Comparative Study of Community Detection Algorithms in Social Networks

Andreas Kalaitzakis

Thesis submitted in Partial Fulfillment of the Requirements for the

Degree of Information Systems & Multimedia Engineering

Technological Educational Institute of Crete
School of Technological Applications
Department of Applied Informatics and Multimedia
Estavromenos, P.O. Box 1939, Heraklion, GR-71404, Greece

Thesis Advisor: Prof. *Paraskevi Fragopoulou*

TECHNOLOGICAL EDUCATIONAL INSTITUTE OF CRETE
DEPARTMENT OF APPLIED INFORMATICS AND MULTIMEDIA

Comparative Study of Community Detection Algorithms in Social Networks

Thesis submitted by

Andreas Kalaitzakis

in partial fulfillment of the requirements for the
Degree of Information Systems & Multimedia Engineering

THESIS APPROVAL

Author: ________________________________
Andreas Kalaitzakis

Committee approvals: ________________________________
Paraskevi Fragopoulou
Professor, Thesis Supervisor

____
Harris Papadakis
PhD, Thesis Co-Supervisor

____
Athanasios Malamos
Associate Professor , Committee member

Heraklion, November 2012

Abstract

In recent years, we all became witnesses of an unprecedented revolution in social media as a consequence of the appearance of the first large social networks, which encouraging for the first time individuals to share their thoughts and ideas with the newly formed web society. The underlying community structures of these networks created scientific and business value in such an extent in which to re attract the interest of the academic community on clustering methods pushing the boundaries of community detection methods. Motivated by the recognized void in comparative studies of community detection methods we ended up dealing with an experimental validation and comparison of five state of the art algorithms on a wide range of benchmark graphs demonstrating the necessity to devise local and efficient community detection techniques that perform well under a variety of changing conditions. Presuming on the revealed strengths and weaknesses of these methods we proceeded with an empirical study of the MySpace Online Social Network (OSN). Its purpose was threefold aiming to capture the evolution of user population, to examine user activity, and finally to characterize community formation surrounding seed nodes and utilizing only local interactions between nodes.

One million user profiles were randomly collected in a month's period and stored in a local database for further processing. For each profile certain attributes were fetched: profile status (public, private, invalid), member since and last login dates, number of friends, number of views, etc. The profiles and their attributes were analyzed in order to reveal the evolution in user population and the activity of the participating members. Significant conclusions were drawn for the synthesis of the population based on profile status, the number of friends, and the duration MySpace members stay active.

Subsequently, a large number of communities were identified aiming to reveal the structure of the underlying social network graph. The collected data were further analyzed in order to characterize community size and density but also to retrieve correlations in the activity among members of the same community. A total of 171 communities were detected with Fortunato's algorithm, while using Clique Percolation this number was 201. Results demonstrate that MySpace members tend to form dense communities. For the first time, strong correlation in the last login date (the main attribute that shows user activity) for members of the same community was documented. It was also shown that members participating in the same community have similar values for other attributes like for example number of friends. Lastly, there is strong evidence that participation of users in communities inhibits them from abandoning MySpace. As a last observation, members that abandoned MySpace shortly after their account creation (the so-called Tourists), have very low connectivity and thus they do not participate in communities.

Περίληψη

Τα τελευταία χρόνια γίναμε μάρτυρες μίας άνευ προηγουμένου επανάστασης στα κοινωνικά μέσα ως συνέπειας της εμφάνισης των πρώτων μεγάλων κοινωνικών δικτύων, τα οποία ενθάρρυναν για πρώτη φορά χρήστες να μοιραστούν ιδέες και σκέψεις με μια νεοσύστατη διαδικτυακή κοινότητα. Η υποκείμενη διάρθρωση των δικτύων αυτών απέκτησε τέτοια επιστημονική αλλά και εμπορική αξία ώστε να προσελκύσει εκ νέου το ενδιαφέρον της ακαδημαϊκής κοινότητας στο πρόβλημα της αναγνώριρης κοινοτήτων σε κοινωνικά δίκτυα ωθώντας τα όρια των σημερινών μεθόδων συσταδοποίησης. Αναγνωρίζοντας το υφιστάμενο βιβλιογραφικό κενό πάνω σε συγκριτικές μελέτες αλγορίθμων αναγνώρισης κοινοτήτων καταλήξαμε στην ανάπτυξη ενός μεθοδολογικού πακέτου για την σύγκριση πέντε διάσημων αλγορίθμων. Χρησιμοποιώντας ένα ευρύ φάσμα συνθετικών δικτύων καταφέραμε να καταδείξουμε την ανάγκη για ανάπτυξη τεχνικών που θα μπορούσαν να λειτουργήσουν αποδοτικά πάνω σε δίκτυα που συνεχώς μεταβάλλουν μία σειρά παραμέτρων. Εκμεταλλευόμενοι τα πλεονεκτήματα αλλά και τις αδυναμίες αυτών των μεθόδων, όπως αυτές αναδείχτηκαν στο πρώτο σκέλος προχωρήσαμε με μία εμπειρική μελέτη του κοινωνικού δικτύου MySpace. Ο σκοπός της ήταν τριπλός και στόχευε στην σύλληψη της εξέλιξης του πληθυσμού του δικτύου, στην εξέταση της διαδικτυακής δραστηρίοτητας των χρηστών καθώς και στην αποτύπωση των ειδικών χαρακτηριστικών που διέπουν τις κοινότητες του συγκεκριμένου δικτύου χρησιμοποιώντας μόνο τοπικές αλληλεπιδράσεις μεταξύ των κόμβων.

Στην κατεύθυνση αυτή, ένα εκατομμύριο τυχαία επιλεγμένα προφίλ χρηστών συλλέχθησαν εντός ενός χρονικού παραθύρου ενός μήνα και αποθηκεύτηκαν σε μία τοπική βάση δεδομένων αναμένοντας περεταίρω επεξεργασία. Μαζί με κάθε προφίλ ανακτήθηκαν από το διαδίκτυο και μία σειρά από χαρακτηριστικά όπως η κατάσταση του χρήστη, οι ημερομηνίες εγγραφής και τελευταίας επίσκεψης, ο αριθμός των φίλων, ο αριθμός εμφανίσεων προφίλ κ.α. Στη συνέχεια τα προφίλ σε συνδυασμό με τις ιδιότητες που τα συνοδεύουν αναλύθηκαν σε μία προσπάθεια να εξάγουμε χρήσιμα συμπεράσματα για την εξέλιξη του πληθυσμού του δικτύου αλλά και της δραστηριότητας των χρηστών.

Ακολούθως προχωρήσαμε με την αναγνώριση ενός σεβαστού πλήθους κοινοτήτων στοχεύοντας στην ανάδειξη της διάρθρωσης του υποκείμενου κοινωνικού δικτύου. Χρησιμοποιώντας τον αλγόριθμο του Fortunato καταφέραμε να αναγνωρίσουμε 171 κοινότητες ενώ εφαρμόζοντας την μέθοδο του Clique Percolation προχωρήσαμε στην αποκάλυψη του εντυπωσιακού αριθμού των 201 κοινοτήτων. Σε συνδυασμό με την ανάλυση των δεδομένων που συλλέξαμε προηγουμένως από το δίκτυο μπορέσαμε να χαρακτηρίσουμε τις κοινότητες που σχηματίζουν οι χρήστες ως προς το μέγεθος, την πυκνότητα καθώς και την ομοιογένεια τους. Επ' αυτού, για πρώτη φορά συναντάμε ενδείξεις σημαντικής ομοιογένειας πάνω στην ημερομηνία τελευταίας επίσκεψης στο δίκτυο ενώ αντίστοιχα χαμηλή απόκλιση τιμών εμφανίζεται και για το πλήθος των φίλων μεταξύ διαφορετικών χρηστών ίδιων κοινοτήτων. Εν τέλει, υπάρχουν ισχυρά στοιχεία που δείχνουν ότι η συμμετοχή χρηστών σε κοινότητες λειτουργεί αποτρεπτικά στην εγκατάλειψη του δικτύου, υπόθεση που ενισχύεται από τη διαπίστωση ότι οι χρήστες που φέρονται να έχουν εγκαταλείψει το MySpace μετά από σύντομη παραμονή στο δίκτυο, επονομαζόμενοι και “Τουρίστες” χαρακτηρίζονται από ιδιαίτερα χαμηλή συνδεσιμότητα, δηλαδή πολύ μικρό αριθμό φίλων και ως εκ τούτου δεν ευνοείται η συμμετοχή τους σε κοινότητες.

Acknowledgements

Before I proceed with this thesis I would like to express my deepest and sincere gratitude and appreciation to my committee chair and thesis supervisor, Professor Paraskevi Fragopoulou. Through her attitude and passion for research she introduced me to the limitless world of knowledge and scholarship, listening with extreme patience every single idea I had, regardless how trivial this was.

I would also like to thank my thesis co-supervisor, Dr. Harris Papadakis. Without his guidance and persistent help this thesis would not have been possible.

When the day ends, there is always a single person with which you share your universe, a person that is there to support you whether things go well or not. In this particular case I also owe this person being an budding scientist. Not only because she discovered a talent I didn't know I had back in 2003 but more importantly because she offered me a perspective away from mediocrity. This thesis is dedicated to her, Sofia Kleisarchaki, PhD student.

In addition, I feel I need to thank my parents for supporting me at all costs all these years. They were always deeply concerned for my education from a very early age therefore I hope I fulfilled their expectations.

Finally I want to thank all my close friends who tolerated with great patience my quirky personality every single time I believed I was the chosen one, acting like I had achieved the most groundbreaking discovery after the one of fire.

Table of Contents

Abstract....................i
Περίληψη....................iii
Acknowledgements....................v
Table of Contents....................vii
List of Figures....................viii
List of Tables....................ix
1 Introduction....................1
  1.1 Motivation & Problem Statement....................2
  1.2 Thesis Organization 3
2 Complex Networks....................4
  2.1 Social Network Analysis....................4
  2.2 Notion of Complex Network....................4
  2.3 Basic Network Definitions....................4
    2.3.1 Non-Directed Network....................5
    2.3.2 Directed Network....................5
    2.3.3 Mixed Network....................6
    2.3.4 Multinetwork....................6
    2.3.5 Weighted Network....................7
    2.3.6 Regular Network....................8
    2.3.7 Complete Network....................8
  2.4 Node Sequences....................9
    2.4.1 Walk....................9
    2.4.2 Trail....................9
    2.4.3 Path....................9
    2.4.4 Distance....................9
  2.5 Node Sets....................10
    2.5.1 Dyad and Triad....................10
    2.5.2 Triangle....................10
    2.5.3 K-Clique....................10
    2.5.4 Component....................10
    2.5.5 Community....................10
  2.6 Networks Properties....................11
    2.6.1 Homophily and Eterophily....................11
    2.6.2 Clustering Coefficient....................11
    2.6.3 Small World Property....................12
    2.6.4 Hierarchy....................13
    2.6.5 Network Resilience....................13
  2.7 Measures....................13
    2.7.1 Density....................14
    2.7.2 Diameter and Average Path....................14
    2.7.3 Centrality....................14
    2.7.4 Degree-Based Measures....................16
    2.7.5 Modularity....................16
  2.8 Common Network Problems....................17
    2.8.1 Minimum Spanning Tree....................17
      2.8.1.1 Simple Minimum Spanning Tree Algorithm....................17

2.8.2 Shortest-Path Problem Definition....18
2.8.2.1 Dijkstra's algorithm....19
3 Community Detection Algorithms....21
3.1 Newman's Algorithm....21
3.2 CiBC....23
3.3 Bridge Bounding....24
3.4 Fortunato's Algorithm....25
3.5 Clique Percolation....26
4 Design and Implementation....28
4.1 System Architecture....28
4.1.1 Synthetic Input API....28
4.1.2 MySpace API....29
5 Experimental Evaluation on Synthetic Data....31
5.1 Summarizing Evaluation....34
6 Applying Community Detection on Real-world Networks....36
6.1 Profile Collection and Methodology....37
6.2 Dynamics of the underlying network....38
6.3 Experimental results....43
6.4 Conclusion....47
7 Thesis Conclusions....48
8 Source Code Appendix....49
8.1 Benchmark Graph Generator....49
8.2 Algorithms Implementations....51
8.2.1 Newman's Algorithm....51
8.2.2 Bridge Bounding Algorithm....59
8.2.3 Fortunato's Algorithm....65
8.2.4 CiBC Algorithm....71
8.3 Modularity Measure Implementation....78
References....82
Published Papers Appendix....84
Community Detection in Collaborative Environments: A Comparative Analysis....85
Evolution of User Activity and Community Formation in an Online Social Network....86

List of Figures

Εικόνα 1: A drawing of a labeled graph on 6 vertices and 7 edges..5
Εικόνα 2: A simple directed acyclic graph..5
Εικόνα 3: A digraph with vertices labeled (indegree, outdegree)..6
Εικόνα 4: A multigraph with multiple edges (red) and several loops (blue)..............................7
Εικόνα 5: A simple weighted network..7
Εικόνα 6: A simple graph's community structure constisting of three communities................11
Εικόνα 7: Network Modeling versus Hierarchical View...13
Εικόνα 8: Yellow node represents the vertex with the highest betweeness centrality value.....15
Εικόνα 9: The minimum spanning tree of a planar graph...17
Εικόνα 10: Nodes (6, 4, 5, 1) and (6, 4, 3, 2, 1) are both paths between vertices 6 and 1........19
Εικόνα 11: Dijkstra's algorithm's execution example...20
Εικόνα 12: A network exhibiting hierarchical structure. 26
Εικόνα 13: Examples of 3-clique percolation clusters on ER random graphs..........................27
Εικόνα 14: Newman’s algorithm..31
Εικόνα 15: Bridge bounding algorithm..32
Εικόνα 16: CiBC algorithm with BFS (depth=2) 32
Εικόνα 17: CiBC algorithm without BFS...33
Εικόνα 18: Fortunato’s algorithm with a = 0.6...33
Εικόνα 19: Fortunato’s algorithm with a = 0.8...34
Εικόνα 20: Fortunato’s algorithm with a = 1.0...34
Εικόνα 21: CDF of user ID based on profile status over total population...............................39
Εικόνα 22: Synthesis of user population. CDF of user ID based on profile status...................39
Εικόνα 23: CDF of number of friends...40
Εικόνα 24: CCDF of (Fetch Date - Last Login) for public profiles...41
Εικόνα 25: CDF of profile number of views...41
Εικόνα 26: Mean value of profiles number of views for profiles with ID less than X............42
Εικόνα 27: CDF of (Last Login - Member Since), i.e. number of days profiles are active......42
Εικόνα 28: (Fetch Date - Last Login) for all profiles...43
Εικόνα 29: (Fetch Date - Last Login) for Tourists only..43
Εικόνα 30: CDF of community size. 44
Εικόνα 31: CDF of community density...45
Εικόνα 32: Standard deviation of communities member number of friends............................46
Εικόνα 33: Percentage of Tourists in communities..46

List of Tables

Πίνακας 1: Table of basic complete networks with k<=10..8
Πίνακας 2: Parameters used for the benchmark graphs...31
Πίνακας 3: Parameters of the revealed community structures..44

1 Introduction

Computer science from its inception until today, has experienced an unprecedented growth unlike any other field. In just a few years, computers transformed from being a privilege of big institutions and governments to a public acquis. Computers size and cost has shrunk over the years while processing capacity doubles every four, providing everyone with the opportunity to own a personal computer. Computers have infiltrated our lives to such an extent as to be an indicator of quality of life. Countries are now judged by the percentage of their people that have access to internet services while persons that lack computer basic knowledge and skills are considered to be electronically illiterate.

The last two decades we became witnesses of a rapid development of distributed computing and computer networks. Users that were initially restricted to access static text data that was available on the internet are now enjoying multimedia content that is even produced by other users in real-time. The increasing proliferation and affordability of internet devices, as well as the ease of publishing, searching and accessing information on the web encourages the individual users to communicate their content with the web society. This gave birth to the idea of social interaction over the internet which in conjunction with the advent of web 2.0 technology led to the appearance of the first large social media. Social media are defined by Kaplan and Haenlein[1] as a group of internet-based applications that build on the ideological and technological foundations of Web 2.0 and that allow the creation and exchange of User Generated Content (UGC). The once one way communication with the end users passively consuming web content turned into many-to-many communication of interactive dialogues and dynamic content. These applications are today responsible for a large proportion of the information that are exchanged through the internet every day. They produce data-sets of massive size that have the tendency to evolve over time.

This major impact of Social Media on Information Technology Industry has emerged a variety of problems concerning complex networks. In the context of network theory, complex networks are defined as graphs of non-trivial topological features occurring in real graphs. Networks in various application domains present an internal structure, where nodes form groups of tightly connected components which are more loosely connected to the rest of the network. These components are mostly known as communities, clusters or groups, terms used interchangeably in the rest of this thesis. Uncovering the community structure of a network is a fundamental problem in complex networks, already successfully applied in a wide range of scientific disciplines including Physics, Biology, Social Sciences, Discrete Mathematics and more recently Computer Science.

The task of community detection attracted once more the interest of the academic community on clustering, putting inevitably under reconsideration the capabilities of the existing community detection methods. The plurality of these algorithms proved to be inefficient in adapting to the modern distributed environment and were rendered obsolete leaving a significant void in scientific literature. This inconvenient truth led to the invention of new methods, able to deal with the new restrictions imposed by modern Online Social Networks (OSNs) including large amount of data with streaming nature.

In recent years, the ubiquity of communication networks speeds up the development of internet applications. Social networking[2] has been driving a dramatic evolution due to the increasing

1 A. M. Kaplan and M. Haenlein, “Users of the world, unite! The challenges and opportunities of Social Media” Business Horizons, 2010.

2 B. Buter, N. Dijkshoorn, D. Modolo, Q. Nguyen, S. van Noort, B. van de Poel, A. Ali, and A. Salah1. Explorative visualization and analysis of a social network for arts: The case of deviantart. Journal of Convergence Volume, 2(1), 2011.

use of Web 2.0 elements such as blogs, micro-blogging services (e.g. Twitter), social networking sites (e.g. MySpace, Facebook, LinkedIn), social media news (e.g. Digg) and wikis, etc. One of the fundamental problems in social networking with a lot of potential applications is to detect effectively the communities that are created by the users' interaction. In a dynamic environment of social networks the network structure evolves rapidly and the content is significantly larger in size. Such observations are unique in the online scenario and challenge the scientists.

Several attempts have been made to provide a formal definition for this generally described "community detection" concept in networks. A strong community was defined as a group of nodes for which each node of the community has more edges to other nodes of the same community than to nodes outside the community[3]. This is a relatively strict definition, in the sense that it does not allow for overlapping communities and creates a hierarchical community structure since the entire graph can be a community itself. A weak community, was later defined as a subgraph in which the sum of all node degrees within the community is larger than the sum of all node degrees toward the rest of the graph[4].

Variations also appear in the method used to identify communities: Most of the algorithms that appear in the literature follow an iterative approach starting by characterizing either the entire network, or each individual node as community, and splitting[5], or merging, respectively. These methods produce a hierarchy of partitions. There is an entire hierarchy of communities, because communities are nested: small communities compose larger ones, which in turn are put together to form even larger ones. By merging or splitting communities one can build a hierarchical tree of community partitions called dendrogram. The modularity criterion[6] defined n is a measure of the quality of a partition, and can be used to identify a single optimal partition, i.e. the one corresponding o the largest modularity value.

1.1 Motivation & Problem Statement

The underlying community structure of real-world networks has created scientific and business value. Although many algorithms exist dealing with the problem of community detection it is recognized that the methods involved have a long way to go leaving significant space for improvement. Motivated from the void in comparative studies on community detection methods in the related literature we ended up dealing with an experimental validation and comparison of five well-known algorithms for community detection.

Furthermore, motivated from the aforementioned observation that the social networks produce vast amount of data on a daily basis with a dynamic nature in forming communities we collected and subsequently studied data coming from the social network of MySpace. In particular, we examined its underlying friendship graph along with other equally interesting profile attributes, in an effort to extract valuable information on the network's population, the evolution that takes place as users abandon the network and the extent to which this mobility can be captured applying community detection techniques.

3 G. W. Flake, S. Lawrence, C. L. Giles, and F. M. Coetzee. Self-organization and identification of web communities. IEEE Computer, 35:66–71, March 2002.

4 D. Katsaros, G. Pallis, K. Stamos, A. Vakali, A. Sidiropoulos, and Y. Manolopoulos. Cdns content outsourcing via generalized communities. IEEE Transactions on Knowledge and Data Engineering, 21:137–151, 2009.

5 M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. Physical Review E, 69(2):026113, Feb 2004.

6 M. Girvan and M. E. J. Newman. Community structure in social and biological networks. Proceedings of the National Academy of Sciences of the United States of America, 99(12):7821–7826, June 2002.

1.2 Thesis Organization

After this introductory section, we proceed with chapter 2, we make an introduction to basic concepts that accompany graphs, structures borrowed from discrete mathematics used in order to model social networks. Having consolidate the previous preliminaries we continue with an analysis of a series of state of the art community detection algorithms that dominate the related literature. Chapter 4 describes the basic architecture of the system that was implemented as part of this thesis highlighting all parameters involved in the development. Among them we can distinguish restrictions imposed from the social networks and the World Wide Web, system requirements and other problems that chaperon data mining and concurrent programming. Chapter 5 introduces a methodological frame that permits as to evaluate the performance of the examined algorithms on a wide range of synthetic networks. Chapter 6 presuming on the knowledge gained from the previous chapter help us extract valuable conclusions on the performance of community detection methods on real-world applications while Chapter 7 is the final chapter of this thesis being this dissertation's summary.

2 Complex Networks

2.1 Social Network Analysis

Social Network Analysis (SNA) is the methodical analysis of social networks. In order to study efficiently a social network in the purpose of understanding its dynamics, a simplified and abstract model is required. A popular modeling method is to represent a network's objects and their relationships by a graph, allowing us to use a large set of generic methods provided by the field of Discrete Mathematics. In this context, individual actors within the network are represented by nodes and the relationships formed between them such as friendship or organizational position are represented by ties. The use of this specific modeling tool was considered a fail-safe choice as the technique had already been successfully applied to a wide range of scientific disciplines including mathematics, physics, biology social sciences and criminology. Tool's flexibility has allowed academic community to tackle with a large set of non-trivial problems. Impressively, network analysis has been already conscripted in an effort to extract valuable information on human DNA and molecules attractions using the same principles that allow us to solve navigation problems or uncover the relationships that lay between two people that share for example the same hobbies, political views, marital status etc.

2.2 Notion of Complex Network

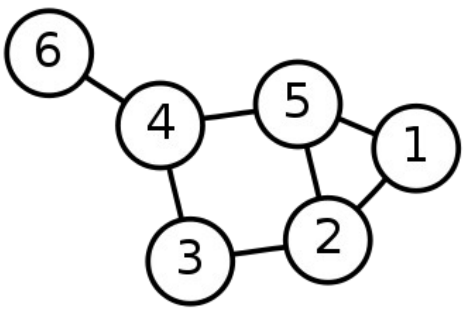
The rapid development of distributed computing domain, in conjunction with web 2.0 technology eventually led to the appearance of the first social networks. Data-sets produced by these networks are characterized by their massive size and the tendency they have to evolve over time. More importantly, the plurality of the existing social networks display substantial non-trivial topological features, with patterns of connection between their elements that are neither purely regular nor purely random. Before we get our hands on the basic problems that chaperon network analysis we must first focus on a series of important preliminaries that concern Network Theory. Objects' semantic content may vary requiring networks to adapt to different occasions and applications. The ability to cope with so diverse data-sets is based on a variety of different Graph definitions.

2.3 Basic Network Definitions

Before we get our hands on the basic problems that chaperon networks we must first focus on a series of important preliminaries that concern Graph Theory, including some basic definitions. In mathematics and computer science, graphs represent mathematical structures used to model pairwise relations between objects from a certain collection. The interconnected objects are represented by mathematical abstractions called vertices or Nodes which are connected by edges. Graphs can be implemented in a visual level by a set of dots or circles for the vertices, joined by lines that simulate the relations between them. What makes Graphs a so powerful mathematical tool is its flexibility. As mentioned before Graphs have the ability to adapt to the nature of the represented data. This ability to cope with so diverse data-sets is based on a variety of different Graph definitions.

2.3.1 Non-Directed Network

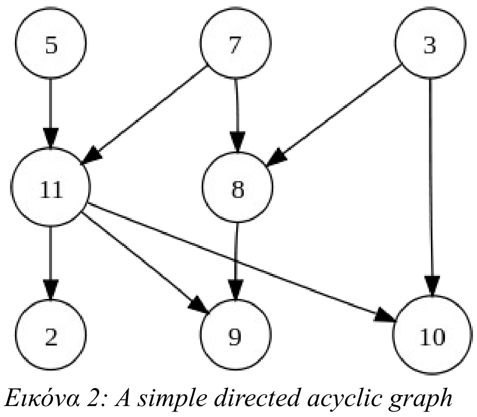
A network whose edges have no orientation is referred as a Non-Directed Network. The edge that connects node *x* to node *y* is exactly the same with the edge that connects node *y* to node *x*. It is the simplest and most common type of networks. Unless stated otherwise, from now on the words "Graph" and the word "Network" for this thesis will refer to this type of graphs.



Εικόνα 1: A drawing of a labeled graph on 6 vertices and 7 edges.

2.3.2 Directed Network

A directed network or digraph is an ordered pair *D* = (*V*, *A*). *V* refers to the set of nodes while *A* refers to a set of ordered pairs of nodes called arcs. An arc *a = (x, y)* is considered to be directed from node *x* to node *y*. *y* is called the head or direct successor, and *x* is called the tail or direct predecessor of the arc. If a path leads from vertex *x* to vertex *y* via one or more other nodes, *y* is considered to be successor or reachable from *x*. A directed graph as described above, is considered to be symmetric if for every arc in *D*, the corresponding inverted arc also exists in *D*. A symmetric loop-less directed graph *D = (V, A)* is equivalent to a simple undirected graph *G = (V, E)*, where the pairs of inverse arcs in *A* correspond one to one with the edges in *E*. As a consequence the edges in *D* are twice as much as the edges in *G*.

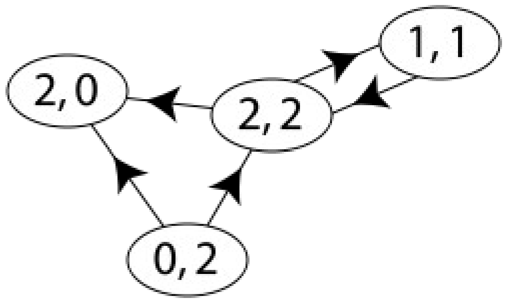


Εικόνα 2: A simple directed acyclic graph

A more strict variation of this definition is called oriented network. The term refers to a network in which no more than one of *(x, y)* and *(y, x)* may exist simultaneously. For a node, the number of head endpoints adjacent to a node is called the in-degree of the node and the number of tail endpoints is its out-degree. The in-degree is denoted $deg^{-}(v)$ and the out-degree as $deg^{+}(v)$. A vertex with $deg^{-}(v) = 0$ is called a source, as it is the origin of each of its incident edges. Similarly, a vertex with $deg^{+}(v) = 0$ is called a sink.

$$\sum_{v\in V} \deg^+(v) = \sum_{v\in V} \deg^-(v) = |A|\,.$$

If for every node $v \in V$, $deg^{+}(v) = deg^{-}(v)$, the graph is called a balanced digraph. Regarding its connectivity, a digraph *G* is called weakly connected if the undirected underlying graph obtained by replacing all directed edges of *G* with undirected edges is a connected graph. A digraph is strongly connected if it contains a directed path from *u* to *v* and a directed path from *v* to *u* for every pair of vertices *u*,*v*. The strong components are the maximal strongly connected sub-graphs.



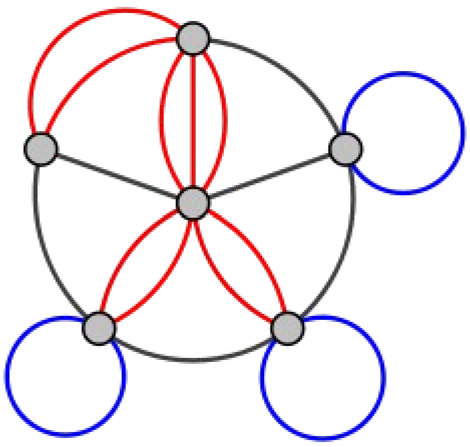
Εικόνα 3: A digraph with vertices labeled (indegree, outdegree)

2.3.3 Mixed Network

A mixed network *G* is graph in which some edges may be directed and some other may not be directed. It can be expressed through an ordered triple *G = (V, A, E)* with *V*, *A* and *E* as defined earlier.

2.3.4 Multinetwork

In discrete mathematics, the term multinetwork refers to a graph that allows the existence of multiple edges. Edges are considered to be multiple or parallel, if they have the same tail and head creating multiple direct paths between these nodes. Some authors also allow multigraphs to have loops, edges that connects a vertex to itself, while others call these pseudo-graphs, reserving the term multigraph for the case with no loops. This definition can be combined with these of directed or undirected graphs resulting in a hybrid graph. For example in flight booking systems, multigraphs might be used to model the possible flight connections offered by an airline.

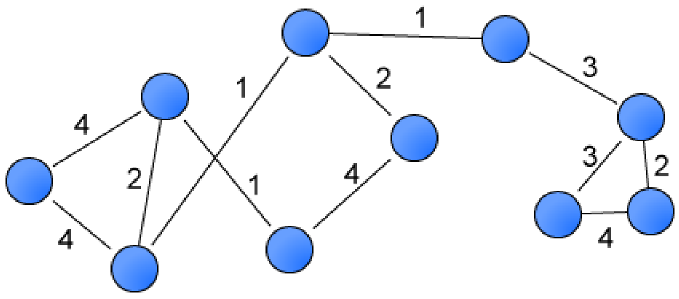


Εικόνα 4: A multigraph with multiple edges (red) and several loops (blue).

2.3.5 Weighted Network

Certain applications demand more information to be represented by the graph. In this case a real or a rational number which is called weight or cost can be assigned to each edge in order to represent, for example, costs, lengths or capacities, etc. depending on the problem at hand. Although some authors use the term network as a synonym for a weighted graph, unless stated otherwise, a graph is always assumed to be unweighted. Classic problems connected with weighted graphs include:

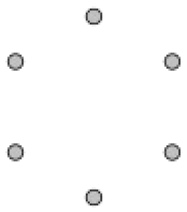
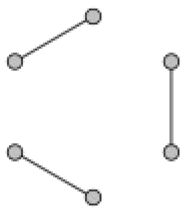
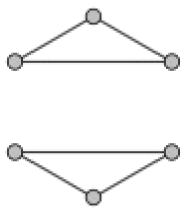
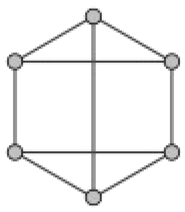
- ***Minimum spanning tree***
- ***Shortest-path problem***
- ***Max-flow/min-cut theorem***



Εικόνα 5: A simple weighted network

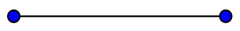
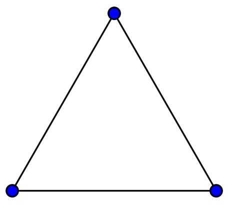
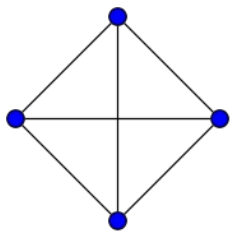
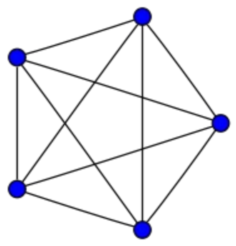
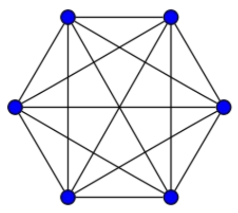
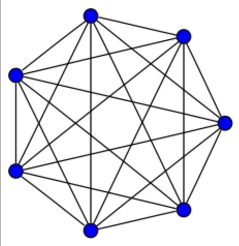
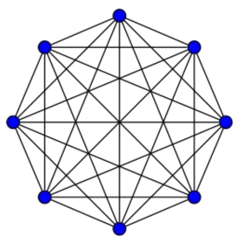
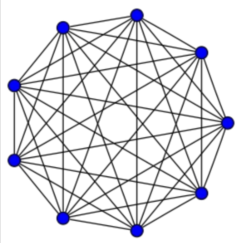
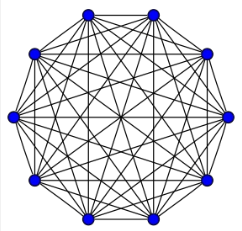
2.3.6 Regular Network

In graph theory, the term regular graph is used to describe a graph where each node has the same degree, in other words the same number of neighbors. In case we are dealing with directed graphs, another restriction must also be satisfied. The in-degree and out-degree for every vertex has to be equal. A regular graph with nodes of k-degree can also be referred as a k-regular graph.

			
0-regular graph	1-regular graph	2-regular graph	3-regular graph

2.3.7 Complete Network

The term “complete”, some times stated as “fully connected”, defines a simple undirected graph in which all pairs of two distinct vertices are connected by a unique edge. The complete graph on n vertices has $n(n - 1)/2$ edges, and is denoted by K_n. Another way to see a complete graph is as a (k-1) regular graph. One very interesting ascertainment on complete graphs is that the only way to disconnect the graph is by removing the complete set of vertices. The complement graph of a fully connected graph is an empty graph. Essentially complete graphs form their own maximal cliques.

				
K_1	K_2	K_3	K_4	K_5
				
K_6	K_7	K_8	K_9	K_{10}

Πίνακας 1: Table of basic complete networks with k<=10

2.4 Node Sequences

2.4.1 Walk

A walk from node *x* to node *y* is any sequence of adjacent edges that begins with an edge containing *x* and ends with an edge containing node *y*. Nodes and edges can be visited several times in a walk, as long as these constraints are not violated.

2.4.2 Trail

On the other hand, trail is defined as a walk where the sequence of adjacent edges does not contain any edge for more than one times. Though, the same constraint does not apply for nodes, so a node can be visited through different edges numerous times.

2.4.3 Path

Path is defined as a walk in which neither edges nor nodes are visited more than once. If the starting node coincides with the destination node then this path is considered to be an Eulerian path. This specialized notion of path was first introduced by Leonhard Euler while solving the famous Seven Bridges of Konigsberg problem in 1736.

2.4.4 Distance

In network theory, the shortest-path problem is the problem of finding a path between two nodes in a graph such that the sum of the weights of its constituent edges is minimized which is analogous to the problem of finding the shortest path between two intersections on a road map. In this context the graph's vertices correspond to intersections and the edges correspond to road segments, each weighted by the length of its road segment. In a non weighted network, the length of a path corresponds to the minimum hops required in order to reach node *y* starting from node *x*. On the other hand, in a weighted network, the length of a path is processed by summing the weights of the edges it includes, resulting in a slightly different definition of the distance. When a node *y* is not reachable from a node *x*, their distance is theoretically infinite. However, in many situations, this infinite value causes problems, and this distance is then considered to be zero. The most important algorithms for solving this problem are:

- ***Dijkstra's algorithm***
- ***Bellman-ford algorithm***
- ***A* search algorithm***
- ***Floyd-Warshall algorithm***
- ***Johnson's algorithm***
- ***Perturbation theory***

2.5 Node Sets

2.5.1 Dyad and Triad

Dyad is the smallest set of nodes. It consists of two nodes which can be linked together using none, one or multiple edges. Triad on the other hand, as its name betokens concerns group of nodes containing three nodes.

2.5.2 Triangle

Triangle's cornerstone is the triplet. A triplet consists of three nodes that are connected by either two (open triplet) or three (closed triplet) non directed edges. A triangle consists of three closed triplets, one centered on each of the nodes.

2.5.3 K-Clique

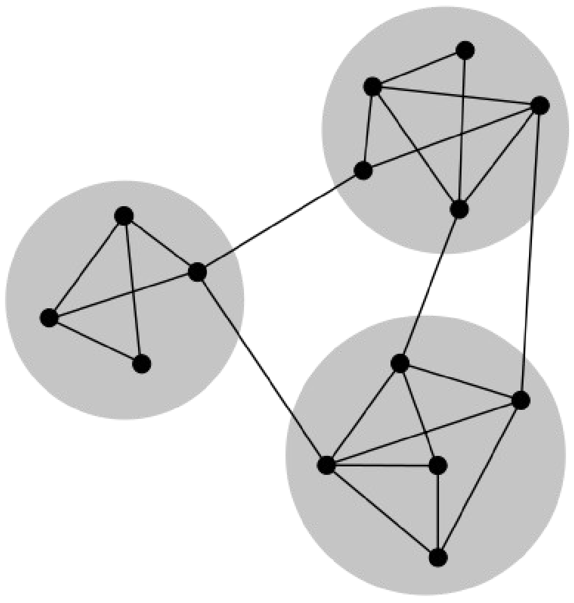
K-Clique corresponds to a maximally connected subset of nodes. Each node must be connected to every other node that participates into the clique. Its size corresponds to the number of nodes it contains.

2.5.4 Component

A component is a sub-network in which any node is reachable from any other one by a walk. Put concisely, it is a maximal connected subgraph. For an undirected network, a component is a set of connected nodes with no links with other nodes from the same network. But for directed networks, it is less straightforward. A component is said to be strongly connected if there is a directed walk between each pair of nodes. It is called weakly connected if there is at least an undirected walk between each pair of nodes. A network with only one component is said to be connected. An isolated node (i.e. a node with a degree zero) is a component of its own.

2.5.5 Community

Several definition on community exist. Though two of them are more popular among the academic community. A strong community is defined as a group of nodes for which each node has more edges to nodes of the same community than to nodes outside the community. This definition is relatively strict, since it does not allow for overlapping communities and creates a hierarchical community structure, since the entire graph can be a community itself. A generalized community, is defined as a subgraph in which the sum of all node degrees within the community is larger than the sum of all node degrees towards the rest of the graph. Other definitions are related to the optimization of some ``fitness'' criterion like for example the intracommunity edge density, or the intercommunity edge cut. In general, community detection is the problem of finding a partition of a graph into subgraphs that maximizes some quality criterion which reflects the density of the subgraph(s).



Εικόνα 6: A simple graph's community structure constisting of three communities

2.6 Networks Properties

2.6.1 Homophily and Eterophily

Homophily refers to the tendency for nodes to connect to other vertices with which they share (respectively don't share) common attributes and characteristics. In other words homophily seems to be responsible for emphasizing or even causing the relationships between the nodes. For example, in a social network where edges represent friendship connections, it is more likely that a user is connected to an equally young person or a user with who shares common interests. In literature this property can be also found as assortative mixing.

2.6.2 Clustering Coefficient

In network theory clustering coefficient is representing the extent to which nodes of a specific network tend to shape clusters. Evidence suggests that in most real-world networks, and in particular social networks, nodes tend to create tightly connected groups characterized by a relatively high density of ties. Impressively, in real-world networks, this likelihood tends to be greater than the average probability of a tie randomly established between two random vertices. Two versions of this metric exist in the literature, depending on their perspective. The global perspective definition was designed to give an overall indication of the clustering in the network, whereas the measure that follows a local approach gives an indication of the embeddedness of single vertices. The global clustering coefficient is based on triplets of nodes and is defined as the number of closed triplets over the total number of triplets. This measure gives us an indication of the clustering in the whole network, and can be applied to both non directed and directed networks.

$$C = \frac{3 \times \text{number of triangles}}{\text{number of connected triples of vertices}} = \frac{\text{number of closed triplets}}{\text{number of connected triples of vertices}}.$$

The local clustering coefficient of a node on the other hand, acts as an indication of how close its neighbors are to form a complete subgraph. A graph $G = (V, E)$ formally consists of a set of vertices V and a set of edges E connecting them. An edge e_{ij} connects vertex i with vertex j. The neighborhood N_i for a vertex u_i is defined as the set that consists of all the nodes that are directly connected to vertex i.

$$N_i = \{u_j : e_{ij} \in E \wedge e_{ii} \in E\}$$

We define k_i as the size of node's i neighborhood. The local clustering coefficient C_i for a vertex u_i then corresponds to the proportion of links between the vertices within its neighbourhood divided by the number of links that could possibly exist between them. For a directed graph, e_{ij} is distinct from e_{ij}, and therefore for each neighbourhood N_i there are links that $k_i(k_i\text{-}1)$ could exist among the vertices within the neighbourhood (k_i is the number of neighbors of a vertex). Though, in non directed networks, e_{ij} and e_{ji} are considered identical. Therefore, if a vertex u_i has $k_{i\,\text{neighbors}}$, $k_i(k_i\text{-}1)/2$ edges could exist among the vertices within the neighborhood. Thus, the local clustering coefficient for undirected graphs can be defined as:

$$C_i = \frac{2|\{e_{jk}\}|}{k_i(k_i - 1)} : v_j, v_k \in N_i, e_{jk} \in E.$$

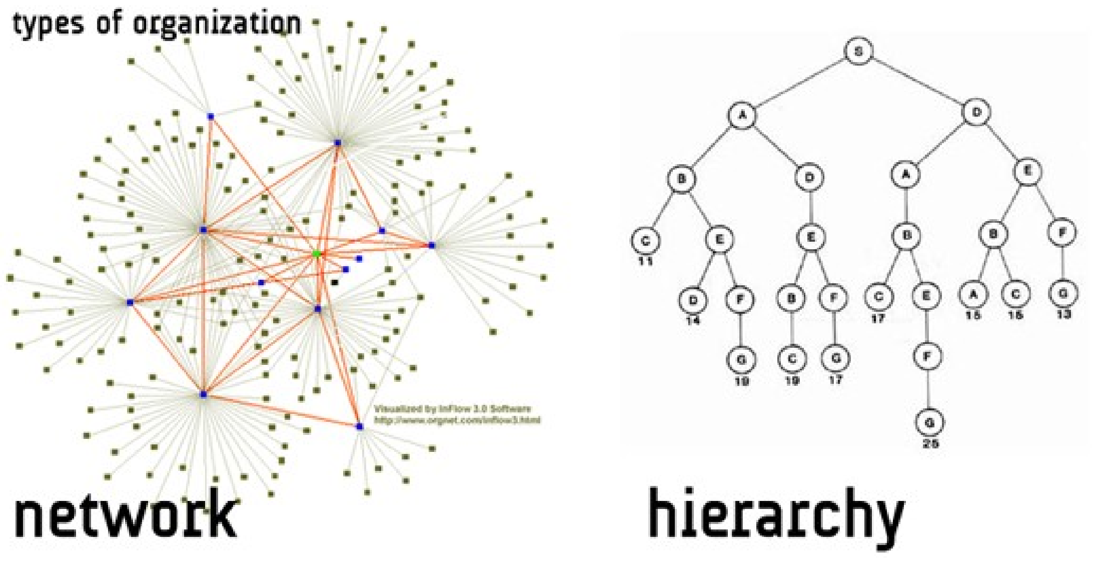
2.6.3 Small World Property

Small world property was first introduced by Stanley Milgram and a group of other researchers on an effort to examine the average path lengths for social networks of people living in the united states. The research proved to be groundbreaking suggesting that human society network is characterized by short path lengths. This characteristic is often associated with the idea of “the six degrees of separation”. According to the idea everyone is on average approximately six steps away,by way of introduction, from any other person in the world, so that a chain of “a friend of a friend” statements can be made, on average, to connect any two people in six steps or fewer. As far as mathematics are concerned, a small-world network is a graph in which most nodes are not neighbors of one another, but most nodes can be reached from every other with a relatively low cost. More specifically, a small-world network is defined to be a network where the typical distance L between two randomly chosen nodes (the number of hops required) grows proportionally to the logarithm of the number of nodes N in the network, that is:

$$L \; x \; \log N$$

2.6.4 Hierarchy

Another very interesting property found on an important number of complex networks and hence on most modern social networks is hierarchy which shares common characteristics with the tree topology, applied in real-world computer networks. Networks that adopt this property tend to have less links between the nodes of different levels , where vertices that participate to the same level are more densely connected with each other.


types of organization
Visualized by InFlow 3.0 Software
http://www.orgnet.com/inflow3.html
network
hierarchy

Εικόνα 7: Network Modeling versus Hierarchical View

2.6.5 Network Resilience

The term signifies the extent to which a network tends to maintain its topological properties when changes occur on its structure. By removing a series of edges or nodes, network's connectivity is reduced having an important impact on information flow which, when passing a critical point can lead to bottlenecks.

2.7 Measures

Different measures are used to quantify the previously described properties. One can distinguish node level (or local) measures and network level (or global) ones. Node level measures are mainly related to centrality.

2.7.1 Density

Density imprints the general level of network's connectivity. It can be defined as the ratio of the number of the existing edges *m* to the number of all possible edges between the vertices of the network. The more density approaches one, the more the network approaches being a complete network. A network is characterized as sparse in case that the number of links is of the same order as the number of nodes. Otherwise a network is considered to be dense.

$$d = \frac{m}{n(n-1)}$$

2.7.2 Diameter and Average Path

Diameter refers to the maximum distance that exists between two nodes of the network and sets the upper-bound for all distances between to any set of vertices that participate into the network. Another basic property that defines the topology of a network is the average distance. It can be formulated as the mean value of all distances over all possible pairs of the network.

$$l = \frac{1}{n(n-1)} \sum i \neq j \, dij$$

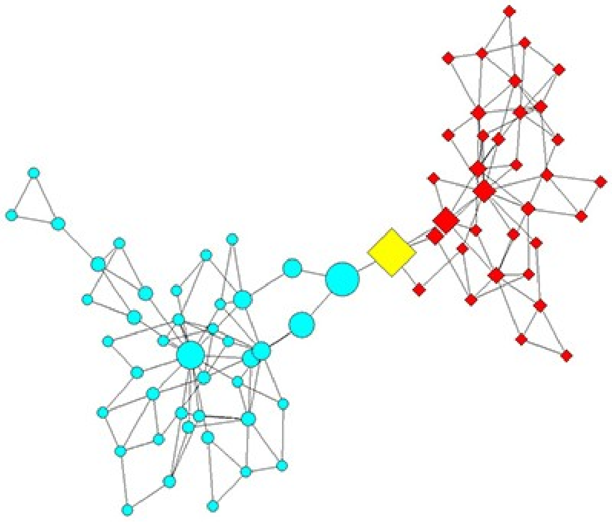
2.7.3 Centrality

Within the scope of graph theory, a node's relative importance can be defined using a series of measures. Although centrality concepts were first developed in social network analysis, and many of the terms used to measure centrality reflect their sociological origin, they have adapted to a wide range of real-world problems. It can be used to value the influence of a person over a social network with the same success that it can be used on urban network routing. The most common centrality metrics found in relative literature include:

- ***Degree centrality***
- ***Betweeness centrality***
- ***Closeness***

Degree centrality is the most trivial measure of them and is defined as the number of edges containing a node. The degree can be interpreted in terms of the immediate risk of a node for catching whatever is traveling through the network. In the case of a directed network, we usually define two separate measures of degree centrality, namely in-degree and out-degree. As mentioned earlier, in-degree is a count of the number of ties directed to the node and out-degree is the number of ties that the node directs to others. When ties are associated to some positive aspects such as friendship or collaboration, in-degree is often interpreted as a form of popularity, and out-degree as gregariousness. Another definition that exists in literature denotes degree centrality as the ratio of actual degree (in case of directed graphs in-degree or out-degree respectively) to the possible edges leading to a specific node which is equal to the size of the graph minus one.

$$C = \frac{Vertex\ Degree}{n-1}$$



Εικόνα 8: Yellow node represents the vertex with the highest betweeness centrality value

Betweeness centrality was first introduced as a metric for quantifying the control of a human on the communication between other humans in a social network by Linton Freeman. In this context, nodes that have a high probability to be encountered on a randomly chosen shortest path between two randomly chosen nodes are characterized by a high betweenness value. The betweenness of a vertex *v* in a graph *G = (V, E)* with *V* vertices can be calculated using the following algorithm:

1. ***For each pair of vertices (s,t), compute all shortest paths that lay between them.***
2. ***For each pair of vertices (s,t), determine the fraction of shortest paths that pass through the vertex in question (here, vertex v)***
3. ***Sum this fraction over all pairs of vertices (s,t)***

The algorithm can be expressed in a more compact form by the following equation:

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

where σ_{st} is total number of shortest paths from node s to node t and $\sigma_{st}(\upsilon)$ is the number of those paths that pass through υ.

Closeness of a node *s* is defined as the inverse of the sum of its distances to all other nodes. Thus, the more central a node is the lower its total distance to all other nodes is. Closeness can be regarded as a measure of how fast it will take to spread information from *s* to all other nodes in a sequential pattern.

2.7.4 Degree-Based Measures

The simplest measure regarding degree is *average degree* which can be defined as the mean degree processed over the entire network population and for non directed graphs can be formulated as follows:

$$DM = \frac{1}{n} \sum ki$$

Unfortunately, mean degree does not offer a clear view over the network's structure as it does not take into consideration the *degree distribution* as no assumptions can be made on properties like homophily or eterophily. In order to extract more valuable information on the network's characteristics, *degree distribution* and *degree standard deviation* must also be examined. Experimental studies on statistical behavior have shown that degree distribution on real-world social networks follow either a power or an exponential law.

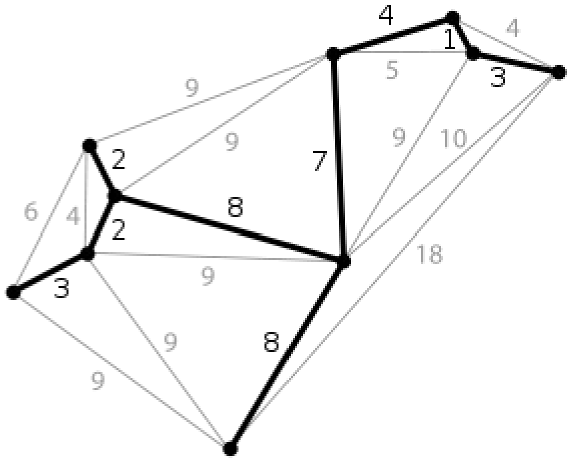
2.7.5 Modularity

Modularity essentially indicates the extent to which a given community partition is characterized by high number of intra-community edges compared to inter-community ones and is calculated as follows: for a network community structure with *l* communities, an *l x l* symmetric matrix *e* is defined whose element e_{ij} is the fraction of all edges in the network that connect nodes in community *i* to nodes in community *j*. The row sums $a_i = \Sigma_j e_{ij}$ of this matrix represent the fraction of edges with an endpoint in community *i*. Modularity *Q* of a community partition is defined as follows: $Q = \Sigma^l_{i=1} e_{ii} - a_i^2$. Modularity measures the fraction of intra-community edges minus the expected value of the same quantity in a network with the same community division and random connections between nodes. If the number of intra-community edges is no better than random, we will get a modularity value close to 0, while modularity values approaching 1 (which is the maximum possible) indicate networks with strong community structure.

2.8 Common Network Problems

2.8.1 Minimum Spanning Tree

Given a non-directed non-weighted network *G = (V, E)*, a spanning tree is a subset of the graph's edges that has the ability to maintain network's connectivity. In other words, the resulting tree-type data structure that contains a subset of *E* ensures that no vertex becomes inaccessible. For each graph, one or more spanning trees exist. The term Minimum Spanning Tree (MST) on the other hand refers to the smallest possible set of edges characterized by this property. If we assign a weight to each edge, representing its cost, we can use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A minimum spanning tree (MST) in this case or minimum weight spanning tree is then a spanning tree with weight less than or equal to the weight of every other spanning tree.



Εικόνα 9: The minimum spanning tree of a planar graph.

2.8.1.1 Simple Minimum Spanning Tree Algorithm

The following function constitutes a simple solution to the problem:

```
function MST(G,W):
    T = {}
    while T does not form a spanning tree:
        find the minimum weigthed edge in E that is safe for T
```

```
                T = T union {(u,v)}
    return T
```

2.8.2 Shortest-Path Problem Definition

The shortest path problem can be defined for a wide range of networks as mentioned earlier. In the case of non-directed networks, two nodes are considered to be adjacent if they are both incident to a common edge. As mentioned earlier path in an undirected graph is a sequence of vertices:

$$P=(v_{1,}v_{2,}\dots,v_n)\in V\times V\dots\times V$$

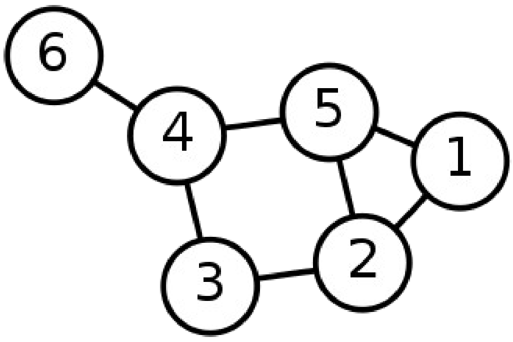
such that v_i is adjacent to v_i+1 for $1 < i < n$. Such a path P is called a path of length n from v_1 to v_n. Let e_{ij} be the edge incident to both v_i and v_j. Given a real-valued weight function $f:E\rightarrow\mathbb{R}$, and a non-directed network G, the shortest path from v to v' is the path $P=(v_{1,}v_{2,}\dots,v_n)$ (where $v_1 = v$ and $v_n = v'$) that over all possible n minimizes the sum:

$$\sum_{i=1}^{n-1} f(e_{ii\ +1})$$

When the graph is unweighted or $f:E\rightarrow\{c\}, c \in \mathbb{R}$, this is equivalent to finding the path with fewest edges. The problem is also sometimes called the single-pair shortest path problem, to distinguish it from the following variations:

- *The **single-source shortest path problem**, in which we have to find shortest paths from a source vertex v to all other vertices in the graph.*
- *The **single-destination shortest path problem**, in which we have to find shortest paths from all vertices in the directed graph to a single destination vertex v. This can be reduced to the single-source shortest path problem by reversing the arcs in the directed graph.*
- *The **all-pairs shortest path problem**, in which we have to find shortest paths between every pair of vertices v, v' in the graph.*

These generalizations have significantly more efficient algorithms than the simplistic approach of running a single-pair shortest path algorithm on all relevant pairs of vertices.



Εικόνα 10: Nodes (6, 4, 5, 1) and (6, 4, 3, 2, 1) are both paths between vertices 6 and 1.

2.8.2.1 Dijkstra's algorithm

The most popular algorithm dealing with the problem was introduced by Edsger Wybe Dijkstra in 1956. Since 1959 when it was published[7], Dijkstra's algorithm has been widely used in routing algorithms, mainly as a subroutine. Algorithm solves the single-source shortest-path problem for weighted networks with non-negative edge costs. For a given node in the network, the algorithm finds the shortest path (i.e with lowest cost) between that node and every other node which is connected to the network. This way it can be used for finding costs of shortest paths from point A to point B by stopping the algorithm once the shortest path to the destination has been determined. For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities, a feature that is used by all modern Geographical Information Systems (GIS) including the well known Google Maps.

The algorithm is uses breadth-first search (BFS) in order to explore vertices by spreading out as new vertices are discovered. We could liken the process with starting a fire on the graph. We light the source vertex, and then the fire spreads to its neighbors while going out at the starting node. The fire then spreads to the unburned neighbors of the burning vertices, and so on until the entire graph is burned. For our purpose we will assume that only one vertex can be burning at a time; that is our fire will always choose to spread to the closest neighbor.

Pick a starting vertex o. We label each vertex v in the graph with a "distance" δ(v) from o. We start out with δ(o) = 0 and guess ∞ for the rest. We start with each vertex in the Unburned state, from which we will remove them as they are burned by the algorithm. Removed nodes are called Burned. For each vertex v we will also keep track of a source vertex called source(v), which is the vertex closest vertex to v along its shortest path to o. The algorithm can be formulated using pseudocode as follows:

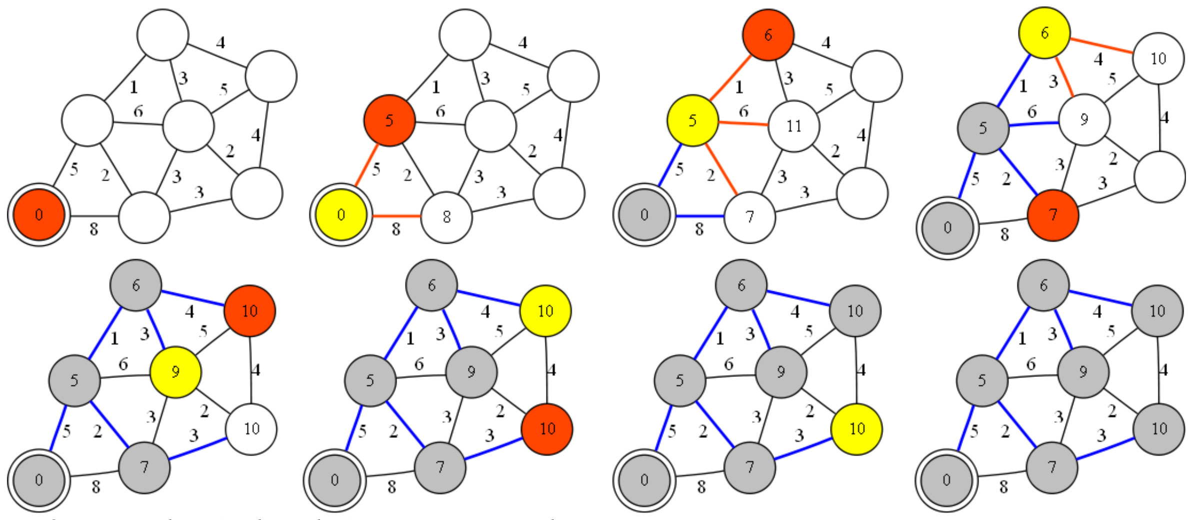
```
Function Dijkstra()
    For each vertex v in G
        Label v as Unburned
        Set δ(v) = &infin
        Set source(v) as undefined
```

7 Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs". Numerische Mathematik 1: 269–271. doi:10.1007/BF01386390.

```
                Set δ(o) = 0

          While any vertex is Unburned
                Call the Unburned vertex with smallest &delta value u
                Label u as Burned

          For each neighbor n of u
                If δ(u) + w(u,n) < &delta(n)
                Set δ(n) = δ(u) + w(u,n)
                Set source(n)=u
    End
```



Εικόνα 11: Dijkstra's algorithm's execution example

In the above picture starting point is marked by a double circle. The Unvisited node with the smallest delta value is orange. It will be visited at the next step in the algorithm. The yellow vertex is the one that is currently selected. Gray vertices have been labeled as already visited by the algorithm. Neighbors if the burning vertex whose delta is being updated have their edges marked in orange. This happens when δ(u) + w(u,n) < delta(n). Edges corresponding to sources are marked in blue. Note that the algorithm will occasionally overwrite the source of a vertex with a closer source. Once all the vertices are Burned the algorithm is burned.

In order to find the shortest-path we simply follow the source edges (blue in the example) beginning this time from the destination until we reach the initial vertex. We also know that we have actually found the shortest path (it is possible to have more than one path tied for being the shortest) since our fire has always chosen to take the shortest steps possible when moving to a new location.

3 Community Detection Algorithms

Variations appear in the methodology used to identify communities. Certain algorithms follow an iterative approach starting by characterizing either the entire network, or each individual node as community, and splitting [2, 6] or merging [3] communities, respectively. These methods produce a hierarchy of nested communities. By merging or splitting communities one can build a hierarchical tree of community partitions called dendrogram. Several re-searcher aim to find the entire hierarchical community dendrogram, while others try to identify only the optimal community partition. More recent approaches aim to identify the community surrounding one or more seed nodes. Some researchers aim to discover distinct (non-overlapping) communities, while others allow for overlaps.

3.1 Newman's Algorithm

One of the well known community finding algorithms was developed by Girvan an Newman[8][9]. This algorithm follows, what is known as, the divisive-agglomerative method, a hierarchical approach based on which communities are detected by removing edges iteratively from the graph. An edge that belongs to many shortest paths between nodes has high betweeness and has to be removed, because it is more likely to be an inter-community edge. By removing gradually edges, the graph is split and its hierarchical community structure is revealed. The algorithm is computationally intensive, because following the removal of an edge, the shortest paths between all pairs of nodes have to be recalculated. However, it reveals not only individual communities, but the entire hierarchical community dendrogram of the graph. An important element of the algorithm is the modularity calculation, which is used to evaluate the quality of a community partition resulting and also as a termination criterion for the algorithm.

Although there is a wide range of betweenness measures available, shortest-path betweenness is used due to the lower computational cost and the satisfactory results. Shortest-path betweenness can be calculated by finding the shortest paths between all pairs of vertices and summing up how many of those run along each edge. Experimental results for implementations of the algorithm based on different betweenness metrics have shown no significant impact on the quality of the community structure output.

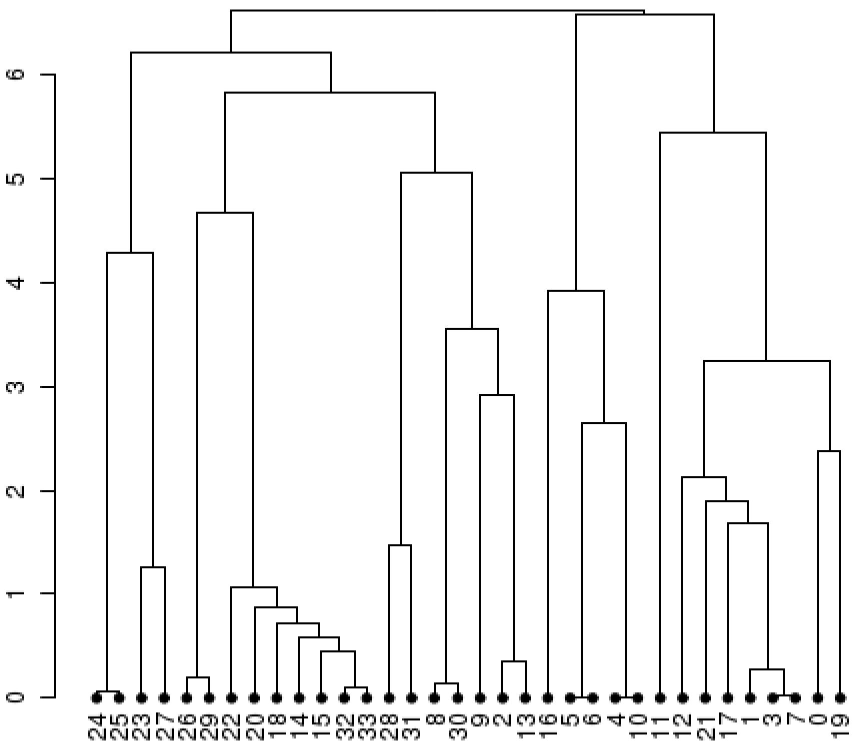
The general form of Newman’s algorithm is as follows:

1. ***Calculate betweenness scores for all edges of the network***
2. ***Find the edge with the highest betweenness value and remove it from the network***
3. ***Recalculate betweenness for all remaining edges***
4. ***Repeat from step (2)***

8 M. Girvan and M. E. J. Newman. Community structure in social and biological networks. Proceedings of the National Academy of Sciences of the United States of America, 99(12):7821–7826, June 2002.

9 M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. Physical Review E, 69(2):026113, Feb 2004.

The output of this process can be represented as a dendrogram depicting the successive splits of the network. The detection process can be stopped at any point the output community structure is judged to be satisfactory. An accurate way to determine if a network division is satisfactory is to use an appropriate evaluation metric. A metric that can carry out this task is the modularity metric. Modularity essentially indicates the extent to which a given community partition is characterized by high number of intra- community edges compared to inter-community ones. Essentially, the algorithm proceeds as long as network partitions with higher modularity are produced after edge removals. An appropriate modularity threshold is applied in order to identity the optimal community structure and the algorithm to terminate.



The key in order to achieve acceptable results is the recalculation step. After the removal of an interconnecting edge, the workload for the remaining edges standing between two communities is increased. The fewer are the remaining edges, the more dramatic becomes the increase. As a consequence a single betweenness calculation followed by the serial removal of edges in descending betweenness order could lead to the faulty removal of an edge and thus poor results for the algorithm.

Summing up, despite being a fairly successful and robust algorithm, Newman is computationally intensive and thus fails to keep up with the perpetual evolution in the field of community detection. This becomes more obvious when we have to deal with large data-sets or streaming data produced by modern web systems.

3.2 CiBC

Compared to other community detection algorithms applied in various scientific fields, CiBC was designed to fulfill the very specific task of identifying Web communities from a web server content, in order to improve the performance of CDNs.

CDN stands for Content Delivery Network. A content delivery network (CDN) is a large distributed system of servers deployed in multiple data centers all over the world. The main purpose of a CDN is to provide web content to end-users with high availability and high performance. CDNs serve a large fraction of the Internet content today, including web objects (text, graphics, URLs and scripts), downloadable objects (media files, software, documents), applications (e-commerce, portals), live streaming media, on-demand streaming media, and social networks more importantly social networks and web 2.0 applications.

The algorithm is based on a slightly different definition from what is traditionally considered as a network community. Usually, a community is defined as a subgraph for which each node has more edges to nodes of the same community than to nodes outside the community (strong community). A more flexible definition, called generalized community, is the following: a community is a sub- graph in which the sum of all node degrees within the community is larger than the sum of all node degrees towards the rest of the graph. Apart from the capability of identifying overlapping communities, CiBC has one more innovative characteristic, its hybrid nature, using both local and global graph's properties in order to accomplish its mission. Community detection is performed in three phases, with each phase including further steps.

In the first phase, the Betweeness Centrality (BC) is calculated for each node of the graph as shown in [10].[10] BC is a metric used to measure how "central" a node is in the graph. Last step before proceeding with phase two includes the sorting of the nodes of the graph by ascending BC value.

The second phase concentrates on the initialization of the cliques. This is achieved using an iterative procedure starting with the nodes with the lowest BC values. Although a high BC value may indicate that a node is central within a community, it can also be an indication of a node that is central within the graph, connecting different communities. Moreover, if we start with a node characterized by a high BC value, it is highly possible to end up with a single community that includes all nodes of the graph. In each iteration, if the currently-selected node *υ* is not assigned to any group yet, a new subset of the graph called clique is created. In this clique, we include all nodes that belong to the neighborhood of *υ*. Moreover, we further expand this clique using Bounded-BFS with typical depth value $\sqrt{N}$ (where *N* is the number of nodes). By applying this procedure, after the completion of all iterations, a set of groups (cliques) will be created. This fatefully leads to phase three of the algorithm.

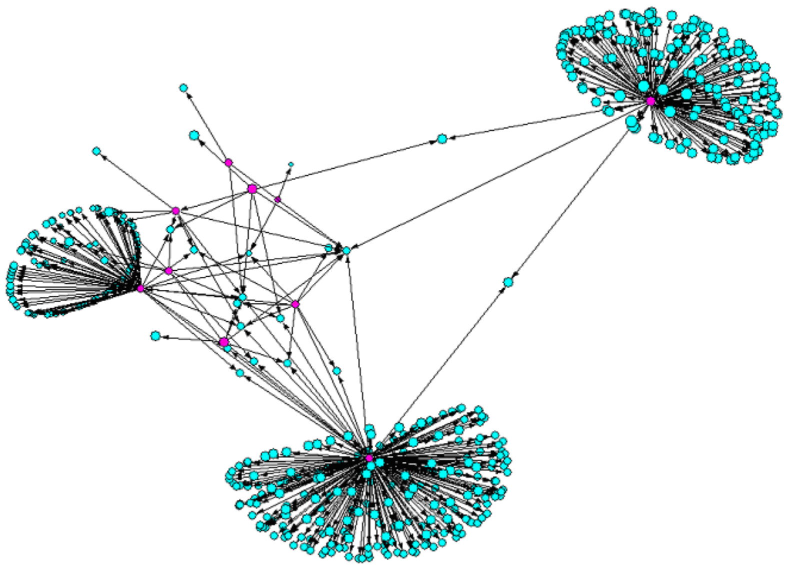
The large number of the created groups raises the need for some sort of minimization, in order to get the desired generalized community structure. This task is carried out by merging these groups through an iterative process. We define an $l \times l$ matrix B , where l refers to the number of the groups produced at phase two. Each element *B [i, j]*, with $i \neq j$ of the matrix corresponds to the number of edges that connect directly nodes assigned in group *i* to nodes assigned in group *j* . On the other hand, each element *B [i, j]* with *i = j* corresponds to the number of edges internal in group *i*. In each iteration, the pair of groups with maximum $\frac{B[i,j]}{B[i,i]}$ value is selected for merging and then the recalculation and repopulation of matrix B is required. The process terminates when there is no pair of groups with $\frac{B[i,j]}{B[i,i]} \geq 1$

10 S. Papadopoulos, A. Skusa, A. Vakali, Y. Kompatsiaris, and N. Wagner. Bridge bounding: A local approach for efficient community discovery in complex networks. Technical Report arXiv:0902.0871, Feb 2009.

3.3 *Bridge Bounding*

The authors of [10] introduce a local methodology for community detection, named Bridge Bounding. The algorithm initiates the community detection from a certain seed node and progressively expands the community trying to identify bridges , i.e. edges that act as community boundaries. The edge clustering coefficient is calculated for each edge, looking at the edge's neighborhood, and edges are characterized as bridges depending on whether their clustering coefficient exceeds a threshold. The method is local, has low complexity and allows the flexibility to detect individual communities. Additionally, the entire community structure of a network can be uncovered starting the algorithm at various unassigned seed nodes, till all nodes have been assigned to a community.

In order to identify a community around a seed node s the algorithm uses a flooding technique. Starting at node s , nodes in the neighborhood of s are gradually attached to the community if the following two conditions are satisfied: neighbor v does not belong to any other community and the edge connecting s to v is not a bridge (community boundary). The term bridge defines an edge connecting two nodes that are members of different communities. The steps described above are repeated for every node until no other node can be attached to the community. Repeating the same procedure for different nodes, inevitably leads to the discovery of the overall community structure of the graph.



In order to identify a community around a seed node s the algorithm uses a flooding technique. Starting at node s , nodes in the neighborhood of s are gradually attached to the community if the following two conditions are satisfied: neighbor v does not belong to any other community and the edge connecting s to v is not a bridge (community boundary). The term bridge defines an edge connecting two nodes that are members of different communities. The steps described above are repeated for every node until no other node can be attached to the community. Repeating the same procedure for different nodes, inevitably leads to the discovery of the overall community structure of the graph.

3.4 Fortunato's Algorithm

An interesting method for community detection appears in [11][11]. This algorithm is developed based on the observation that network communities may have overlaps, and, thus, algorithms should allow for the identification of overlapping communities. Based on this principle, a local algorithm is devised developing a community from a seed node and expanding around it. A community is identified as a subgraph that has a certain fitness. The authors provide an appropriate fitness function, whose calculation is based on the number of inter- and intra-community edges and a tunable parameter a . Starting at a node, at each iteration, the community is either expanded by a neighboring node that increases the community fitness, or shrinks by omitting a node if this action results in higher fitness for the community. The algorithm stops when the insertion of any neighboring node would lower the fitness of the community. This algorithm is local, and able to identify individual communities. The entire overlapping and hierarchical structure of complex networks can also be found. For a community G of the graph, the fitness fG is calculated as follows:

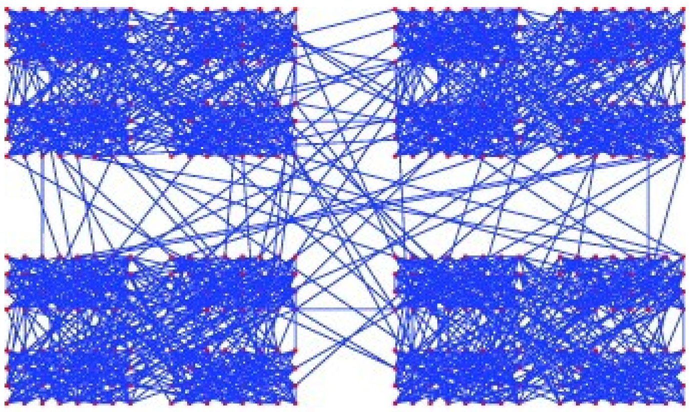
$$f_G = \frac{K_{in}^G}{(K_{in}^G + K_{out}^G)^{\alpha}}$$

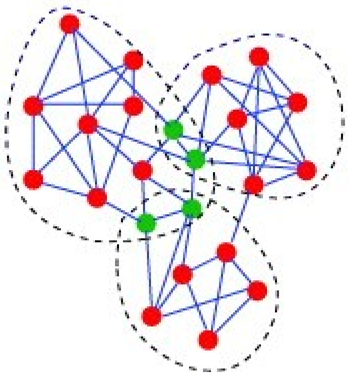
where K^G_{in} and K^G_{out} refer to the total internal and external degrees of community G respectively, and a is a positive real-valued parameter which controls the size of the community. As mentioned above, in order to reveal the entire community structure of a network, each node should belong to at least one community. To achieve this goal we apply the process summarized below:

1. ***Select at random node A***
2. ***Discover the natural community of node A***
3. ***Randomly select a node B that has not been assigned to a community***
4. ***Discover the natural community of B, by exploring all the candidate nodes regardless of whether they belong to other communities (allow for overlaps)***
5. ***Repeat from step (3)***

A major advantage of the above approach is the lower computational cost. In summary, Fortunato's algorithm is a modern and highly successful algorithm as it is revealed by its experimental results.

11 A. Lancichinetti, S. Fortunato, and J. Kertész. Detecting the overlapping and hierarchical community structure in complex networks. New Journal of Physics, 11(3):033015+, March 2009.





Εικόνα 12: A network exhibiting hierarchical structure.

3.5 Clique Percolation

Recognizing the importance of identification of overlapping community structures, Palla, Derenyi and Vicsek[12] introduced Clique Percolation in 2005. The method was applied in a variety of well established scientific disciplines extracting valuable information over network structures.

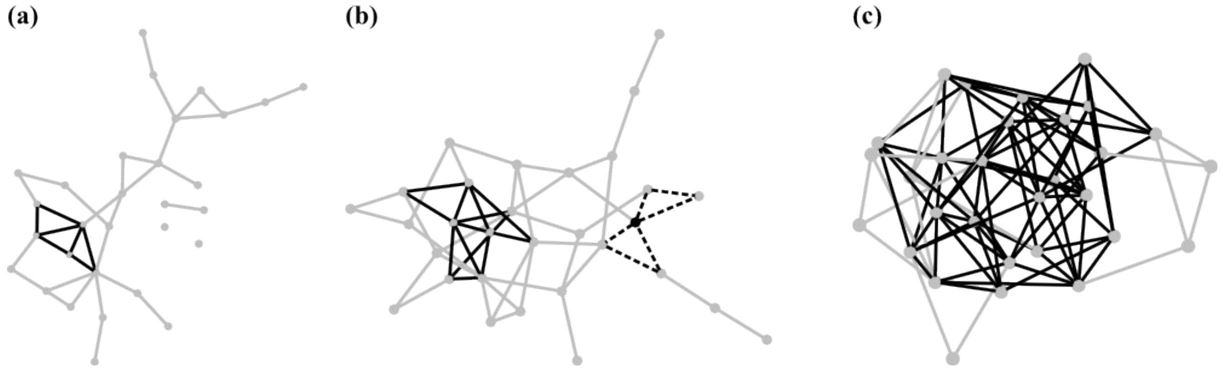
As the algorithm's name betokens, the method's cornerstone is the K-clique which correspond to a fully connected subgraph of K nodes. Two K-cliques are considered to be adjacent if they share K-1 nodes. A community is defined as the maximal union of K-cliques that can be reached from each other through a series of adjacent K-cliques.

The whole procedure can be implemented with the help of a K-clique template which is an object isomorphic to a complete graph of K nodes. Such a template can be placed onto any K-clique in the graph, and rolled to an adjacent K-clique by locating one of its nodes and keeping its other K-1 nodes fixed. Thus, a K-clique community of a network is a subgraph hat can be fully explored by rolling a k-Clique template in it.

The key in order to achieve acceptable results is the selection of the K parameter. Usually

12 I. Derenyi, G. Palla, and T. Vicsek. Clique Percolation in Random Networks.Physical Review Letters, 94(16):160–202, Apr 2005.

choosing K = 3 or K = 4 helps us extract valuable information, and currently these values of K have yielded, to our knowledge, the most relevant communities in practical applications. In our implementation K = 3 was chosen in order to maximize output communities' size and the possibility to successfully start the detection of the surrounding community from a random node.



Εικόνα 13: Examples of 3-clique percolation clusters on ER random graphs

4 Design and Implementation

In order to evaluate the performance and confirm the special dynamics that chaperon the previously discussed algorithms, a relatively complex system had to be developed as part of this thesis. The current chapter discusses the basic architecture of the developed system and highlights a series of special implementation issues.

4.1 System Architecture

The system that has been developed as part of this thesis had to meet a series of requirements. These requirements emanated from the need to be able to process a large number of data-sets with different characteristics originating from different sources. In case of synthetic data-sets which are produced by the appropriate component, graphs' information is stored on simple text files where in case of real world data-sets, friendship graphs' information is stored on a local database server after being fetched from the web.

In an effort to reduce the experiment time to the bare minimum, the system should also be able to take advantage of the latest multi-core processors as laboratory's resources include several cluster computers. This feature should allow us not only process different data-sets at the same time but also run single data-sets faster by paralleling independent sections of the previously mentioned community detection methods whereas possible. Moreover, considering that this thesis does not mark the end of this work, additional algorithms implemented in the future must be easily adapted into the system. Taking into consideration the previously set requirements we proceeded with building our system following a multiple layer architecture exploiting the capabilities of Java Object Oriented Language (OOL).

4.1.1 Synthetic Input API

The Synthetic Input API (S.I.A) is oriented to synthetic data manipulation and consists of five layers, each one providing information to the higher ones. Data-sets are stored in text files, with each line containing the id of a vertex and an id of another vertex connected to it, so our first concern is to retrieve this information. In a bottom to the top view, the first level we encounter is the physical level which is responsible for all the low level Input/Output operations regarding retrieving and writing data to files. After reading process has reached its end, output which is a list of doublets is provided to the higher level.

One layer higher a number of basic abstract models and the appropriate methodological frame for graphs representation are provided. This permits us to support different graph definitions as defined in chapter two, with the most important being the non-directed network and the directed network definitions. Based on this model, the system is now able to interpret information retrieved from the physical layer, in order to build network structures which will be used as input by the higher layer.

One layer higher, we find the Algorithms Layer which consists of the implementations of all the algorithms examined in chapter three. In this layer we also find a series of classes which inherit attributes of the basic graph models provided by the previous layer, extended with special requirements set from the community detection algorithms.

The fourth layer, consists of all appropriate components in order to evaluate the output community structure provided by the Algorithms level. The implemented measure for evaluation is Modularity, however the interfaces that constitute this layer permit easy measures adaptions in the future.

As mentioned earlier, the system was developed taking into account the resources of "Parallel and Distributed Computing Laboratory" and more importantly the availability of three cluster computers which had an invaluable contribution to the experiments, reducing the total time for processing a large number of synthetic networks. However, in order to exploit these multicore units concurrent programming techniques must be applied. Fortunately, java through its API provides a very extensive package for multi-threading development. Therefore, the role of the fifth layer is to provide a framework to support multiple executions at the same time. This is accomplished through two classes, one that represents a process and one that represents a batch of multiple instances of the first class.

At the top layer of our API's architecture we encounter the Graphical User Interface (GUI). This layer, thanks to a successful conceptual model allows users to organize their experiments and draw important conclusions exploiting the good informational awareness provided by the machine-user interface.

4.1.2 MySpace API

Unlike Synthetic Data API, MySpace API is oriented to real-world networks manipulation. Because of the special nature of the data and the restrictions imposed by the world wide web and the applications themselves MySpace API varies significantly from the SDI API as it is expected to operate on information which is fetched from the internet. Apart from the input, these two APIs also produce a different output. When dealing with synthetic data, the expected output includes several communities whose aggregation constitutes the whole graph. On the other hand, due to the enormous size that characterizes the underlying friendship graphs of real-world networks like MySpace, when dealing with real-world networks, the output consists only of the surrounding community of a selected vertex. This requires only a fraction of the friendship graph which can be acquired in parts while community detection progresses in a procedure which shares similarities with the flooding technique already known from computer networks.

We will now proceed with down to the top analysis of the API's architecture. Just like the Synthetic Data API, the first layer we encounter is the Physical Layer. The first thing that changes when dealing with real-world networks is the number of times we need to retrieve stored data. In the case of synthetic data we only need to access the disk once for every data-set allowing us to follow a simpler approach based on files. Moreover, data saving also takes place only once for storing the evaluation report for the under examination graph. On the other hand, MySpace API requires constant reading and writing data which are retrieved in real time from the web rendering files obsolete and forcing us to obvert to more complex solutions. The first solution we examined was storing all data connected to the detected community in the random access memory of the computer, a solution that was quickly abandoned due to the insufficient size of it, especially considering the need for uncovering multiple communities at the same time. Therefore we decided to proceed with using a database for the maintenance of our data choosing the well known MySQL database managements system. Summarizing, the Physical Layer consists of appropriate entity relationship model that represents the underlying friendship graph and all the classes that are connected with storing and retrieving tuples from the database.

On layer higher we find the component whose role is to mine information regarding MySpace's network's structure and population. Fortunately enough, in contrast with the social networks majority, information about the underlying graph and the profiles themselves are easily accessible via simple HTTP requests. Therefore, this layer is responsible for two tasks, managing HTTP requests and processing HTTP responses through a crawling process. In practice what happens is that higher levels as they progress their process of uncovering community structures constantly demand information without any concern whether they are already fetched and are available locally at the database. These requests are conveyed to the first two levels. If information is available the process can continue without any delay where in the case that this is not true, information are gathered from the web and stored at the database satisfying the initial request. At this point we must also mention that special care was taken in order take advantage of information already fetched for previous community detections. This feature saves us significant time when trying to uncover the natural surrounding community of the same vertex using different algorithms, or using the same algorithm with different parameters like the learning degree in the case of Fortunato's algorithm. Moreover, despite the vast proportions of MySpace and social networks in general, it is not uncommon, mainly due to the small-world property, to demand information regarding a profile already encountered during a detection started from another root vertex. It is important to understand that while it is very convenient base the detection process on information mined for previous detections, there is always the risk that a specific user has abandoned the network or has changed his friendship status by adding or removing friends. However, we have ascertained that at least for MySpace network users tend to change status with a relatively low rate. Therefore, choosing a good date-time threshold after which available information has to be updated or at least confirmed is a crucial task that can have a striking impact on the detected community's correctness. For the purposes of our experiment, and after evaluating different threshold values, we concluded that very few users change significantly their friendship status on the same day choosing this interval.

At the third layer of the API's architecture we encounter all the classes and interfaces that implement the different community detection algorithms. Of course, as we are doomed to be content to a very restricted fraction of the whole network we can only proceed with the implementations of algorithms that are characterized by a local perspective and lack any global subroutine. Therefore from the previously mentioned five community detection algorithms, the only ones that meet the criteria are Clique Percolation and Fortunato's Algorithm. While Bridge Bounding method at first site follows a local approach, it includes a global initial step only performed once, and is thus eliminated.

The last two upper layers include a simple task scheduler which is responsible for managing a queue of community detection instances and a simple Graphical User Interface which shares many similarities with the one developed for the Synthetic Data API.

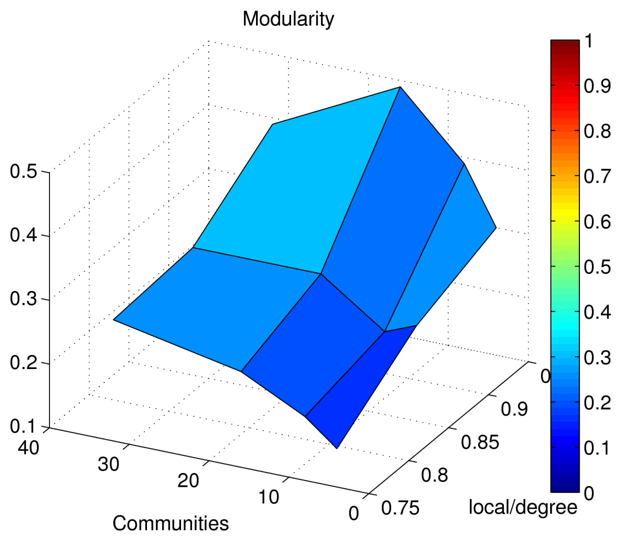
5 Experimental Evaluation on Synthetic Data

In this section we describe the experimental framework, namely the way benchmark graphs were created, the modularity metric used for the comparison of the algorithms, and finally the a comparative analysis of the algorithms' performance. We have created a variety of benchmark graphs with known community structure to test the accuracy of our algorithm. Benchmark graphs are essential in the testing of a community detection algorithm, since there is an apriori knowledge of the structure of the graph and thus one is able to accurately ascertain the accuracy of the algorithm. Our benchmark graphs were generated randomly given the following set of parameters: number of graph nodes N , number of communities Comm, node degree deg r ee, and finally ratio of intra-community edges to node degree local/degree . The parameters used for the creation of the benchmark graphs and their corresponding values are shown in the following table:

N	512, 1024
Comm	4, 8, 16, 32
Degree	10, 20, 30
Density	0.75, 0.85, 0.95

Πίνακας 2: Parameters used for the benchmark graphs.

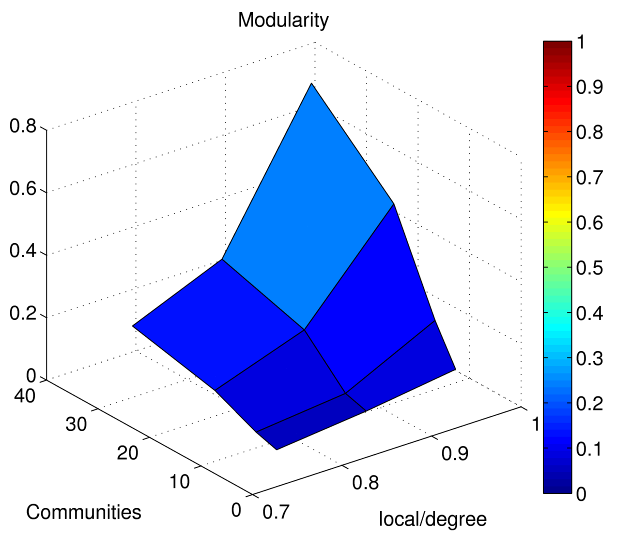
In order to compare the performance of the algorithms, we use the well established modularity metric. Modularity as mentioned earlier indicates the extent to which a given community partition is characterized by high number of intra-community edges compared to inter-community ones. If the number of intra-community edges is no better than random, we will get a modularity value close to 0, while modularity values approaching 1 indicate networks with strong community structure.



Εικόνα 14: Newman's algorithm

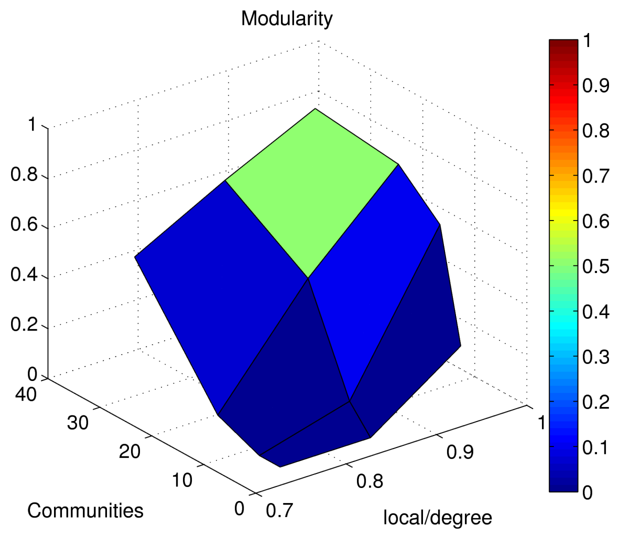
Figures show the performance of the four algorithms on the benchmark graphs we created with respect to modularity. Modularity is plotted against two parameters that were shown to play the most important role in the performance of the algorithms. For all algorithms these two parameters are the number of communities in the graph and the local/degree which is an indication of the density of the communities. The experimental results on the given benchmark graphs demonstrate that although Newman is a computationally intensive community finding algorithm, it is not very effective in identifying communities in the given graphs, and starts to be effective only when community density becomes very high.

Similarly, Bridge Bounding is mostly effective for graphs with very dense communities, where local degree approaches 0.9 and higher. However, Bridge Bounding (BB) is a local algorithm and requires very little time. We can thus conclude that BB can be safely used on graphs with very dense communities.



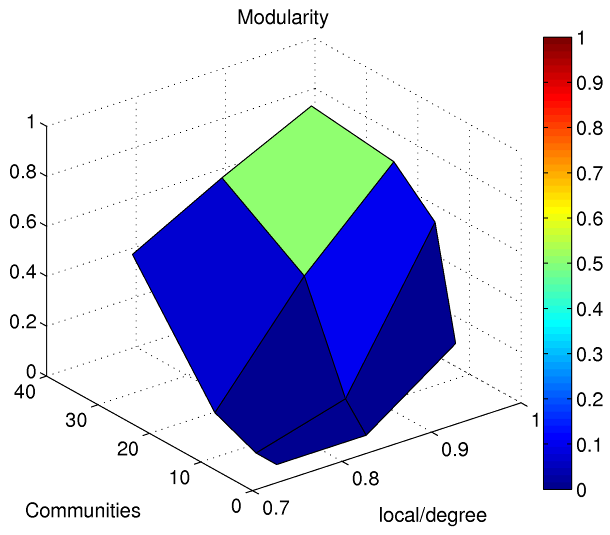
Εικόνα 15: Bridge bounding algorithm

CiBC seems to perform better than the previous two algorithms. When cliques are not expanded with BFS , CiBC seems to give decent results when community density exceeds 0.8 (local degree > 0.8), as shown in the following figure.



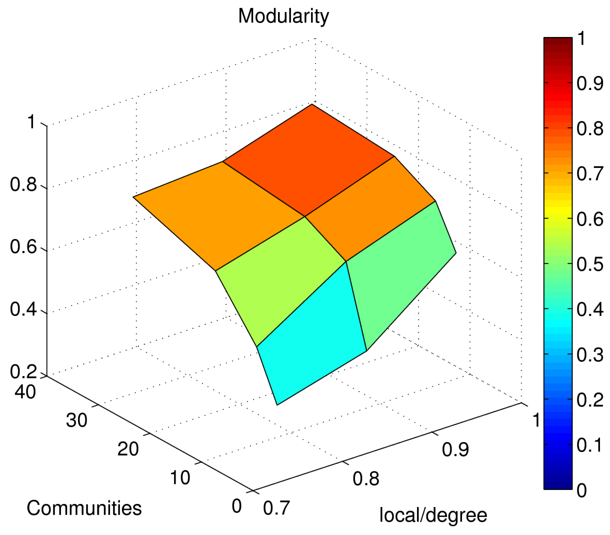
Εικόνα 16: CiBC algorithm with BFS (depth=2)

However, comparing CiBC to Bridge Bounding, we need to emphasize that CiBC is not a local algorithm as Bridge Bounding is, and its operation requires a global view of the entire graph. However, starting with individual nodes and merging them into larger communities, renders it far less computationally intensive compared to Newman which uses the inverse approach, namely, starts with the entire graph and splits it into communities by gradually removing edges. Using community merging, CiBC never reaches the point to manipulate the entire graph as a whole.

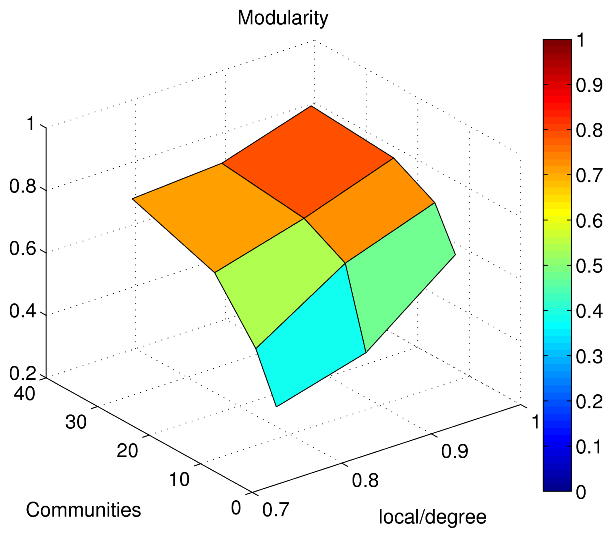


Εικόνα 17: CiBC algorithm without BFS

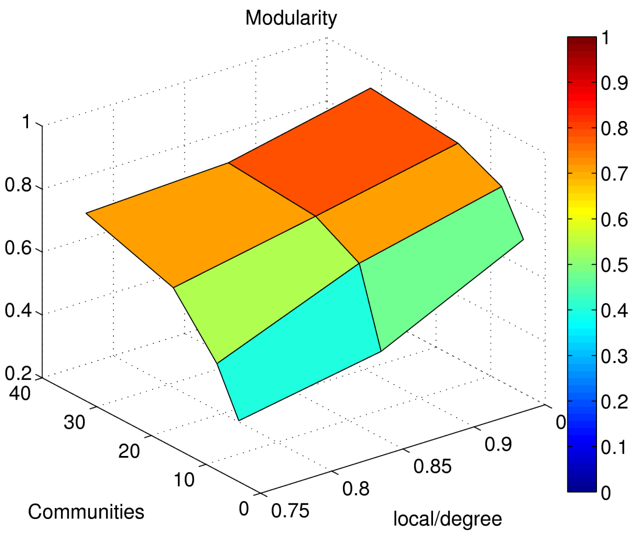
Figs. 18, 19 and 20 demonstrate Fortunatos performance for two different values of a , namely a =0.6, a = 0.8 and a =1.0. In all cases Fortunato outperforms its previous counterparts demonstrating its ability to identity communities even in graphs with community density local degree > 0.7. Furthermore, Fortunato is a local algorithm a little more computationally efficient compared to Bridge Bounding. The advantage of Fortunato is that the fitness function it utilized to expands a community from a seed node is directly related to the density of the community. Thus, Fortunato is the best of all candidates and could be safely used to identify communities in various application domains. The fact that it is a local algorithm makes it easy to apply in situations where a global view of the network is not easy to obtain. Finally, we have to mention that Fortunatos algorithm allows for further improvement.



Εικόνα 18: Fortunato's algorithm with a = 0.6



Εικόνα 19: Fortunato's algorithm with a = 0.8



Εικόνα 20: Fortunato's algorithm with a = 1.0

5.1 Summarizing Evaluation

We compared the performance of four community detection algorithms, each one following a different approach. The performance of the algorithms was demonstrated on a variety of benchmark graphs with known community structure. In general, based on the above observations we can derive the conclusion that although community finding is a problems that appears in many different version and exhibits a richness of solutions, there is still plenty of room for improvement of existing solutions and for the derivation of new ones that would allow the manipulation of graphs from various new and emerging application domains like social networks and other types of collaborative environments.

There is an emerging need to devise community detection algorithms for dynamic graphs, i.e. graphs whose structure evolved over time. Such algorithms would be able to capture for example the dynamic evolution of social networks.

6 Applying Community Detection on Real-world Networks

As mentioned earlier, in recent years, social networking[13] has been driving a dramatic evolution due to the increasing use of Web 2.0. The major impact of Online Social Networks (OSNs), such as MySpace[14] and Facebook[15], on Information Technology paved the way for popular Internet applications and attracted hundreds of million of users. This evolution also attracted the interest of the academic community and prompted several researchers to examine population synthesis, user activity, evolution, and structure, of popular OSNs[16] [17] [18] [19] [20].

First efforts on this direction included the analysis of the friendship relations of popular social networking sites from data collected online. Some studies focused on the evolution of user population based on data collected over a period of time [4], [19]. They also tried to capture the evolution of the friendship relations over time, considering this as the main element that characterize activity in social networks. In [18] authors aim to capture the decline in use activity of the MySpace OSN. Their research is based on the collection and analysis of a large number of user profiles and provides significant evidence on the activity of MySpace users from several different points of view. The authors claim that cap- turing decline in the activity of an OSN is a nontrivial task since network administrators tend to hide such events from public view, as much as possible, in order not to influence other members.

In this paper, we collect a large number of profiles from the MySpace OSN and try to characterize the way the synthesis of the population changed over time, the friendship graph, and the way user activity evolves in the network. To this purpose one million user profiles were randomly collected over a period of time, and they were analyzed accordingly. One of the important features of MySpace, is the fact that it enables researches to collect user profiles along with all accompanied attributes and to analyze them, deriving interesting results for the network.

Apart from characterizing user activity based on collective analysis of individual profiles, we go one step further, trying to derive interesting results by extracting a large number of communities from the underlying social graph, using the two previously analyzed community detection algorithms, the Fortunato et al.[21], and the Clique Percolation[22], [23] algorithms.

We performed an empirical study of the MySpace OSN, collecting not only a large number of

13 B. Buter, N. Dijkshoorn, D. Modolo, Q. Nguyen, S. van Noort, B. van de Poel, A. Ali, and A. Salah1. Explorative visualization and analysis of a social network for arts: The case of deviantart. Journal of Convergence Volume, 2(1), 2011.
14 MySpace. http://www.myspace.com.
15 Facebook. http://www.facebook.com.
16 Y.-Y. Ahn, S. Han, H. Kwak, M. S., and H. Jeong. Analysis of Topological Characteristics of Huge Online Social Networking Services. May 2007.
17 W. W. Mojtaba Torkjazi, Reza Rejaie. Hot Today, Gone Tomorrow: On the Migration of MySpace Users. WOSN'09, Barcelona, Spain, pages 43–48, August 17 2009.
18 J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. August 2005.
19 R. Kumar, J. Novak, and A. Tomkins. Structure and the Evolution of Online Social Networks. August 2006.
20 G. Pallis, D. Zeinalipour-Yazti, and M. D. Dikaiakos. Online Social Networks: Status and Trends.New directions in Web Data Management, 1, 2011.
21 A. Lancichinetti and S. Fortunato. Community detection algorithms: a comparative analysis.Physical Review E, 80(5 Pt 2):056117, Sep 2009.
22 S. E. Schaeffer. Graph clustering.Computer Science Review, 1(1):27 – 64, 2007.

user profiles in order to characterize population synthesis, evolution in user population, and user activity, but also extracting a large number of community structures in order to analyze the underlying social network graph and to correlate characteristics and activity of users participating in communities. Two local algorithms were used to extract communities, namely, the Fortunato and the Clique Percolation methods. The main conclusions can be summarizes as follows: Population synthesis changed over time, as the difference between public and private profiles shrunk, and invalid profiled overpassed first private ones and subsequently the public ones. Use activity was shown to be diminishing, while the number of new users joining the network remains high. Drawing safe conclusions regarding MySpace was based on the extraction of a combination of data for every single profile, including among others, the status, the Member Since and Last Login dates, the number of friends. Indeed, these information alone had been very enlightening helping us to shed light to all crucial questions mentioned before.

Based on the analysis of the communities, important conclusions were drawn on the size of the communities that are formed in the network, but also in the community density as shown by the number of edges among participating member. As an important result, strong correlation was demonstrated in the Last Login date among members of the same community, and other attributes like the number of friends. We draw the conclusion that participation in strong communities inhibits users from abandoning the OSN in order to migrate to other more popular networks that emerge at times. As a last observation, we notice that members that abandoned MySpace shortly after their account creation (the so-called Tourists), have very low connectivity and thus they do not participate in communities.

The remaining of the paper is organized as follows: In section II, we present the methodology we followed to collect the profiles used in our experiments. In section III, we analyze population evolution and user activity based on the collected profiles. The community detection algorithms we use are briefly presented in Section IV, and the experimental results related to the community analysis of the MySpace OSN are provided in Section V. Finally, we conclude in Section VI.

6.1 Profile Collection and Methodology

MySpace is a popular OSN with several hundreds of millions members [15]. MySpace allocates numeric user IDs in a sequential fashion. This claim was documented in [18] which collected and analyzed a large number of user profiles in February 2009. Another important characteristic of MySpace that facilitates its analysis, is that user profiles are accessible via HTTP requests, and unlike other social networks, like for example Facebook, one does not need to rely on the availability of enormous static datasets, released periodically by the company. A user, writing his own scripts is able to dynamically collect random user profiles according to its needs. For a certain numeric user ID, the corresponding profile is easily accessed through an http request with the following URL http://www.myspace.com/ID. The file downloaded from this URL provides all the available information for the user with the specified ID. Using html parsing to preprocess the downloaded profiles we extracted all necessary attributes for our research.

Each user profile records a number of information regarding the user. Apart from the numeric user ID, there is a Status attribute with values in {Public, Private, Invalid}. Other at- tributes are the Alias, the number of friends, and the number of views. Furthermore, for each profile the following dates are recorded, Member Since date, which is the date a user joined the OSN, and the Last Login date, which is the most recent date a user logged in the network. We verified that a user’s Last Login is indeed updated each time a user logs into its MySpace account. Last Login and Member Since information is available only for public profiles. The relative age of user accounts can be also inferred by the user ID since, as we already mentioned, user IDs are monotonically assigned, thus a smaller ID has been created earlier in time compared to a larger one. Finally, for each profile we recorder the date it was fetched (Fetch Date) since the profiles were collected over a period of several days.

We collected a total of 999.937 user profiles over a period of one month, from the 2nd to the 28th of September 2011. To summarize, for each user profile we retrieved the following attributes:

- ***ID***
- ***Status: Public, Private or Invalid***
- ***Alias***
- ***Number of friends***
- ***Number of views***
- ***Last login date***
- ***Member since date***
- ***Profile fetch date***

Based on Status (Public, Private, Invalid) the user population has the following composition:

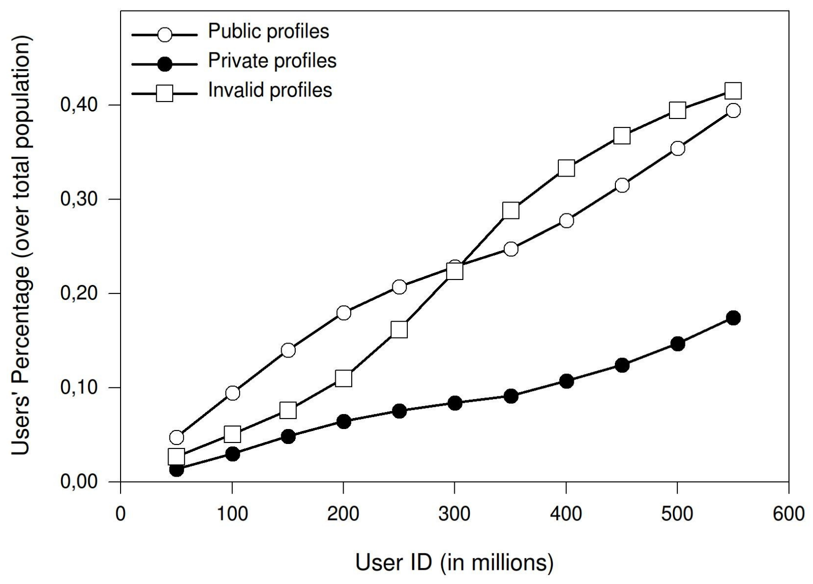
Status	Fetched Profiles
Public	403183
Private	177664
Invalid	419090
Total	999937

The first indications regarding population's composition are confirming our assumptions based on previous work [16]. Out of 999.937 randomly selected profiles, 419.090 were found to be invalid, corresponding to 41.91% of our sample population and allowing to safely conclude that a large number of MySpace users abandoned the network. The remaining 580.847 profiles (or 58.09% of the total) is shared between public and private, with public profiles being 403.183, corresponding to 40.32% for the sampled population, while private profiles are only 177.664, corresponding to 17.76% of the sampled population.

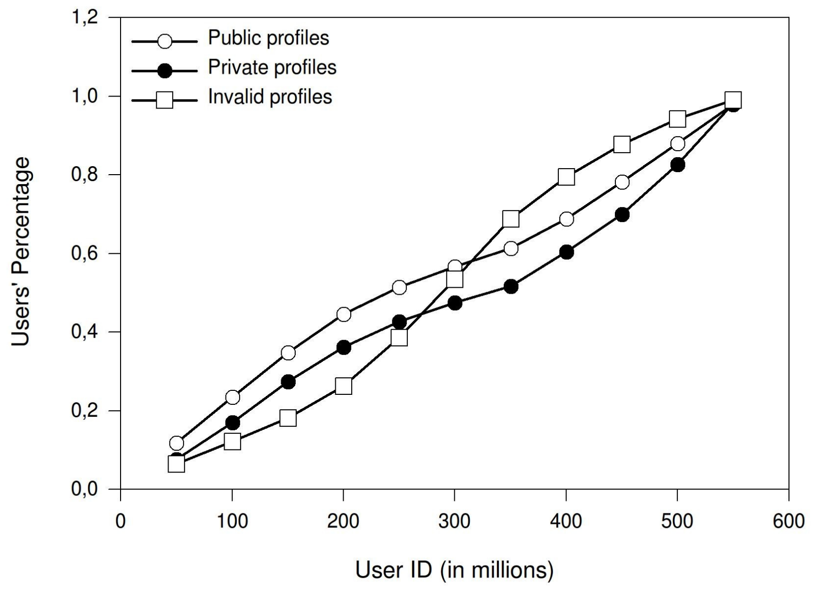
Compared to [16] which collected and analyzed a large number of MySpace user profiles in February 2009, at the time, the largest allocated user ID was 455 millions while at the time of our experiments the largest allocated user ID was 650 millions. Thus in a period of one year and seven months 200 million new MySpace user accounts were created, indicating that while MySpace users tend to abandon the OSN, the arrival of new users continues at a rate of approximately 348.000 new user accounts per day.

6.2 Dynamics of the underlying network

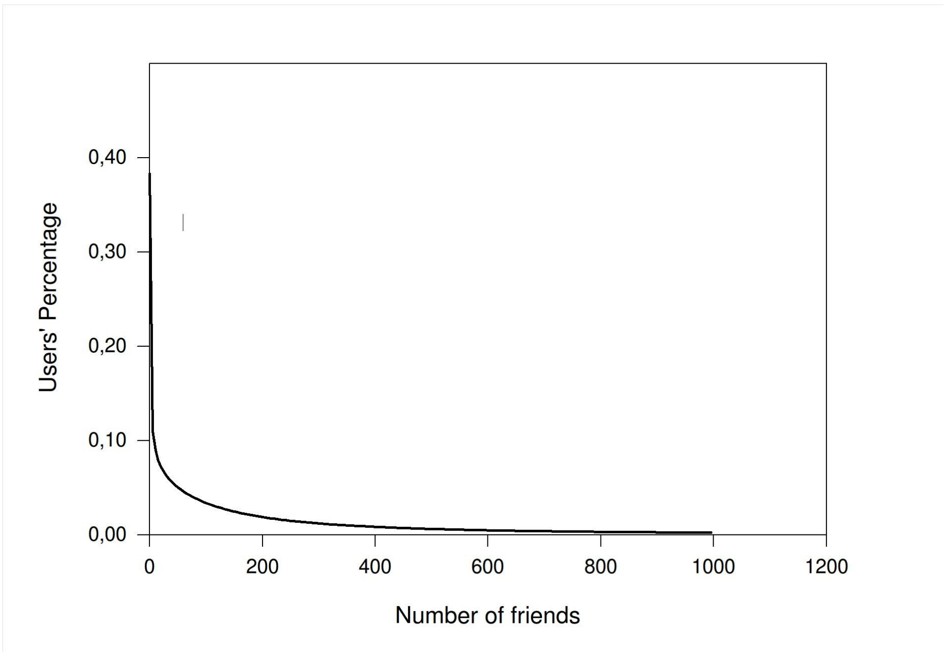
For all recorded IDs we followed the change in the synthesis of the population based on the percentage of profile status over time. Figures 21 and 22 present this information. Figure 21 shows the CDF (Cumulative Distribution Function) for each different type of profile (Public vs Private vs Invalid), while Figure 22 shows the CDF for each profile type as a percentage of the total population. These figures show the distribution of each group of profiles across the entire range of user IDs. The results indicate that for a large initial portion of the ID space, the percentage of public profiles was higher than that of the private profiles, while towards the end of the ID space this difference is vanishing. The results are even more noteworthy if we take into consideration Invalid profiles. While in the beginning of the ID space the percentage of Invalid profiles was lower compared to that of Private and Public ones, it increased over time and at some point, clearly shown in the figures, Invalid profiles overpassed Private ones, and a little later they also exceeded Private profiles. This is an evidence that users either abandoned MySpace OSN and migrated to other OSNs that gained popularity at the given time period, or OSN administrators aggressively removed profiles for violation of use.



Εικόνα 21: CDF of user ID based on profile status over total population.



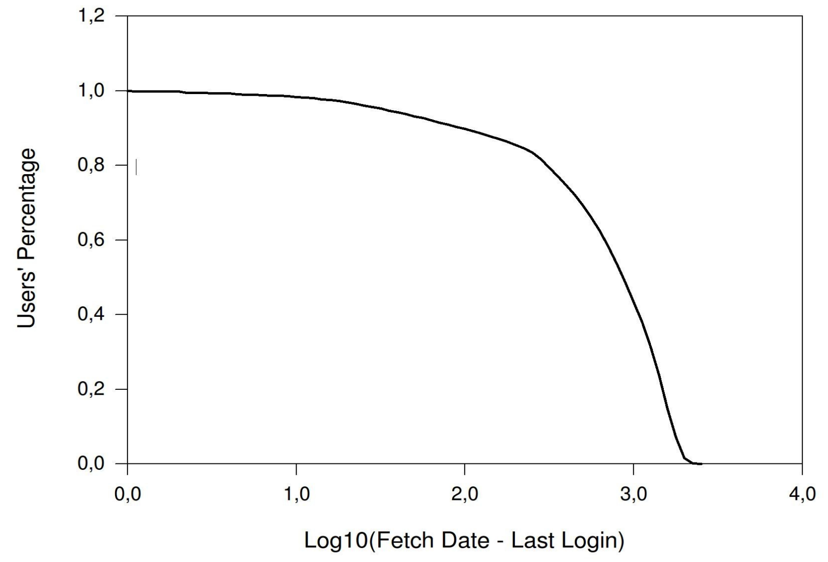
Εικόνα 22: Synthesis of user population. CDF of user ID based on profile status.



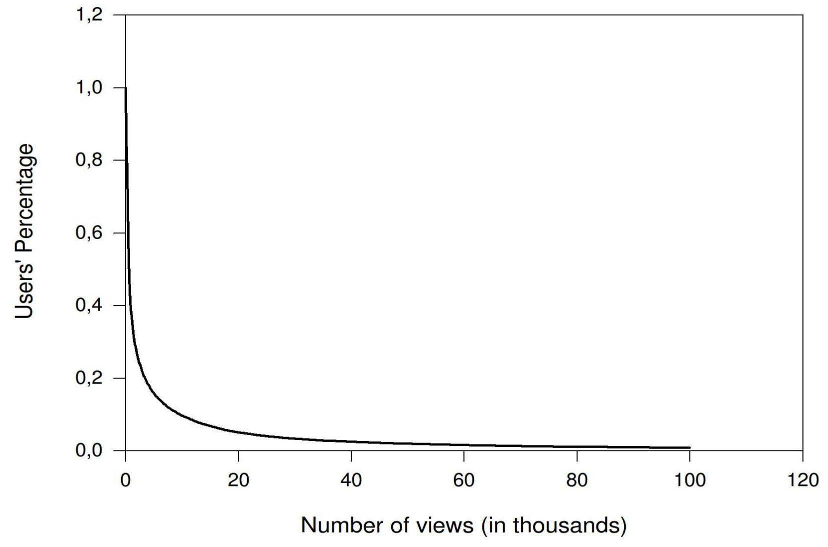
Εικόνα 23: CDF of number of friends.

Following the population synthesis based on ID Status, we approached the term activity in three different ways. The easiest way to determine the level of activity of a specific user is to examine its number of friends. In this context, a user that counts, for example, one hundred friends is considered to be more active than a user with a smaller number of friends. Figure 23 presents the CDF of IDs based on number of friends. From this figure we notice that a large number of MySpace users have a small number of friends. We could mention that about 5% of users have more that 100 friends. The second way to characterize activity is based on the difference, expressed in days, between the date that the user had logged in for the last time (Last Login) and the date its profile was fetched for the experiments (Fetch Date). Figure 24 depicts the CCDF of the difference in days between a users Last Login and the date his profile was fetched (Fetch Date). According to our statistics only 1.8% of profiles did actually login, up to ten days before the fetch process. On the other hand, 8,3% were the users that logged in up to one hundred days before profile fetching took place. This leaves us with almost 90% of the profiles not having logged in for more than one hundred days.

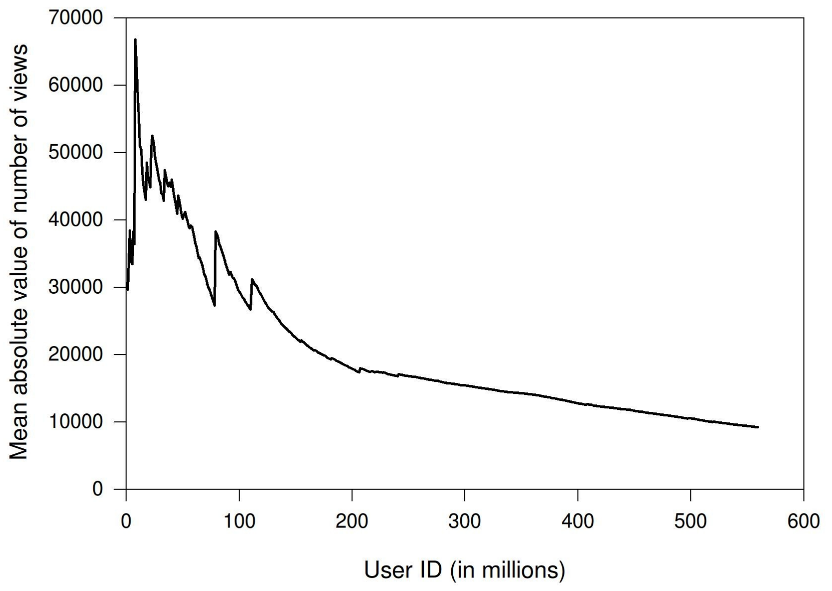
Another way to monitor the activity of users is by analyzing the number of profile views. The results are shown in Figures 25 and 26. Subsequently, we analyze a special category of users, the so-called Tourists [17]. The basic characteristic of tourists is the very small time-span between the date they created their account and the date they logged in to the system for the last time. As shown in Figure 27, about 33% percent of the users abandoned the network between one to ten days after they created their account. Moreover, another 13% percent has been found to be active for a period of ten to one hundred days. Practically this means that these users had a short stay in the network.



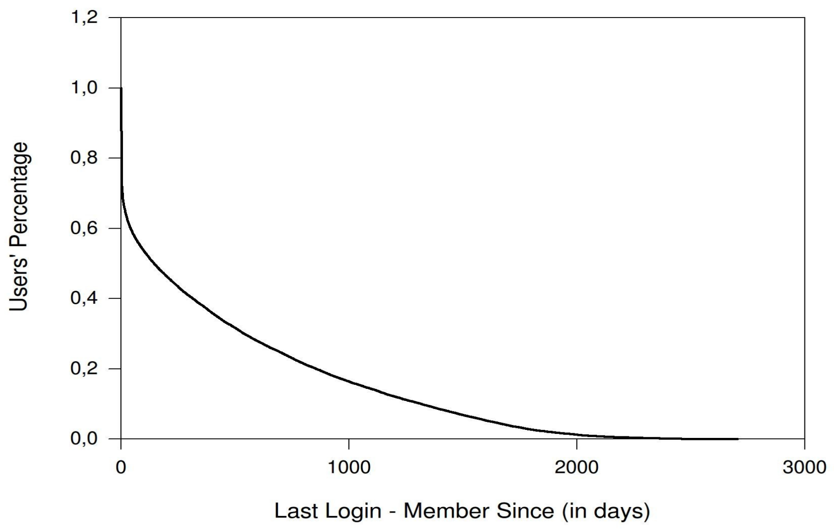
Εικόνα 24: CCDF of (Fetch Date - Last Login) for public profiles.



Εικόνα 25: CDF of profile number of views.

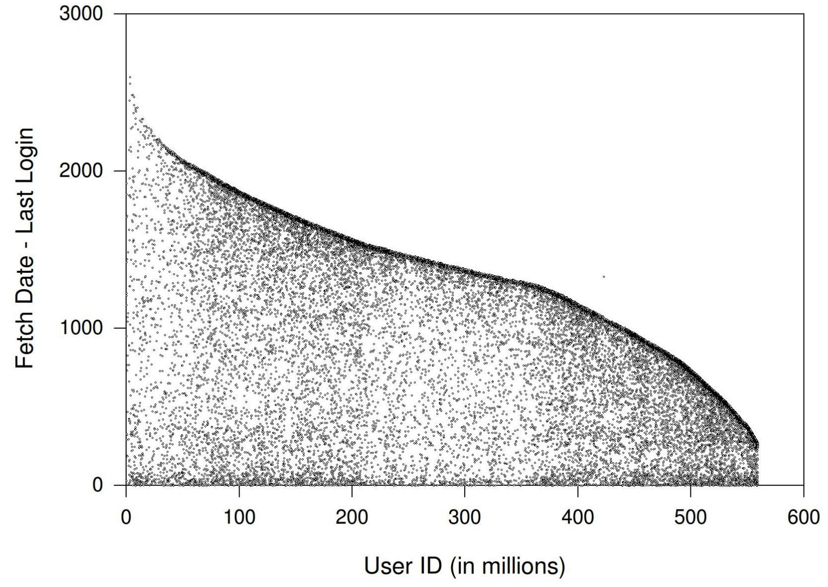


Εικόνα 26: Mean value of profiles number of views for profiles with ID less than X

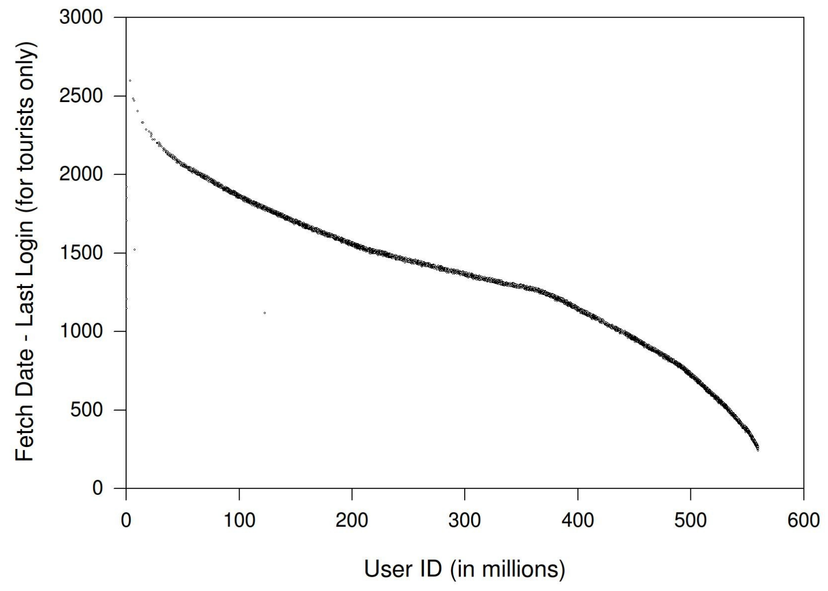


Εικόνα 27: CDF of (Last Login - Member Since), i.e. number of days profiles are active.

As a last indication for user activity, we study the relationship between a user ID and the users Last Login date, trying to collect evidence on the activity of users compared to account creation time. Figure 28 is a scattered plot presenting the Fetch date - Last login for Public profiles. The monotonic decrease of this plot is understandable and the appearance of an clear edge at the top is fully explained by the existence of Tourists, whose Last Login occurred shortly after their account creation time. Thus, in addition to Figure 27, the sharp edge isolated in Figure 29 also shows the distribution of Tourists over time.



Εικόνα 28: (Fetch Date - Last Login) for all profiles.



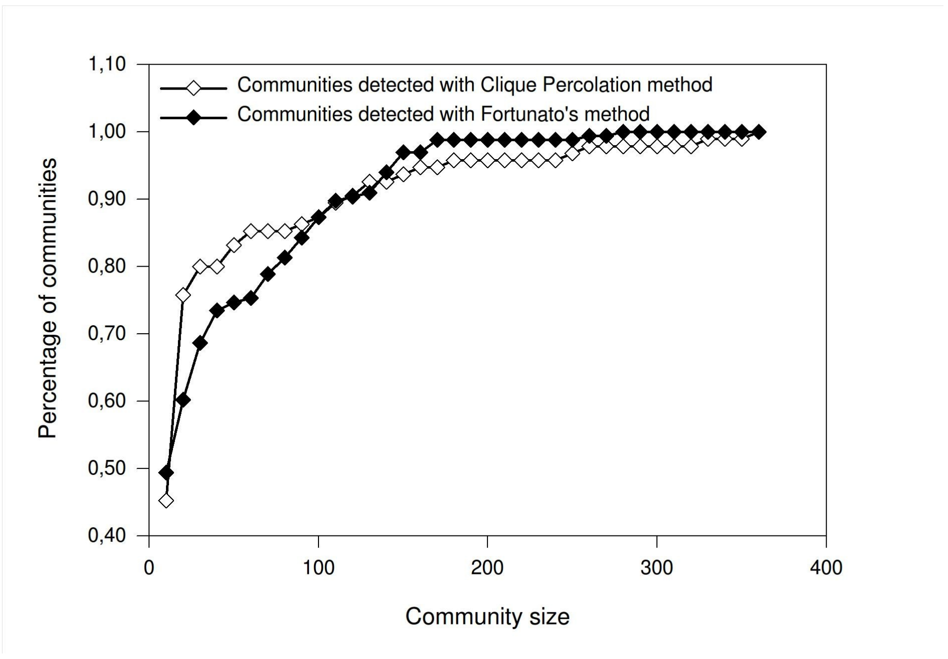
Εικόνα 29: (Fetch Date - Last Login) for Tourists only.

6.3 Experimental results

Using Fortunato's algorithm to extract communities, we managed to successfully identify 171 different communities. A total of 6137 different profiles belonged to these communities, thus the average number of members per community is about 37. In order to extract the 171 successfully detected communities, we had to fetch a large number of nodes surrounding the communities. These were 18259 distinct nodes. Using the Clique Percolation algorithm we identified a total of 201 communities. The total number of profiles that belonged to the communities is 3576, thus an average of about 18 members per community. In order to extract the 201 successfully detected communities, we had to fetch 176577 distinct nodes. Thus to summarize:

	Fortunato	Clique Percolation
Communities successfully detected	171	201
Total number of community members	6132	3576
Average members in each community	36.718	17.791
Total number of profiles fetched	18259	176577

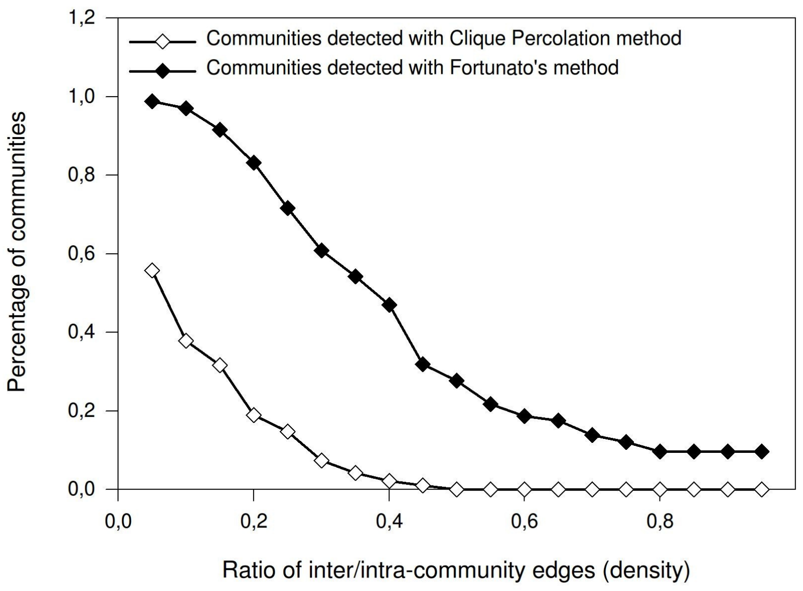
Πίνακας 3: Parameters of the revealed community structures.



Εικόνα 30: CDF of community size.

A first question we are trying to answer is “What type of communities do nodes form?” in the MySpace OSN. We proceeded with analyzing the surrounding community of a series of randomly selected profiles identified with the two algorithms, Fortunato’s (with a=1) and Clique Percolation (with K=3). Figure 30 shows the CDF of the community sizes identified with the two methods. It is interesting to mention that Fortunato’s algorithm identifies larger communities com- pared to the Clique Percolation method. Figure 31 depicts the CDF of the community density for the two methods. Density is defined as the ration of the number of intra- to inter-community edges (mentioned in the figure as short/ total links ratio). The more this ratio approaches one, the more the specific community is dense and thus tends to be an independent community structure.

As shown in the plot, in case of Clique Percolation, 60% of the detected communities have a greater than 0.05 short/ total ratio value. For density values greater than 0.10, percentage drops to about 40%. At the same time, members of communities detected by Fortunato's method, seem to be more closely connected achieving density values as high as 0.80. Moreover, almost half of the detected communities are characterized by a very satisfying 0.40 density ratio. Clique Percolation's Achilles' heel comes as a result of the conditions that needs to be met in order to be possible for a node to participate into a community. A node's participation in an adjacent triangle alone is sufficient, regardless of the risk it carries to have disproportionately more edges leading to nodes outside the community. Apart from the qualitative difference that density represents, output also differs in terms of communities' size. Although difference is not as striking as in the case of density, we can still observe a difference in small to medium communities, counting a few dozens nodes as shown in Figure 31.



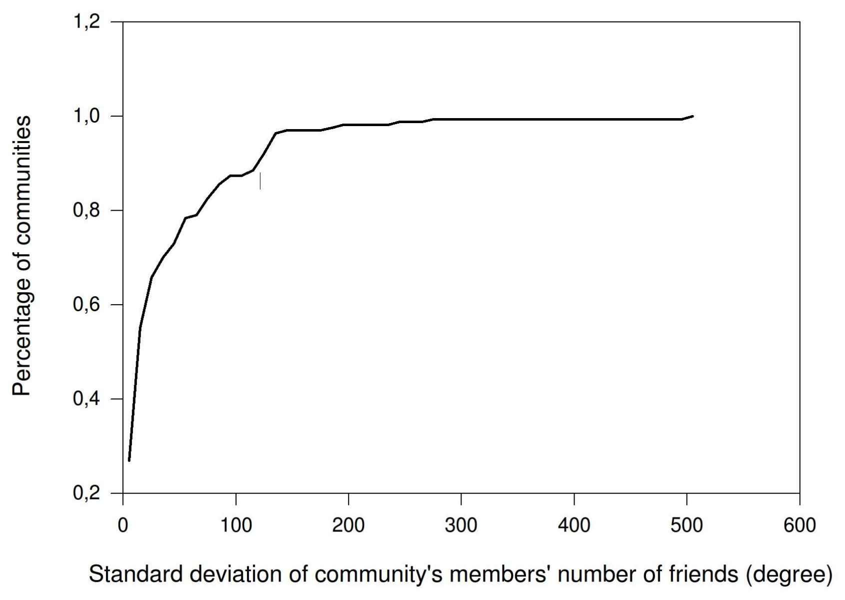
Εικόνα 31: CDF of community density

The next, less trivial questions we are trying to answer are "Do inactive users tend to abandon MySpace individually or the same tendency exists for their surrounding community?" and "Do tourists participate in community structures?". Taking into account the previously mentioned differences and considering the fact that Clique Percolation demands more information to be fetched in order to reveal the surrounding community of a node, we continued our analysis based on Fortunato's results.

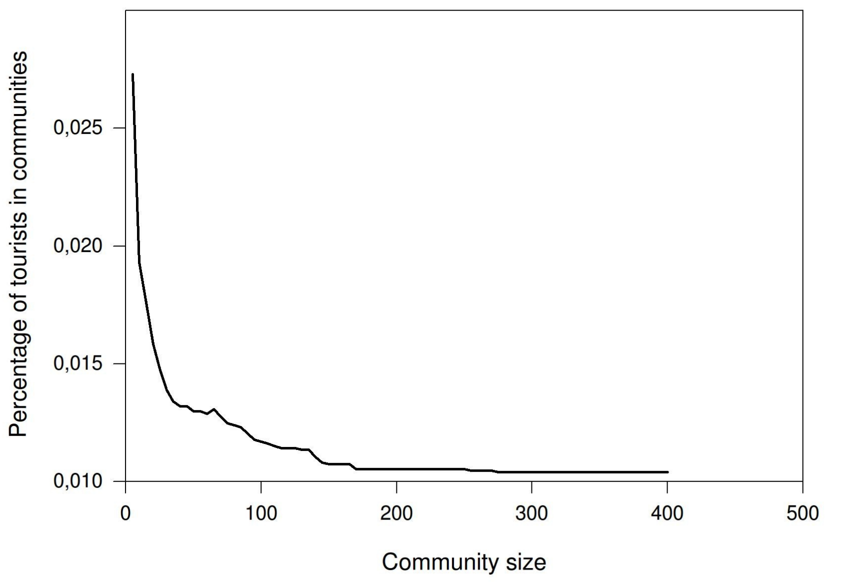
In order to answer the first question we focus our efforts into locating members that have not given any signs of life for more than one hundred days. According to the collected data, the majority of community members is found to be satisfyingly active. It is characteristic that more than 85% of the communities consist of almost 90% of users that showed their presence in the network at least once in the last one hundred days. The vast majority of the inactive users are either tourists whose short stay period did not allow them to get connected with other profiles, or users with a very high degree. Both types are not favored by the different community definitions nor by the majority of the existing community detection methods which are operating by trying to maximize community's density and are thus eliminated. This can be also confirmed by the low to moderate average members' degree which characterizes the plurality of the communities. More interestingly, our revealed community structures show a significant homogeneity in members' level of activity. Impressively, the members of the detected communities show a significantly low dispersion over the total degree. The maximum standard deviation value is lower than 150. Moreover 60% of the communities have a standard deviation lower than 20. When percentage increases by another 20%, standard deviation increases by

30. In conjunction with the previous observation this demonstrates an important correlation between the nodes. Although we have only indications, all information provided above permits us to make the assumption that the participation of a user in a community, inhibits him to abandon MySpace, regardless of the existence of more attractive social networks.

Concerning the question “Do tourists participate in community structures?”, Figure 33 clearly shows the very low participation of tourists in communities. Users that have been active for such a small period of time usually have a very low number of friends and are not favored by the different community definitions nor by the majority of the existing community detection methods which are operating by trying to maximize community’s density.



Εικόνα 32: Standard deviation of communities member number of friends.



Εικόνα 33: Percentage of Tourists in communities.

6.4 Conclusion

We performed an empirical study of the MySpace OSN, collecting not only a large number of user profiles in order to characterize population synthesis, evolution in user population, and user activity, but also extracting a large number of community structures in order to analyze the underlying social network graph and to correlate characteristics and activity of users participating in communities. Two local algorithms were used to extract communities, namely, the Fortunato and the Clique Percolation methods.

The main conclusions can be summarizes as follows: Population synthesis changed over time, as the difference between public and private profiles shrunk, and invalid profiled over- passed first private ones and subsequently the public ones. Use activity was shown to be diminishing, while the number of new users joining the network remains high.

Based on the analysis of the communities, important conclusions were drawn on the size of the communities that are formed in the network, but also in the community density as shown by the number of edges among participating member. As a last and most important result, strong correlation was demonstrated in the Last Login date among members of the same community, and other attributes like the number of friends. We draw the conclusion that participation in strong communities inhibits users from abandoning the OSN in order to migrate to other more popular networks that emerge at times. As a last observation, we notice that members that abandoned MySpace shortly after their account creation (the so- called Tourists), have very low connectivity and thus they do not participate in communities.

7 Thesis Conclusions

The performed analysis on the redefined problem of community detection and its applications proved to by quite successful. By applying a series of state of the art community detection algorithms each one following a different approach, both on benchmark graphs and real-world data we were able to draw safe conclusions regarding the methods' strengths and weaknesses. Algorithms proved to be more or less robust when dealing with the synthetic networks, however they struggled in detecting the surrounding community of a vertex when applied in real data-sets leading to very poor results. Although literature exhibits a richness of solutions, a significant void was recognized suggesting a delay in adapting into the new parameters that chaperon the modern social networks and the revolution that takes place in the domain of the social media. Motivated by this observation we took one step further in an effort to conclude into the crucial parameters that leads to the inadequacy of the plurality of today's community detection algorithms by examining the underlying structure of MySpace social network.

The most important observed characteristic of MySpace and hence of all social networks and collaborative environments has to be its tendency to evolve. Just like human beings or services, network appeared to have a cycle of life with its population synthesis changing over time. In the same direction, user activity was shown to be diminishing while the number of new users joining the network remained high. Both observations translate into significant changes on the underlying friendship graph over time, with new edges being added and removed constantly, information that has to be assimilated by the clustering methods.

Based on the analysis of the retrieved communities, important conclusions were also drawn on the size of the communities that constitute the network. Size-wise community structures proved to be decent allowing algorithms to proceed their operation enhancing their situational awareness. This is not very intuitive however becomes clearer if we recall the measures that algorithms base their operation on. It is proven that early additions that take place in the early stages of the detection process rely on significantly less information hiding the risk of misclassified nodes.

On the other hand although strong correlation was demonstrated on the number of friends among members of the same community, the revealed communities are characterized by poor density demanding much higher resolution from the algorithms, a irreplaceable quality in order to decide whether a vertex participates or not into the under detection community.

All the above show that there is still plenty of room for improvement of existing solutions and for the derivation of new ones that would be able to capture the dynamic evolution that characterizes graphs from various new and emerging application domains like social networks and other types of web 2.0 applications.

8 Source Code Appendix

8.1 Benchmark Graph Generator

```
import java.io.File;
import java.io.FileWriter;

public class SimpleGraphGenerator
{
    static int getDegree(double d)
    {
      int res = (int)Math.floor(d);
      if (Math.random() < d - res) res++;
      return res;
    }

    static int randomRange(int from, int to, int excl)
    {
      int res;
      to++;
      double prob;
      do
      {
          prob = Math.random();
          res = from + (int)(prob*(to-from));
      }
      while (prob == 1 || res == excl);
      return res;
    }

    public static void main(String []args)
    {
      //nr of peers, nr of communities, short links, long links
      if (args.length < 4)
      {
          System.err.println("Params: nr of peers, nr of communities, short 
links, long link prob");
          return;
      }
      try
      {
          int nrOfPeers = Integer.parseInt(args[0]);
          int nrOfComs = Integer.parseInt(args[1]);
          double SLprob = Double.parseDouble(args[2]);
          double LLprob = Double.parseDouble(args[3]);
          FileWriter out = new FileWriter(new File("simple-
graph_"+nrOfPeers+"_"+nrOfComs+"_"+SLprob+"_"+LLprob+".txt"), false);

          SLprob /= 2;
          LLprob /= 2;
          double []pops = new double[nrOfComs];
          for (int i = 0; i < pops.length; i++)
            //pops[i] = 0.5+Math.random()/2;
            pops[i] = 1;
          double sum = 0;
          for (int i = 0; i < pops.length; i++)
            sum += pops[i];
          for (int i = 0; i < pops.length; i++)
            pops[i] = (int)(nrOfPeers*(pops[i]/sum));
          sum = 0;
          for (int i = 0; i < pops.length; i++)
          {
            System.err.println(sum+" -> "+(sum+pops[i]));
            sum += pops[i];
          }

          for (int i = 0; i < nrOfPeers; i++)
          {
            int Cfrom = 0, Cto = 0;
            for (int j = 0; j < pops.length; j++)
            {
                if (Cfrom + pops[j] > i)
                {
                  Cto = Cfrom + (int)pops[j];
                  break;
                }
                Cfrom += pops[j];
            }
            int SL = getDegree(SLprob);
            for (int j = 0; j < SL; j++)
            {
                out.write(i+"\t"+randomRange(Cfrom, Cto-1, i)+"\n");
                out.flush();
            }
            int LL = getDegree(LLprob);
            for (int j = 0; j < LL; j++)
            {
                int n;
                do
                {
                  n = randomRange(0, nrOfPeers-1, i);
                }
                while(n >= Cfrom && n < Cto);
                out.write(i+"\t"+n+"\n");
                out.flush();
            }
          }
          out.close();
      }
      catch (Exception e){e.printStackTrace();}
    }
}
```

8.2 Algorithms Implementations

8.2.1 Newman's Algorithm

```
public class NewmanVertex extends AdvancedVertex {

    private int distance, weight;

    public NewmanVertex(int id) {
        super(id);
        distance = -1;
        weight = -1;
    }

    public void setDistance(int newDistance) {
        distance = newDistance;
    }

    public int getDistance() {
        return distance;
    }

    public void setWeight(int newWeight) {
        weight = newWeight;
    }

    public int getWeight() {
        return weight;
    }
}
```

```
import export.ThreadedResults;
import graph.Vertex;
import graph.UndirectedGraph;
import graph.BasicVertexInterface;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Vector;
import strengthquantifier.Modularity;

public class NewmanAlgorithm extends Thread {

    private UndirectedGraph g, subgraph1, subgraph2;
    private ThreadedResults results;
    private Modularity modularity;
    private Vector<Vector<NewmanVertex>> adjacencyList;
    private Hashtable<Vertex, Double> verticesScores;
    private Hashtable<String, Double> edgesScores, edgesBetweeness;
    private int checked;

    public NewmanAlgorithm(UndirectedGraph g, ThreadedResults
results, Modularity modularity) {
        this.g = g;
        this.results = results;
        this.modularity = modularity;
        adjacencyList = new Vector();
        edgesScores = new Hashtable();
        edgesBetweeness = new Hashtable();
        verticesScores = new Hashtable();
        checked = 0;
    }

    public void run() {
        String edgeWithHighestBetweeness = null;
        while (g.getSize() > 1) {
            NewmanVertex root = null;
            Enumeration<BasicVertexInterface> en = g.getVertices();
            while (en.hasMoreElements()) {
                root = (NewmanVertex) en.nextElement();
                checked++;
                System.out.println("Traversing for new root:" +
root.getID() + " Already checked:" + checked);
                this.traverseGraph((NewmanVertex) root);
                this.setVerticesEdgesScores();
                this.adjacencyList.clear();
                this.setBetweeness();
                this.verticesScores.clear();
                this.edgesScores.clear();
                this.resetVertices();


            }
            edgeWithHighestBetweeness =
this.removeEdgeWithMaxBetweeness();
            System.out.println("Edge:" + edgeWithHighestBetweeness);
            this.edgesBetweeness.clear();
            //Copy all nodes from g to sg1
            subgraph1 = new UndirectedGraph();
            this.copyNodes(g, subgraph1);
            subgraph2 = this.createSubgraphs(root);
            if (subgraph1.getSize() == 0) {
                //Graph did not split in two.

                //empty subgraph1 & subgraph2
                subgraph1.removeAllVertices();
                subgraph2 = null;
            } else {
                //Graph did split in two
                double currentModularity =
modularity.getStrength(results);
                //If results contain g remove it.
                if (results.containsCommunity(g)) {
                    results.removeCommunity(g);
                }
                //Add subgraph1 & subgraph2 to results
                results.addCommunity(subgraph1);
                results.addCommunity(subgraph2);
                //Calculate newStrength without g but with subgraph1
& subgraph2
                double newStrength =
modularity.getStrength(results);
                System.out.println("New Strength:" + newStrength);
                if (newStrength < currentModularity) {
                    //addEdge Again
                    NewmanVertex source = (NewmanVertex)
this.getSource(edgeWithHighestBetweeness);
                    NewmanVertex destination = (NewmanVertex)
this.getDestination(edgeWithHighestBetweeness);
                    source.addNeighbor(destination);
                    destination.addNeighbor(source);
                    //remove subgraph1 & subgraph2 from results
                    results.removeCommunity(subgraph1);
                    results.removeCommunity(subgraph2);
                    //add g to results again
                    results.addCommunity(g);
                    results.unregisterThread();
                    return;
                } else {
                    results.registerThread();
                    NewmanAlgorithm detectCommunity1 = new
NewmanAlgorithm(subgraph1, results, modularity);
                    detectCommunity1.start();
                    results.registerThread();
                    NewmanAlgorithm detectCommunity2 = new
NewmanAlgorithm(subgraph2, results, modularity);
                    detectCommunity2.start();
                    return;
                }
            }
        }

        results.addCommunity(g);
        results.unregisterThread();
    }

    private void traverseGraph(NewmanVertex root) {
        Vector<NewmanVertex> edges;
        Queue<NewmanVertex> qe = new LinkedList();

        NewmanVertex vertex, neigh;
        int newDistance, newWeight;
        qe.add(root);
        root.setDistance(0);
        root.setWeight(1);
        while (!qe.isEmpty()) {
            edges = new Vector();
            vertex = qe.remove();
            edges.add(vertex);
            newDistance = vertex.getDistance() + 1;
            for (int i = 0; i < vertex.neighborhoodSize(); i++) {
                neigh = (NewmanVertex) vertex.getNeighbor(i);
                if (neigh.getDistance() == -1) {
                    qe.add(neigh);
                    newWeight = vertex.getWeight();
                    neigh.setDistance(newDistance);
                    neigh.setWeight(newWeight);
                    edges.add(neigh);
                } else {
                    if (neigh.getDistance() == newDistance) {
                        newWeight = neigh.getWeight() +
vertex.getWeight();
                        neigh.setWeight(newWeight);
                        edges.add(neigh);
                    }
                }
            }
            adjacencyList.add(edges);

        }
        qe.clear();
    }

    private void resetVertices() {
        NewmanVertex vertex;

        Enumeration<BasicVertexInterface> en = g.getVertices();

        while (en.hasMoreElements()) {
            vertex = (NewmanVertex) en.nextElement();
            vertex.setDistance(-1);
            vertex.setWeight(-1);
        }
    }

    private BasicVertexInterface getSource(String edge) {
        String temp[];
        temp = edge.split(":");
        return g.getVertex(Integer.parseInt(temp[0]));
    }

    private BasicVertexInterface getDestination(String edge) {
        String temp[];
        temp = edge.split(":");
        return g.getVertex(Integer.parseInt(temp[1]));
    }

    private UndirectedGraph createSubgraphs(NewmanVertex vertex) {
        UndirectedGraph community = new UndirectedGraph();
        Vector<BasicVertexInterface> visitedVertices = new Vector();
        NewmanVertex neigh;
        Queue<BasicVertexInterface> qe = new LinkedList();
        qe.add(vertex);
        visitedVertices.add(subgraph1.getVertex(vertex.getID()));
        while (!qe.isEmpty()) {
            vertex = (NewmanVertex) qe.remove();
            for (int i = 0; i < vertex.neighborhoodSize(); i++) {
                neigh = (NewmanVertex) vertex.getNeighbor(i);
                if (!visitedVertices.contains(neigh)) {
                    visitedVertices.add(subgraph1.getVertex(neigh.ge
tID()));
                    qe.add(neigh);
                }
            }
        }
        qe.clear();
        Enumeration<BasicVertexInterface> en =
visitedVertices.elements();
        while (en.hasMoreElements()) {
            vertex = (NewmanVertex) en.nextElement();
            community.addVertex(subgraph1.removeVertex(vertex.getID(
)));
        }
        return community;
    }

    private String removeEdgeWithMaxBetweeness() {
        double maxBetweeness = 0, currentBetweeness;
        String currentEdge, edgeWithMaxBetweeness = null;
        Enumeration<String> en = edgesBetweeness.keys();
        while (en.hasMoreElements()) {
            currentEdge = en.nextElement();
            currentBetweeness = edgesBetweeness.get(currentEdge);
            if (currentBetweeness >= maxBetweeness) {
                maxBetweeness = currentBetweeness;
                edgeWithMaxBetweeness = currentEdge;
            }
        }
        NewmanVertex source = (NewmanVertex)
this.getSource(edgeWithMaxBetweeness);
        NewmanVertex destination = (NewmanVertex)
this.getDestination(edgeWithMaxBetweeness);
        source.removeNeighbor(destination);
        destination.removeNeighbor(source);
        return edgeWithMaxBetweeness;
    }

    private void setBetweeness() {
        String edge;
        Enumeration<String> en = edgesScores.keys();
        while (en.hasMoreElements()) {
            edge = en.nextElement();
            if (edgesBetweeness.contains(edge)) {
                edgesBetweeness.put(edge, edgesBetweeness.get(edge)
+ edgesScores.get(edge));
            } else {
                edgesBetweeness.put(edge, edgesScores.get(edge));
            }
        }
    }

    private void setVerticesEdgesScores() {
        double sum;
        NewmanVertex vertex, neigh;
        Vector<NewmanVertex> currentEdges;
        for (int i = adjacencyList.size() - 1; i >= 0; i--) {
            sum = 0;
            currentEdges = adjacencyList.get(i);
            if (currentEdges.size() > 1) {
                vertex = currentEdges.firstElement();
                for (int j = 1; j < currentEdges.size(); j++) {
                    neigh = currentEdges.get(j);
                    if (!verticesScores.contains(neigh)) {
                        edgesScores.put(vertex.getID() + ":" +
neigh.getID(), new Double(vertex.getWeight() / neigh.getWeight()));
                        sum += vertex.getWeight() /
neigh.getWeight();
                    } else {
                        sum += verticesScores.get(neigh) + 1;
                        edgesScores.put(vertex.getID() + ":" +
neigh.getID(), verticesScores.get(neigh) + 1);
                    }
                }
                verticesScores.put(vertex, sum);
            }
        }
    }

    private void copyNodes(UndirectedGraph g1, UndirectedGraph g2) {
        Enumeration<BasicVertexInterface> vertices =
g1.getVertices();
        while (vertices.hasMoreElements()) {
            g2.addVertex(vertices.nextElement());
        }
    }
}
```

8.2.2 Bridge Bounding Algorithm

```
package bridgeBounding;

import graph.Vertex;

/**
 *
 * @author epp1640
 */
public class BridgeBoundingVertex extends Vertex {

    private boolean assigned;

    public BridgeBoundingVertex(int id){
        super(id);
        assigned = false;
    }

    public boolean isAssigned(){
        return assigned;
    }
    public void assign(){
        assigned = true;
    }
    public void reset(){
        assigned = false;
    }

}

package bridgeBounding;

import graph.BasicVertexInterface;
import graph.UndirectedGraph;
import java.util.Enumeration;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Vector;
import export.ThreadedResults;
import java.util.Hashtable;

public class BridgeBoundingTraverser extends Thread {

    private BridgeBoundingVertex root;
    private double threshold;
    private Vector<BridgeBoundingVertex> frontier;
    private ThreadedResults results;
    private Hashtable<String, Double> firstOrder;
    private boolean order;

    public BridgeBoundingTraverser(BridgeBoundingVertex root, double
threshold, Vector<BridgeBoundingVertex> frontier, ThreadedResults results,
Hashtable<String, Double> firstOrder, boolean order) {
        this.root = root;
        this.threshold = threshold;
        this.frontier = frontier;
        this.results = results;
        this.firstOrder = firstOrder;
        this.order = order;
    }


    public void run() {
        UndirectedGraph community = new UndirectedGraph();
        Vector<BridgeBoundingVertex> visitedVertices = new Vector();
        Queue<BridgeBoundingVertex> q = new LinkedList();
        q.add(root);
        visitedVertices.add(root);
        while (!q.isEmpty()) {
            BridgeBoundingVertex vertex = q.remove();
            visitedVertices.add(vertex);
            community.addVertex(vertex);
            vertex.assign();
            BridgeBoundingVertex link = null;
            Enumeration<BasicVertexInterface> links = vertex.getLinks();
            while (links.hasMoreElements()) {
                System.out.println("Next Element");
                link = (BridgeBoundingVertex) links.nextElement();
                if (!visitedVertices.contains(link)) {
                    if (!link.isAssigned() && !this.isBridge(vertex, link))
                    {
                        q.add(link);
                        visitedVertices.add(link);
                    } else if (this.isBridge(vertex, link)) {
                        System.out.println("Adding link to frontier!!");
                        frontier.add(link);
                    }
                }
            }
        }
        q.clear();
        visitedVertices.clear();
        results.addCommunity(community);
        results.unregisterThread();
    }
    private boolean isBridge(BasicVertexInterface vertex,
BasicVertexInterface link) {
        if (this.getMetric(vertex, link, order) <= threshold) {
            return false;
        } else {
            return true;
        }
    }


    private double getMetric(BasicVertexInterface vertex,
BasicVertexInterface link, boolean order) {
        if (order) {
            double metric = 0, sum = 0;
            metric += 0.7 * this.getMetric(vertex, link, !order);
            Enumeration<BasicVertexInterface> vertexNeighbors =
vertex.getLinks();
            while (vertexNeighbors.hasMoreElements()) {
                BasicVertexInterface vertexNeighbor =
vertexNeighbors.nextElement();
                if (vertexNeighbor != link) {
                    sum += this.getMetric(vertex, vertexNeighbor, !order);
                }
            }


            Enumeration<BasicVertexInterface> linkNeighbors =
link.getLinks();
            while (linkNeighbors.hasMoreElements()) {
                BasicVertexInterface linkNeighbor =
linkNeighbors.nextElement();
                if (linkNeighbor != vertex) {
                    sum += this.getMetric(link, linkNeighbor, !order);
                }
            }
            metric += 0.3 * sum / (vertex.degree() + link.degree() - 2);
            return metric;
        } else {
            return firstOrder.get(vertex.getID() + ":" + link.getID());
        }


    }
}
```

```
package bridgeBounding;

import graph.BasicVertexInterface;
import graph.UndirectedGraph;
import java.util.Enumeration;
import java.util.Vector;
import export.ThreadedResults;
import java.util.Hashtable;

public class BridgeBoundingAlgorithm extends Thread {
    private UndirectedGraph g;
    private double threshold;
    private Vector<BridgeBoundingVertex> frontier;
    private Hashtable<String, Double> firstOrder;
    private ThreadedResults results;
    private boolean order;
    private Vector<BridgeBoundingVertex> visitedFrontier;

    public BridgeBoundingAlgorithm(UndirectedGraph g, double threshold,
ThreadedResults results, boolean order) {
        this.g = g;
        this.threshold = threshold;
        this.results = results;
        this.order = order;
        frontier = new Vector();
        firstOrder = new Hashtable();
        visitedFrontier = new Vector();
    }

    public void run() {
        this.initMetrics();
        frontier.add(this.getStartingVertex());
        while (!frontier.isEmpty() || results.inProgress()) {
            if (!frontier.isEmpty()) {
                if (visitedFrontier.contains(frontier.get(0))) {
                    frontier.remove(0);
                } else {
                    visitedFrontier.add(frontier.get(0));
                    System.out.println("NEW THREAD");
                    BridgeBoundingTraverser thread = new
BridgeBoundingTraverser(frontier.remove(0), threshold, frontier, results,
firstOrder, order);
                    thread.start();
                    results.registerThread();
                }
            }
        }
        System.out.println("Exited.");
    }

    private BridgeBoundingVertex getStartingVertex() {
        Enumeration<BasicVertexInterface> vertices = g.getVertices();
        if (vertices.hasMoreElements()) {
            return (BridgeBoundingVertex) vertices.nextElement();
        }
        return null;
    }

    private double getBLest(BasicVertexInterface a, BasicVertexInterface b)
{

        double intersection = this.getIntersection(a, b).size();
        double relativeDegree = Math.min(a.degree(), b.degree());
        //System.out.println("1 - INTERSECTION:"+intersection+"/ RELATIVE
DEGREE:"+relativeDegree);
        double BLest = 1 - intersection / relativeDegree;
        return BLest;
    }

    private Vector<BasicVertexInterface>
getIntersection(BasicVertexInterface a, BasicVertexInterface b) {
        Vector<BasicVertexInterface> intersection = new Vector();
        Enumeration<BasicVertexInterface> aLinks = a.getLinks();
        while (aLinks.hasMoreElements()) {
            BridgeBoundingVertex aLink = (BridgeBoundingVertex)
aLinks.nextElement();
            if (b.hasLink(aLink)) {
                intersection.add(aLink);
            }
        }
        return intersection;
    }

    private void initMetrics() {
        Enumeration<BasicVertexInterface> vertices = g.getVertices();
        while (vertices.hasMoreElements()) {
            BasicVertexInterface vertex = vertices.nextElement();
            Enumeration<BasicVertexInterface> neighbors =
vertex.getLinks();
            while (neighbors.hasMoreElements()) {
                BasicVertexInterface neighbor = neighbors.nextElement();
                double BLest = this.getBLest(vertex, neighbor);
                firstOrder.put(vertex.getID() + ":" + neighbor.getID(),
BLest);
            }
        }
    }
}
```

8.2.3 Fortunato's Algorithm

```
package fortunato;

import graph.BasicVertexInterface;
import graph.Vertex;
import java.util.Enumeration;

/**
 *
 * @author epp1640
 */
public class FortunatoVertex extends Vertex {
    private double dFitness;
    public FortunatoVertex(int id) {
        super(id);
        dFitness = 0;
    }
    public double getDFitness() {
        return dFitness;
    }

    public void setDFitness(double fitness) {
        this.dFitness = fitness;
    }

    public long getShortLinks(FortunatoCommunity community) {
        long shortLinks = 0;
        Enumeration<BasicVertexInterface> neighbors = this.getLinks();
        while (neighbors.hasMoreElements()) {
            if (community.contains(neighbors.nextElement().getID())) {
                shortLinks++;
            }
        }
        return shortLinks;
    }

    public long getLongLinks(FortunatoCommunity community) {
        return this.degree() - this.getShortLinks(community);
    }
}

package fortunato;

import graph.BasicVertexInterface;
import graph.UndirectedGraph;
import java.util.Enumeration;

public class FortunatoCommunity extends UndirectedGraph {

    ;
    @Override
    public BasicVertexInterface removeVertex(int id) {
        this.getTable().remove(id);
        return this.getVertex(id);
    }
```

```
      public long shortLinksCount() {
        long shortLinks = 0;
        Enumeration<BasicVertexInterface> vertices = this.getVertices();
        while (vertices.hasMoreElements()) {
            FortunatoVertex vertex = (FortunatoVertex)
vertices.nextElement();
            shortLinks += vertex.getShortLinks(this);
        }
        return shortLinks / 2;
    }

      public long totalLinksCount() {
        long totalLinks = 0;
        Enumeration<BasicVertexInterface> vertices = this.getVertices();
        while (vertices.hasMoreElements()) {
            FortunatoVertex vertex = (FortunatoVertex)
vertices.nextElement();
            totalLinks += vertex.degree();
        }
        totalLinks -= this.shortLinksCount();
        return totalLinks;
    }

}

package fortunato;

import export.Results;
import graph.BasicVertexInterface;
import graph.GraphInterface;
import graph.UndirectedGraph;
import java.util.Enumeration;
import java.util.Vector;

public class FortunatoAlgorithm extends Thread {

    private UndirectedGraph g;
    private double a;
    private Results results;
    private double currentFitness;

    public FortunatoAlgorithm(UndirectedGraph g, double a, Results results)
{
        this.g = g;
        this.a = a;
        this.results = results;
    }

    public void run() {
        results.setProgress(true);
        Enumeration<BasicVertexInterface> vertices = g.getVertices();
        while (vertices.hasMoreElements()) {
            FortunatoVertex vertex = (FortunatoVertex)
vertices.nextElement();
            if (this.isAssigned(vertex)) {
                continue;
            }
            FortunatoCommunity community = new FortunatoCommunity();
            currentFitness = 0;
            Vector<FortunatoVertex> candidates = new Vector();
            community.addVertex(vertex);
            this.recruitCandidates(candidates, community);
            while (!candidates.isEmpty()) {
                this.interviewCandidates(candidates, community);
                FortunatoVertex bestCandidate =
this.getBestCandidate(candidates);
                if (bestCandidate == null) {
                    break;
                }
                community.addVertex(bestCandidate);
                currentFitness = bestCandidate.getDFitness();
                this.rateEfficiency(community);
                this.recruitCandidates(candidates, community);
            }
            results.addCommunity(community);
        }
        results.setProgress(false);
    }

    private boolean isAssigned(FortunatoVertex v) {
        Enumeration<GraphInterface> communities = results.getCommunities();
        while (communities.hasMoreElements()) {
            if (communities.nextElement().contains(v.getID())) {
                return true;
            }
        }
        return false;
    }

    private void recruitCandidates(Vector<FortunatoVertex> candidates,
FortunatoCommunity community) {
        candidates.clear();
        Enumeration<BasicVertexInterface> vertices =
community.getVertices();
        while (vertices.hasMoreElements()) {
            Enumeration<BasicVertexInterface> neighbors =
vertices.nextElement().getLinks();
            while (neighbors.hasMoreElements()) {
                FortunatoVertex neighbor = (FortunatoVertex)
neighbors.nextElement();
                if (!community.contains(neighbor.getID()) && !
candidates.contains(neighbor)) {
                    candidates.add(neighbor);
                }
            }
        }
    }

    private void interviewCandidates(Vector<FortunatoVertex> candidates,
FortunatoCommunity c) {

        long previousShortLinks = c.shortLinksCount();
        long previousTotalLinks = c.totalLinksCount();
        for (int i = 0; i < candidates.size(); i++) {
            long currentShortLinks = previousShortLinks +
candidates.get(i).getShortLinks(c);
            long currentTotalLinks = previousTotalLinks +
(candidates.get(i).degree() - candidates.get(i).getShortLinks(c));
            double nextFitness = currentShortLinks / 
Math.pow(currentTotalLinks, a);
            candidates.get(i).setDFitness(nextFitness);
        }
    }


    private FortunatoVertex getBestCandidate(Vector<FortunatoVertex> 
candidates) {
        FortunatoVertex bestCandidate = candidates.get(0);
        for (int i = 1; i < candidates.size(); i++) {
            if (candidates.get(i).getDFitness() > 
bestCandidate.getDFitness()) {
                bestCandidate = candidates.get(i);
            }
        }
        if (bestCandidate.getDFitness() > currentFitness) {
            return bestCandidate;
        } else {
            return null;
        }
    }


    private void rateEfficiency(FortunatoCommunity c) {
        double newFitness;
        Enumeration<BasicVertexInterface> vertices;
        boolean found = true;
        while (found && c.getSize() > 2) {
            found = false;
            long previousShortLinks = c.shortLinksCount();
            long previousTotalLinks = c.totalLinksCount();
            vertices = c.getVertices();
            while (vertices.hasMoreElements()) {
                FortunatoVertex vertex = (FortunatoVertex) 
vertices.nextElement();
                long newShortLinks = previousShortLinks - 
vertex.getShortLinks(c);
                long newTotalLinks = previousTotalLinks - (vertex.degree() 
- vertex.getShortLinks(c));
                newFitness = newShortLinks / Math.pow(newTotalLinks, a);
                if (newFitness > currentFitness) {
                    c.removeVertex(vertex.getID());
                    found = true;
                    currentFitness = newFitness;
                    break;
                }
            }
        }
    }

    private double getCurrentFitness(FortunatoCommunity c) {
        if (c.getSize() > 1) {
            return c.shortLinksCount() / Math.pow(c.totalLinksCount(), a);
        } else {
            return 0;
        }
    }
}
```

8.2.4 CiBC Algorithm

```
package cibc;

import graph.Vertex;

public class CiBCVertex extends Vertex {

    private int distance, weight;

    public CiBCVertex(int id) {
        super(id);
        distance = -1;
        weight = -1;
    }

    public void setDistance(int newDistance) {
        distance = newDistance;
    }

    public int getDistance() {
        return distance;
    }

    public void setWeight(int newWeight) {
        weight = newWeight;
    }

    public int getWeight() {
        return weight;
    }

}


package cibc;

import export.Results;
import graph.BasicVertexInterface;
import graph.GraphInterface;
import graph.UndirectedGraph;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Vector;

public class CiBCAlgorithm extends Thread {

    private Vector<UndirectedGraph> cliques;
    private Hashtable<CiBCVertex, Integer> bcTable;
    private CiBCVertex[] verticesArray;
    private Integer[] valuesArray;
    private UndirectedGraph g;
    private Results results;
    private int[][] matrix;
```

```
public CiBCAlgorithm(UndirectedGraph g, Results results) {
    this.g = g;
    this.results = results;
    cliques = new Vector<UndirectedGraph>();
    bcTable = new Hashtable<CiBCVertex, Integer>();
}

private void initMatrix() {
    matrix = new int[cliques.size()][cliques.size()];
}

private void populateArrays() {
    CiBCVertex vertex;
    Vector<CiBCVertex> bcVertices = new Vector();
    Vector<Integer> bcValues = new Vector();
    Enumeration<CiBCVertex> keys = bcTable.keys();
    while (keys.hasMoreElements()) {
        vertex = keys.nextElement();
        bcVertices.add(vertex);
        bcValues.add(bcTable.get(vertex));
    }
    verticesArray = (CiBCVertex[]) bcVertices.toArray();
    valuesArray = (Integer[]) bcValues.toArray();
}

private void sortArrays() {
    boolean sorted = false;
    for (int top = valuesArray.length - 1; top > 0 && !sorted; top--) {
        sorted = true;

        for (int i = 0; i < top; i++) {
            if (valuesArray[i] > valuesArray[i + 1]) {
                sorted = false;
                int tempValue = valuesArray[i];
                CiBCVertex tempVertex = verticesArray[i];
                valuesArray[i] = valuesArray[i + 1];
                valuesArray[i + 1] = tempValue;
                verticesArray[i] = verticesArray[i + 1];
```

```
                    verticesArray[i + 1] = tempVertex;
                }
            }
        }
    }

    private void allocateValues() {
        GraphInterface communityI, communityJ;
        for (int i = 0; i < cliques.size(); i++) {
            for (int j = 0; j < cliques.size(); j++) {
                communityI = cliques.get(i);
                communityJ = cliques.get(j);
                if (i == j) {
                    if (communityI.getSize() == 1) {
                        matrix[i][j] = 0;
                    } else {
                        matrix[i][j] = communityI.getShortLinks().size() /
2;
                    }
                } else {
                    matrix[i][j] =
communityI.getLongLinks(communityJ).size();
                }
            }
        }
    }

    private void calculateBC() {
        Enumeration<BasicVertexInterface> vertices = g.getVertices();
        while (vertices.hasMoreElements()) {
            CiBCVertex root = (CiBCVertex) vertices.nextElement();
            this.traverseGraph(root);
            this.sumWeights();
            this.resetVertices();
        }
    }

    private void createCliques() {
        for (int i = verticesArray.length - 1; i > 0; i--) {
            CiBCVertex vertex = verticesArray[i];
            if (!this.isAssigned(vertex)) {
                UndirectedGraph clique = new UndirectedGraph();
                clique.addVertex(vertex);
                Enumeration<BasicVertexInterface> en = vertex.getLinks();
                while (en.hasMoreElements()) {
                    CiBCVertex neighbor = (CiBCVertex) en.nextElement();
                    if (!this.isAssigned(neighbor)) {
                        clique.addVertex(neighbor);
                    }
                }
                cliques.add(clique);
            }
        }
    }

    public void run() {
        this.calculateBC();
        this.populateArrays();
        this.sortArrays();
        this.createCliques();
        this.merge();
        this.updateResults();
    }

    private void merge() {
        boolean mergedCliques = false;
        while (cliques.size()>1 && !mergedCliques) {
            mergedCliques = false;
            this.initMatrix();
            this.allocateValues();
            for(int i=0; i<cliques.size(); i++){
                int shortLinks = matrix[i][0];
                for(int j=1; j<cliques.size(); j++){
                    if(shortLinks<=matrix[i][j]){
                        //do the merging
                        this.mergeCliques(cliques.get(0), cliques.get(j));
                        //remove clique
                        cliques.remove(j);
                        mergedCliques = true;
                        break;
                    }
                }
                if(mergedCliques){
                    break;
                }
            }

        }
    }

    private void updateResults(){
        for(int i=0; i<cliques.size(); i++){
            results.addCommunity(cliques.get(i));
        }
    }

    private void traverseGraph(CiBCVertex root) {
        Queue<CiBCVertex> qe = new LinkedList();
        CiBCVertex vertex, neigh;
        int newDistance, newWeight;
        qe.add(root);
        root.setDistance(0);
        root.setWeight(1);
        while (!qe.isEmpty()) {
            vertex = qe.remove();
            newDistance = vertex.getDistance() + 1;
            for (int i = 0; i < vertex.degree(); i++) {
                neigh = (CiBCVertex) vertex.getLink(i);
                if (neigh.getDistance() == -1) {
                    qe.add(neigh);
                    newWeight = vertex.getWeight();
                    neigh.setDistance(newDistance);
                    neigh.setWeight(newWeight);
                } else {
                    if (neigh.getDistance() == newDistance) {
                        newWeight = neigh.getWeight() + vertex.getWeight();
                        neigh.setWeight(newWeight);
                    }
                }
            }
        }
        qe.clear();
    }

    private void resetVertices() {
        CiBCVertex vertex;

        Enumeration<BasicVertexInterface> en = g.getVertices();

        while (en.hasMoreElements()) {
            vertex = (CiBCVertex) en.nextElement();
            vertex.setDistance(-1);
            vertex.setWeight(-1);
        }
    }

    private void sumWeights() {
        CiBCVertex vertex;
        Enumeration<BasicVertexInterface> en = g.getVertices();
        while (en.hasMoreElements()) {
            vertex = (CiBCVertex) en.nextElement();
            if (!bcTable.containsKey(vertex)) {
                bcTable.put(vertex, vertex.getWeight());
            } else {
                int currentValue = bcTable.get(vertex);
                int newValue = currentValue + vertex.getWeight();
                bcTable.put(vertex, newValue);
            }
        }
    }

    private boolean isAssigned(CiBCVertex vertex) {
        for (int i = 0; i < cliques.size(); i++) {
            if (cliques.get(i).contains(vertex.getID())) {
                return true;
            }
        }
        return false;
    }

    private void mergeCliques(GraphInterface communityI, GraphInterface
communityJ){
        Enumeration<BasicVertexInterface> en = communityJ.getVertices();
        while(en.hasMoreElements()){
            communityI.addVertex(en.nextElement());
        }
    }

}
```

8.3 *Modularity Measure Implementation*

```
package strengthquantifier;

import export.DetectionResultsInterface;
import graph.BasicVertexInterface;
import graph.GraphInterface;
import graph.UndirectedGraph;
import java.util.Enumeration;
import java.util.Vector;

public class Modularity implements QuantifierInterface {

    private double[][] matrix;
    private DetectionResultsInterface results;
    private double q;

    public double getStrength(DetectionResultsInterface results) {
        this.results = results;
        q = 0;
        this.initMatrix();
        this.allocateValues();
        q = this.calculateQ(this.getRows());
        System.out.println("Modularity:" + q);
        return q;
    }

    private void initMatrix() {
        matrix = new double[results.getSize()][results.getSize()];
    }

    private void allocateValues() {
        double totalLinks = this.getTotalLinks();
        GraphInterface communityI, communityJ;
        for (int i = 0; i < results.getSize(); i++) {
            for (int j = 0; j < results.getSize(); j++) {
                communityI = results.getCommunity(i);
                communityJ = results.getCommunity(j);
                if (i == j) {
                    if (communityI.getSize() == 1) {
                        matrix[i][j] = 0;
                    } else {
                        double shortLinks = (double)
communityI.getShortLinks().size() / 2.0;
                        System.out.println("ShortLinks["+i+"]:"+shortLinks)
;
                        System.out.println("TotalLinks["+i+"]:"+totalLinks)
;
                        double eij = shortLinks / totalLinks;
                        matrix[i][j] = eij;
                        System.out.println("Short Links
Fraction["+i+","+i+",]:" + matrix[i][i]);
                    }
                } else {
                    double longLinks = (double)
communityI.getLongLinks(communityJ).size();
                    System.out.println("LongLinks["+i+","+j+"]:"+longLinks)
;
                    System.out.println("totalLinks["+i+","+j+"]:"+totalLink
s);
                    double eij = longLinks / totalLinks;
                    matrix[i][j] = eij;
                    System.out.println("Long Links Fraction[" + i + "," + j
+ "]:" + matrix[i][j]);
                }
            }
        }
    }

    private double[] getRows() {
        double rows[] = new double[results.getSize()];
        for (int i = 0; i < rows.length; i++) {
            double sum = 0;
            for (int j = 0; j < rows.length; j++) {
                if (i != j) {
                    sum += matrix[i][j];
                }
            }
            rows[i] = Math.pow(sum, 2);
            System.out.println("Row[" + i + "]:" + rows[i]);
        }
        return rows;
    }

    private double calculateQ(double[] rows) {
        double q = 0;
        for (int i = 0; i < rows.length; i++) {
            q += matrix[i][i] - rows[i];
        }
        return q;
    }

    private double getTotalLinks(){
        double totalLinks = 0;
        Vector<BasicVertexInterface> visitedVertices = new
Vector<BasicVertexInterface>();
        for(int i=0; i<results.getSize(); i++){
            GraphInterface communityI = results.getCommunity(i);
            Enumeration<BasicVertexInterface> vertices =
communityI.getVertices();
            while(vertices.hasMoreElements()){
                BasicVertexInterface vertex = vertices.nextElement();
                if(!visitedVertices.contains(vertex)){
                    totalLinks += vertex.degree();
                    visitedVertices.add(vertex);
                }
            }
        }
        return totalLinks/2.0;
    }
}

package strengthquantifier;

import export.DetectionResultsInterface;

public interface QuantifierInterface {
    public double getStrength(DetectionResultsInterface results);
}
```

References

(1) A. M. Kaplan and M. Haenlein, “Users of the world, unite! The challenges and opportunities of Social Media” Business Horizons, 2010.

(2) B. Buter, N. Dijkshoorn, D. Modolo, Q. Nguyen, S. van Noort, B. van de Poel, A. Ali, and A. Salah1. Explorative visualization and analysis of a social network for arts: The case of deviantart. Journal of Convergence Volume, 2(1), 2011.

(3) G. W. Flake, S. Lawrence, C. L. Giles, and F. M. Coetzee. Self-organization and identification of web communities. IEEE Computer, 35:66–71, March 2002.

(4) D. Katsaros, G. Pallis, K. Stamos, A. Vakali, A. Sidiropoulos, and Y. Manolopoulos. Cdns content outsourcing via generalized communities. IEEE Transactions on Knowledge and Data Engineering, 21:137–151, 2009.

(5) M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. Physical Review E, 69(2):026113, Feb 2004.

(6) M. Girvan and M. E. J. Newman. Community structure in social and biological networks. Proceedings of the National Academy of Sciences of the United States of America, 99(12):7821–7826, June 2002.

(7) Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs". Numerische Mathematik 1: 269–271. doi:10.1007/BF01386390.

(8) M. Girvan and M. E. J. Newman. Community structure in social and biological networks. Proceedings of the National Academy of Sciences of the United States of America, 99(12):7821–7826, June 2002.

(9) M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. Physical Review E, 69(2):026113, Feb 2004.

(10) S. Papadopoulos, A. Skusa, A. Vakali, Y. Kompatsiaris, and N. Wagner. Bridge bounding: A local approach for efficient community discovery in complex networks. Technical Report arXiv:0902.0871, Feb 2009.

(11) A. Lancichinetti, S. Fortunato, and J. Kertész. Detecting the overlapping and hierarchical community structure in complex networks. New Journal of Physics, 11(3):033015+, March 2009.

(12) I. Derenyi, G. Palla, and T. Vicsek. Clique Percolation in Random Networks.Physical Review Letters, 94(16):160–202, Apr 2005.

(13) B. Buter, N. Dijkshoorn, D. Modolo, Q. Nguyen, S. van Noort, B. van de Poel, A. Ali, and A. Salah1. Explorative visualization and analysis of a social network for arts: The case of deviantart. Journal of Convergence Volume, 2(1), 2011.

(14) MySpace. http://www.myspace.com.

(15) Facebook. http://www.facebook.com.

(16) Y.-Y. Ahn, S. Han, H. Kwak, M. S., and H. Jeong. Analysis of Topological Characteristics of Huge Online Social Networking Services. May 2007.

(17) W. W. Mojtaba Torkjazi, Reza Rejaie. Hot Today, Gone Tomorrow: On the Migration of MySpace Users. WOSN’09, Barcelona, Spain, pages 43–48, August 17 2009.

(18) J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. August 2005.

(19) R. Kumar, J. Novak, and A. Tomkins. Structure and the Evolution of Online Social Networks. August 2006.

(20) G. Pallis, D. Zeinalipour-Yazti, and M. D. Dikaiakos. Online Social Networks: Status and Trends.New directions in Web Data Management, 1, 2011.

(21) A. Lancichinetti and S. Fortunato. Community detection algorithms: a comparative analysis.Physical Review E, 80(5 Pt 2):056117, Sep 2009.

(22) S. E. Schaeffer. Graph clustering.Computer Science Review, 1(1):27 – 64, 2007.

(23) G. Palla, I. Derenyi, I. Farkas, and T. Vicsek. Uncovering the overlapping community structure of complex networks in nature and society, June 2005.

Published Papers Appendix

The paper at the following page was published and presented in the 4th International Conference on Pervasive Technologies Related to Assistive Environments organized by the University of Texas at Arlington, USA.

The paper at the following page was published and presented in the 2012 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining organized by Kadir Has University, Istanbul, Turkey and sponsored by ACM and IEEE.