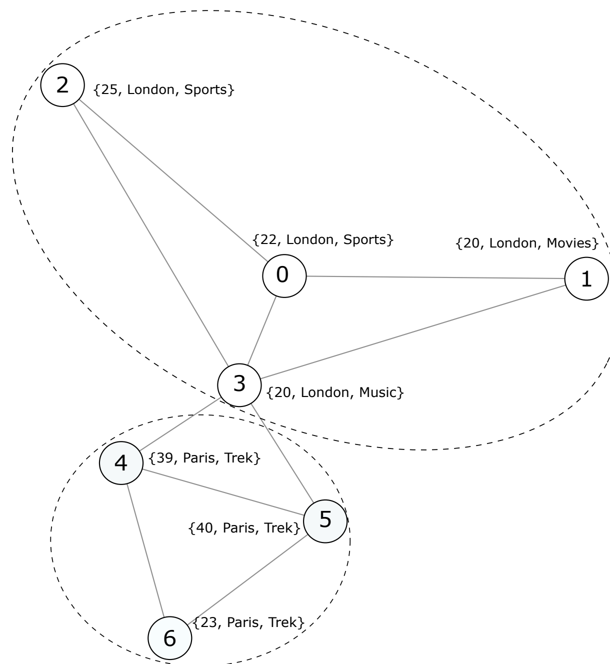

Figure 2.3. Graph with node attributes and two cohesive communities.

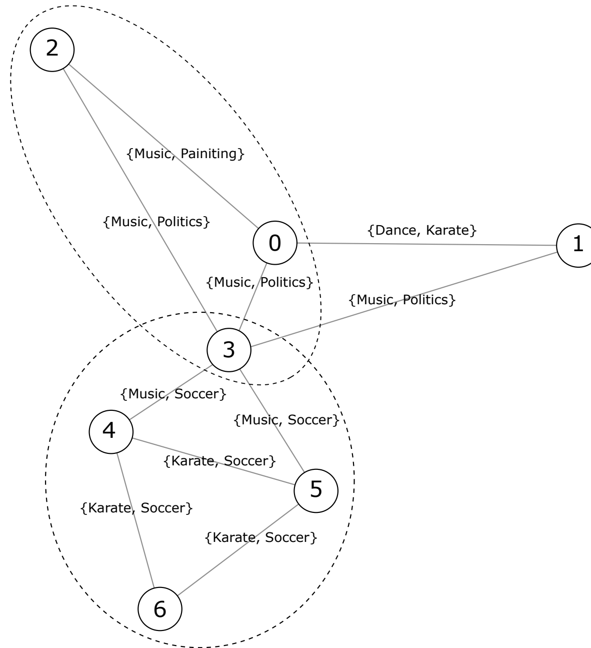

Figure 2.4. Graph with edge attributes and two cohesive communities.

2.3.3. Edge attributes

Edges can have properties too and they are modeled as edge attributes. For example, in a social network the length of the relationship measured in time between two friends is an attribute of their relationship. In a chemical network the strength of the bond between two molecules can be modeled as an edge attribute. Edge attributes are typically modeled as a vector of attributes corresponding to each edge in the graph. Figure 2.4 shows another sample social network with edge attributes. The attribute on a edge shows the type of online activity between two people. Vertices (3,4,5,6) forms a community where all edges have {soccer} as a common attribute. Edge-based content is much more challenging, because the different interests of the same individual may be reflected in different edges.

In an cohesive community detection approach, the communities are not only matched for graph topology but also for attribute similarity. Many cohesive approaches combining graph topology data with attribute data have been proposed. Some rely on full space clustering of attributes [67] while others consider sub space clustering [36, 30, 18]. Full space clustering often leads to poor results in high dimensional dataset because there is a high probability of some irrelevant attribute to obfuscate the cluster.

2.4. Enumeration algorithms for cohesive community detection

This section will briefly introduce some of the state of the art algorithms for mining cohesive communities. All of these algorithms requires three parameters; a density threshold γ which controls the density of output modules, an attribute profile threshold t and number of cohesive attributes threshold s_{min} which together controls the cohesiveness of each module in the community

2.4.1. GAMer

The GAMer algorithm [36] proposed an enumeration approach which uses quasi-clique definition for community or cluster density. GAMer integrates the community detection with vertex attribute sub space clustering technique which identifies locally relevant (similar) subsets of attributes for each community. The quasi clique definition and sub space clustering together form a cluster definition for GAMer. Quasi-clique definition is not anti monotone, i.e., the quasi clique density of the subgraph cannot be used to prune the search space in the enumeration tree. However attribute subspace clustering is an anti-monotone property; if at any search node the number of similar attributes falls below a threshold, then the entire subtree rooted at the current search node can be pruned.

2.4.2. DME

The quasi-clique density is a little restrictive as each vertex is still required to have a minimum degree. Georgii et al. proposed the Dense Module Enumeration (DME) for weighted networks [30]. This algorithm uses a relaxed definition of density which generates more clusters than GAMer. Similar to GAMer, DME also builds an enumeration tree and outputs dense and cohesive clusters. The density definition used in DME is not anti-monotone. However DME employs the reverse search technique [3] which traverses the tree in a way such that the density property going down the enumeration tree is always decreasing.

2.4.3. DECOB

Like the above algorithms, the DECOB algorithm exhaustively finds maximal dense connected biclusters [18]. DECOB starts with the cohesive or similar edges and adds a new neighbor vertex to each edge. If the new pattern is dense and cohesive DECOB keeps it in a list otherwise discards it. Building this way, the algorithm finds out all the dense and cohesive patterns that have 3 vertices (since it started from an edge containing two vertices). The algorithm iteratively works on this list to find maximal dense and cohesive patterns at each pattern size level. Unlike the GAMer and DME this algorithm walks the patterns in Breadth First Search (BFS) manner while the two preceding algorithms walk the enumeration tree in Depth First Search (DFS) manner.

3. MINING DENSE COHESIVE SUBNETWORKS

In this chapter we look at the problem of mining dense and cohesive community or cluster. We first establish the definition of our dense and cohesive cluster and later show our algorithm followed by the results. Figure 3.1 shows a graph with node attributes. The figure shows two communities which are both dense and similar in their attributes.

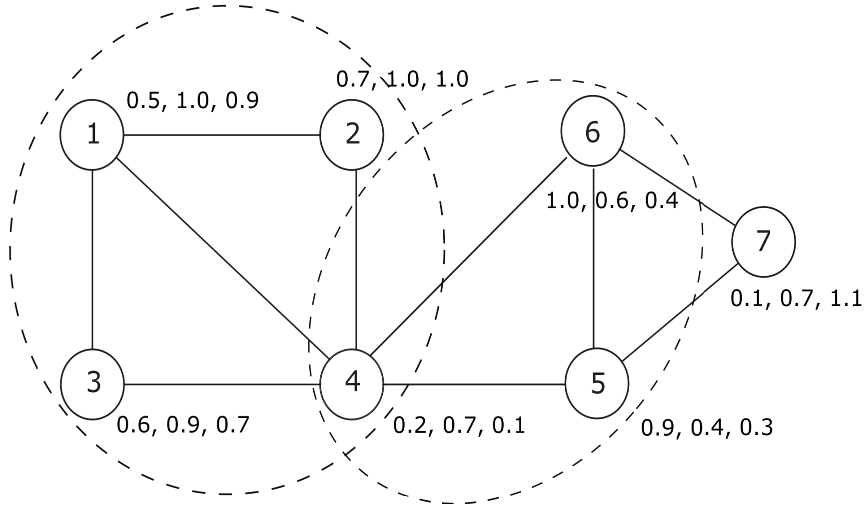

Figure 3.1. Example node attribute graph. The graph shows two communities, vertex 4 belongs to both communities.

3.1. Problem description

In this section, we introduce some preliminary definitions that are used throughout the paper. We then describe the problem of mining maximal dense cohesive subgraphs.

Definition 1. *A graph $G = (V, E, f)$ is an undirected graph, where $V = \{v_1, ..., v_n\}$ is the set of vertices, $E \subseteq V \times V$ is the set of edges, and $f : V \rightarrow \mathbb{R}^d$ is a function that maps a vertex to a d-dimensional real vector.*

The number of vertices and number of edges in G are denoted as $|V|$ and $|E|$, respectively. We use the d-dimensional vector to represent the attributes associated with a vertex. The attributes of all vertices can be represented by an attribute matrix $X \in \mathcal{R}^{n \times d}$, where x_{ij} is the attribute value of the ith vertex in j^{th} attribute. The ith row of the matrix X is the attribute vector of the ith vertex.

From figure 3.1; we have $V = \{1, 2, 3, 4, 5, 6, 7\}$, $E = \{(1, 2), (1, 3), (1, 4), (2, 4), \cdots, (6, 7)\}$, and $f(v_1) = (0.5, 1.0, 0.9)$

For any subset $U \subseteq V$, we denote $G[U] = (U, E[U])$ as the subgraph of G induced by U, i.e. $E[U]$ is the set of edges of G whose endpoints are both in U.

We define the *density* property (denoted as ρ) of an induced subgraph $G[U]$ as the ratio of the number of edges in the induced subgraph ($E[U]$) by the total possible edges in $G[U]$. In Figure 3.1, for $U = \{1, 2, 3, 4\}$, $\rho(U) = 5/6 = 0.83$.

$$\rho(G[U]) = \rho(U) = \frac{2|E[U]|}{|U|(|U| - 1)}$$

Definition 2. *Given a tolerance threshold t and a set of vertices U; where each vertex has d dimensional vector representing attributes. The kth attribute is considered a* ***cohesive attribute*** *for vertices in U if the kth attribute values for all vertices in U differ by at most t.*

$$\forall u_i, u_j \in U : |f(u_i)[k] - f(u_j)[k]| \leq t$$

For a threshold t, let $A(U, t)$ denotes the set of cohesive attributes, for simplicity we refer to $A(U, t)$ as $A(U)$:

$$A(U) = \{k_1, k_2, \cdots, k_l\}, 1 \leq k_i \leq d$$

In Figure 3.1, for $U = \{4, 5, 6\}$ and $t = 0.3$, $A(U) = \{2, 3\}$ since the three vertices have 'similar' values in the 2nd and 3rd attributes, i.e. the maximum difference between the attribute values for the three vertices in U for each of the 2nd and 3rd attribute is less than or equal to t.

Definition 3. *Given a tolerance threshold t, a dimensionality threshold s_{min}, an induced subgraph $G[U]$ is said to be a* ***cohesive subgraph*** *if the cardinality of the set of cohesive attributes is at least s_{min}, i.e. $|A(U)| \geq s_{min}$*

The dimensionality threshold s_{min} is the minimum number of 'similar' attributes a set of vertices must have in order to form a cohesive subgraph. In Figure 3.1, for $t = 0.3$ and $s_{min} = 2$, the subgraph induced by $U = \{4, 5, 6\}$ is a cohesive subgraph.

Definition 4. *Given a density threshold* θ*, an attribute tolerance threshold* t *and a dimensionality threshold* s_{min}*:* $G[U]$ *is a* ***dense cohesive subgraph*** *if it satisfies the following conditions.*

1. *Density of the subgraph* $G[U]$ *is atleast equal to the density threshold,* $\rho(U) \geq \theta$*.*

2. *The number of relevant attributes should be at-least equal to dimensionality threshold, i.e.* $|A(U)| \geq s_{min}$ *from definition 3*

We can see from figure 3.1 that the subgraph induced by $U = \{1, 2, 4\}$ vertices is both dense and cohesive. $\rho(U) = \frac{3}{3} = 1$ and vertices in U have similar values in 1st and 2nd attributes for $t = 0.5$.

According to Definition 4, a dense cohesive subgraph is any subgraph that can satisfy the density condition and has cohesive set of attributes. However, a single vertex is dense by definition and has absolute similarity among its own attributes. Also, a cluster of two vertices with a single edge has a density of 1. It only needs to satisfy the second condition of definition 4 in order to be considered a cohesive subgraph. It is obvious that we need a way to keep these kinds of unmeaningful subgraphs out of our result set. A common solution is to mine for the *maximal* subgraphs. A subgraph is considered maximal if it has no direct superset which is cohesive and satisfies the density threshold condition. In this way we will not output every possible sub graph like $\{1, 2, 4\}$ and $\{1, 3, 4\}$ which are subsumed in the maximal cluster $\{1, 2, 3, 4\}$ from figure 3.1

Definition 5. *A cohesive dense subgraph induced by* U *is* ***maximal*** *if no superset* $U' \supseteq U$ *is dense and cohesive.*

Problem Definition: Given an attributed graph $G = (V, E, f)$, three thresholds θ, t, s_{min}, the problem of mining the set of **maximal dense cohesive subgraphs** is to find the set:

$$\mathcal{P} = \{U_1, U_2, U_3, \cdots, U_{|\mathcal{P}|}\}$$

such that every $U_i \in \mathcal{P}$ is a maximal dense cohesive subgraph. Each U_i is a tuple $\{G_i, A_i\}$ containing a subgraph and its relevant attributes.

3.2. Algorithm

3.2.1. RedCone approach

In this section we introduce our algorithm for mining **RE**presentative **D**ense **CO**hesive sub**NE**tworks (RedCone). As the name suggests, the algorithm discovers maximal dense cohesive clusters in a graph. It later tries to find representative clusters for all such cohesive dense clusters.

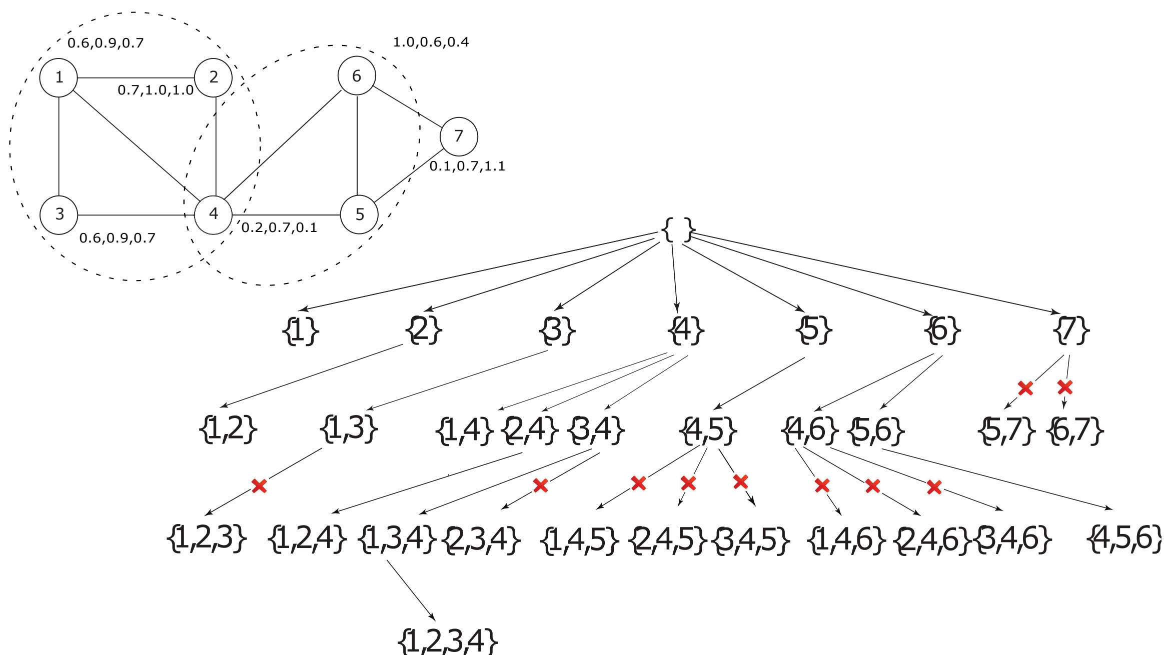

Figure 3.2. Example graph (a) and its enumeration tree (b). $\theta = 0.7$. Crosses show which branches are pruned. The discovered maximal clusters are in green.

We adapt the cluster enumeration approach as described in DME [30]. The cluster enumeration approach starts with an empty set and then iteratively grows into larger sets by adding one vertex at a time. The algorithm builds an enumeration tree (Figure 3.2) where each search node represents a dense cohesive cluster. Even though the density constraint is not anti-monotone, the reverse search technique [3] traverses the tree in a way such that the density property going down the enumeration tree is always decreasing while the node size is increasing. This is achieved by following a strict definition of parent-child relationship in the enumeration tree [30]. Essentially at every given search node all possible child search nodes are generated and only valid child search nodes are explored. The valid child search nodes maintain the density monotonicity property and

Algorithm 1 Maximal Dense Cohesive Cluster Discovery

Input:
$G = (V, E, f)$: an attributed graph
min_size: the minimum size of cluster to include in results
θ: density threshold
t: tolerance threshold between two attribute values in a single subspace
s_{min}: minimum number of similar attributes per cluster

Output:
$\mathcal{P}$: maximal cohesive clusters

```
 1: Remove all non cohesive edges from input graph
 2: P = {}
 3: MineDenseclusters({})
 4: function MineDenseclusters(U)
 5:     locally_maximal ← true
 6:     for v ∈ V\U do
 7:         Let U' = U ∪ v
 8:         if ρ(U') ≥ θ and |A(U')| ≥ s_min then
 9:             locally_maximal ← false
10:             if isChild(U', U) then
11:                 MineDenseclusters(U')
12:             end if
13:         end if
14:     end for
15:     if locally_maximal and |U| ≥ min_size then
16:         P = P ∪ U
17:     end if
18: end function
19: return P
```

also follow a strict ordering which ensures that each search node will be visited only once. A function ord defines a strict total ordering on the nodes, i.e. for each node pair u, v with $u \neq v$ either $ord(u) > ord(v)$ or $ord(u) < ord(v)$ holds. With this, the parent-child relationship for modules is defined as follows. Given U and $v \in V \setminus U$. $U^* := U \cup v$ is a child of U if and only if

$$\forall u \in U : (deg_{U^*}(v) < deg_{U^*}(u)) \vee (deg_{U^*}(v) = deg_{U^*}(u) \wedge ord(v) < ord(u))$$

Here $(deg_{U^*}(v)$ stands for the degree of vertex v in the subgraph induced by U^*. We obtain the parent of a module by removing the smallest ordered vertex with the least degree.

Algorithm 6 shows the pseudo code for our cluster discovery process. The recursive function builds an enumeration tree like the example in figure 3.2. Note that we only consider search node expansion if the conditions given in definition 4 are met (line 16). If a cluster doesn't have a cohesive and dense superset then that cluster by definition 5 is maximally dense and cohesive. The result of this algorithm is the set of maximal dense cohesive clusters $\mathcal{P}$. From figure 3.1 one can observe that

there are two maximal dense cohesive clusters $\{1, 2, 3, 4\}$ and $\{4, 5, 6\}$ which are similar in atleast 2 attributes. Figure 3.2 demonstrates how our algorithm arrives at these two clusters.

We employ different pruning strategies to avoid visiting branches that will not result in cohesive clusters. The algorithm starts by removing edges from the input graph which are not cohesive according to definition 3. Due to the anti-monotonicity of the cohesive constraint, pruning an edge will not result in missing any clusters because the pair of vertices (end points) cannot be together in any cohesive cluster. In figure 3.2 (a) the edge between vertices 5 and 7 is not cohesive in two attributes for $t = 0.6$, therefore we can remove that edge from the graph without missing any clusters.

The second pruning is based on reverse search enumeration. Since the reverse search principle guarantees that the search nodes are grown in a decreasing order of density, we can safely assume that if any search node does not meet the density threshold, θ, then we can prune that search node. This helps by eliminating the entire subtree from the search space.

Before a new child search node can be created, the algorithm checks to see if the potential child search node is cohesive. If it is cohesive then the algorithm creates the child node and recursively extends it. However if no cohesive dense child node exists for the current search node then the current search node is maximally cohesive and it can be added to $\mathcal{P}$ (line 24). Both of these two pruning strategies are enforced in line 16 of algorithm 6.

3.2.2. Multithreaded RedCone

Recall that RedCone requires density and profile thresholds to reduce the search space of the input graph. For relaxed constraints, the search space is huge which in turn takes a very long time to enumerate all qualifying clusters. For reference, RedCone , DME, Gamer and DECOB algorithms took multiple days to completely enumerate all clusters in the BioGRID dataset, which has 6249 vertices and $224,587$ edges. The result set (output space) contained several million clusters which qualified a very relaxed input constraints on density and cohesive profile. As this example shows the above algorithms including RedCone don't scale very well for even a modestly sized input graph or as constraints are further relaxed.

In this section we propose a multithreaded implementation for RedCone , called MT_Redcone to address the issues of scale.

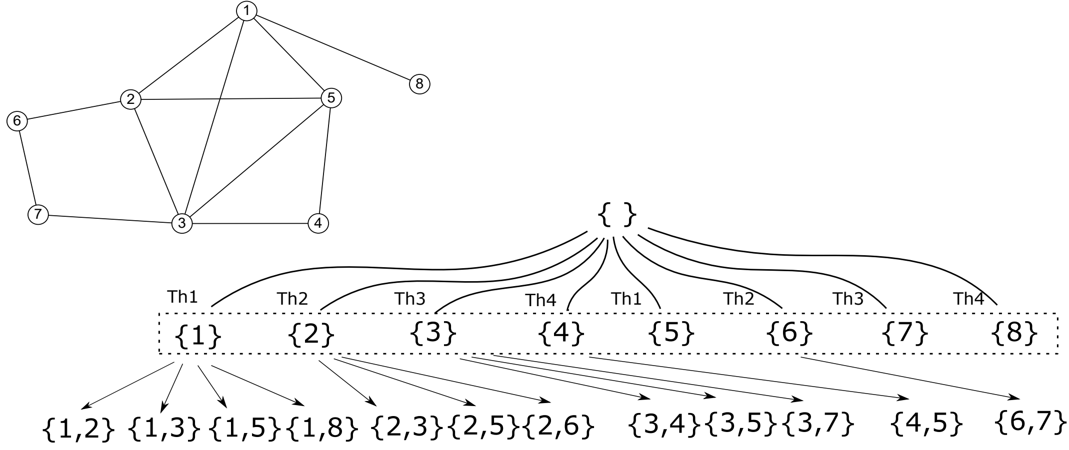

Figure 3.3. The input graph and its corresponding enumeration tree. There are four threads which build the enumeration tree independently. There are 4 threads and each thread builds subtree for 2 first level children.

RedCone mines maximal dense cohesive subgraphs by following a reverse search enumeration technique [3]. The reverse search technique guarantees that the enumeration of child search node is only dependent on its parent search node and is independent of any shared structure. This property can be exploited to parallelize the enumeration tree traversal.

Figure 3.3 shows an example of the enumeration tree that created by MT_Redcone for the input graph shown in figure 3.1. Utilizing the reverse search principle, the subtrees rooted under each first level node in the enumeration tree can be enumerated independently. This suggests that we can spawn multiple threads at the root, and each thread creates the sub tree under each of the first level nodes. Algorithm 3.2.2 shows the psuedo code for MT_Redcone . Apart from the usual inputs such as graph G, density threshold θ, tolerance threshold t and dimensionality threshold s_{min}, MT_Redcone also requires a number of threads input numthreads. The algorithm begins by spawning the requested number of threads (line 3). Each thread then iterates over the first level nodes, selects a vertex and traverses its enumeration subtree (line 9). The output of this algorithm $\mathcal{P}$ is a list of maximal cohesive dense clusters.

3.3. Representative Set

The number of maximal dense cohesive clusters can be astronomically large, depending on the density and cohesive constraints. Moreover, these clusters have overlap in both the vertices and their relevant attributes. For analysis, often a summarized set of all the reported clusters is desired. This set should be representative of the reported clusters such that all the clusters not

Algorithm 2 Multi-threaded Maximal Dense Cohesive Cluster Discovery

Input:
$G = (V, E, f)$: an attributed graph
min_size: the minimum size of cluster to include in results
θ: density threshold
t: tolerance threshold between two attribute values in a single subspace
s_{min}: minimum number of similar attributes per cluster
$num_{threads}$: Number of threads to spawn for parallel execution

Output:
$\mathcal{P}$: maximal cohesive clusters

```
 1: Remove all non cohesive edges from input graph
 2: P = {}
 3: threads[] = spawn_threads(num_threads)
 4: start_all_threads(threads[], ThreadStart)
 5: join_all_threads(threads[])
 6: function ThreadStart
 7:     for v ∈ V do
 8:         execute_thread(t, MineClusters(v))
 9:     end for
10: end function
11: MineClusters({})
12: function MineClusters(U)
13:     locally_maximal ← true
14:     for v ∈ V\U do
15:         Let U' = U ∪ v
16:         if ρ(U') ≥ θ and |A(U')| ≥ s_min then
17:             locally_maximal ← false
18:             if isChild(U', U) then
19:                 MineClusters(U')
20:             end if
21:         end if
22:     end for
23:     if locally_maximal and |U| ≥ min_size then
24:         P = P ∪ U
25:     end if
26: end function
27: return P
```

in the representative set should have at least one 'similar' cluster in the representative set. We propose an approach for selecting a representative set of clusters for the maximal cohesive clusters.

3.3.1. Finding Similarity Scores

In the first step, we introduce a similarity measure to quantify the similarity between two maximal dense cohesive clusters and calculate the similarity scores between all pairs of the reported clusters.

Given two cluster U, U', let $S_{UU'}$ denotes the Jaccard similarity coefficient between the sets of vertices of the two clusters.

$$S^v_{UU'} = \frac{|U \cap U'|}{|U \cup U'|} \tag{3.1}$$

The relevant attribute similarity between the two clusters is captured by the Jaccard similarity coefficient between the sets of relevant attributes.

$$S^a_{UU'} = \frac{|A(U) \cap A(U')|}{|A(U) \cup A(U')|} \tag{3.2}$$

We define the cluster similarity as linear combination of the vertices and relevant attribute similarities as shown below, where α is a user-defined parameter ($0 \leq \alpha \leq 1$) to control the contribution of the vertices similarity to the pair wise cluster similarity.

$$S_{UU'} = \alpha * S^v_{UU'} + (1 - \alpha) * S^a_{UU'} \tag{3.3}$$

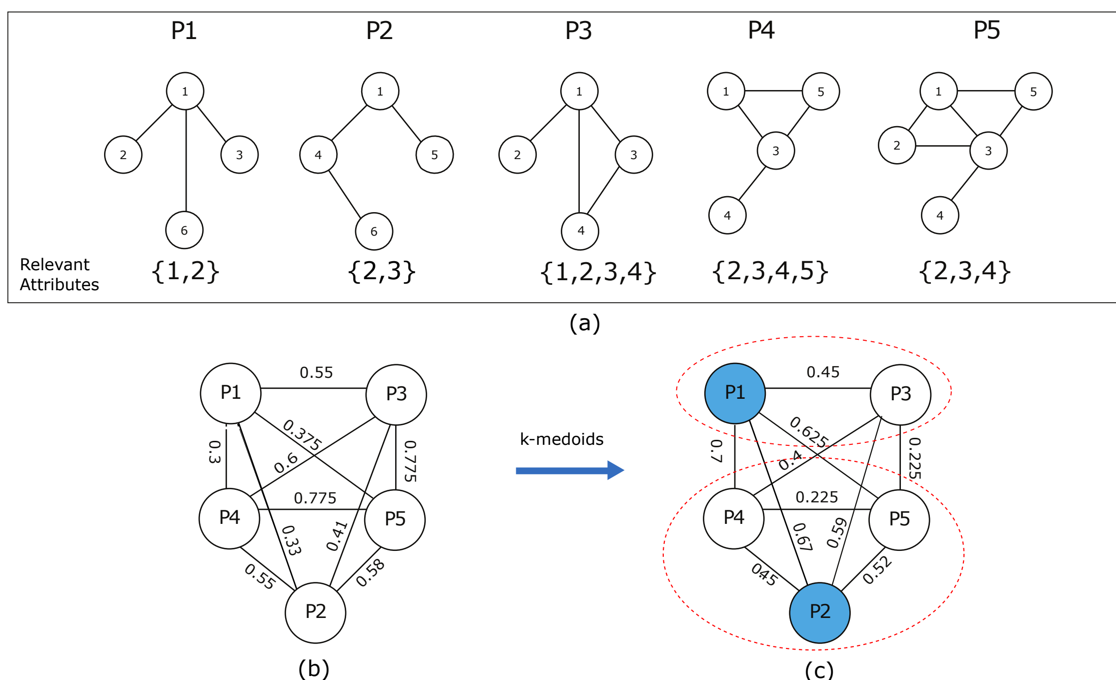

Figure 3.4. Finding representative clusters. (a) The maximal cohesive clusters. (b) The cluster similarity graph for $\alpha = 0.5$, edges show dissimilarity between clusters; (c) Applying k-medoids algorithm results in finding clusters and medoids (highlighted in blue), edges show distances between clusters.

The cluster similarity between the first two clusters shown in Figure 3.4(a) is calculated as follows (for alpha=0.5):

$$S_{P_1P_2} = 0.5 * \frac{2}{6} + 0.5 * \frac{1}{3} = 0.33$$

The two clusters share two vertices out of the six vertices and one relevant attribute out of the three relevant attributes.

3.3.2. Similarity Graph

The cluster similarity graph represents the result clusters as nodes and its similarity as distance on the edges to other nodes. More formally, we construct the cluster similarity graph $G_P = (V_P, E_P)$, where $V_P = \{U_1, U_2, \cdots, U_k\}$ represent the set of maximal dense cohesive clusters and there is an edge between two vertices in this graph where the distance of edge (u_i, u_j) is $S_{U_i U_j}$.

Figure 3.4(a) shows 5 clusters and Figure 3.4(b) shows the cluster similarity of these 5 clusters. Each node in cluster similarity graph is a subgraph as shown in 3.4(a) and edge weights represent the similarities between clusters. The similarities range from 0 to 1, with 0 similarity score indicating no overlap in both the vertices and the relevant attributes and 1 indicating complete overlap.

3.3.3. Representative clusters - Set cover approach

The problem of selecting representative clusters from a large set of result clusters has been previously studied in [9]. In that paper finding the representative clusters problem was mapped to the problem of finding the dominating set of minimum size from the similarity graph. Given the undirected similarity graph $G_P = (V_P, E_P)$, the problem of selecting the smallest dominating set is to select the smallest set of nodes (clusters) $S \subseteq V_p$ such that every node not in S is connected to at least one node in S.

3.3.4. Representative clusters - K-Medoids approach

The problem of selecting representative clusters can be mapped to the problem of finding k medoids from multiple observations using the k-medoids algorithm. k-medoids is a classical partitioning technique that clusters the data set of n nodes into k clusters. Each of these clusters have a center or medoid. A medoid can be defined as the node of a cluster whose average distance to all the nodes in the cluster is minimal, i.e., it is the most centrally located node in the cluster. The objective of this algorithm is to partition all the n nodes into k clusters such that average distance of all the nodes in each cluster from its corresponding medoid is minimized. As a by product the algorithm finds k medoids (nodes) which are most centrally located within their own clusters in the graph. Given an undirected similarity graph $G_P = (V_P, E_P)$, the problem of finding representative result nodes is to find the k medoids of that graph.

Figure 3.5 shows the modified K-medoids algorithm in action with a few output maximal cohesive dense clusters $\{p_1, p_2, ..., p_5\}$. Figure 3.5 (a) shows the network structure of these maximal cohesive dense clusters and also lists the cohesive attributes of each cluster below. These cohesive clusters are projected as points in space and K random points are selected as medoids. This process is shown in figure 3.5 (b).

After the random selection of points, the modified K-Medoids algorithm detects partitions around these random medoids as shown in figure 3.5 (c). After detecting the partitions, new medoids are again determined for each partition as shown in figure 3.5 (d). This process repeats itself by finding new partition again around the new medoids. Finally this process stops when a steady state is reached, i.e, new medoids are exactly same as the previous medoids. The final medoids at the steady state are the representative maximal cohesive dense clusters.

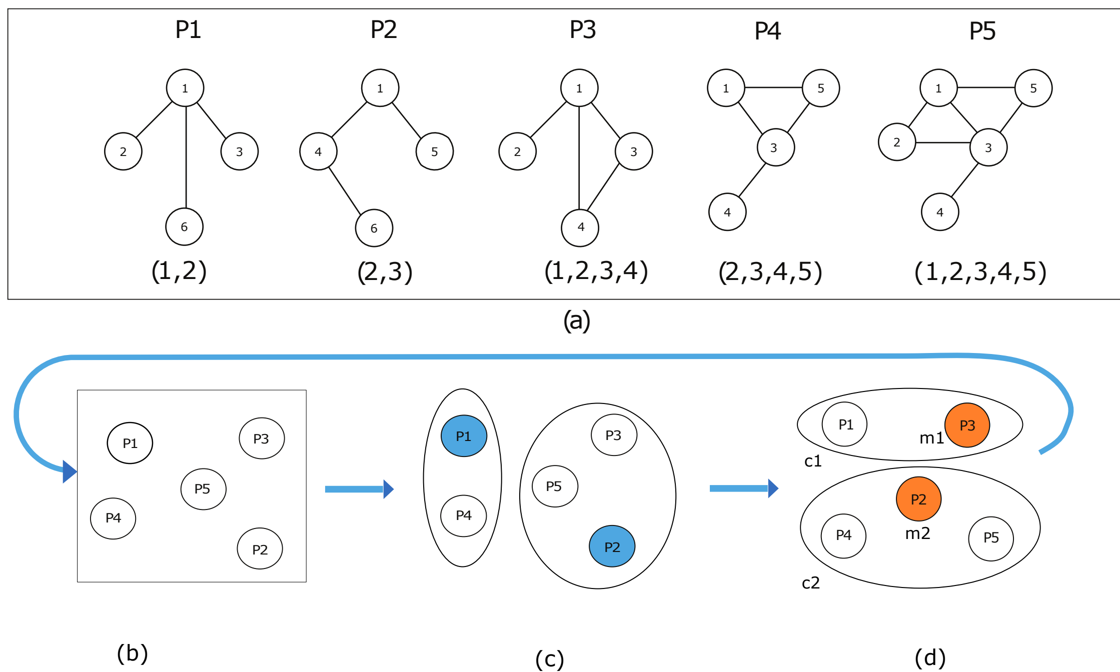

Figure 3.5. Representative clusters. (a) Set of maximal cohesive dense clusters, (b) Projecting maximal cohesive dense clusters to points in space. (c) Finding partitions around the random medoids. (d) Detecting steady state medoids $m1 = P3$ and $m2 = P2$ and its partitions $c1$ and $c2$ respectively. The representative maximal cohesive dense clusters are the set of steady state medoids $m1, m2$

This modified K-Medoids algorithm, distributes a given set of points $\{p_1, p_2, ..., p_n\}$ into K partitions or sets $\{c_1, c_2, ..., c_k\}$ to maximize the similarity between points p and their respective

partition c which can be represented as

$$\sum_{i=1}^{k}\sum_{y\in c_i} S_{ym_i}$$

where m_i represent the medoids and S represents the similarity between two maximal cohesive dense cluster.

Now we shall define some metrics to understand the quality of these output medoids (representative clusters) as produced by the K-Medoids algorithm.

Definition 6. *Given a set of points $\{p_1, p_2, ..., p_n\}$, k partitions $\{c_1, c_2, ..., c_k\}$ and their respective medoids $\{m_1, m_2, ..., m_k\}$, the average intra partition similarity is defined as average sum of the similarities of each point $p \in c_x$ to their corresponding medoid m_x.*

$$AvgPartitionSim = (\sum_{i=1}^{k}\sum_{y\in c_i} S_{ym_i})/|\{p_1, p_2, ..., p_n\}|$$

The $AvgPartitionSim$ similarity captures the quality of all partitions. A high value indicates that each medoid is centrally located in its partition, and, has high similarity with all other points in the partition. A low value indicates that the medoid is not equally similar to all other points in its partition, suggesting poor partitioning.

Definition 7. *Given a set of points $\{p_1, p_2, ..., p_n\}$, k partitions $\{c_1, c_2, ..., c_k\}$ and their respective medoids $\{m_1, m_2, ..., m_k\}$, the average inter medoid similarity is defined as average of sum of the pair wise similarities between the set of medoids.*

$$AvgMedoidSim = (\sum_{i=1}^{k}\sum_{j=i+1}^{k} S_{m_i m_j})/(M * (M-1)/2)$$

where $M = |\{m_1, m_2, ..., m_k\}|$

As opposed to the $AvgPartitionSim$, $AvgMedoidSim$ indicates the dissimilarity between the detected medoids. If the medoids are sufficiently similar to all points in their partition than they should be dissimilar to other medoids. In high quality partitions, we expect a high value for $AvgPartitionSim$ and a low value for $AvgMedoidSim$ indicating dissimilarity with other medoids.

In this perspective these two similarities complement each other. We report our findings on representative sets using these metrics.

Finally, the runtime of K-Medoids is $O(K*n*iter)+O(K^2*iter)$ where $iter$ represents the number of iterations to reach to steady state and K represents the desired number of representative clusters. Since both K and $iter$ are compile time constants, K-Medoids is better when compared to set cover algorithm mentioned in section 3.3.3 which took $O(n^2)$.

3.4. Experiments

We compare our algorithm against GAMer [36] and DECOB [18] using two real-world networks and their associated attribute data: *High Confidence Yeast (YeastHC)* [37] and the *BioGRID* [11]. All experiments were run independently on an Arch Linux operating system with an Intel Core i5-2500K (3.3GHz) processor and 8 Gigabytes of main memory.

3.4.1. Cohesive clusters in YeastHC

For this dataset, we represent the yeast interaction network as a graph and gene profile data as attributes. The interaction network contains 4,008 vertices and 9,857 edges in its graph. We include gene profile information which denotes the differential expression value of each gene when exposed to 173 different experiments [29]. We used real (floating point) attribute values and varied the attribute tolerance threshold t in increments of 0.1.

Table 3.1 compares the topological properties reported by RedCone and GAMer for this dataset. In the table, $|N|$ denotes the number of resulting clusters and $\overline{N}$ represents the average size. RedCone reports notably higher number of these clusters because it has a relaxed density constraint and also because GAMer reports summarized modules. The average cluster size reported by RedCone is also high for higher tolerance thresholds which can be again attributed to its relaxed density constraint. Since DECOB reports the same results as RedCone we only compare against DECOB for the running time.

We also performed biological enrichment analysis using the **D**atabase for **A**nnotation, **V**isualization, and **I**ntegrated **D**iscovery — DAVID [41, 42] on the YeastHC dataset. In order to verify the significance of our results, we attempted to find enrichment of Gene Ontology process terms (GOTERMS) as well as KEGG pathway enrichment in our resulting clusters.

Figure 3.6 plots the resulting clusters listed by RedCone with the percentage of modules enriched in GOTERMs and KEGG pathways. As we can see for densities 0.7 to 0.9 almost all

Table 3.1. Topological properties of cohesive dense clusters for Yeast dataset.

Parameters			RedCone		GAMer	
θ/γ	t	s_{min}	$\|N\|$	$\overline{N}$	$\|N\|$	$\overline{N}$
0.6	0.2	20	379	4.01	184	4.00
0.6	0.2	30	10	4.00	0	0
0.6	0.3	20	48398	4.22	14112	4.17
0.6	0.3	30	5529	4.13	2279	4.06
0.6	0.4	20	452333	5.55	100391	5.12
0.6	0.4	30	114874	4.69	37427	4.55
0.7	0.2	20	192	4.00	93	4.00
0.7	0.2	30	1	4.00	0	0
0.7	0.3	20	14711	4.42	7100	4.28
0.7	0.3	30	2354	4.2	1134	4.11
0.7	0.4	20	170971	5.52	54272	5.12
0.7	0.4	30	40692	4.98	19934	4.65
0.8	0.2	20	192	4.00	93	4.00
0.8	0.2	30	1	4.00	0	0
0.8	0.3	20	12466	4.26	6112	4.17
0.8	0.3	30	2236	4.14	1058	4.05
0.8	0.4	20	56575	5.52	38883	5.11
0.8	0.4	30	24389	4.74	15699	4.56
0.9	0.2	20	101	4.00	93	4.00
0.9	0.2	30	1	4.00	0	0
0.9	0.3	20	5940	4.29	6005	4.13
0.9	0.3	30	1102	4.16	1062	4.05
0.9	0.4	20	21764	5.43	30693	4.78
0.9	0.4	30	10524	4.73	14044	4.37

resulting clusters are enriched in GOTERMs. As density goes down RedCone outputs very large number of clusters because of relaxed density constraint, hence not all clusters are enriched.

Figure 3.7 shows two maximal clusters from the RedCone's output on the YeastHC dataset and two matrices illustrating the attribute data for the vertices in the two clusters. The matrix shows the vertices on the rows and attributes on the columns. The first 20 columns in the matrix are the attributes where these vertices are similar and therefore show very little deviation in its gray shade. The last 20 columns are a sample of 20 attributes from the remaining 153 attributes, where these vertices are not similar. This similarity is evident by the homogenity of gray shade in the first 20 columns versus variation of gray shade in the last 20 columns. This image is helpful in understanding that the output clusters are not only dense in the graph structure but they also have similar values in a number of attributes.

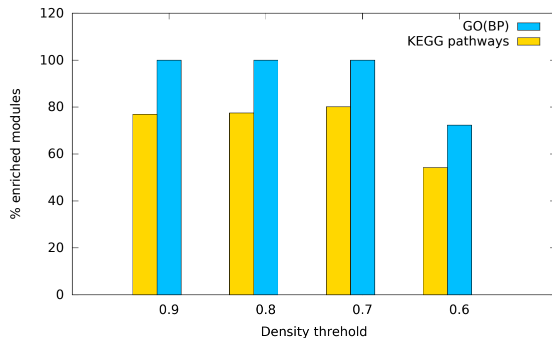

Figure 3.6. Functional interpretation of patterns: GOTERMs and KEGG pathways enrichment for the YeastHC dataset for $t = 0.3, s_{min} = 30$.

3.4.2. Cohesive clusters in BioGRID

The second experimental dataset used was the BioGRID along with its gene profile data. The interaction network contains 6249 vertices and 224587 edges in its graph and attributes collected from 173 different experiments. Like the YeastHC data we used real attribute values and varied the tolerance threshold t in increments of 0.1. Table 3.2 shows a summary of the reported results. We could not gather results for higher tolerances as GAMer did not finish in a reasonable period of time, this restricted us to show results for $t = 0.2$ to $t = 0.3$. We think that since GAMer summarizes over all clusters (not restricted to maximal) it would take GAMer a lot of time to produce results for higher tolerances especially when there are millions of these clusters.

3.4.3. Running Time

The reverse search algorithm for mining all maximal dense cohesive subgraphs with nominal attributes is a polynomial time delay algorithm [30, 71]. In the proposed algorithm, the additional checking of the cohesive constraints over real attributes for each cluster takes $O(dn)$, where d is the number of attributes and n is the maximum number of nodes in a cluster. Therefore, the proposed algorithm is a polynomial-delay algorithm which means that the computation time between reporting two clusters is polynomial in the input size. The running time of the algorithm thus depends on the number of reported maximal dense cohesive clusters which is controlled by the density and cohesive thresholds.

We compare the running time of RedCone with DECOB and GAMer for varying parameters on the YeastHC and BioGRID datasets. We used the implementation provided by the authors for the DECOB and GAMer algorithms. Figure 3.8 and 3.9 show that RedCone outperforms both

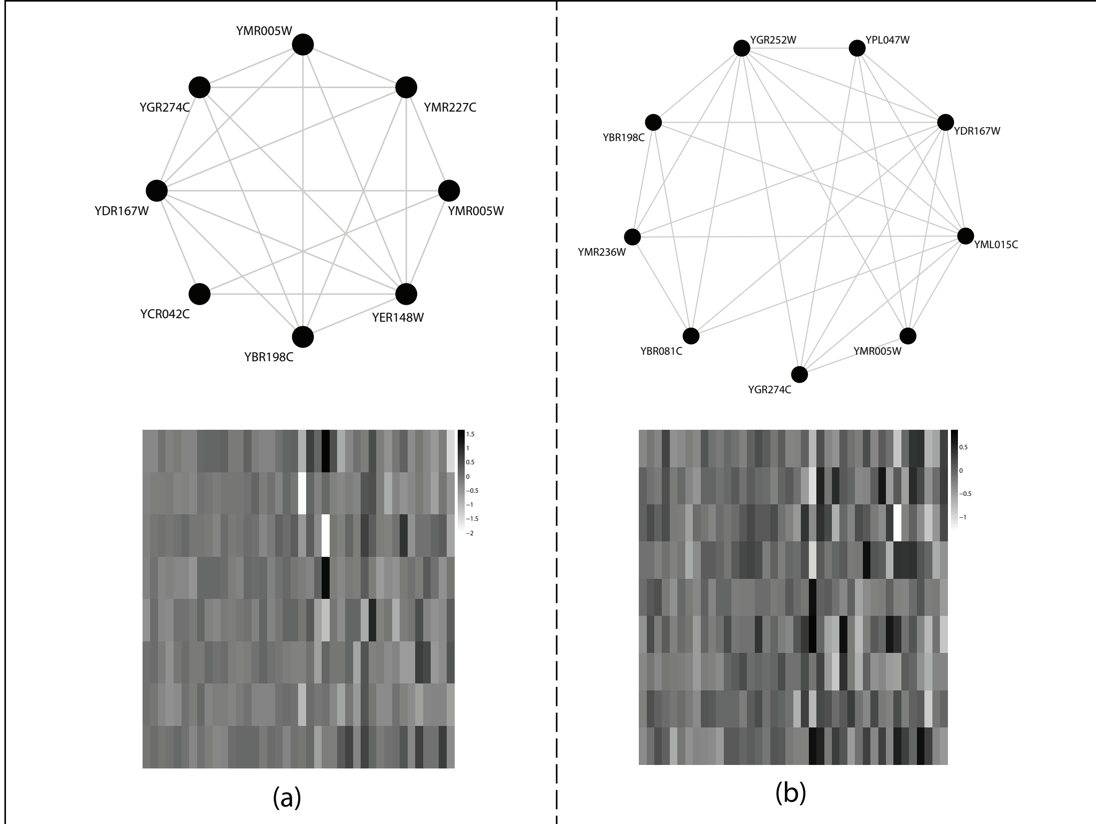

Figure 3.7. Representation of a dense and cohesive patterns from the YeastHC dataset, showing their network structure and attribute similarity of nodes in each pattern (a) number of vertices = 8 (b) number of vertices = 9; Parameters: $\theta = 0.7$, $t = 0.5$ and $s_{min} = 20$ (Only 40 attributes are shown)

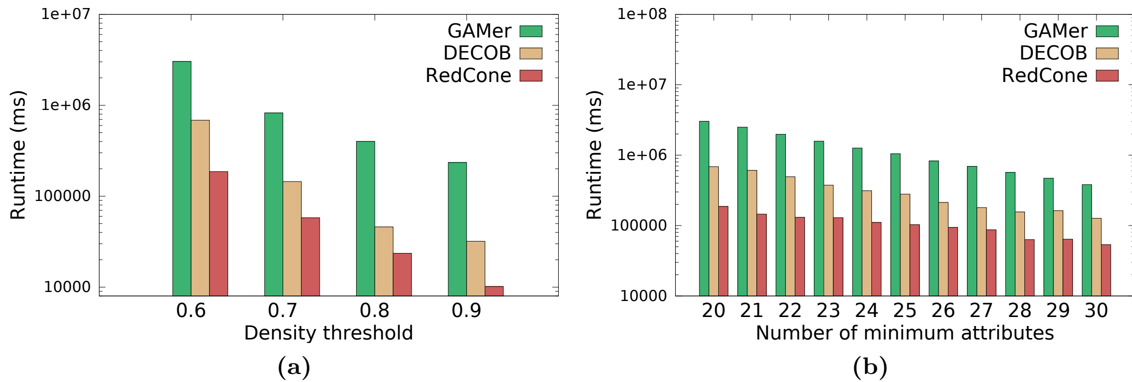

Figure 3.8. Runtime comparison of GAMer, DECOB and RedCone on the YeastHC dataset (a) parameters : $t = 0.4$ and $s_{min} = 20$ (b) parameters : $\theta = 0.6$ and $t = 0.4$

DECOB and GAMer in every case. For parameters $s_{min} = 20$, tolerance $t = 0.4$ and density $\theta = 0.6$, RedCone is more than twenty times faster than GAMer on the YeastHC dataset. Also for

Table 3.2. Topological properties of cohesive dense clusters for Biogrid dataset.

Parameters			RedCone		GAMer	
θ/γ	t	s_{min}	$\|N\|$	$\overline{N}$	$\|N\|$	$\overline{N}$
0.6	0.2	60	1769	4.60	312	4.03
0.6	0.2	75	572	4.59	134	4.07
0.6	0.3	60	49465	4.81	7036	4.02
0.6	0.3	75	8022	4.80	897	4.04
0.7	0.2	60	187	4.86	14	4.21
0.7	0.2	75	63	4.97	8	4.38
0.7	0.3	60	8250	4.59	1431	4.06
0.7	0.3	75	1005	4.82	83	4.16
0.8	0.2	60	108	4.20	11	4.00
0.8	0.2	75	31	4.39	5	4.00
0.8	0.3	60	5340	4.08	1352	4.00
0.8	0.3	75	525	4.11	170	4.00
0.9	0.2	60	10	4.30	11	4.00
0.9	0.2	75	4	4.75	5	4.00
0.9	0.3	60	1654	4.06	1345	4.01
0.9	0.3	75	77	4.12	70	4.00

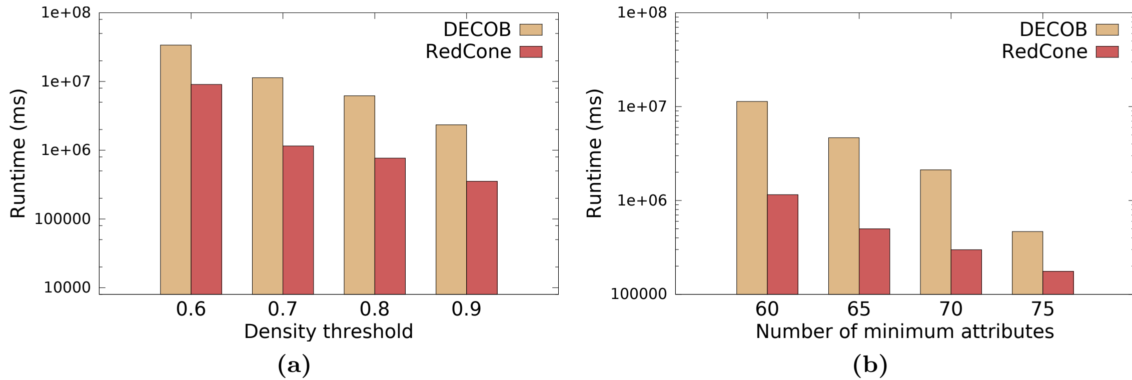

Figure 3.9. Runtime comparison of DECOB and RedCone on the BioGRID dataset (a) parameters : $t = 0.4$ and $s_{min} = 60$ (b) parameters : $\theta = 0.7$ and $t = 0.4$

parameters $s_{min} = 70$, tolerance $t = 0.4$ and density $\theta = 0.7$, RedCone is almost ten times faster than DECOB on the BioGRID dataset.

3.4.4. Multithreaded Runtime

We chose slightly different input datasets to show the effectiveness of the multithreaded algorithm on the runtime.

1. Yeast: We use the Yeast protein-protein interaction network with $6,249$ vertices and $224,587$ edges from the Biological General Repository for Interaction Datasets (BioGRID) [61]. Gene profile attribute information correspond to the differential expression value of each gene when exposed to 173 different experiments [29]. Each gene has 173 real attributes. For reference, YeastHC has 4,008 vertices and 9,857 edges in its graph.

2. Human: We use the Human protein-protein interaction network with $20,313$ vertices and $230,845$ interactions from the BioGRID [61]. For attribute data, we use the dysregulation profile of genes in 13 different cancers (attributes) where '1' indicates that the gene is dysregulated [43].

We compare the runtime of MT_Redcone with varying number of threads. Figures 3.10 and 3.11 plots the runtime for varying values of density (θ) and dimension (S_{min}), respectively, for the Yeast dataset. Figures 3.12 and 3.13 plots the runtime for varying values of density (θ) and dimension (S_{min}), respectively, for the Human dataset. We ran multiple experiments with varying number of threads, beginning from a single thread (RedCone) to parallelizing with upto 32 threads. MT_Redcone is multiple times faster than the single thread execution of RedCone . The speedup is bounded by the number of cores in the CPU which is 8 in our machine. Moreover, as we increase the number of threads beyond 8, no gain in speedup is obtained and we start seeing the impact of the computational overhead (this behavior can be seen in figure 3.10).

3.4.5. Representative set

As noted previously finding a reduced representative set is important when there are millions of output maximal cohesive dense clusters. A reduced representative set (of maximal cohesive dense clusters) is more manageable and is better suited for analysis as they closely represent the entire population of output clusters. We will look at the results from this reduction process on YeastHC dataset.

We ran the modified K-Medoids algorithm on the output space of MT_Redcone for varying parameters of both MT_Redcone and K-Medoids . Table 3.3 presents some of the results of K-Medoids algorithm. θ, γ, t, s_{min} represent the parameters for the MT_Redcone and $|N|$ denotes the number of output maximal cohesive dense clusters. K and α are the parameters for K-Medoids where K represents the number of desired clusters in representative set and α is a user defined

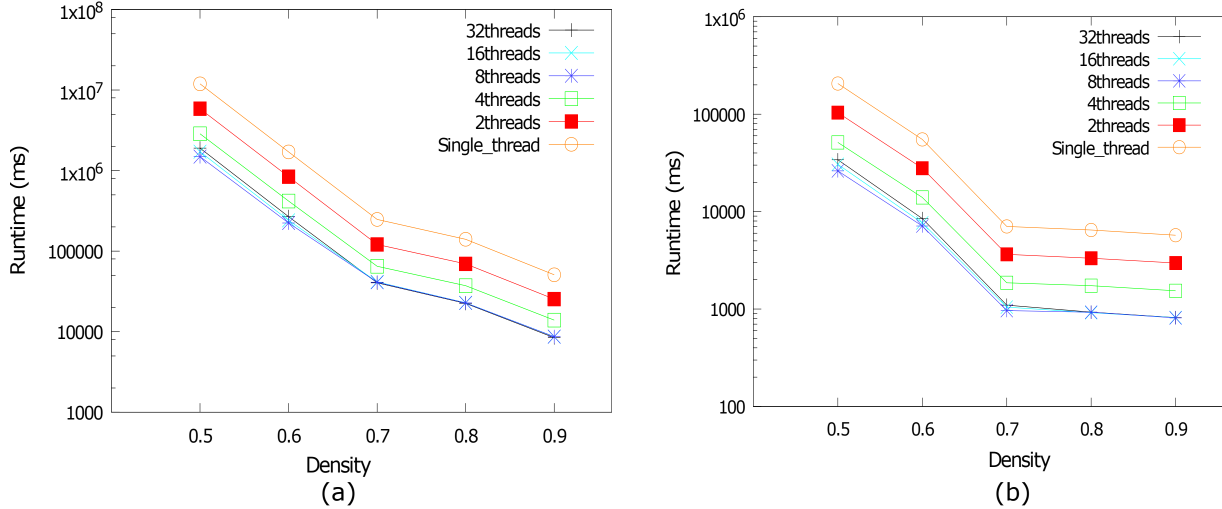

Figure 3.10. Runtime comparison of multiple threads on the Yeast dataset with varying density, parameters : (a) $t = 0.4$ and $s_{min} = 60$ (b) $t = 0.3$ and $s_{min} = 65$

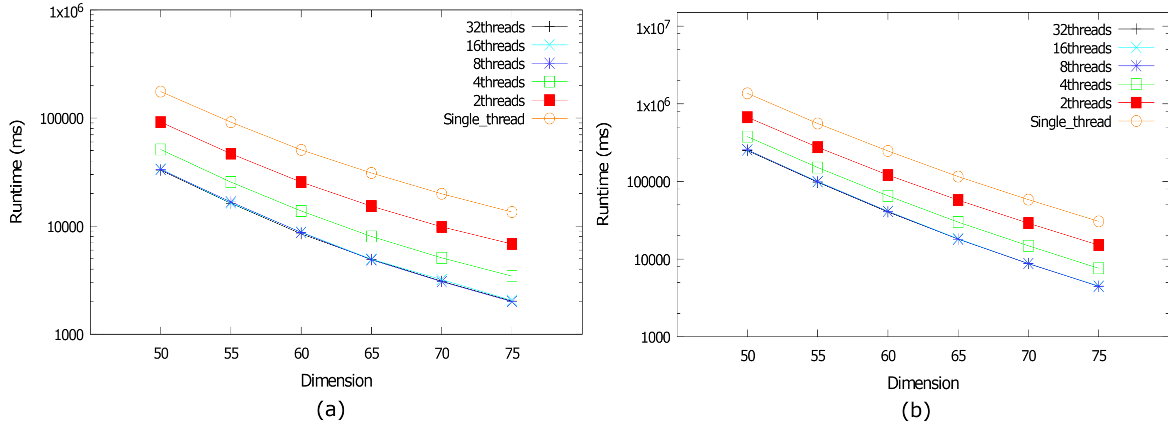

Figure 3.11. Runtime comparison of multiple threads on the Yeast dataset with varying dimension, parameters : (a) $t = 0.4$ and $\theta = 0.9$ (b) $t = 0.4$ and $\theta = 0.7$

parameter to control the similarity between clusters. $K\%$ is the percent of $|N|$ clusters desired in a representative set. We ran multiple experiments varying both sets of parameters and calculated the average intra partition similarity and average inter medoid similarity as defined previously. Figure 3.14 and 3.15 plot the average intra partition similarity and average inter medoid similarity for varying values of α and K.

In figure 3.14, as α is increasing, the similarity calculation between cohesive dense clusters is biased more on the network structure than the attributes of clusters. Notice as α is increasing the $AvgPartitionSim$ is increasing and $AvgMedoidSim$ is decreasing. As both metrics complement each other, they indicate in reinforcement, that the output space (maximal cohesive dense

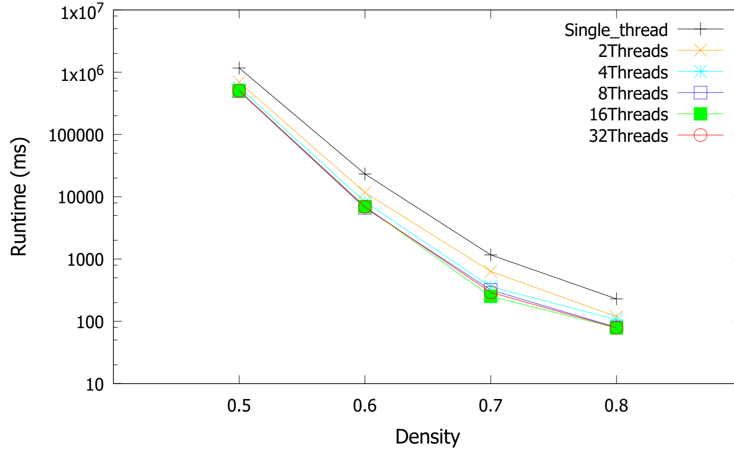

Figure 3.12. Runtime comparison of multiple threads on the Human dataset with varying density, parameters : $s_{min} = 4$

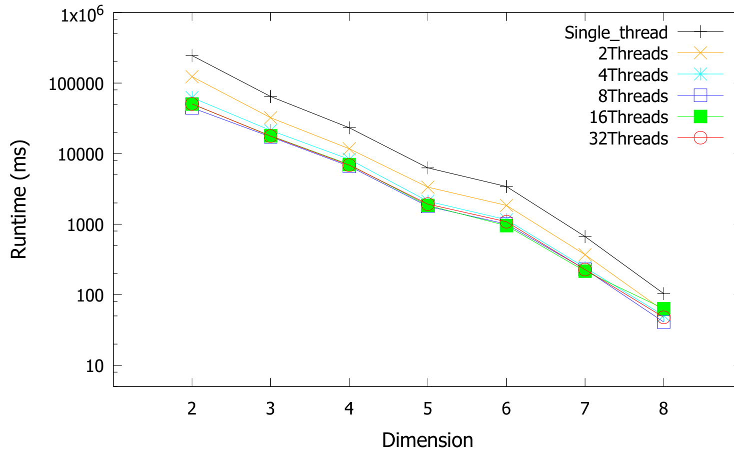

Figure 3.13. Runtime comparison of multiple threads on the Human dataset with varying dimension, parameters : $\theta = 0.6$

clusters) forms better partitions when the similarity metric is biased towards the network structure of maximal cohesive dense clusters. In other words, the maximal cohesive dense clusters resulting from this YeastHC dataset has more similarity in its network structure over their attributes. This trait is demonstrated by the above figure as increasing α values increases similarity among these clusters and hence form better partitions.

In figure 3.15, we observe the trend in $AvgPartitionSim$ and $AvgMedoidSim$ values, while increasing the desired number of representative clusters. As K increases, we increase the number of partitions in the output space. We see a steady increase in $AvgPartitionSim$ while $AvgMedoidSim$ is pretty much flat. This is expected because with increasing number of partitions

Table 3.3. Similarities of medoids in Representative Set.

K-Medoids				K-Medoids		Similarities	
θ/γ	t	s_{min}	$\|N\|$	$K\%$	α	$AvgPartitionSim$	$AvgMedoidSim$
0.5	0.2	15	996	10%	0.5	0.53	0.053
0.7	0.2	15	2417	10%	0.5	0.48	0.085
0.8	0.2	10	6100	10%	0.5	0.50	0.038
0.9	0.3	15	9989	10%	0.5	0.55	0.059
0.5	0.2	15	996	70%	0.5	0.88	0.052
0.7	0.2	15	2417	70%	0.5	0.86	0.077
0.8	0.2	10	6100	70%	0.5	0.86	0.038
0.9	0.3	15	9989	70%	0.5	0.87	0.059
0.5	0.2	15	996	10%	0.9	0.58	0.024
0.7	0.2	15	2417	10%	0.9	0.53	0.007
0.8	0.2	10	6100	10%	0.9	0.57	0.022
0.9	0.3	15	9989	10%	0.9	0.61	0.031

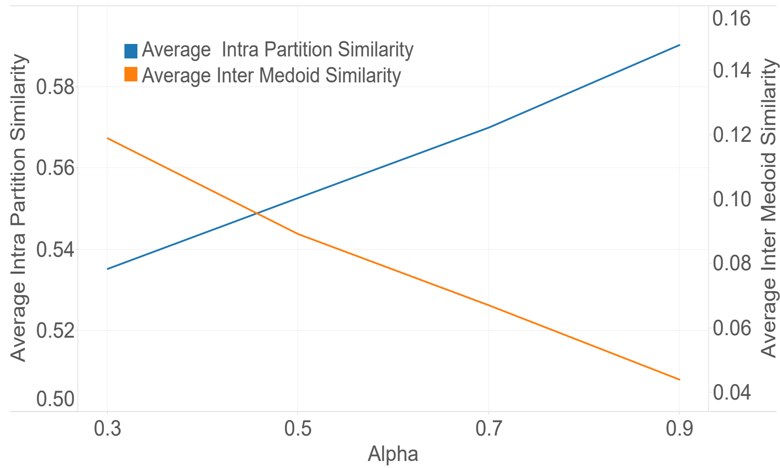

Figure 3.14. Average intra partition similarity $AvgPartitionSim$ and average inter medoid similarity $AvgMedoidSim$ with varying alpha values.

the members in each partition decrease, and, the similarity of each member in the partition to their respective medoid increases. This increase in similarity is captured by the increasing trend of $AvgPartitionSim$. At 100% value of K, each cluster is its own partition and medoid and has a perfect simliarity score of 1. On the other hand, $AvgMedoidSim$ remains invariable because the average distance between medoids does not change with increasing number of medoids.

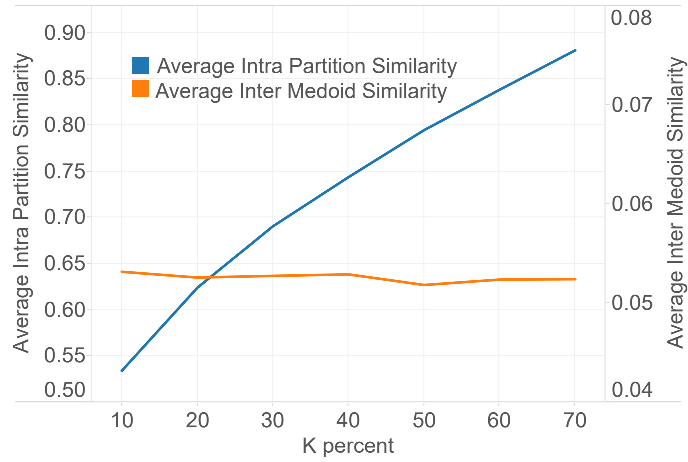

Figure 3.15. Average intra partition similarity $AvgPartitionSim$ and average inter medoid similarity $AvgMedoidSim$ with varying K values.

4. MINING COHESIVE SUBNETWORKS

In this chapter we look at the problem of mining cohesive networks. Highly cohesive clusters may not always be densely connected and hence we need a new definition and approach for finding such patterns. Figure 4.1 shows two identical graphs with node attributes which highlight two cohesive subnetworks. Figure 4.1 (a) shows a graph which has a highly dense cohesive subnetwork highlighted in black. The density of this dense cohesive network is 0.8. Figure 4.1 (b) shows another highly cohesive subnetwork but it is not dense, in fact its density is 0.4.

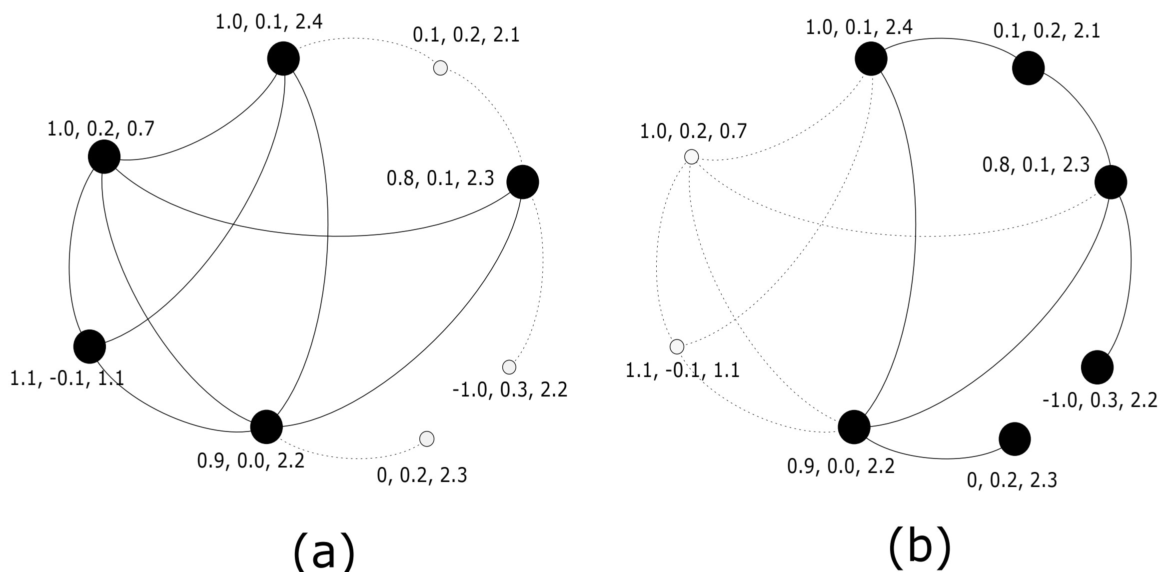

Figure 4.1. Example of cohesive network. Graph containing two communities, (a) An example of dense cohesive subgraph and (b) an example of a cohesive subgraph

The previous definition of dense cohesive networks cannot be extended to the problem of finding cohesive networks, because no density threshold exists in this problem. Moreover even if a very low density threshold is given as an input, there can always be cohesive network which has lower density then the input threshold value and hence might remain undiscovered.

4.1. Problem description

Given an attributed graph $G = (V, E, f)$, two thresholds t and s_{min}, the problem of mining the set of **maximal cohesive subgraphs** is to find the set:

$$\mathcal{M} = \{M_1, M_2, M_3, \cdots, M_{|\mathcal{M}|}\}$$

such that every $M_i \in \mathcal{M}$ is a maximal cohesive subgraph. Each M_i is a tuple $\{G_i, A_i\}$ containing a subgraph and its relevant attributes. A cohesive subgraph M_i is called a maximal cohesive subgraph, if $\nexists M_i'$ such that $M_i' \supseteq M_i$ and M_i' is cohesive.

4.2. Algorithm

A naïve way of solving this problem is by enumerating all possible patterns and then reporting only the unique cohesive patterns or clusters. Each pattern found is added to a list. Before a new pattern is added to the list, we check if a super pattern exists in the list which subsumes the current pattern. This ensures that a pattern has not been visited before and keeps the discovery limited to unique patterns. The next section describes this algorithm in detail. We also propose an algorithm MinCone to address the computational issues in the naïve approach.

4.2.1. Brute force approach

The basic idea behind this brute force approach is to build an enumeration tree and only add a maximal cohesive pattern to an output list if the list doesn't contain a super pattern that subsumes the current pattern. Algorithm 3 shows the pseudo code for brute force approach. Line 17 shows the condition to check the list before adding a new pattern. Another useful optimization is to check whether a child pattern exists in the list before actually creating the child pattern. If its already in the list then the entire subtree rooted at the current node can be pruned because this child pattern has been visited once before and will yield the exact same subtree as before. This optimization to the brute force approach can be seen in line 8. The enumeration tree can also be pruned on the cohesive constraint as shown in line 9 as the number of similar attributes is an anti-monotone property.

The biggest problem with brute force approach is that we have to constantly check the list to output unique patterns. For very large graphs the checking for duplicates in the list becomes more costly than the actual recursive enumeration and cohesive constraint check. This severely

restricts the scalability of this algorithm for large graphs. To address the limitations of the brute force approach we propose an efficient algorithm for finding cohesive clusters in the next section. Moreover this new approach can be parallelized for even better runtime efficiency.

Algorithm 3 Brute force : Maximal Cohesive Cluster Discovery

Input:
$G = (V, E, f)$: an attributed graph
min_size: the minimum size of cluster to include in results
t: tolerance threshold between two attribute values in a single subspace
s_{min}: minimum number of similar attributes per cluster

Output:
$\mathcal{M}$: maximal cohesive clusters
Q : List for storing visited nodes

```
 1: Remove all non cohesive edges from input graph
 2: M = {}
 3: BruteForceMineclusters({})
 4: function BruteForceMineclusters(U)
 5:     locally_maximal ← true
 6:     for v ∈ V\U do
 7:         Let U' = U ∪ v
 8:         if NotVisited(U', Q) then
 9:             if |A(U')| ≥ s_min then
10:                 locally_maximal ← false
11:                 AddToVisitedQueue(U', Q)
12:                 BruteForceMineclusters(U')
13:             end if
14:         end if
15:     end for
16:     if locally_maximal and |U| ≥ min_size then
17:         if ∄M_i ∈ M such that U ⊆ M_i then
18:             M = M ∪ U
19:         end if
20:     end if
21: end function
22: return M
```

4.2.2. MinCone approach

We propose an efficient approach to mine cohesive clusters named MinCone (**MIN**ing **CO**hesive **NE**tworks). This algorithm builds an enumeration tree and utilizes the reverse search principle technique to grow patterns such that algorithm does not visit the same pattern twice. This means that a there exists a unique path from the root to a pattern in the enumeration tree. This eliminates the need to check whether a pattern or even a super pattern has been visited before. At any given search node only valid child search nodes are explored. The valid child search nodes maintain the cohesive property.

4.2.2.1. Minimum spanning tree

The MinCone algorithm utilizes the mininum spanning tree for enumerating patterns. A spanning tree of a graph is a tree which connects all the vertices together. A single graph can have multiple spanning trees. If all the edges of the graph have unique weights then the sum of the edges in the spanning tree can represent a weight for the spanning tree. A minimum spanning tree (MST) is then a spanning tree with the least weight among all possible spanning trees.

Let $G = (V, E, w)$ be a weighted undirected graph, where $w : E \rightarrow \mathbb{R}$ is a weight function defined on its edges. Every edge in G has a unique weight or in other words function w defines a strict total ordering on the edge weights, i.e., for any two edges e_i, $e_j \in E[G]$ with $e_i \neq e_j$ either $w(e_i) > w(e_j)$ or $w(e_i) < w(e_j)$ holds. The weight of a spanning tree $g = (V, E_g)$ is defined to be $w(g) = \sum_{e \in E_g} w(e)$. As stated earlier a minimum spanning tree (MST) is a spanning tree whose weight is less than or equal to the weight of every other spanning tree. So we can say that there exists a unique spanning tree of G represented by $MST(G)$ such that $w(MST(G)) \leq \forall i \, w(\mathbb{S}_i(G))$ where $\mathbb{S}$ represents the set of spanning trees for G. Figure 4.2 (c) shows a graph and its corresponding MST in Figure 4.2 (d).

4.2.2.2. Parent child relationship

Now we describe the parent child relationship as utilized by MinCone . Figure 4.2 (a) shows a sample enumeration tree where A, B, U, U^* are all cohesive clusters. U is the parent cluster of U^* such that $U^* := U \cup v$. In figure 4.2 (b) and (c) we see the cohesive clusters $U = \{1, 2, 3, 4, 5, 6, 8\}$ and $U^* = \{1, 2, 3, 4, 5, 6, 7, 8\}$ respectively represented as graphs. To establish the parent child relationship we need to find the MST of U^* which is shown in 4.2 (d). The vertex 7 is highlighted in figure 4.2 (d) because its the only vertex which is connected to a single edge with the smallest edge weight. Note the condition of connected to a single edge in turn restricts the degree of the vertex to 1. The set of vertices having a degree of 1 in the MST is $\{4, 5, 7, 8\}$. Out of these vertices, vertex $v = \{7\}$ has the smallest weight of $w = 3$. A child cluster is said to be a valid child of its parent if the vertex v which is used to grow the child cluster from the parent is the same vertex which has a degree of 1 and the smallest edge weight in minimum spanning tree of the child cluster.

Lets assume a graph $G = (V, E, f, w)$ be a weighted undirected graph with node attributes, where w gives weight for each edge and f gives the attribute for each vertex. For a given set of

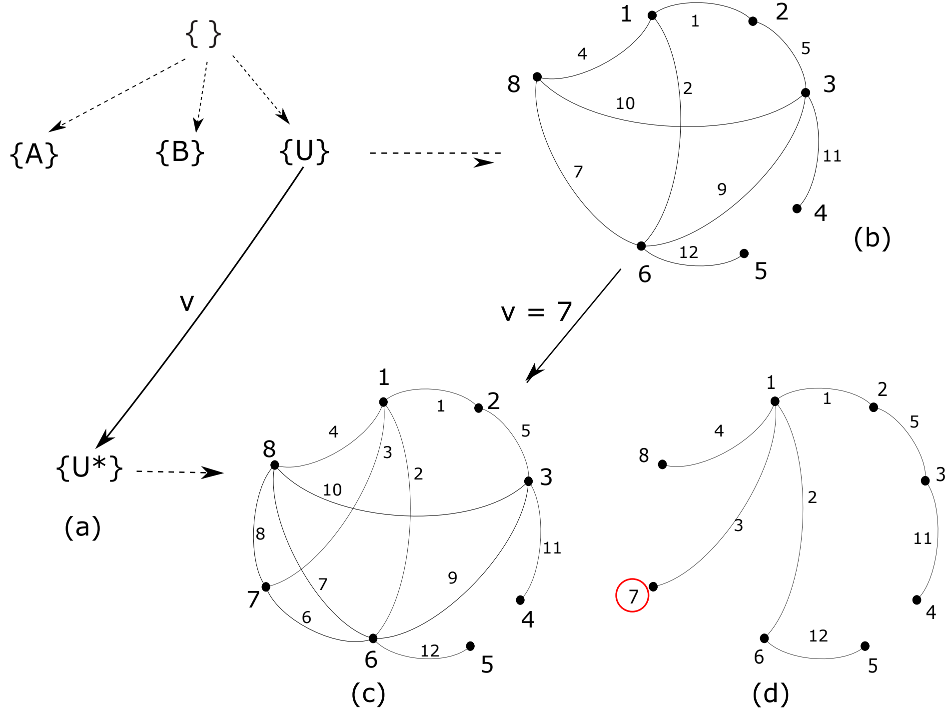

Figure 4.2. MinCone approach.(a) Sample enumeration tree where A, B, U, U^* are all cohesive clusters, cluster U is the parent of the cluster U^* (b) cohesive cluster represented by U (c) child cohesive cluster represented by U^* (d) MST of U^*.

vertices U, we define $MST(U)$ as the minimum spanning tree of the graph induced by $G[U]$. Given U and $v \in V \setminus U$. $U^* := U \cup v$ is a child of U if and only if

$$\forall e_i \in E[MST(U^*)] : deg_{MST(U^*)}(v) = 1 \wedge w(indEdge(U^*, v)) < w(e_i)$$

where $indEdge(U^*, v)$ is the edge connecting the vertex v in $MST(U^*)$. Since the degree of v in $MST(U^*) = 1$, $indEdge(U^*, v)$ uniquely represents one edge. Also $deg_{MST(U^*)}(v)$ represents the degree of vertex v in the graph represented by $MST(U^*)$.

4.2.2.3. Algorithm

Algorithm 4 shows the pseudo code for the MinCone . The recursive function builds an enumeration tree like the example in Figure 3.2. The result of this algorithm is the set of maximal cohesive clusters $\mathcal{M}$.

We employ a couple pruning strategies to reduce the size of the enumeration tree. At first all non cohesive edges are removed from the input graph as shown in line 1. The end points

Algorithm 4 MinCone : Maximal Cohesive Cluster Discovery

Input:
$G = (V, E, f)$: an attributed graph
min_size: the minimum size of cluster to include in results
t: tolerance threshold between two attribute values in a single subspace
s_{min}: minimum number of similar attributes per cluster
Output:
$\mathcal{M}$: maximal cohesive clusters

```
 1: Remove all non cohesive edges from input graph
 2: M = {}
 3: MINECLUSTERS({})
 4: function MINECLUSTERS(U)
 5:     locally_maximal ← true
 6:     for v ∈ V\U do
 7:         Let U' = U ∪ v
 8:         if |A(U')| ≥ s_min then
 9:             locally_maximal ← false
10:             if ISCHILD(U', U) then
11:                 MINECLUSTERS(U')
12:             end if
13:         end if
14:     end for
15:     if locally_maximal and |U| ≥ min_size then
16:         M = M ∪ U
17:     end if
18: end function
19: return M
```

(vertices) of these edges will never be a part of any cohesive cluster together hence they can be safely removed without losing any clusters. The second pruning is based on the cohesion of the edges while growing a parent node. Before a child node can be created the algorithm checks to see if the potential child node is cohesive. If it is cohesive then the algorithm recursively creates the child node. If no cohesive child node exists for the current search node then the current search node is maximally cohesive and it can be added to $\mathcal{M}$ (line 16).

4.2.3. Multithreaded MinCone approach

In this section we extend MinCone algorithm by proposing a parallel implementation called MT_MinCone. Recall that MinCone uses a special reverse search enumeration technique [3] that guarantees a unique child parent relationship, i.e., a child search node will always have a unique parent node in this tree. Following this corollary of this unique child parent relationship, the enumeration tree built by MinCone ensures that every node has a unique path from the root of the enumeration tree. This property eliminates the need to check whether a subgraph has been visited

before because it will only ever be visited once. We exploit this unique child parent relationship to develop a parallel implementation of mincone.

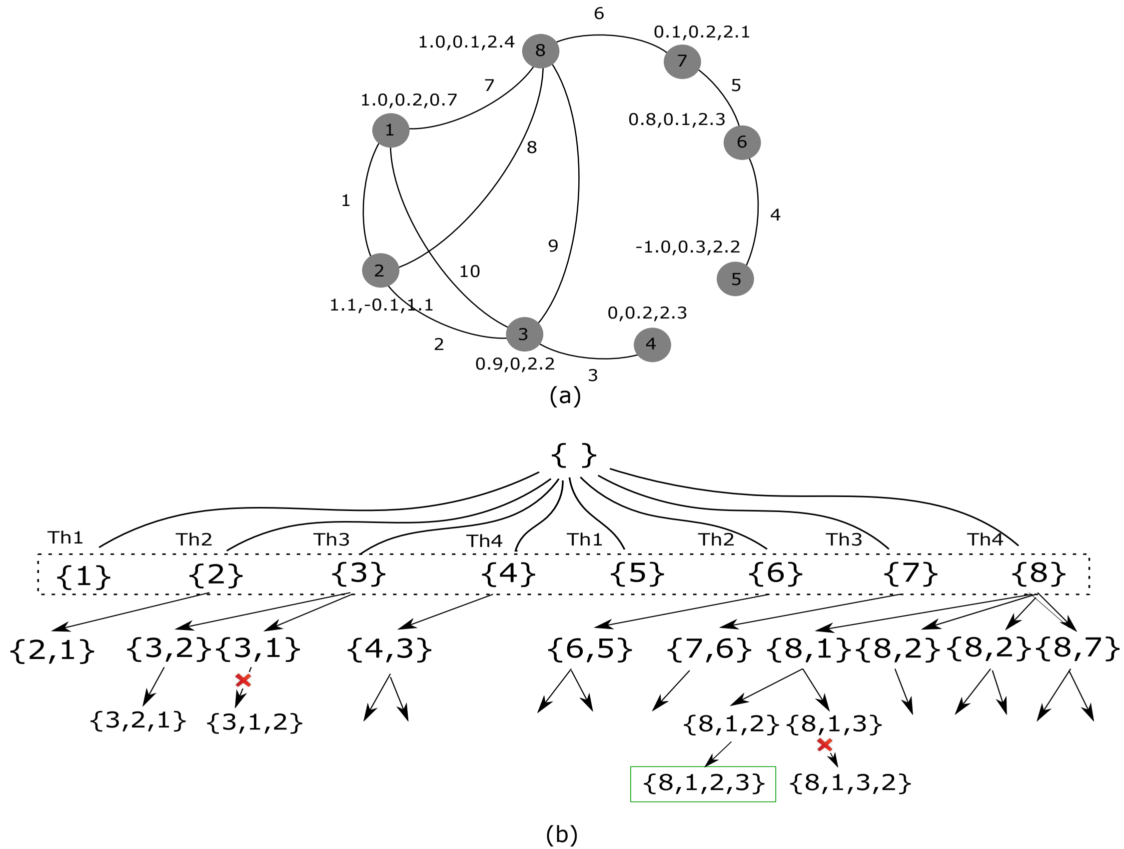

Figure 4.3. The input graph and a portion of the corresponding enumeration tree built by MT_Mincone . There are four threads which build parts of the enumeration tree independently. Each thread builds two subtrees from the first level children. Crosses show which branches are pruned. The discovered maximal cluster is highlighted by a green box.

Figure 4.3 (a) shows an input graph (as seen earlier in Figure 4.1), and Figure 4.3 (b) shows the enumeration tree as created by MinCone . Utilizing this reverse search principle, the subtrees rooted under each first level node in the enumeration tree (figure 4.3) can be enumerated independently. This suggests that we can spawn multiple threads at the root, and each thread creates the subtree under each of the first level nodes. Each thread is going to enumerate its subtree concurrently and because of unique child parent relationship this algorithm will never miss any search nodes.

Algorithm 6 shows the pseudo code for MT_Mincone. The inputs include graph, G, tolerance threshold, t, dimensionality threshold, s_{min}, and a number of threads, $num_{threads}$. The algorithm begins by spawning the requested number of threads (line 3). Each thread then iterates over the first level nodes, selects a vertex and traverses its enumeration sub tree (line 9). The output of this algorithm $\mathcal{M}$ is a list of maximal cohesive subgraphs. Line 16 checks the cohesive constraint and only grows child nodes if the current node is cohesive. In line 18 we compute the MST of U' and determine whether U' is a valid child of U, according to child parent definition established above.

Algorithm 5 Pseudo-code for a parallel (Multithreaded) algorithm for Maximal Cohesive subgraph Detection

Input:
$G = (V, E, f)$: An attributed graph
min_size: Minimum size of cohesive subgraphs
t: Tolerance threshold in a single subspace
s_{min}: Minimum number of similar attributes
$num_{threads}$: Number of threads

Output:
$\mathcal{P}$: maximal cohesive clusters

```
 1: Remove all non cohesive edges from input graph
 2: P = {}
 3: threads[] = spawn_threads(num_threads)
 4: start_all_threads(threads[], ThreadStart)
 5: join_all_threads(threads[])
 6: function ThreadStart
 7:     while there are more unexplored vertices do
 8:         Ensuring mutual exclusion, choose a vertex, v
 9:         execute_thread(MineClusters({v}))
10:     end while
11: end function
12: function MineClusters(U)
13:     locally_maximal ← true
14:     for v ∈ V\U do
15:         Let U' = U ∪ v
16:         if (|A(U', D, t)| = T) ∧ |D| ≥ s_min then
17:             locally_maximal ← false
18:             if isChild(U', U) then
19:                 MineClusters(U')
20:             end if
21:         end if
22:     end for
23:     if locally_maximal and |U| ≥ min_size then
24:         P = P ∪ U
25:     end if
26: end function
27: return P
```

4.3. Experiments

We compare MinCone against the brute force approach. We used a real-world network and its associated attribute data: *High Confidence Yeast (YeastHC)*. All experiments were run independently on an Arch Linux operating system with an Intel Core i5-2500K (3.3GHz) processor and 8 Gigabytes of main memory.

4.3.1. Cohesive clusters in the YeastHC

Table 4.1 shows the topological properties reported by MinCone for the YeastHC dataset. In the table, $|N|$ denotes the number of resulting clusters and $\overline{N}$ represents the average size. Both MinCone and brute force approach outputs the exact same clusters hence results from brute force are not explicitly shown in the table.

Table 4.1. Topological properties of cohesive clusters for the YeastHC dataset.

Parameters		MinCone	
t	s_{min}	$\|N\|$	$\overline{N}$
0.300	40	1645	4.14
0.300	50	255	4.19
0.300	60	441	4.32
0.300	70	9	4.78
0.325	40	3919	4.19
0.325	50	627	4.13
0.325	60	98	4.33
0.325	70	18	4.44
0.350	40	15730	4.24
0.350	50	2382	4.14
0.350	60	485	4.10
0.350	70	64	4.27
0.375	40	38976	4.27
0.375	50	54514	4.21
0.375	60	1102	4.10
0.375	70	168	4.23
0.400	40	122783	4.4
0.400	50	19326	4.31
0.400	60	3485	4.2
0.400	70	825	4.09

Figure 4.4 shows a maximal cluster from the MinCone's output on the YeastHC dataset and a matrix illustrating the attribute data for the vertices in this cluster. The matrix shows the

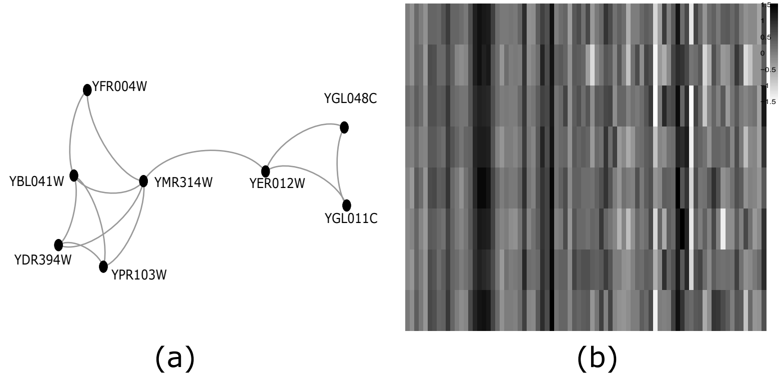

Figure 4.4. Representation of a cohesive pattern from the YeastHC dataset (a) Network structure of the pattern (b) Attributes of the nodes in the pattern; Parameters: $t = 0.4$ and $s_{min} = 40$ (Only 80 attributes are shown). Notice the pattern is not particularly dense

vertices on the rows and attributes on the columns. The first 40 columns in the matrix are the attributes where these vertices are similar and therefore show very little deviation in its gray shade. The last 40 columns are a sample of 40 attributes from the remaining 133 attributes, where these vertices are not similar.

4.3.2. Running Time

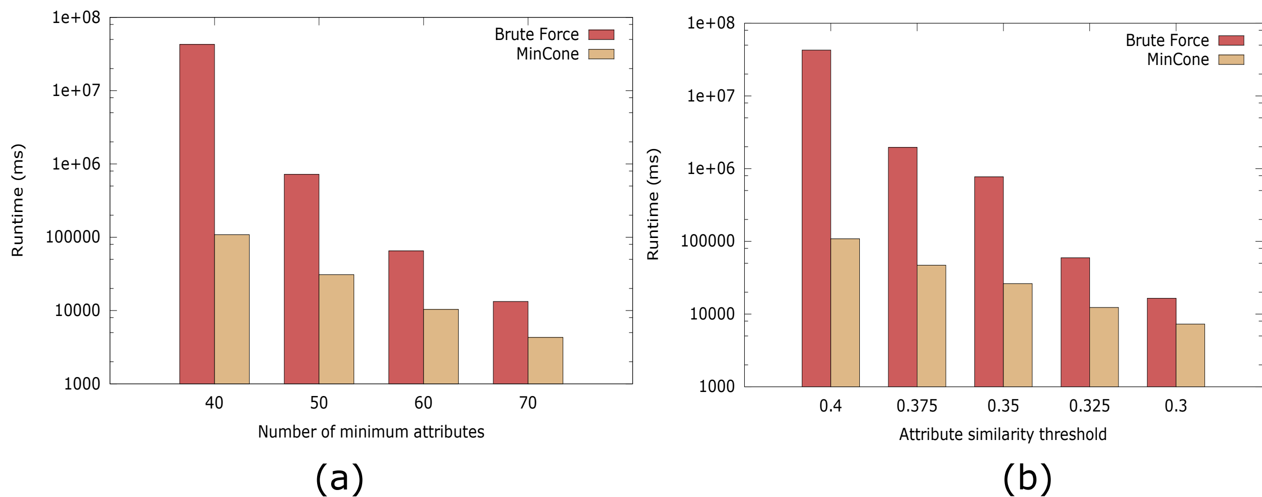

Figure 4.5. Runtime comparison of brute force approach and MineCone on the YeastHC dataset (a) parameters : $t = 0.4$ (b) parameters : $s_{min} = 40$

Similar to RedCone, MinCone performs the cohesive constraint check over real attribute for each cluster which takes $O(dn)$, where d is the number of attributes and n is the maximum

number of nodes in a cluster. MinCone also finds a minimum spanning tree for each cluster in an attempt to create a child node. We employ Kruskal's spanning tree algorithm to find MST for each cluster. Kruskal's algorithm takes an additional $O(n^2 logn)$. MinCone is a polynomial-delay algorithm which means that the computation time between reporting two clusters is polynomial in the input size. The running time of the algorithm thus depends on the number of reported maximal cohesive clusters which is controlled by the thresholds.

We compare the running time of MinCone with brute force approach for varying parameters on the YeastHC dataset. Figure 4.5 show that MinCone outperforms brute force in every case. For example, in Figure 4.5 (a) for, t = 0:4 and smin = 40, Mincone is almost 400 times faster than the baseline approach.

4.3.3. Multithreaded Runtime

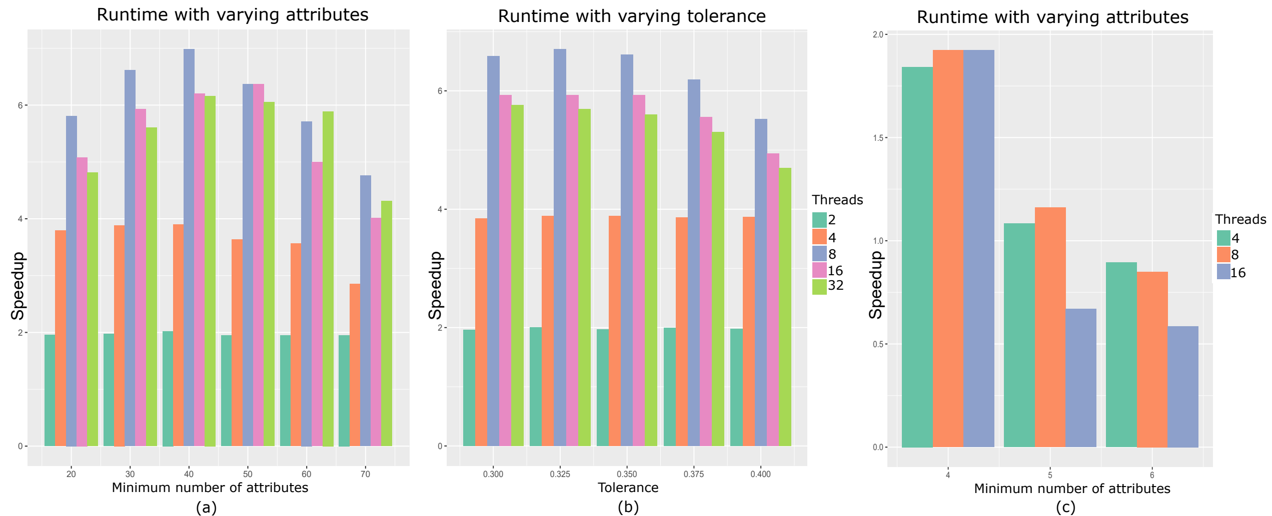

Figure 4.6. Speedup in runtime for multiple threads (a) on the Yeast dataset with varying dimension while tolerance is fixed at $t = 0.35$ (b) on the Yeast dataset with varying tolerance while dimension is fixed at $s_{min} = 30$ (c) on Human dataset with varying dimension

In this section, we compare the runtime of MT_Mincone with varying number of threads. Figures 4.6 plots the speedup in runtime for varying parameters in Yeast and Human datasets respectively. We ran multiple experiments with varying number of threads, beginning from a single thread to 32 threads. MT_Mincone is multiple times faster than the single thread execution of MinCone . For instance, a single thread takes roughly 20 minutes for finding patterns with

parameters $t = 0.375$ and $s_{min} = 20$, however the parallel execution of the same experiment with 8 threads takes less than 4 minutes.

We noticed a maximum speedup of 7 as shown in 4.6 (a) for $s_{min} = 40$ and 8 threads. The speedup is bounded by the number of cores in the CPU which is 8 in our machine. The maximum speedup is achieved with 8 threads, as we increase the number of threads beyond 8, no gain in speedup is obtained and we start seeing the impact of the computational overhead.

5. SAMPLING DENSE AND COHESIVE NETWORKS

Several recent studies [50, 19] have generated tremendous amounts of genomic data. Human PPI data in BioGRID database increased quite considerably over the last few years now totaling to 368,417 raw interactions as of March 2017 update [12].

The module detection algorithms discussed previously will not be able to scale to such large graphs [36, 30, 34]. Although the multithreaded clustering algorithm [32] tried to speedup execution by utilizing multiple threads, it may not be enough and we need a new way to detect modules with an ever increasing database. The biggest issue with all the above algorithms is that they all enumerate the entire output space which is often exponential to the input graph size. Enumerating an exponential output space over an input graph of roughly 350K interactions (current human PPI network) is prohibitively expensive. This restriction creates an opportunity to find a new technique of detecting modules without needing to enumerate the entire output space.

5.1. Sampling

In this section, we propose a sampling technique which can output a reduced set of cohesive and dense modules without enumerating the entire output space. To further understand this approach we need to define the partial order graph of a graph. The partial order graph POG of a graph G is a graph where each node represents a subgraph in G and an edge is drawn between two subgraphs (nodes of POG) when they differ by only one vertex in G. We will define this further in the problem definition section.

Basically, this algorithm performs a random walk on the partial order graph, and returns modules from the partial order graph when the walk converges to a stationary distribution. The stationary distribution of the random walk is established to match the preferred qualities of the output modules such as density and profile constraints. The qualities desired of the output modules can be converted into a score and plugged into the algorithm. In the stationary distribution the output modules with higher score have a higher probability of random walk visit. Finally we rank the output modules by their visit count and return the top k modules where k is a user supplied parameter.

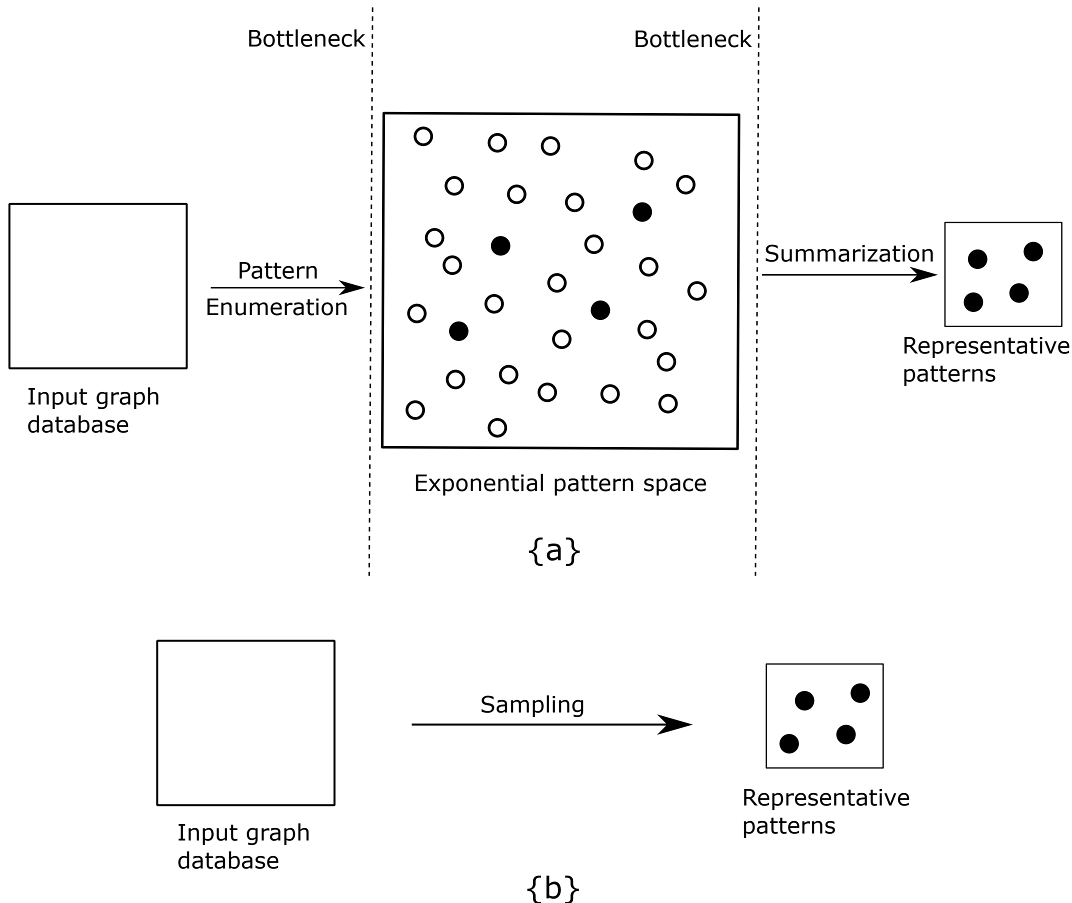

Figure 5.1. (a) Module enumeration process enumerates all modules from the input graph. Modules are shows as circles and representative modules are shown by filled circles. Summarization process then tries to reduce this exponential output set to a representative set. (b) Sampling technique outputs a reduced set directly from the input graph database without enumeration.

The biggest benefit of this technique is the scalability of the algorithm. As the algorithm builds the partial order graph POG locally (while its sampling), it doesn't need to enumerate the entire output space ahead of time. Another benefit is that it eliminates the need for summarization as it only outputs the most visited (high quality) modules. This way the expensive operation of finding representative modules from an astronomically large output space is also removed.

5.2. Related work

Sampling from a graph has been widely studied and has diverse applications, for example, survey hidden population in sociology [66], visualize social graph [49, 63] and scale down internet autonomous systems graph [47]. A complete survey of graph sampling is discussed here [40].

There are multiple sampling techniques, some of them are described below. A graph G represented as $G = \{V, E\}$ where V is the set of vertices and E is the set of edges. A sample graph is represented by $G_s = \{V_s, E_s\}$ where $V_s \subset V$ and $E_s \subset E$.

5.2.1. Vertex based sampling

The idea here is to select a subset of vertices V_s from V, where V_s can be selected either randomly or according to a target distribution. The edges between the vertices of V_s are only kept in the edges E_s, $E_s = (u, v) \in E | u \in V_s, v \in V_s$. Vertex sampling has been applied to estimate the network size and network density of very large graphs in [48].

5.2.2. Edge based sampling

Similar to vertex based sampling, a subset of edges E_s, $E_s \subset E$ is selected. The vertices found in the endpoints of the edge set E_s are kept in V_s, i.e., $V_s = (u, v | (u, v) \in E_s$. Benczur et. al. uses edge sampling to find the minimum cut and flow problems in large graphs [5].

5.2.3. Traversal based sampling

In this technique the sampling starts with a set of initial vertices (or edges) and expands the sample based on current observations. Snowball Sampling [31] is a traversal based sampling, where the sample starts with an initial set of nodes in the graph and new nodes are obtained from the neighbors of the initial nodes. This is a non-probabilistic algorithm because all neighbors of the selected vertex are added to the sample.

Forest Fire [52] is a probabilistic version of snowball sampling, instead of selecting all neighbors of the current node, x outgoing edges are selected based on a geometric distribution. The other endpoints of these x outgoing edges are stored and then these x outgoing edges are burned or destroyed. The process recursively repeats at the stored endpoints. Continuing this way the process ends when there are no further outgoing edges. All the edges that are burned form a sample. Another paper [51] proposes a sampling method and compares with the original graph back in time when the original graph was the size of the sample. This paper uses forest-fire methods and show good accuracy for sample sizes down to about 15% of the original graph.

Metropolis-Hastings random walk [57] is a traversal based approach where new nodes are chosen randomly from the set of neighbors of the current node. Unlike travsersal based approach, random walk only depends on their previous state as a node can be revisited in random walk technique. Metropolis-Hastings can be applied to obtain a desired distribution of vertices in the

sample. Hasan et.al [1] proposed an output space sampling technique based on Metropolis-Hastings algorithm to mine dense subgraphs only.

To the best of our knowledge sampling dense and cohesive subgraphs from a large graph is very novel and has not been addressed before.

5.3. Problem Description

Definition 8. *Partial Order Graph (POG) : Given a graph G, parameters θ, t and s_{min} and S which contains the set of all the cohesive dense modules in G satisfying these parameters. The partial order graph M is a graph where each vertex of the M is a cohesive dense module in G for the supplied parameters. $M = (V', E')$ where every $v \in V'$, $v \in S$ holds. Edges of this graph denote the sub graph relationship between the cohesive dense modules, i.e., for any edge $e \in E'$ which connects two vertices u and v either of $u \subset v$ or $v \subset u$ holds, where both $u, v \in S$.*

Figure 5.2 (a) shows a sample input graph G and Figure 5.2 (b) shows the POG, M. We assume that every possible module of G is cohesive and dense for the given parameters. We can see the M starts from an empty set and recursively adds new vertices till it enumerates all possible modules. Vertices in M represent cohesive and dense modules and edges represent an extension of a module to another module by adding a new vertex from G. For e.g., in figure 5.2 (b) the node {B} grows to node {A,B} by adding vertex A from G to node B. This is similar to an enumeration tree and various algorithms [36, 34] have suggested different traversal techniques to mine modules from this enumeration tree.

Definition 9. *Module score : Given a graph G, parameters θ, t and s_{min} and S which contains the set of all the cohesive dense modules in G satisfying the parameters. We define three more parameters α, β and γ which are used to calculate a score for each module $U \in S$. The score of each module is calculated as*

$$\Delta_U = (|U|^{\alpha}) * (\rho(U)^{\beta}) * (|A(U)|^{\gamma})$$

The three terms are size of module $|U|$, density of module $\rho(U)$ and number of cohesive attributes $|A(U)|$. The score quantifies the interestingness of a module. The idea behind the sampling algorithm is that we will mine the modules with better scores over others. Also the

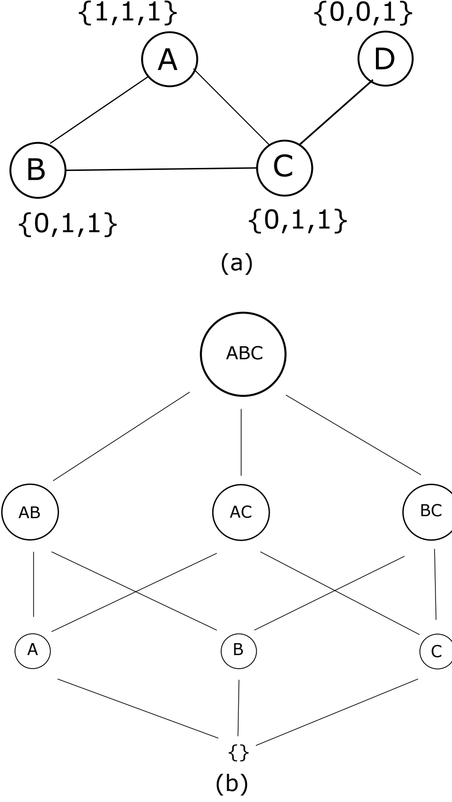

Figure 5.2. A input graph G and the corresponding POG represented by M, assuming every module of G is cohesive and dense.

score parameters (α, β, γ) provide a way to bias the sampling algorithm to find bigger modules or modules which are more dense or modules with high cohesiveness [36].

The score is a very important feature and greatly impacts the inner workings of the sampling algorithm. The score of a module is plugged into the sampling algorithm to perform random walk in POG. As the score reflects the desirability of a module and the random walk is biased to visit higher score modules, the sampling algorithm ends up visiting the higher score modules more than others. When the random walk converges we take the top k most visited modules.

Problem Definition: Given an attributed graph G, three thresholds θ, t, s_{min} and a desired number of output modules k ($k < |N|$), S contains the set of all the cohesive dense modules in G, the problem is to sample a set of cohesive and dense modules

$$\mathcal{S} = \{U_1, U_2, U_3, \cdots, U_{|k|}\}$$

such that every $U_i \in S$ is a dense cohesive module with respect to input parameters and U_i should have a high score Δ_{U_i}, indicating that we will sample the modules with higher score over others.

5.4. Sampling algorithm

As discussed before the true problem of an enumeration algorithm is to enumerate an exponential number of modules which is very time consuming. Secondly, to assist analysis we need to reduce this exponential output space to a manageable set of modules. The sampling algorithm can help alleviate these issues by directly sampling the representative modules (reduced set of modules) without enumerating the output space entirely. We use the Metropolis-Hastings algorithm [65] to sample modules from the output space, such that the samples form a representative set of the most cohesive and dense modules from the output space. The sampling algorithm not only avoided the exponential enumeration of output space but also removed the need to summarize the exponential output space.

The main idea behind this algorithm is that it can draw samples from a population (output space) provided we have a function $f(x)$ that is proportional to the target distribution D. The target distribution D, describes the distribution of modules in the population. In the case of a uniform sampling where every module is equally likely to be in the representative set (sample) the probability of any module is $1/|N|$ where $|N|$ is the number of modules in the exponential output space. However, we are interested in mining cohesive and dense modules and we would like to bias the sampling probability to modules which are more dense and cohesive than others. In other words we have a distinct target distribution for all modules in the output space which favors some modules more than others. We model the probabilities such that the probability of a module which doesn't meet the parameter constraints is 0. The reduced constraint to have a function $f(x)$ which is proportional to the target distribution D makes Metropolis-Hastings a very useful method, as calculating the normalizing factor $|N|$ for the target distribution is very difficult (recall $|N|$ is exponential).

This algorithm iteratively generates samples and as more samples are produced the distribution of samples approximates the target distribution. The algorithm asymptotically converges to a steady state distribution which matches the target distribution. At each iteration the algorithm generates a new sample which is only dependent on the current sample. This new sample is either accepted or rejected based upon the scores of the current and the new sample.

5.4.1. Metropolis-Hastings algorithm

Metropolis-Hastings is a Markov chain monte carlo method. A Markov chain is a mathematical model which captures a stochastic process's transitions from one state to another. For a given set of states $T = \{t_1, t_2, ..., t_n\}$ and a process which can transition from one state to another with some probability, a Markov Chain can be represented by a *Transition probability Matrix*, where $P_{i|j}$ represents the conditional probability of transitioning to state i from j $(1 \leq i \leq n, 1 \leq j \leq n)$. For all $i, j \in T$, we have $0 \leq P(i|j) \leq 1$, and $\sum_i P(i|j) = 1$, i.e., the sum of probabilities in any given row adds up to 1. This indicates that sum of the probabilities of transitioning from a state j to all other state is 1.

For a given transition probability matrix P, the matrix P^k gives the transition probabilities from one state to another after k hops or transitions. As $\lim_{k \to \infty} P^k = W$, i.e., the transition probability matrix P approaches a stationary (steady) state represented by W. At stationary state, we have a stationary distribution w (a row vector over states T) for the Markov chain such that $w = wP$ relation holds, where w is the left eigen vector of the matrix P corresponding to the eigenvalue 1.

A sufficient but not necessary condition for the existence of the stationary distribution is the condition of detailed balance which is shown below.

$$w(i)P(j|i) = w(j)P(i|j), \forall i, j \in T \tag{5.1}$$

This condition establishes the existence of the stationary distribution by requiring that each transition i to j is reversible, i.e., the probability of being in state i and transitioning to state j must be equal to the probability of being in state j and transitioning to state i. We can start the derivation of the algorithm from two states at initial state.

$$\begin{aligned} P(i)P(j|i) &= P(j)P(i|j), \forall i, j \in T \\ &= \frac{P(j|i)}{P(i|j)} = \frac{P(j)}{P(i)} \end{aligned} \tag{5.2}$$

The transition probability $P(j|i)$ can be expressed in two steps

- Proposal probability : The proposal probability $g(j|i)$ is a conditional probability of selecting the next state j given the current state of i.

- Acceptance probability : The acceptance probability $\alpha(j|i)$ is a conditional probability of accepting the proposed state j

$$P(j|i) = g(j|i)\alpha(j|i) \tag{5.3}$$

Using in equation 5.2, we have

$$\frac{\alpha(j|i)}{\alpha(i|j)} = \frac{P(j)g(i|j)}{P(i)g(j|i)} \tag{5.4}$$

Metropolis choice for acceptance is

$$\alpha(j|i) = Min\left\{1, \frac{P(j)g(i|j)}{P(i)g(j|i)}\right\} \tag{5.5}$$

$P(j)$ and $P(i)$ is the target distribution for states i and j, such that $P(i) = a_i/|N|$, where a_i is some positive number. Recall $|N|$ is exponential in the output space and is difficult to calculate. However, according to Metropolis-Hastings algorithm we can substitute another function $score(i)$ which is proportional to $P(i)$. The function *score* relates to the interestingness of the module. Substituting $score(i)$ in equation 5.5

$$\alpha(j|i) = Min\left\{1, \frac{score(j)g(i|j)}{score(i)g(j|i)}\right\} \tag{5.6}$$

After calculating acceptance probability a new random value between 0 to 1 is generated called the rejection probability. The new module is accepted only if the acceptance probability is greater than the rejection probability. This technique encourages the algorithm to select the samples with higher score and to remain there while occasionally picking the low score modules.

Another condition for the stationary distribution is the requirement of a unique stationary distribution which is guaranteed by the ergodicity of the Markov chain. Ergodicity implies that the Markov chain is finite, irreducible and aperiodic. The proof of this claim is provided here [1].

5.4.2. Random walk

Sampling in the output space involves a random walk process which selects modules from the *POG*. In this context, we can project the modules of the partial order graph $M = (V', E')$ as states ($T = V'$) and the sampling algorithm as a random walk process which jumps from one module to next using the edges E' of the POG. The transition probability matrix P shows the probability of the random walk process to transition from one module to another. At stationary state, P converges to W which implies that every output module has a definite target probability distribution regardless of the initial starting state. This convergence is reached irrespective of the starting state as long as the the conditions of detailed balance and ergodicity are met.

We don't have to construct the complete *POG* for the random walk, instead the algorithm picks a random module i and locally calculates the neighbors adjacent to this module represented by $neigh_i$. As the random walk process traverses along the edges of the *POG* it can only transition to one of the neighbors of i. After identifying all neighbors, the algorithm calculates the proposal and acceptance probabilities from the current module i to each neighbor j where $j \in neigh_i$. As described before in equation 5.5 we use *score* function to calculate the acceptance probability which is proportional to the target distribution.

5.4.3. Uniform sampling

In the case of uniform sampling all modules are equally likely, therefore the target distribution is $1/|N|$ for any module in the output space. $|N|$ represents the total number of modules in the output space. Since every module is equally likely in uniform sampling the *score* function is set to 1.

A random walk in the POG when picking modules uniformly would gravitate towards the degrees of the modules. In other words the modules with high degree in the POG would be sampled more than low degree modules, in fact the steady state distribution is directly proportional to degree of the module [17]. To counter this effect we design the proposal distribution $g(i|j) = 1/d_i$ where d_i represents the degree of module i in the POG. Substituting the values of $g(i|j)$ in equation 5.5 we can obtain the acceptance probability for uniform sampling

$$\alpha(j|i) = Min\left\{1, \frac{d_j}{d_i}\right\} \tag{5.7}$$

5.4.4. Targeted sampling - sampling dense and cohesive modules

To sample cohesive and dense modules we need to introduce a bias in the acceptance probability which will encourage the random walk to move towards more cohesive and dense modules and remain there. We use module score defined in definition 9 to introduce this bias. Recall that by properly selecting the parameters we can adjust the score to favor bigger, denser and more cohesive modules.

We use the same proposal probability as uniform distribution to offset for the bias on the degree of modules. Substituting $score = \Delta$ in in equation 5.5 we can obtain the acceptance probability for targeted sampling

$$\alpha(j|i) = Min\left\{1, \frac{d_j * \Delta_i}{d_i * \Delta_j}\right\} \tag{5.8}$$

Algorithm 6 shows the random walk process in detail. The algorithm starts with an empty map S which will contain all the visited modules and their visit count. Next the algorithm finds a random module u in the POG to begin processing as shown in line number 3. The randomly selected module u, adheres to the input constraints of density and cohesiveness. The algorithm does not create the POG ahead of time, instead it finds a random module by randomly selecting a vertex from G (input graph) and extending it by one of its neighbors in G till it creates a random initial module.

Once a random module is found, the algorithm starts the random walk process. It iterates over all qualified neighbors $neigh_u$ (all neighbors in this set satisfy the density and cohesive constraints) of the random module u and calculates the acceptance probability $accept_prob$ of each neighbor. It then jumps to one of the neighbors if the acceptance probability is greater than a rejection probability. If the algorithm transitions to a new module v then it adds the current module u to the map S and increments it visit count by 1. If the algorithm does not jump to any of the neighbors then it finds another random module as shown in line 18. Continuing this way the algorithm transitions from one module to another till the maximum number of iterations has reached. This usually implies that the random walk process has converged to a steady state distribution and the map S contains the modules in relation to the target distribution. Finally the algorithm returns the N most visited modules from S as requested.

Note that the random module search also has a maximum iterations check beyond which the algorithm gives up and exits, this avoids the runoff problem and guarantees the exit of the algorithm if a random module cannot be located.

Algorithm 6 Pseudo-code for a sampling dense and cohesive modules

Input:
$G = (V, E, f)$: An attributed graph
θ: Density threshold
t: Tolerance threshold in a single subspace
s_{min}: Minimum number of similar attributes
N: Number of output samples desired

Output:
$M < module, int >$: Map of dense cohesive cluster samples and its visit count

1: $\mathcal{S} = \{,\}$
2: $max_iter = maximum_number_of_iterations$
3: $u = find_random_module()$
4: **for** $i \in 1$ to max_iter **do**
5: $transition = false$
6: Let $neigh_u$ be set of neighbors for u
7: **for** $v \in neigh_u$ **do**
8: $accept_prob = score(v) * g(v|u)/score(u) * g(u|v)$
9: $reject_prob = uniform(0, 1)$
10: **if** $accept_prob \geq reject_prob$ **then**
11: $AddOrIncrementVisitCount(\mathcal{S}, u)$
12: $u = v$
13: $transition = true$
14: $break$
15: **end if**
16: **end for**
17: **if** $transition = false$ **then**
18: $u = find_random_module()$
19: **end if**
20: **end for**
21: **return** top N most visited samples from S

5.5. Experiments

In this section we discuss the experiments and results from the execution of our proposed algorithm. To evaluate the sampling algorithm and the modules generated from the sampling algorithm we first start with a small dataset such that its output space can be easily enumerated. We run the sampling algorithm and take sufficiently large number of samples which gives us a sampling distribution. We then compare the sampling distribution with the desired target distribution. Later, we run the proposed algorithm on large real world datasets where it is difficult to enumerate the output space.

We run the sampling algorithm in both uniform sampling mode and also targeted sampling mode. In uniform sampling mode we expect all modules to be sampled with equal probability while in targeted sampling the probabilities are biased for more cohesive and dense modules. Finally, we also analyze the generated modules and show that they are biologically relevant. We have used two real world protein-protein interaction networks and associated attribute data.

1. Yeast: We use the Yeast protein-protein interaction network High Confidence Yeast (YeastHC) [4]. This network has 4008 vertices (genes) and 9857 edges (interactions). Gene profile attribute information correspond to the differential expression value of each gene when exposed to 173 different experiments [29]. Each gene has 173 real attributes.

2. Human: The HPRD network is a database of curated proteomic information pertaining to human proteins. The interaction network contains $21,429$ vertices (genes) and $288,229$ edges (interactions). Attribute data for the HPRD network is a binary dataset. We compiled a dataset of 13 experiments (attributes) from the Gene Expression Omnibus (GEO) database [22].

This algorithm is implemented in C++ and experiments were run on a system with an Intel Xeon (3.3GHz) processor, 8 cores, 16GB RAM and Ubuntu operating system.

5.5.1. Uniform sampling

In uniform sampling, the random walk algorithm samples modules from the POG uniformly. We run this experiment on a smaller curated graph. There are $|N| = 107$ number of cohesive and dense modules in this output space. We ran the sampling algorithm for $|N| * 100$ (10700) iterations. Figure 5.3 shows the result of this experiment. Figure 5.3 (a) shows the visit count of each of the modules. Figure 5.3 (b) plots the histogram of the visit counts where the x axis represents the visit count bins and the y axis shows the number of modules whose visit count falls in that bin. This histogram looks like a normal distribution. The summary statistics of the visit count distribution is shown in Table 5.1 which shows the minimum, maximum, mean and the standard deviation (SD) of the visit counts.

We compared the sampling algorithm against a random sampler. To achieve this we converted the uniform sampling problem to sampling numbers from a set of integers. We created a set of 107 numbers and generated 10700 random samples with replacement. The summary statistics

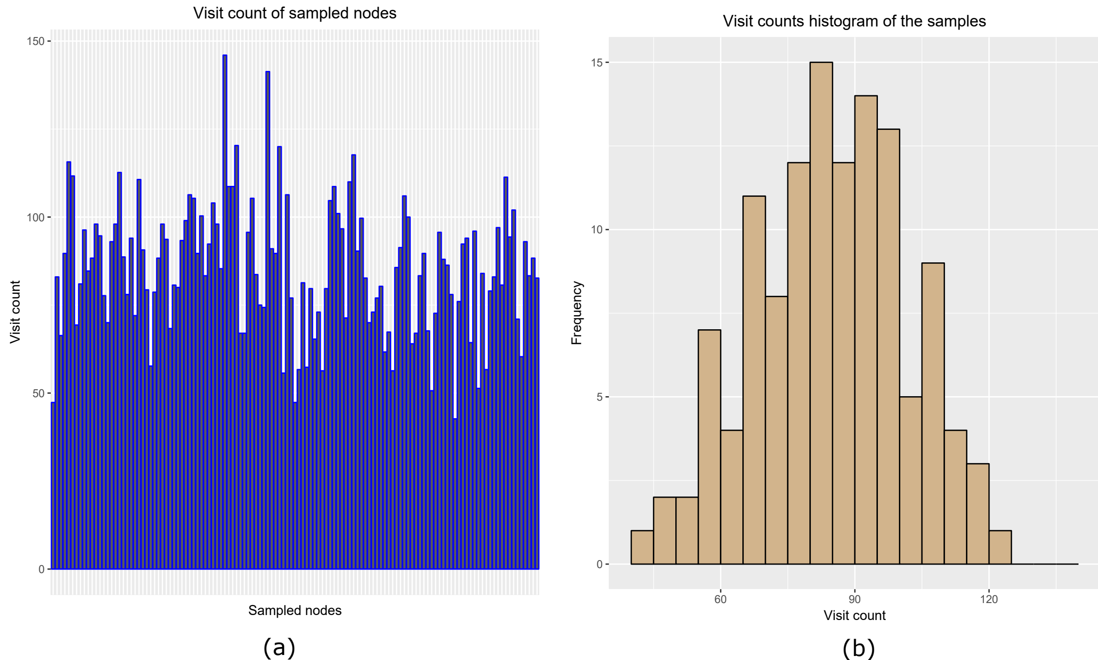

Figure 5.3. Uniform sampling where $|N| = 107$, sampling algorithm ran for 10700 iterations (a) Visit counts of each sample (b) Histogram of visit counts

Table 5.1. Summary statistics of the visit counts of the samples in uniform sampling which includes minimum, maximum, mean and standard deviation of the visit counts.

Algorithm	*Min*	*Max*	*Mean*	*SD*
Uniform sampling	42	146	85	18.6
Random sampling	75	134	100	9.4

from the random sampler is shown in Table 5.1. The summary statistics of the visit count in random sampler looks very similar to sampling algorithm. Both exhibit a normal distribution with the highest visit counts at the mean.

5.5.2. Targeted sampling

For targeted sampling we bias the acceptance probability towards selection of dense and cohesive modules over others. We expect that more dense and cohesive modules will have a higher visit count over other modules. We ran this algorithm on Yeast and Human datasets for varying parameters θ, t and S_{min}.

We first ran our enumeration algorithm on Yeast dataset with strict parameters because we can enumerate all patterns (modules) easily. For each of the experiment we ran our sampling algorithm such that the number of iterations is several times more than the total number of patterns.

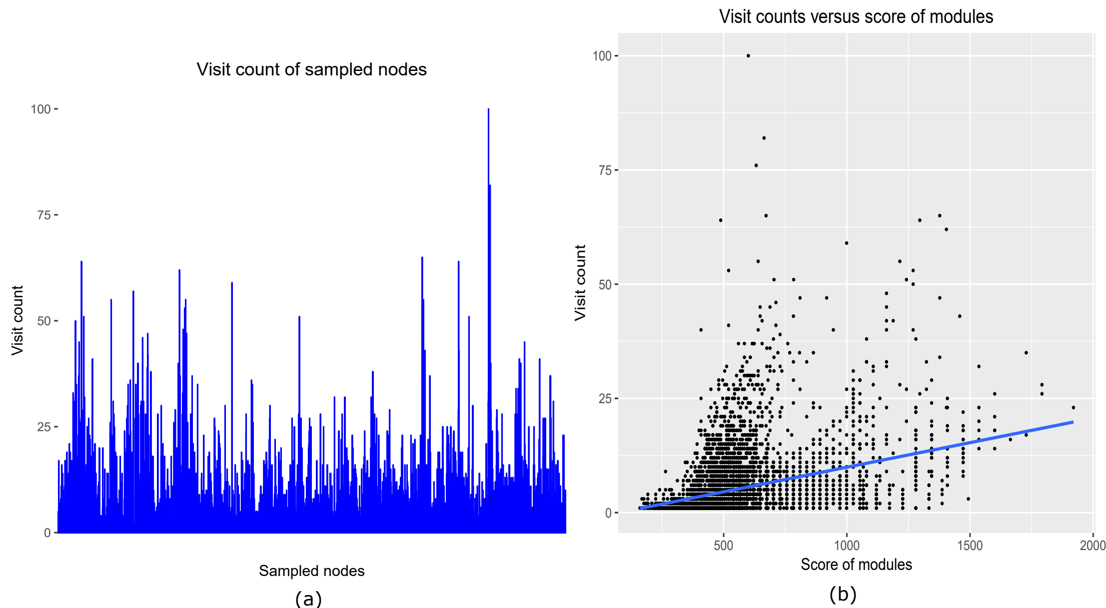

Figure 5.4. Targeted sampling where $|N| = 3819$, sampling algorithm ran for 75000 iterations (a) Visit counts of each sample (b) Scatter plot graph between score of the module and the visit count of each module. The blue line indicates the positive correlation between score of the module and visit count.

Table 5.2. Results of sampling algorithm on Yeast dataset.

θ	t	s_{min}	$\|N\|$	γ	r
0.8	0.2	15	7238	10	0.39
0.8	0.2	20	3819	20	0.39
0.8	0.2	25	1680	30	0.32
0.9	0.2	15	6716	10	0.45
0.9	0.2	20	3819	10	0.39
0.9	0.2	25	3729	30	0.35

By sampling a high number of patterns we ensure that the algorithm gets a fair chance of visiting most patterns and also arrive at a steady state distribution. Table 5.2 shows the results of our experiments on Yeast dataset with multiple parameters where $|N|$ represents the total number of output modules and γ represents a multiplier factor such that the total number of iterations can be calculated by the product of $|N|$ and γ. For e.g., the experiment with parameters $\theta = 0.9$, $t = 0.2$, $S_{min} = 15$ (fourth row in Table 5.2), the total number of output modules $|N|$ is 6716 and γ is 10 and the total number of iterations = 67160 ($|N| * \gamma$). r represents the correlation between the visit count of samples and their Δ scores.

In 67160 iterations, the sampling algorithm visited 5365 out of 6716 modules. 59 out of 6716 modules have high module score such that their scores are in the top 33% of the module score distribution. The sampling algorithm visited 100% of the top scored modules (all 59), while it only visited 60% of the remaining modules. This indicates that the sampling algorithm favors visiting modules with higher score.

The average visit count of all 5365 modules was 7 and the average module score was 53.3. In contrast, the most visited module by the algorithm had a score of 118.8 and was visited 166 times, also the module with the highest score of 212.5 was visited 55 times. The sampling algorithm spent about 32% of the time sampling the top 10% of the most visited modules with an average module score of 87.3. These statistics show that the sampling algorithm favors sampling modules with higher score.

There are 3819 cohesive and dense modules satisfying these parameters $\theta = 0.8$, $t = 0.2$, $S_{min} = 20$ in the Yeast dataset (second row in Table 5.2). We ran the sampling algorithm for 75,000 iterations. Figure 5.4 (a) shows the distribution of the visit count of the samples. As expected there are some modules which are sampled more aggressively than others. We also show that the modules which were sampled more tend to have higher Δ scores, which in turn suggest that cohesive and dense modules were sampled more than others. Figure 5.4 (b) shows a plot between module score and sampling frequency. We can see that module score and sampling frequency are positively correlated which suggests that modules with higher score are sampled more. The column labeled r in Table 5.2 shows the positive correlation for each of the experiments with different parameters.

Next, we ran our sampling algorithm on a large Human dataset. The enumeration algorithms would not be able to complete with such large datasets in a reasonable amount of time. We ran MultiRedcone [32] and our sampling algorithm for for varying amounts of time and compared the output in Table 5.3. We ran both the algorithms for 30, 60 and 90 minutes each and then compared the output modules generated so far. The $|Patterns|$ column shows the number of patterns enumerated by the MultiRedcone algorithm while the $|Samples|$ column shows the number of patterns generated by the sampling algorithm in the same amount of time. For e.g., the experiments with parameters $\theta = 0.5$, $S_{min} = 2$ and $runtime = 30$ (first row in table 5.3), the MultiRedcone algorithm generated over $101,933$ patterns while for the same parameters the sampling algorithm

generated 3,968 patterns. The enumeration algorithm visits every pattern once while the sampling algorithm can visit the same pattern multiple times.

The sampling algorithm beats the enumeration algorithm in the quality of the output modules. For e.g., the maximum score in the patterns enumerated by MultiRedcone algorithm for experiment $\theta = 0.5$, $S_{min} = 2$ and $runtime = 30$ is 22.5 while the maximum score in the samples generated by the sampling algorithm is 40, which shows that the sampling algorithm has found a bigger, denser and more cohesvive module over the MultiRedcone algorithm. Recall that the score of a pattern is a function of density, size and cohesiveness of a pattern.

Table 5.3. Results of sampling algorithm on Human dataset.

θ	s_{min}	$runtime$	$\|Patterns\|$	$\|Samples\|$	$NS_{patterns}$	$NS_{samples}$
0.5	2	30	101,933	3,968	0.130	0.248
0.5	2	60	113,981	6,279	0.179	0.332
0.5	2	90	221,176	13,375	0.218	0.33
0.5	3	30	137,553	13,253	0.352	0.408
0.5	3	60	503,032	26,496	0.360	0.391
0.5	3	90	1,014,937	35,735	0.326	0.390
0.5	4	30	812,084	24,375	0.403	0.434
0.5	4	60	1,621,483	28,013	0.461	0.472
0.5	4	90	2,084,805	27,883	0.428	0.457

Table 5.4. Traversal of enumeration and sampling algorithm.

θ	s_{min}	$runtime$	$MVFS_{enum}$	$MVFS_{samp}$
0.5	2	30	82.94	21.64
0.5	2	60	84.59	16.46
0.5	2	90	91.69	19.37
0.5	3	30	98.51	23.65
0.5	3	60	96.11	22.62
0.5	3	90	94.42	25.30
0.5	4	30	93.07	32.03
0.5	4	60	93.74	32.08
0.5	4	90	90.84	30.94

Table 5.3 also shows the normalized average score of patterns ($NS_{patterns}$) enumerated by MultiRedcone algorithm and sampling algorithm ($NS_{samples}$) respectively. The normalized average score normalizes the score of patterns to a range between 0 to 1. We can see that for every experiment the sampling algorithm has higher normalized score, for e.g., for the experiment $\theta = 0.5$, $S_{min} = 2$ and $runtime = 60$ (second row in Table 5.3) the enumeration algorithm produced $113,981$ patterns with an average score of 0.179 while the sampling algorithm generated $6,279$ patterns with an average score of 0.332.

We further analyzed the traversal pattern of the enumeration and the sampling algorithm. Table 5.4 shows the percentage of time spent by each algorithm in exploring patterns. We calculated the time spent by each algorithm in the top 3 most visited first level sub trees. $MVFS_{enum}$ col shows the percent of time spent by enumeration algorithm in the most visited first level subtrees and $MVFS_{samp}$ shows the percent of time spent by sampling algorithm in most visited first level subtrees. As the sampling algorithm does not generate patterns in any specific order we sorted the patterns by vertexIds to calculate this metric. The enumeration algorithm spends more than 80% of the time exploring patterns under 3 out of $20,000$ sub trees in every experiment. This is expected because the enumeration tree is essentially a depth first search technique and will get stuck enumerating patterns under a large sub tree. In contrast, the most time spent by sampling algorithm in the top 3 most visited sub trees is only 30%. The sampling algorithm randomly jumps from one node to another in the POG. This technique helps the sampling algorithm explore more patterns while the enumeration technique would be restricted to a few sub trees.

In summary, we can claim that the sampling algorithm helps in finding fewer and better modules over the enumeration algorithms, and achieves its initial proposal of generating a reduced set of high quality dense and cohesive modules without enumerating the entire output space.

5.5.2.1. Biological Analysis

To assess the biological significance of the reported cohesive subnetworks, we performed enrichment analysis of these gene sets. If a biological annotation is overrepresented in the genes in a cohesive subnetwork, the subnetwork is marked as enriched. We used the DAVID functional annotation tool for performing the enrichment analysis [41, 42] and assessed the enrichment for the KEGG pathways and the Gene Ontology biological process. Table 5.5 shows the result of enrichment analysis of GO terms, KEGG pathways and gene-disease association (DGN) for various

Table 5.5. Biological enrichment analysis on modules generated by sampling algorithm in human dataset.

θ	s_{min}	$runtime$	$\|samples\|$	$GO\%$	$KEGG\%$	$DGN\%$
0.5	4	30	6,250	100	100	100
0.5	4	60	12,749	100	99.1	99.2
0.5	4	90	17,669	100	98.6	99.1
0.5	3	30	3,053	100	100	100
0.5	3	60	5,718	100	99.7	100
0.5	3	90	8,028	100	99.9	100
0.5	2	30	869	100	100	100
0.5	2	60	1,995	99.9	92.5	100
0.5	2	90	3,022	99.4	100	100

sampling experiments. We can see that all of the modules generated by the sampling algorithm are biologically very significant. For e.g., for the experiment where $\theta = 0.5$, $S_{min} = 4$ and $runtime = 30$ (first row in Table 5.5) the GO, KEGG and DGN enrichment is at 100% for sampling.

6. CONCLUSION AND FUTURE WORK

6.1. Conclusion

In this dissertation we proposed new algorithms for mining cohesive sub networks from graphs with node attributes. We will summarize our findings and provide a future outlook in this chapter.

Real world graphs always contain wealth of related information about nodes and edges. In social networks we have basic user information such as age, city and interests which can be modeled as node attributes. Communities which show similarity in node attributes in addition to their network structure are called cohesive communities and have higher fidelity in their application. This dissertation addresses the problem of efficiently mining cohesive communities from graphs.

6.1.1. Mining dense and cohesive sub networks

We started with the problem of mining dense and cohesive sub graphs (modules) from graphs. We proposed a RedCone [34] algorithm which builds an enumeration tree and lists all qualifying sub graphs. The enumeration tree can contain exponential number of nodes. RedCone utilizes a reverse search enumeration technique to efficiently reduce the search space in this enumeration tree.

We ran the RedCone algorithm on real world protein protein interaction networks and compared our results against the state of the art algorithms GAMer and DECOB. We noted that RedCone is multiple times faster than both GAMer and DECOB for various experiments. We extended RedCone algorithm by implementing a parallel approach [32] to RedCone utilizing multiple threads. We compared our results against the single threaded execution and noticed multiple times of improvement in runtime. The speedup was bound by the total number of cores in the cpu. We also analyzed the biological significance of the reported modules. We tested our results against a known biological database DAVID and found that our result modules were always enriched in GO Terms (100%) and showed a high percent in KEGG pathways (80%).

For very relaxed constraints RedCone will generate millions of output modules. These modules also have a high overlap among themselves. Subsequent analysis often requires a small set of output modules which are fairly separated from each other, yet representative of the entire

output modules. To this problem, we implemented a K-Medoids based summarization technique which generates a small set of representative modules.

6.1.2. Mining cohesive sub networks

Cohesive modules need not be dense, in fact density constraint can sometimes force an algorithm to miss modules which are cohesive in high number of attributes as shown in example figure 4.1. With this limitation in mind, we proposed a MinCone [35] algorithm to detect cohesive only modules without density constraint. MinCone is also an enumeration based approach which utilizes reverse search technique to effectively reduce the search space. We compared our MinCone algorithm against a baseline approach to mine cohesive modules. Mincone algorithm is several time faster than the baseline approach.

We implemented a parallel approach algorithm [33] for MinCone utilizing multiple threads. We achieved a maximum speedup of 7 by running 8 threads for parallel MinCone . The speedup was bound by the number of cores in the cpu which was 8 in our case.

6.1.3. Sampling dense and cohesive sub networks

Many real world graphs are huge with millions and billions of nodes in the graph. Most of the enumeration algorithms discussed here would not be able to handle graphs of this scale. The biggest problem of the enumeration techniques is that they enumerate an output space which is exponential to the number of nodes in the graph. With hundreds of millions of nodes in a graph the task of enumerating the entire output space is enormous and very time consuming. In post processing we would often require a small set of representative modules for analyses. The second task of reducing this output space of modules into a small representative set is even more time consuming.

To combat this dual problem we proposed a sampling algorithm which promises to return fewer and higher quality modules without enumerating the entire output space. The sampling algorithm is random walk technique on the partial order graph of the input graph. The POG is a graph where each node is a subgraph of the input graph as shown in figure 5.2. The random walk transitions from one node to another in this POG and tries to bias jumps to a higher quality node. The quality score quantifies the density, size and cohesiveness of a module. A high score module is denser, bigger and is cohesive in more number of attributes than others. The algorithm outputs the top k most visited modules, all of which have a high module score.

We ran the sampling algorithm on protein protein interaction networks and showed that the sampling algorithm visits the higher quality nodes in POG far more than others. Recall the nodes in POG are actual subgraphs of the input graph. We ran the sampling algorithms against RedCone for 30, 60 and 90 minutes and compared the results produced by both the algorithms respectively. We noticed that sampling algorithm generated a fraction of modules generated by RedCone and yet the average quality of the sampled modules was 50% more than that of RedCone.

6.2. Future work

This chapter explores some of the areas in which this current research can be extended or applied in the future.

6.2.1. Cohesive communities with noisy node attributes

The node attributes in the clustering models we had discussed so far were absolute or noiseless. In reality however this is often not the case, for example, people usually have incomplete profile information in social networks or the gene differential expression values in protein protein interaction networks the can contain noise. One possible extension of the current research is to detect cohesive communities in a graph even if the node attributes are incomplete or contains noise. The new definition of a cohesive community would contain a probabilistic model which can score the cohesiveness of a community.

Fortunately, in the domain of computer vision, the problem of subspace clustering with noisy data has already been researched recently [74, 73]. In this extension we would like to marry the techniques of subspace clustering with noise to graph mining.

6.2.2. Cohesive communities with edge attributes

Mining dense and cohesive networks with attributes on the edges is another important area for further research and study. This is a logical extension to both RedCone and MinCone which currently use node attributes. Also edges provide a much richer characterization of community behavior, because the content models the characteristics of pairwise interactions rather than individual entities [62].

Figure 6.1 shows a sample social network with the edge attributes. This graph shows two cohesive communities between vertices (3,4,5,6) and (0,2,3). The vertex 3 is a member of two communities. This reflects the fact that a given individual may have different facets to their life, which are revealed only in their interactions with other individuals. The individual represented by

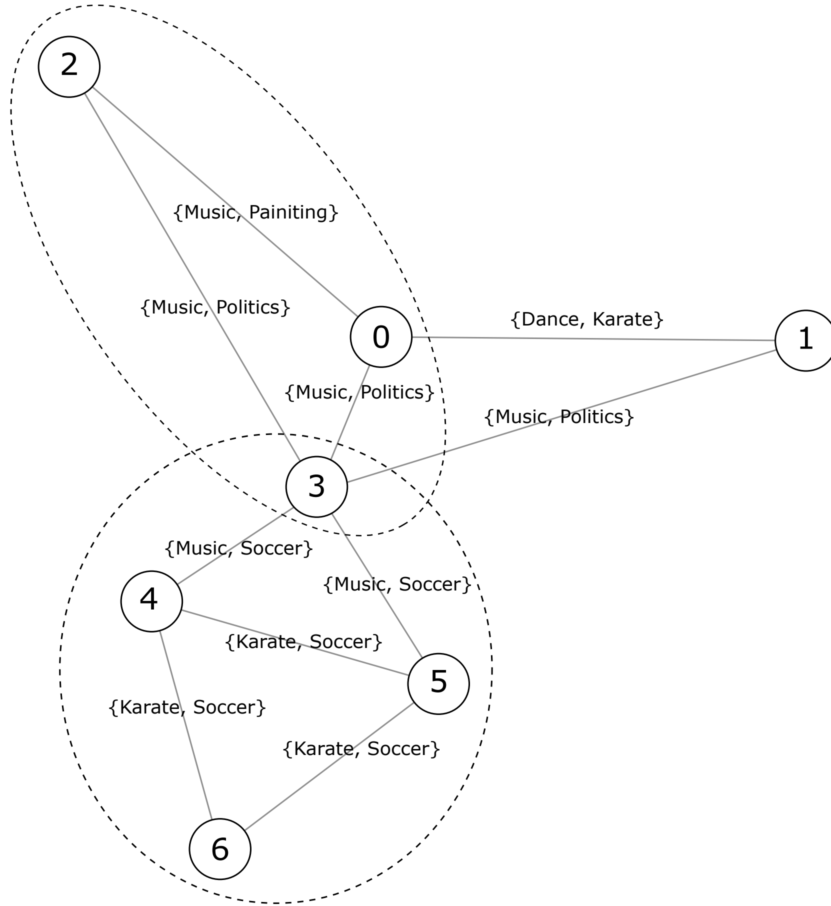

Figure 6.1. Graph with edge attributes and two communities.

vertex 3 in this social graph displays attributes of {soccer}, {music} and {politics} however they are only exhibited in their interactions with other individuals.

6.2.3. Cohesive communities with ranked edge interaction data

The edge attributes discussed in the above section contribute equally in forming a cohesive community, i.e., there is no relative priority or rank among the edge attributes while forming communities. In social networks, there are often various types of interactions among users and these interaction types emphasize the interest of the user in a topic. For example, two users engaging in conversations related to music (posting replies to each other) are more likely to be members of a community on music than someone who has occasionally broadcasted posts on music (for example, during a music festival or concert). This kind of relative importance between user interactions is not captured in the edge attributes. In addition a user can have interaction with other users

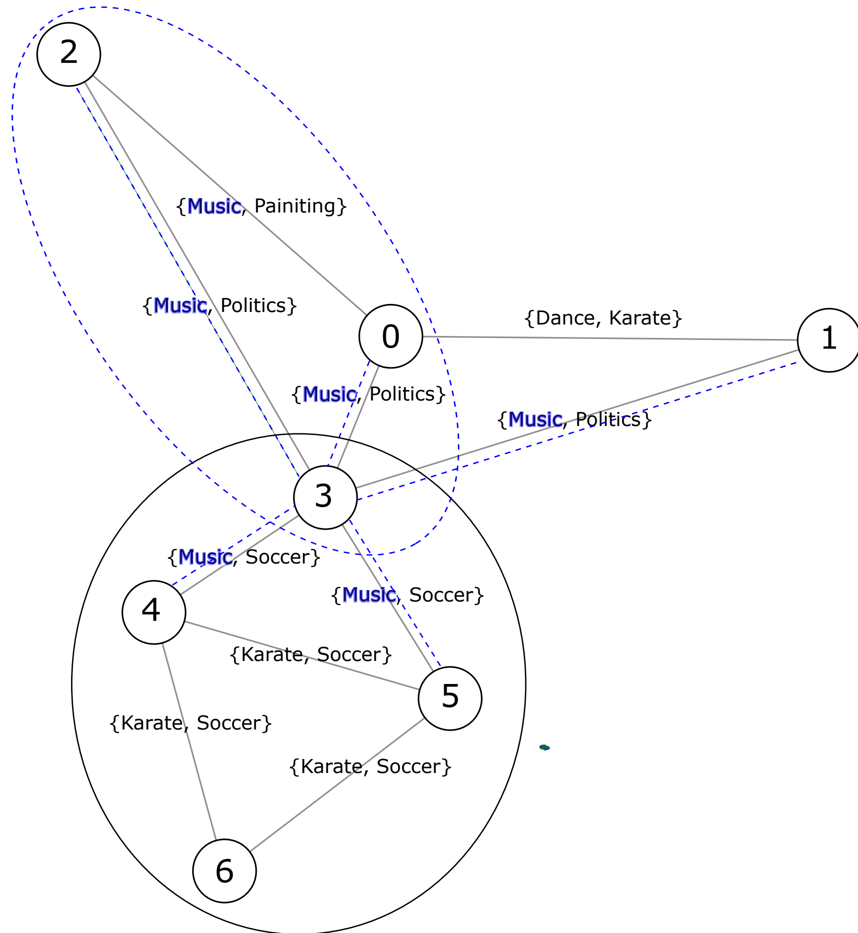

Figure 6.2. Graph with ranked edge attributes. The dashed ellipse shows a community formed by including the topics from a broadcast message.

related to other topics, i.e., a user can belong to multiple communities and these communities can be related to multiple topics.

Figure 6.2 shows a sample social network graph where users belong to multiple communities based on topics decoded from their interaction data. The graph shows solid and dashed edges. The dashed edges represent a broadcast post from user 3 on the topic of music which was sent to all of the friends of user 3 (perhaps because of a music concert that user 3 attended). The solid lines indicate direct conversation between users. The dashed edges can be characterized as weak edges while the solid edges are strong edges revealing the true interest of the user. The community formed between users (0,2,3) is solely on the premise that each of these users have inherent interest in the topics identified in their interaction. Since the community formed by (0,2,3) is cohesive in {music} attribute, its expected that each of the users to have interest in music. However as noted above,

broadcast posts are one of the occasional messages which may or may not be accurately identify the users interests. If we ignore the broadcast message (as identified by the dashed edges) then the cohesive community formed by users (0,2,3) would not exist. Instead of completely ignoring a broadcast message, we can also model the relative importance of user's interest as reflected in their interaction. We will explore ways to integrate edge attributes with importance or priority.

6.2.4. Sampling cohesive communities with multi relational edge attribute data

In real world graphs there are more than one kind of relationship among the nodes in a network. For example, in DBLP dataset (http://dblp.uni-trier.de/xml/) the nodes represent authors. The edges can represent co-authorship when two authors have collaborated for a paper or the edges can represent citation when two authors have cited each other. So between two nodes there are two edges one representing co-authorship while the other representing citation. Furthermore, the edges can have attributes such as year of the paper or total number of citations. The graphs which have the same vertices but different topology are called multi relational or multi layer graphs. An example of this graph is shown in Figure 6.3

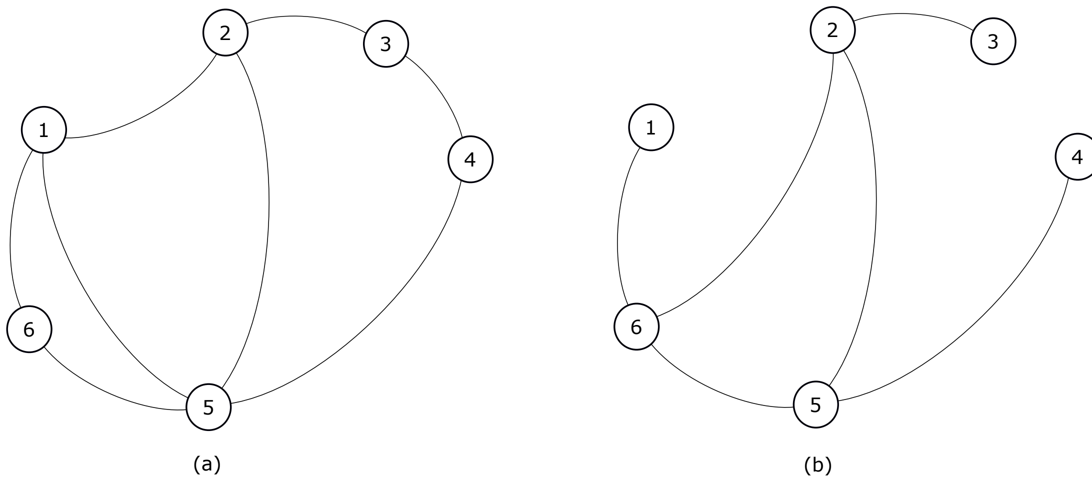

Figure 6.3. A graph containing multi relational edges, (a) An example of co-authorship network where two authors co-authored a paper and (b) an example of citation network where authors cited each other. In this example the community formed by $\{1, 2, 5, 6\}$ is dense in both co-authorship and citation network with a minimum density of 0.6, assuming they are cohesive in their edge attributes

Mining communities in multi relational edge attributed networks has been studied previously in [10, 59, 7, 76]. [10] discusses hidden features in a one dimensional edge network. In [76] the multi-relational input network is converted into a one dimensional network by unionizing the edges

across multiple networks into one network and labeling the edges with attributes. The authors in [59, 7] discuss an enumeration tree strategy and try to mine cohesive dense communities on a subspace of multi relational edges.

The mining algorithms in a multi relational graph needs to be highly scalable because a multi relational real world graph can grow very quickly by either adding new vertices or new edges in the input graph. To address the issues of scale, we recommend a sampling technique which samples the desired sub graphs instead of enumerating all, and then finding a few representative sub graphs. Similar to the sampling approach for node attributed graphs in chapter 5, the sampling technique uses a random walk algorithm.

In Figure 6.4 (a) we have a sample DBLP graph with three authors and two types of edges representing the co-authorship and citation networks. Figure 6.4 (b) shows the POG for co-authorship network and Figure 6.4 (c) shows the POG for citation network. We can also have another POG for both co-authorship and citation network together which is not shown here for simplicity. The random walk discussed previously was transitioning from one search node in the *POG* to another. As the different layers in a multi relational graph have different edge sets (topology) the *POG* for each layer would be different. For example, in Figure 6.4 (a) the subgraph $\{A, C\}$ only exists in co-authorship layer which means that the *POG* for citation layer will not contain the node $\{A, C\}$ and will be missed out if the random walk is restricted to *POG* of citation layer.

Unlike the random walk algorithm discussed previously, the random walk for multi relational graph needs to jump from one *POG* to another in addition to traversing within a *POG*. The random walk algorithm has three different transitions.

- Transitions to another search node in the same *POG* by adding a new vertex to the search node. Jumping from $\{A, B\}$ to $\{A, B, C\}$ in the figure 6.4 (b)

- Transitions to another search node in a different *POG* by selecting a new edge layer or a combination of edge layers. Jumping from $\{A, B\}$ in Figure 6.4 (b) to $\{A, B\}$ in the Figure 6.4 (c)

- Transitions to another search node in a different *POG* by adding both a new vertex and selecting a new edge layer or a combination of edge layers. Jumping from $\{A\}$ in Figure 6.4 (c) to $\{A, C\}$ in the Figure 6.4 (b)

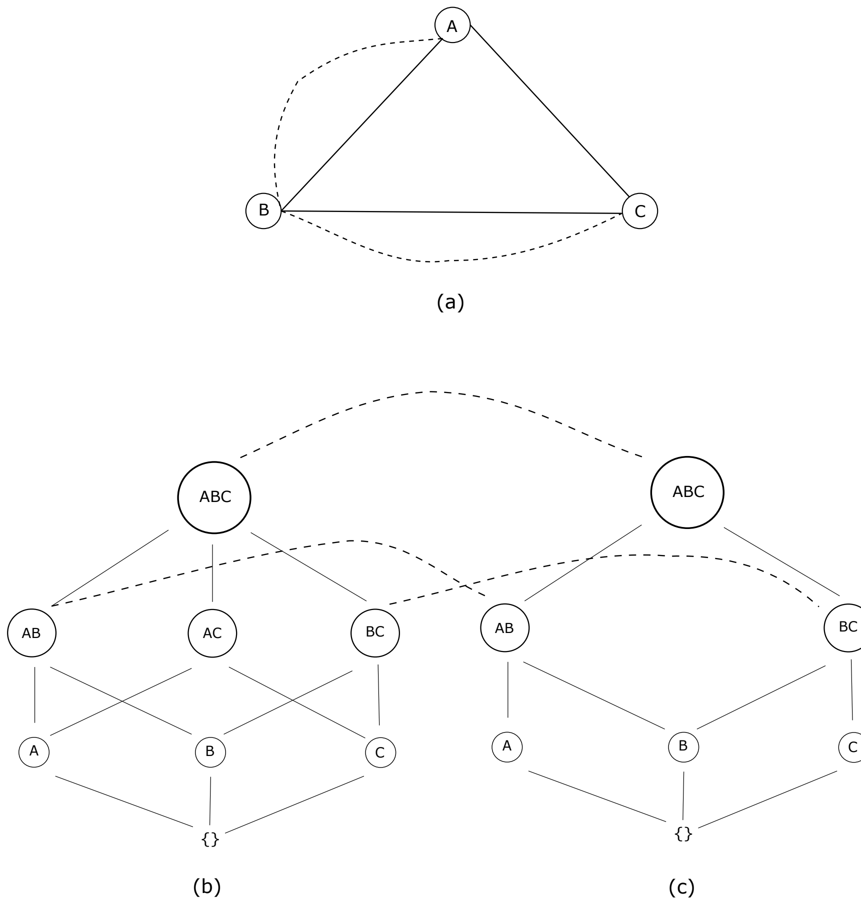

Figure 6.4. DBLP graph. (a) An example DBLP graph with multiple relational edges, solid edges are co-authorship and dashed edges are citation edges. (b) Partial order graph for input graph in (a) with co-authorship topology. (c) Partial order graph for input graph in (a) with citation topology. Notice that subgraph $\{A, C\}$ does not exist in POG for citation topology.

Recall that a *POG* is created locally around a given search node, so we don't need to build different *POG*'s ahead of time. At each search node, the random walk algorithm computes all possible neighbors both within and outside of *POG*. After enumerating the neighbors we calculate the transition probabilities to each of these neighbors. The random walk transitions randomly to one of its neighbors and repeats the process. If at any point there are no good neighbors, the random walk will transition to random search node and recomputes its neighbors. Finally when the walk converges to a stationary state we take the top k search nodes with the highest visit count.

6.2.5. Parallel approach to sampling

Chapter 5 explored the idea of sampling dense and cohesive subgraphs instead of traditional mining. One future extension to this work is creating a parallel implementation of sampling utilizing

multiple threads. The basic idea is to spawn n threads and to have them start at n random nodes in the POG. These threads will start their random walk as usual, exploring neighbors and randomly jumping to any other node in POG when required. At the end of the total number of desired iterations (or total runtime) each of the thread will output its k most visited patterns. A final set of output patterns can be generated by creating a union set of the individual patterns generated by each thread.

Recall the every node in POG is a dense and cohesive subgraph of G. For relaxed parameters or constraints the POG can be dense as more subgraphs of G will qualify. When the POG is dense having multiple threads performing random walk in the different areas of POG will greatly improve performance. The final output will be the best of the patterns as reported by each thread.

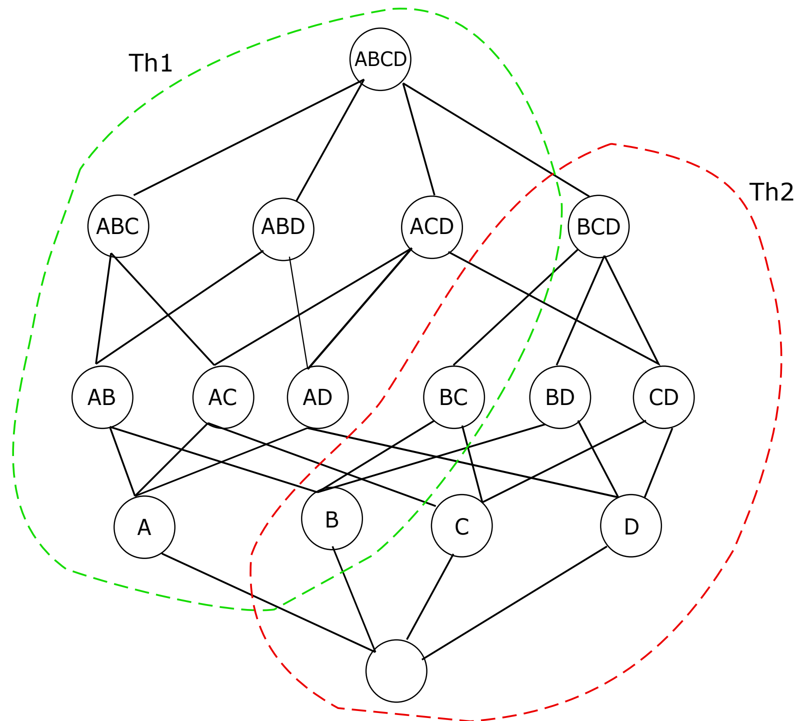

Figure 6.5. Multi threaded approach to sampling in POG with two threads (Th1 and Th2) performing parallel random walk. Each thread is restricted to its respective area highlighted by color.

There is a possibility that these n threads will start visiting the same patterns. In order to prevent the n threads from visiting the same patterns, we can force the random pattern generation to be localized to a specific area in the POG. In a random walk the algorithm frequently jumps

to a random node in the POG if none of the neighbors of the current node in POG satisfies the constraints. By restricting the random jump to a certain area in the POG we can force the threads to remain local to their respective areas in the POG. This will not necessarily guarantee that the threads will be visiting mutually exclusive areas but to a large extent the threads will remain localized to their areas. Figure 6.5 shows a POG with two threads performing a random walk in parallel. Each thread has its own localized area which can overlap with another thread.

In an another approach towards a parallel implementation, we spawn n threads and each of threads will start the random walk over the entire POG starting from n random yet distinct nodes. Each of the threads however will have a different *score* function for a module. The employed scoring function has significant impact on the output patterns. For example, one *score* function can be tuned to find larger sub graphs and another can be tuned to find denser sub graphs and so on. By having different *score* function for each thread we will force each thread to output different modules. At the end of the desired number of iterations the algorithm will union the top k most visited modules by each thread. In this way we can get the best of patterns as output by each thread which are all optimized to find different types of modules.

REFERENCES

[1] Mohammad Al Hasan and Mohammed J Zaki. Output space sampling for graph patterns. *Proceedings of the VLDB Endowment*, 2(1):730–741, 2009.

[2] Sitaram Asur and Bernardo A Huberman. Predicting the future with social media. In *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2010 IEEE/WIC/ACM International Conference on*, volume 1, pages 492–499. IEEE, 2010.

[3] David Avis and Komei Fukuda. Reverse search for enumeration. *Discrete Applied Mathematics*, 65(1):21–46, 1996.

[4] Nizar N Batada, Teresa Reguly, Ashton Breitkreutz, Lorrie Boucher, Bobby-Joe Breitkreutz, Laurence D Hurst, and Mike Tyers. Still stratus not altocumulus: further evidence against the date/party hub distinction. *PLoS biology*, 5(6):e154, 2007.

[5] Andras Benczur and David R Karger. Randomized approximation schemes for cuts and flows in capacitated graphs. *arXiv preprint cs/0207078*, 2002.

[6] Zhong-Rui Bian, Juan Yin, Wen Sun, and Dian-Jie Lin. Microarray and network-based identification of functional modules and pathways of active tuberculosis. *Microbial Pathogenesis*, 2017.

[7] Brigitte Boden, Stephan Günnemann, Holger Hoffmann, and Thomas Seidl. Mimag: mining coherent subgraphs in multi-layer graphs with edge labels. *Knowledge and Information Systems*, 50(2):417–446, 2017.

[8] Immanuel M Bomze, Marco Budinich, Panos M Pardalos, and Marcello Pelillo. The maximum clique problem. In *Handbook of combinatorial optimization*, pages 1–74. Springer, 1999.

[9] Tyler Brazier. Mining Representative Cohesive Dense Subgraphs. Master's thesis, North Dakota State University, 2014.

[10] Deng Cai, Zheng Shao, Xiaofei He, Xifeng Yan, and Jiawei Han. Community mining from multi-relational networks. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 445–452. Springer, 2005.

[11] Andrew Chatr-aryamontri, Bobby-Joe Breitkreutz, Sven Heinicke, Lorrie Boucher, Andrew Winter, Chris Stark, Julie Nixon, Lindsay Ramage, Nadine Kolas, Lara O'Donnell, et al. The biogrid interaction database: 2013 update. *Nucleic acids research*, 41(D1):D816–D823, 2013.

[12] Andrew Chatr-aryamontri, Rose Oughtred, Lorrie Boucher, Jennifer Rust, Christie Chang, Nadine K Kolas, Lara O'Donnell, Sara Oster, Chandra Theesfeld, Adnane Sellam, et al. The biogrid interaction database: 2017 update. *Nucleic Acids Research*, page gkw1102, 2016.

[13] Wei Chen, Chi Wang, and Yajun Wang. Scalable influence maximization for prevalent viral marketing in large-scale social networks. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1029–1038. ACM, 2010.

[14] Eunjoon Cho, Seth A Myers, and Jure Leskovec. Friendship and mobility: user movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1082–1090. ACM, 2011.

[15] Salim A Chowdhury, Rod K Nibbe, Mark R Chance, and Mehmet Koyutürk. Subnetwork state functions define dysregulated subnetworks in cancer. *Journal of Computational Biology*, 18(3):263–281, 2011.

[16] Han-Yu Chuang, Eunjung Lee, Yu-Tsueng Liu, Doheon Lee, and Trey Ideker. Network-based classification of breast cancer metastasis. *Molecular systems biology*, 3(1), 2007.

[17] Fan RK Chung. *Spectral graph theory*, volume 92. American Mathematical Soc., 1997.

[18] Recep Colak, Flavia Moser, Jeffrey Shih-Chieh Chu, Alexander Schönhuth, Nansheng Chen, and Martin Ester. Module discovery by exhaustive search for densely connected, co-expressed regions in biomolecular interaction networks. *PloS one*, 5(10):e13348, 2010.

[19] Javier De Las Rivas and Celia Fontanillo. Protein–protein interactions essentials: key concepts to building and analyzing interactome networks. *PLoS Comput Biol*, 6(6):e1000807, 2010.

[20] William E Donath and Alan J Hoffman. Lower bounds for the partitioning of graphs. *IBM Journal of Research and Development*, 17(5):420–425, 1973.

[21] Janusz Dutkowski and Trey Ideker. Protein networks as logic functions in development and cancer. *PLoS computational biology*, 7(9):e1002180, 2011.

[22] Lash AE Edgar R, Domrachev M. Gene Expression Omnibus: NCBI gene expression and hybridization array data repository. *Nucleic Acids Res.*, 30(1):207–2010, 2002.

[23] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 251–262. ACM, 1999.

[24] Miroslav Fiedler. Algebraic connectivity of graphs. *Czechoslovak Mathematical Journal*, 23 (2):298–305, 1973.

[25] Andrew Fiore and Jeff Heer. UC Berkeley Enron email analysis. `http://bailando.sims.berkeley.edu/enron_email.html`, 2005.

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

[27] Linton Freeman. The development of social network analysis. *A Study in the Sociology of Science*, 2004.

[28] Michael R Garey and David S Johnson. Computer and intractability. *A Guide to the Theory of NP-Completeness*, 1979.

[29] Audrey P Gasch, Paul T Spellman, Camilla M Kao, Orna Carmel-Harel, Michael B Eisen, Gisela Storz, David Botstein, and Patrick O Brown. Genomic expression programs in the response of yeast cells to environmental changes. *Molecular biology of the cell*, 11(12):4241–4257, 2000.

[30] Elisabeth Georgii, Sabine Dietmann, Takeaki Uno, Philipp Pagel, and Koji Tsuda. Enumeration of condition-dependent dense modules in protein interaction networks. *Bioinformatics*, 25(7):933–940, 2009.

[31] Leo A Goodman. Snowball sampling. *The annals of mathematical statistics*, pages 148–170, 1961.

[32] Aditya Goparaju and Saeed Salem. A multithreaded algorithm for mining maximal cohesive dense modules from interaction networks with gene profiles. In *Proceedings of the 7th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, pages 671–678. ACM, 2016.

[33] Aditya Goparaju and Saeed Salem. A parallel algorithm for mining maximal cohesive subgraphs. In *9th International Conference on Bioinformatics and Computational Biology (BICOB)*. ISCA, 2017.

[34] Aditya Goparaju, Tyler Brazier, and Saeed Salem. Mining representative maximal dense cohesive subnetworks. *Network Modeling Analysis in Health Informatics and Bioinformatics*, 4(1):1–11, 2015.

[35] Aditya Goparaju, Bassam Qormosh, and Saeed Salem. Mining maximal subnetworks from interaction network with node attributes. In *Bioinformatics and Biomedicine (BIBM), 2015 IEEE International Conference on*, pages 1630–1635. IEEE, 2015.

[36] Stephan Gunnemann, Ines Farber, Brigitte Boden, and Thomas Seidl. Subspace clustering meets dense subgraph mining: A synthesis of two paradigms. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*, pages 845–850. IEEE, 2010.

[37] Jing-Dong J Han, Nicolas Bertin, Tong Hao, Debra S Goldberg, Gabriel F Berriz, Lan V Zhang, Denis Dupuy, Albertha JM Walhout, Michael E Cusick, Frederick P Roth, et al. Evidence for dynamically organized modularity in the yeast protein–protein interaction network. *Nature*, 430(6995):88–93, 2004.

[38] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. The elements of statistical learning. 2001. *NY Springer*, 2001.

[39] Qiaona Hong, Sunghun Kim, S. C. Cheung, and Christian Bird. Understanding a developer social network and its evolution. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*. IEEE, September 2011.

[40] Pili Hu and Wing Cheong Lau. A survey and taxonomy of graph sampling. *arXiv preprint arXiv:1308.5865*, 2013.

[41] Da Wei Huang, Brad T. Sherman, and Richard A. Lempicki. Systematic and integrative analysis of large gene lists using david bioinformatics resources. *Nature Protoc.*, 4(1):44–57, 2009.

[42] Da Wei Huang, Brad T. Sherman, and Richard A. Lempicki. Bioinformatics enrichment tools: paths toward the comprehensive functional analysis of large gene lists. *Nucleic Acids Res.*, 37 (1):1–13, 2009.

[43] Wei Jiang, Ramkrishna Mitra, Chen-Ching Lin, Quan Wang, Feixiong Cheng, and Zhongming Zhao. Systematic dissection of dysregulated transcription factor–mirna feed-forward loops across tumor types. *Briefings in bioinformatics*, page bbv107, 2015.

[44] Ruoming Jin, Scott Mccallen, C Liu, Y Xiang, E Almaas, and XH Zhou. Identify dynamic network modules with temporal and spatial constraints. In *Pacific Symposium on Biocomputing*, 2009.

[45] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell system technical journal*, 49(2):291–307, 1970.

[46] Bryan Klimt and Yiming Yang. The enron corpus: A new dataset for email classification research. In *Machine learning: ECML 2004*, pages 217–226. Springer, 2004.

[47] Vaishnavi Krishnamurthy, Michalis Faloutsos, Marek Chrobak, Li Lao, J-H Cui, and Allon G Percus. Reducing large internet topologies for faster simulations. In *International Conference on Research in Networking*, pages 328–341. Springer, 2005.

[48] Maciej Kurant, Carter T Butts, and Athina Markopoulou. Graph size estimation. *arXiv preprint arXiv:1210.0460*, 2012.

[49] Maciej Kurant, Minas Gjoka, Yan Wang, Zack W Almquist, Carter T Butts, and Athina Markopoulou. Coarse-grained topology estimation via graph sampling. In *Proceedings of the 2012 ACM workshop on Workshop on online social networks*, pages 25–30. ACM, 2012.

[50] Benjamin Lehne and Thomas Schlitt. Protein-protein interaction databases: keeping up with growing interactomes. *Human genomics*, 3(3):291, 2009.

[51] Jure Leskovec and Christos Faloutsos. Sampling from large graphs. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 631–636. ACM, 2006.

[52] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 177–187. ACM, 2005.

[53] Hongyan Li, Xiaonan Zhao, Jing Wang, Minru Zong, and Hailing Yang. Bioinformatics analysis of gene expression profile data to screen key genes involved in pulmonary sarcoidosis. *Gene*, 596:98–104, 2017.

[54] Guimei Liu and Limsoon Wong. Effective pruning techniques for mining quasi-cliques. In *Machine Learning and Knowledge Discovery in Databases*, pages 33–49. Springer, 2008.

[55] Heikki Mannila and Hannu Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data mining and knowledge discovery*, 1(3):241–258, 1997.

[56] Miller McPherson, Lynn Smith-Lovin, and James M Cook. Birds of a feather: Homophily in social networks. *Annual review of sociology*, pages 415–444, 2001.

[57] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[58] Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[59] Jian Pei, Daxin Jiang, and Aidong Zhang. On mining cross-graph quasi-cliques. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 228–238. ACM, 2005.

[60] Juan F Poyatos and Laurence D Hurst. How biologically relevant are interaction-based modules in protein networks? *Genome biology*, 5(11):R93, 2004.

[61] TS Keshava Prasad, Renu Goel, Kumaran Kandasamy, Shivakumar Keerthikumar, Sameer Kumar, Suresh Mathivanan, Deepthi Telikicherla, Rajesh Raju, Beema Shafreen, Abhilash Venugopal, et al. Human protein reference database—2009 update. *Nucleic acids research*, 37 (suppl 1):D767–D772, 2009.

[62] G-J Qi, Charu C Aggarwal, and Thomas Huang. Community detection with edge content in social media networks. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 534–545. IEEE, 2012.

[63] Davood Rafiei. Effectively visualizing large networks through sampling. In *Visualization, 2005. VIS 05. IEEE*, pages 375–382. IEEE, 2005.

[64] Alexander W Rives and Timothy Galitski. Modular organization of cellular networks. *Proceedings of the National Academy of Sciences*, 100(3):1128–1133, 2003.

[65] Reuven Y Rubinstein and Dirk P Kroese. *Simulation and the Monte Carlo method*. John Wiley & Sons, 2016.

[66] Matthew J Salganik and Douglas D Heckathorn. Sampling and estimation in hidden populations using respondent-driven sampling. *Sociological methodology*, 34(1):193–240, 2004.

[67] Motoki Shiga, Ichigaku Takigawa, and Hiroshi Mamitsuka. A spectral clustering approach to optimally combining numericalvectors with a modular network. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 647–656. ACM, 2007.

[68] Stefan Stieglitz and Linh Dang-Xuan. Emotions and information diffusion in social media—sentiment of microblogs and sharing behavior. *Journal of management information systems*, 29(4):217–248, 2013.

[69] Chenhao Tan, Lillian Lee, Jie Tang, Long Jiang, Ming Zhou, and Ping Li. User-level sentiment analysis incorporating social networks. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1397–1405. ACM, 2011.

[70] Amy Hin Yan Tong, Becky Drees, Giuliano Nardelli, Gary D Bader, Barbara Brannetti, Luisa Castagnoli, Marie Evangelista, Silvia Ferracuti, Bryce Nelson, Serena Paoluzi, et al. A combined experimental and computational strategy to define protein interaction networks for peptide recognition modules. *Science*, 295(5553):321–324, 2002.

[71] Takeaki Uno. An efficient algorithm for solving pseudo clique enumeration problem. *Algorithmica*, 56(1):3–16, January 2010.

[72] Stijn Marinus Van Dongen. Graph clustering by flow simulation. 2001.

[73] Yining Wang, Yu-Xiang Wang, and Aarti Singh. Graph connectivity in noisy sparse subspace clustering. In *Artificial Intelligence and Statistics*, pages 538–546, 2016.

[74] Yu-Xiang Wang and Huan Xu. Noisy sparse subspace clustering. *The Journal of Machine Learning Research*, 17(1):320–360, 2016.

[75] Stanley Wasserman. *Social network analysis: Methods and applications*, volume 8. Cambridge university press, 1994.

[76] Zhiang Wu, Wenpeng Yin, Jie Cao, Guandong Xu, and Alfredo Cuzzocrea. Community detection in multi-relational social networks. In *International Conference on Web Information Systems Engineering*, pages 43–56. Springer, 2013.

[77] Jaewon Yang, Julian McAuley, and Jure Leskovec. Community detection in networks with node attributes. In *Data Mining (ICDM), 2013 IEEE 13th International Conference on*, pages 1151–1156. IEEE, 2013.

[78] Jun-Bo Yang, Rong Luo, Yan Yan, and Yan Chen. Differential pathway network analysis used to identify key pathways associated with pediatric pneumonia. *Microbial Pathogenesis*, 101: 50–55, 2016.

[79] Shuang-Hong Yang, Bo Long, Alex Smola, Narayanan Sadagopan, Zhaohui Zheng, and Hongyuan Zha. Like like alike: joint friendship and interest propagation in social networks. In *Proceedings of the 20th international conference on World wide web*, pages 537–546. ACM, 2011.

[80] Mao Ye, Peifeng Yin, and Wang-Chien Lee. Location recommendation for location-based social networks. In *Proceedings of the 18th SIGSPATIAL international conference on advances in geographic information systems*, pages 458–461. ACM, 2010.

[81] Zhiping Zeng, Jianyong Wang, Lizhu Zhou, and George Karypis. Coherent closed quasi-clique discovery from large dense graph databases. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 797–802. ACM, 2006.