Algorithms For Discovering Communities In Complex Networks

Hemant Balakrishnan
University of Central Florida

ALGORITHMS FOR DISCOVERING COMMUNITIES IN COMPLEX NETWORKS

by

HEMANT BALAKRISHNAN
B.E. Bharathiar University, 2000
M.S. University of Texas at Dallas, 2002

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2006

Major Professor: Narsingh Deo

ABSTRACT

It has been observed that real-world random networks like the WWW, Internet, social networks, citation networks, etc., organize themselves into closely-knit groups that are *locally dense* and *globally sparse*. These closely-knit groups are termed *communities*. Nodes within a community are similar in some aspect. For example in a WWW network, communities might consist of web pages that share similar contents. Mining these communities facilitates better understanding of their evolution and topology, and is of great theoretical and commercial significance. Community related research has focused on two main problems: community discovery and community identification. *Community discovery* is the problem of extracting all the communities in a given network, whereas *community identification* is the problem of identifying the community, to which, a given set of nodes belong.

We make a comparative study of various existing community-discovery algorithms. We then propose a new algorithm based on bibliographic metrics, which addresses the drawbacks in existing approaches. *Bibliographic metrics* are used to study similarities between publications in a citation network. Our algorithm classifies nodes in the network based on the similarity of their neighborhoods. One of the drawbacks of the current community-discovery algorithms is their computational complexity. These algorithms do not scale up to the enormous size of the real-world networks. We propose a hash-table-based technique that helps us compute the bibliometric similarity between nodes in O(m Δ) time. Here m is the number of edges in the graph and Δ, the largest degree.

Next, we investigate different centrality metrics. *Centrality metrics* are used to portray the importance of a node in the network. We propose an algorithm that utilizes centrality metrics of the nodes to compute the importance of the edges in the network. Removal of the edges in ascending order of their importance breaks the network into components, each of which represent a community. We compare the performance of the algorithm on synthetic networks with a known community structure using several centrality metrics. Performance was measured as the percentage of nodes that were correctly classified.

As an illustration, we model the ucf.edu domain as a *web graph* and analyze the changes in its properties like densification power law, edge density, degree distribution, diameter, etc., over a five-year period. Our results show super-linear growth in the number of edges with time. We observe (and explain) that despite the increase in average degree of the nodes, the edge density decreases with time.

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Narsingh Deo, for introducing me to the beautiful area of real-world graphs, and for his continued mentorship and encouragement. Without his support this dissertation would not have been possible. I would also like to thank the members of my research committee, professors Charles Hughes, Ronald Dutton and Liqiang Ni for their help, guidance and advice during the entire process.

I wish to thank my fellow graduate students in the Center for Parallel Computation, especially Aurel Cami, Sanjeeb Nanda and Zoran Nikoloski, for many stimulating discussions and insightful comments. I am indebted to the faculty and the staff of the Computer Science Department for providing the environment and the resources, which made this effort possible.

Special thanks are due to my wife Nirupa, for her love and support; my parents Balakrishnan and Susheela, for their faith and inspiration; and my friends Manish Billa, Sagar Karandikar, Abhishek Karnik, Suyog Pathak, and Aniket Vartak who have been there when I needed them.

TABLE OF CONTENTS

LIST OF FIGURES x

LIST OF TABLES xiii

1. INTRODUCTION 1

1.1 Real-world networks 3

1.2 Properties 5

1.2.1 Small world network 5

1.2.2 Scale-free property 6

1.2.3 Power-law degree distribution 7

1.2.4 Network transitivity 8

1.2.5 Community structure 9

1.3 Community 9

1.4 Random graph models 13

1.4.1 Classical random graph model 14

1.4.2 Watts and Strogatz model 17

1.4.3 Preferential attachment model 20

1.4.4 Copy model 21

1.4.5 Models with embedded communities 23

2. EVOLUTION OF THE WORLD WIDE WEB 26

2.1 Importance of this work 26

2.2 Related work 27

2.3 Experimental setup 27

2.4 Empirical observations 29

2.4.1 New nodes 29

2.4.2 Densification power law and average out degree 30

2.4.3 Edge density 32

2.4.4 Degree distributions 34

2.4.5 Diameter and average distance between node pairs 37

2.5 Conclusion 38

3. SURVEY OF EXISTING METHODS 40

3.1 Community related problems 40

3.2 Existing algorithms 41

3.3 Similarity metrics 42

3.3.1 Node/Edge independent paths 42

3.3.2 Edge betweenness 46

3.3.3 Edge clustering co-efficient 50

3.3.4 Random walks 51

3.3.5 Current betweenness 54

3.3.6 Euclidean distance 57

3.3.7 Pearson correlation coefficient 58

3.3.8 Flow network based 58

3.4 Spectral methods 61

3.4.1 HITS 63

3.4.2 HITS Communities 65

3.5 Conclusion 66

4. PROPOSED CENTRALITY BASED COMMUNITY DISCOVERY 68

4.1 Centrality metrics 68

4.1.1 Degree centrality 68

4.1.2 Closeness centrality 69

4.1.3 Betweenness centrality 70

4.1.4 Straightness centrality 72

4.1.5 Information centrality 72

4.1.6 Clustering co-efficient 73

4.2 Centrality of a graph 75

4.3 Community discovery using centrality measures 75

4.3.1 Proposed approach 76

5. BIBLIOMETRIC APPROACH 81

5.1 Inspiration 81

5.2 Bibliometric similarity 81

5.3 Performance on computer generated models 84

5.4 Performance on real-world networks 85

6. COMMUNITIES IN LARGE NETWORKS 93

6.1 Requirement 93

6.2 Proposed algorithm 94

6.2.1 Complexity analysis 97

6.3 Results 97

7. UCFBOT CRAWLER 100

7.1 Introduction 100

7.2 The Crawling Algorithm 101

7.3 Related Work 104

7.4 UcfBot Architecture 105

7.5 urlsToVisit Data Structure 107

7.6 DNS Lookup module 110

7.7 Page Fetch 111

7.8 Parser 112

7.8.1 libwww library 113

7.8.2 Removing duplicates 114

7.8.3 Robots META Element 115

7.8.4 Filtering 115

7.9 urlsEncountered Data Structure 116

7.10 Checkpointing 117

7.11 A Polite Crawler 118

7.11.1 Avoiding DOS attacks 119

7.11.2 Robot Exclusion Protocol and META elements 119

7.12 Mirroring 119

7.13 Conclusion 120

8. CONCLUSION AND FUTURE DIRECTIONS 122

8.1 Overlapping communities 122

8.2 New definition 123

8.3 Quality of communities 123

8.4 Algorithms for community identification 125

8.5 Centrality based community identification 125

8.6 Parallel algorithms 126

8.7 Applications 127

8.7.1 Visualization of search results 127

8.7.2 Automated web directory generation 128

8.7.3 Focused crawling 129

8.7.4 Network security enhancement 129

8.7.5 Other applications 130

8.8 Conclusion 130

LIST OF REFERENCES 131

LIST OF FIGURES

Figure 1: Out degree distribution of WWW. ... 7

Figure 2: Communities in a high school friendships network. ... 10

Figure 3: Hubs and authorities. ... 11

Figure 4: A bipartite core. ... 11

Figure 7: Watts and Strogatz models obtained by varying *p* between 0 and 1. ... 18

Figure 8: Number of new nodes introduced in the network over time. ... 30

Figure 9: Number of edges m_t versus number of nodes n_t in log-log scales. ... 31

Figure 10: Average node out-degree over time. ... 32

Figure 11: Edge density over time. ... 33

Figure 12: Degree distributions from 5 years of ucf.edu domain web graphs. ... 36

Figure 13: Diameter over time. ... 37

Figure 14: Average distance between node pairs. ... 38

Figure 15: Community discovery. ... 40

Figure 16: Community identification. ... 41

Figure 17: (a) edge independent paths, (b) node independent paths. ... 42

Figure 18: Edge betweenness of edges. ... 47

Figure 19: Edge clustering coefficient of edges. ... 50

Figure 20: Example resistor network. ... 54

Figure 21: Graph obtained by using the algorithm in [36]. ... 60

Figure 22: Hubs and Authorities 64

Figure 23: Degree centrality of a star, $K_{1,8}$, and a cycle C_8. 69

Figure 24: Closeness centrality of a star, $K_{1,8}$, and a cycle C_8. 70

Figure 25: An example for betweenness centrality 71

Figure 26: An example for information centrality. 72

Figure 27: An example for clustering coefficient. 74

Figure 28: (a) degree centrality of the nodes in the graph, (b) corresponding edge centralities. . 77

Figure 29: Performance of community discovery using different centrality measures. 80

Figure 30: (a) bibliographic coupling (b) co-citation coupling. 81

Figure 31 : Performance on computer-generated models. 85

Figure 32: The Zachary Karate Club Network. 85

Figure 33: Communities identified in the Zachary Karate Club Network 86

Figure 34: Communities identified in the College Football Network. 88

Figure 35: The largest component of the Santa Fe Institute collaboration network. 90

Figure 36: A few communities identified in the Roget's thesaurus network. 91

Figure 37: Communities identified in the largest component of the Santa Fe Institute collaboration network. 92

Figure 38: Output for various threshold values. 99

Figure 39: The architecture of UcfBot. 106

Figure 40: Implementation of the urlsToVisit data structure 108

Figure 41: The architecture of the Parser module 112

Figure 42: Degree distribution of a small portion of www.ucf.edu domain. 121

Figure 43: A snapshot of the network of the www.ucf.edu domain. 121

Figure 44: Overlap in communities. 122

LIST OF TABLES

Table 1: Size and order of the web graphs over the five-year period. t represents time. 29

Table 2: Comparing the complexity of the various algorithms.. 67

1. INTRODUCTION

Ever since Euler’s paper on the Bridges of Königsberg [5], published in 1736, graph theory has been used to solve a wide range of problems in computer science, math, physics and chemistry. The four-color problem posed in 1852 by Francis Guthrie is considered as the birth of graph theory. This problem asks if it is possible to color, using only four colors, any map of countries such that no two neighboring countries have the same color. Appel and Haken [6] provided a solution to this problem in 1976.

Graph theory has proven itself to be a very powerful tool in solving a number of real-world problems. This is accomplished by first, modeling it as a graph and then, applying graph theoretic techniques to solve the problem. One of its first applications was on electric circuit analysis, where Gustav Kirchhoff used it to compute voltage and current in electric circuits. Many problems of practical interest can be described as graphs. Transforming a real-world problem into a graph is by itself a challenging task. For example, consider testing printed circuit boards (PCB) for short circuits. This can be accomplished by modeling the PCB into a graph such that each component on the PCB represents a vertex and an edge is drawn between two components if there is a potential for a short. Now one could perform graph coloring on this model to partition it into groups, consisting of vertices of the same color. Components represented by one group, could be simultaneously tested for shorts against all other components, thereby reducing the time required for testing. Over the past decade researchers have shown a lot of interest in graph modeling and study of properties of the graphs thus obtained.

Before proceeding further I will go over certain basic definitions, that might help the reader better understand the content of this thesis. Other definitions will be provided as and when required. For more advanced concepts and definitions in graph theory refer to [31].

A *graph*, G, is an ordered pair consisting of two sets (V, E), where V is a set of *vertices*, and E is a set of *edges*. The number of vertices, $|V|$, is termed the order of the graph and the number of edges, $|E|$, is termed its size. Every edge connects a pair of vertices in V, and is represented as (u, v) or $e_{u,v}$. Here u, $v \in V$ and are called the *endpoints* of the edge. The graph is termed *directed* if the edge (u, v) represents an ordered pair. In this case the vertex u is called the *tail* and vertex v is called the *head* of the *directed edge*. Most of the graphs mentioned in this thesis are undirected unless specified otherwise. Physicists often refer to the graph as a *network*, vertices as *nodes* and edges as *links*. Two vertices u and v are said to be *adjacent* or *neighbors* if they are the endpoints of the same edge. The *neighborhood* of a vertex u, denoted N_u, is a set consisting of all the vertices that are adjacent to u. Two edges are *incident* with each other if they have a common endpoint. A *loop* is an edge whose endpoints are equal. Two edges are said to be *parallel* if they have the same set of endpoints. A vertex u is called *isolated* if it is not an endpoint of any edge. The *degree* of a vertex u, denoted d_{u}, is the number of edges that have u as its endpoint. In a directed graph one can talk about in-degree and out-degree of a vertex. The *in–degree* of a vertex u refers to the number of edges that have u as its head and *out-degree* of a vertex u refers to the number of edges that have u as its tail. A *complete graph* on n vertices K_n = (V, E), is such that $V = \{u_1, u_2, \ldots, u_n\}$ and every pair of vertices are connected by an edge *i.e.* $E = \{\{u_i, u_j\} : i, j \in \{1, 2, \ldots, \mathrm{n}\}, i < j\}$.

Graphs are being widely used to describe complex real-world systems such as the World Wide Web, Internet, energy power grids, social friendships, neural networks, etc. Despite their diversity, most of these real-world graphs share a lot of their organizational structure. Research in this area has concentrated on local (or microscopic) properties involving individual vertices or vertex pairs (*e.g.*, shortest path between vertices, transitivity, degree distribution, robustness, etc.). Though these properties are important, it is equally important to understand their global (or macroscopic) properties. The microscopic and macroscopic properties together, aid in better understanding these systems.

1.1 Real-world networks

Real-world networks are used to model complex systems, which consist of components that interact. These systems have most of the following features: *agent based* - the basic building blocks are characteristics and activities of individual agents in the environment under study; *dynamic* – the characteristics of each of these agents vary over time, the system rarely achieves an equilibrium and the changes it undergoes are non-linear and often chaotic; *organization* – the agents organize themselves into groups that are well structured and these structures influence their evolution.

Network models are built to study the system and conduct scientific experiments. This involves identifying the mechanisms that characterize the dynamics of the system and translating them into sets of rules that can be implemented and investigated computationally. Care has to be taken to build simple models that provide coarse-grained description of the system. Once a model is built researchers analytically identify new properties of the system and then validate the new properties against real-world data. Examples of complex systems and their models:

World Wide Web - The Web is an actively evolving network, composed of html pages that point to one another by means of a hyperlink. Superficially, it looks like a giant directed graph (*web graph*) where vertices are html pages and edges are the hyperlinks [55].

Internet – The Internet refers to the interconnected system of networks that connect computers and routers around the world. Its model consists of vertices representing computers or routers and edges representing physical connections between them [3].

Neural network – A neural network is collection of brain cells (neurons) transferring impulses to each other across via synaptic connections (axons). In this network model each vertex represents a neuron, and edges, the synaptic connections [93].

Citation network – A citation network consists of publications and their references. Vertices in this model represent publications and there exists an edge from one vertex to the other if the document being represented by the former vertex refers to the one being represented by the later [41].

Airline network – An airline network is a transportation network with vertices representing the various cities that have an airport and an edge is drawn between two vertices if there is an airline connecting the cities being represented by the vertices [48].

Semantic network – A semantic network consists of words represented by vertices and an edge between two vertices indicates that the corresponding words are either synonyms or antonyms [78].

Movie actors' network – Nodes in this network represent actors and an edge is introduced between two vertices, if the corresponding actor's appear in a movie together [92].

For additional examples see [3, 32, 67].

1.2 Properties

Due to the randomness involved in their evolution, classical random graphs were initially used to represent real-world networks. Classical random graphs also known as Erdos and Renyi random graphs were first defined by Erdos and Renyi in the year 1959 [34]. Section 1.4.1 provides more information on these graphs. Recent studies on real-world networks have revealed some interesting structural and topological properties, which suggest that these networks are significantly different from classical random graphs. The next few sections describe a few of these properties.

1.2.1 *Small world network*

A *small world network* is a network in which every vertex can reach every other vertex within a small number of hops. The idea is an extension of the small world phenomenon (*small world effect*) in social networks that hypothesizes that everyone in the world can be reached through a short chain of social acquaintances. Psychologist Stanley Milgram conducted an experiment and found that any two random US citizens were connected by an average of six acquaintances [63]. Milgram sent 60 letters to various people in Omaha, Nebraska and asked them to forward them to a stockbroker in Sharon, Massachusetts. The participants were required to forward the letters to their acquaintances, whom they thought might be able to reach the stockbroker. Though Milgram's initial experiment had a very poor completion rate, subsequent experiments by Milgram and other researchers had good completion rates. Watts and Strogatz showed the small-world phenomenon is common in a variety of realms ranging from C. elegans neurons to power grids [92]. They showed that by adding a handful of random links they could turn a disconnected network into a highly connected one. This has both positive and negative implications: a vast communication network like the internet could be made a few hops wide by addition of a few judicious routers; in contrast, in a social network, it places an individual a mere six people away from a deadly disease such as SARS. The average distance between vertices in such small world networks is of the order of log log *n* [27].

1.2.2 *Scale-free property*

Barabàsi, Albert and Jeong working to study the topology of the World Wide Web [7] modeled the web as a directed graph with vertices representing HTML documents and edges representing hyperlinks or URLs that point from one document to another. They discovered that the graph thus formed had a small number of vertices with a high in/out-degree and a large number of vertices with small in /out-degree. This property was independent of the size of the graph and hence termed *scale-free*. This distribution of degrees was remarkably different from the Poisson distribution usually found in classical random graphs. Such a heavy tailed distribution is known as power-law and is explained further in the following section. The high degree vertices act as hubs and increase the overall connectivity of the graph thus reducing the average distance between vertex pairs. Scale-free networks are very resistant to random failure of vertices and show no sign of degradation. Due to high connectivity of hub vertices and the low probability of failure under random conditions make these networks robust. However a planned attack on the hub vertices could bring down a scale-free network in very little time. When a graph is said to exhibit the scale-free property it is also a small world [7].

1.2.3 *Power-law degree distribution*

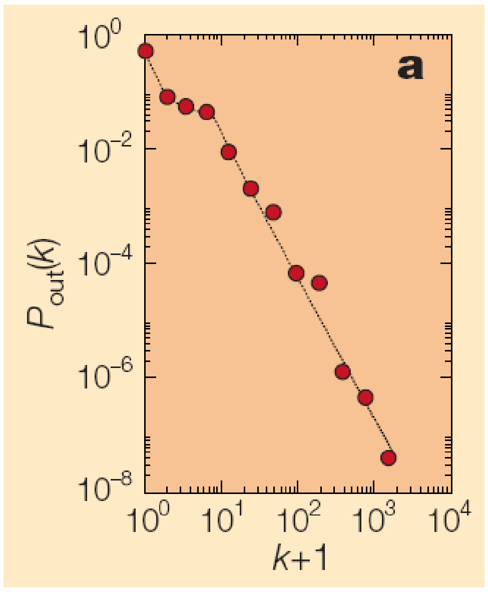


Figure 1: Out degree distribution of WWW.

A *power-law* relationship between two scalar quantities x and y can be expressed as shown below

$$y = ax^k .$$

Here *a* is some proportionality constant and *k* is the exponent of the power-law. *Degree distribution* of a graph is a function that describes the total number of vertices in the graph with a given degree. Statistically it has been determined that for most of the real-world networks the degree distribution follows a power-law and is given by

$$P\{d_u = d\} = \frac{1}{d^{\gamma}},$$

Here $P(d_u = d)$, is the probability that a vertex *u* has the degree *d* and γ is the exponent of the power-law. Chung and Lu claim that the real-world networks have a γ value between 2 and 3 [27]. Figure 1 shows the out degree distribution of nodes in the World Wide Web, the data used was obtained from a crawl of the nd.edu domain [4].

1.2.4 *Network transitivity*

Watts and Strogatz [92] first introduced the concept of clustering co-efficient in real world networks. It refers to the property that, two vertices, which are both neighbors of a third vertex, have a very high probability of being neighbors of one another. Section 4.1.6 provides more insight into this property. Network transitivity was introduced as an alternative to clustering co-efficient by Newman, Watts, and Strogatz [72]. The transitivity of a graph is defined as:

$$T_G = \frac{3 \times N_{\Delta}}{N_{\Lambda}},$$

where N_{Δ} is the number of triples of vertices (v_1, v_2, v_3) in the graph (*triangles*), where each member of the triple is connected to the remaining two (subgraph K_3). N_{Λ} is the number of triples of vertices (v_1, v_2, v_3) such that, at least one member of the triple is connected to the remaining two. The 3 in the numerator signifies the fact that every triangle contains has three sets of triples such that, one vertex is connected to the remaining two. The resulting value of T_G lies between zero and one. The null graph has a transitivity of zero and a complete graph has the transitivity of one. Real-world networks have a transitivity value that is typically between 0.1 and 0.5 [44]. Comparatively classical random graphs have very low transitivity.

1.2.5 *Community structure*

One property of recent interest is the *community structure*, it turns out the real-world networks organize themselves such that they are locally dense and globally sparse [70]. Each of these locally dense regions constitutes a *community*. The nodes of a community are similar in some aspect. Identifying these communities is the primary goal of my research. The community structure of a network would reflect the natural divisions of the network into densely connected subgraphs. Figure 2 shows the dominant communities present in the high school friendships network, color-coded to represent students from different races [69]. We will investigate the community structure property in detail in the following section.

1.3 Community

The term *community* has several definitions. Initially people thought of using *cliques* and *near-cliques* to define communities in graphs with the idea that high connectivity corresponds to similarity between vertices [9].

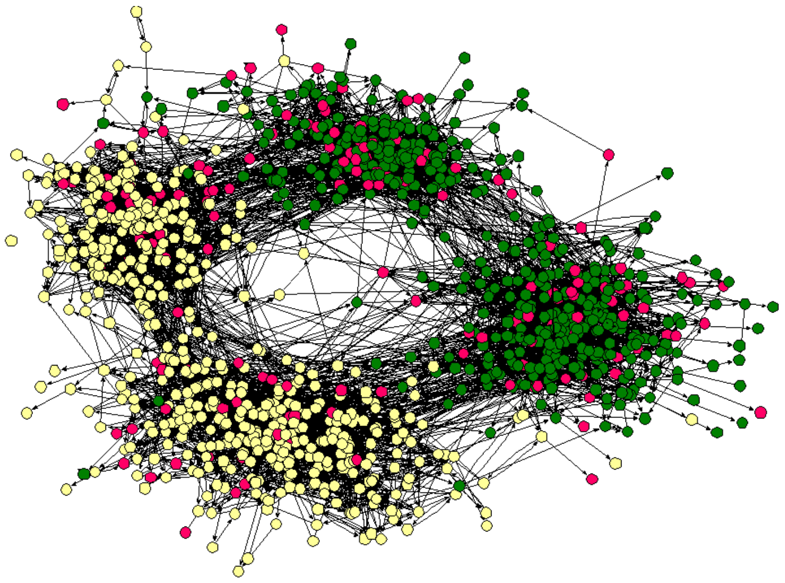


Figure 2: Communities in a high school friendships network**.**

Kleinberg while studying web graphs introduced the concept of “authorities” and “hubs”. *Authorities* are web pages, which are highly referenced (vertices with high in-degree), and *hubs* are web pages that reference many (high out-degree) authority pages. Later Gibson, Kleinberg and Raghavan define communities in web graph as a core of central, authoritative pages connected together by hub pages [42].

Kumar et al. define communities as bipartite cores: a *bipartite core* in a graph G consists of two (not necessarily disjoint) sets of vertices L and R, such that every vertex in L links to every vertex in R [57]. Vertices within L or R could be adjacent to one another.

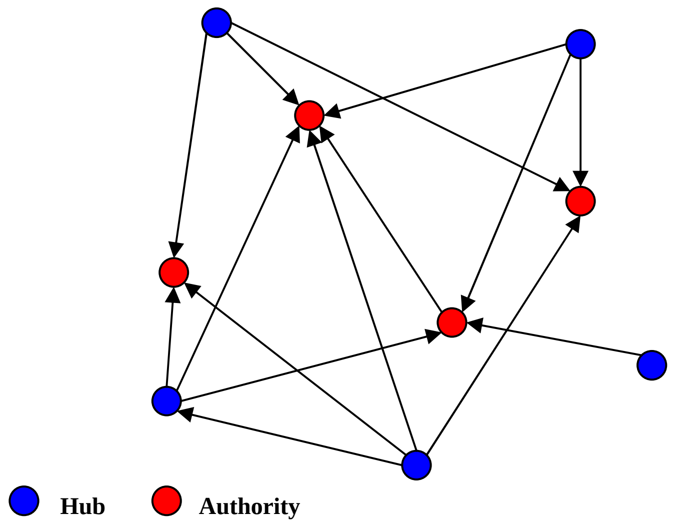


Figure 3: Hubs and authorities.

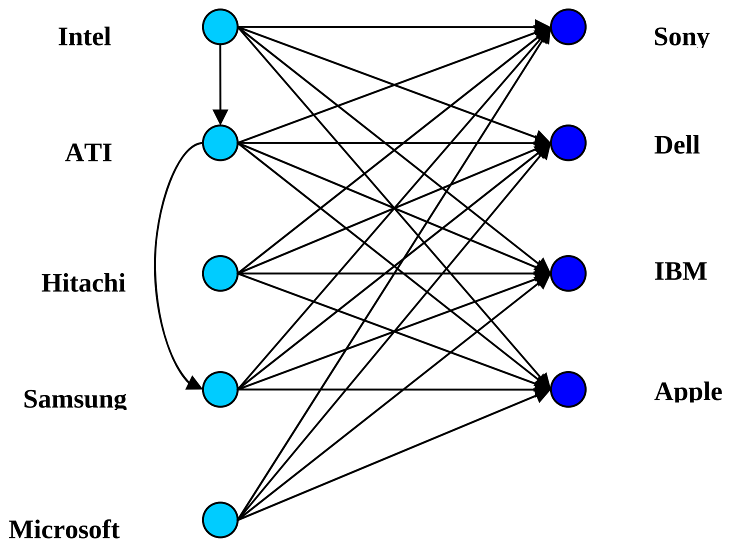


Figure 4: A bipartite core.

Flake, Lawrence and Giles define them as a set of vertices C in a graph G that have more edges to members of the community than to non-members [36]. This definition of community is similar to the concept of defensive alliances introduced by Hedetniemi et al. [49]. A *defensive alliance* in a graph $G = (V, E)$ is defined as a non-empty set of vertices $S \subseteq V$ such that for every vertex $v \in S$, $|N[v] \cap S| \geq |N[v] - S|$ where $N[v]$ represents the closed neighborhood of vertex v.

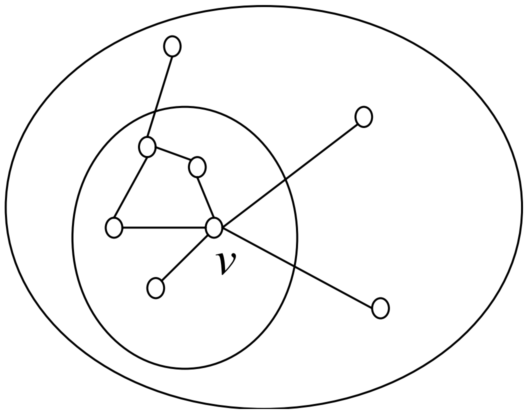


Figure 5: Communities based on graph alliances.

Girvan and Newman define communities based on edge density: subsets of vertices

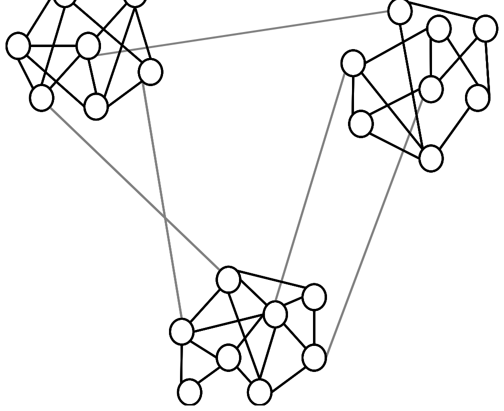


Figure 6: Communities based on edge density.

within which edges are dense, but between which edges are sparse [44].

The existence of no one clear definition for community makes the task of extracting them from the graph more difficult. The general approach up until now has been to come up with a definition for the term community and then devise algorithms that would extract such structures from the input graph.

1.4 Random graph models

Though the first random graph model [34] was developed in 1959, no significant developments were made till the late 90's. Recent years have seen a high level of interest in the topology of complex networks. This has resulted in development of a number of random graph models. Most of the current work concentrates on real-world random graphs. Unlike the classical random graphs, real-world graphs are not uniformly random. They posses certain organizational-traits which are predictable. The general approach in designing these models has been as follows: a few experimentally deduced properties of real-world networks are used to design a mathematical model that exhibits these properties. The Model is then analyzed to deduce additional properties and finally the newly derived properties are validated against real-world data. This process is repeated several times to obtain models that are as accurate as possible. Having an accurate model serves several purposes: (i) it would help us understand the evolution of such networks; (ii) it would provide us with a synthetic graph to test the algorithms, developed for the real-world networks; (iii) understanding the topology of these networks would aid us in designing efficient algorithms for such networks. Despite the dynamic nature and enormous size, one of the main hurdles in working with random graphs is their non-deterministic nature.

Random graph models can be classified into two (i) static and (ii) dynamic. In a *static* model the vertices are added first and then edges are introduced one by one between pairs of vertices selected at random with a biased or uniform probability. In a *dynamic* model, at every iteration, one new vertex and a few edges are introduced that connect the new vertex to the exiting vertices (selected using a biased or uniform probability). The following sections provide some insight into the evolution of different network models and their characteristics.

1.4.1 *Classical random graph model*

The theory of random graphs lies in the intersection between probability theory and graph theory. The $G_{n,m}$ model introduced by Erdos and Renyi [34] and the $G_{n,p}$ model introduced by Gilbert [43] are known as classic random graphs. Both these models are static models. The $G_{n,m}$ model is defined as follows: $G_{n,m}$ represents a set of all undirected, simple and labeled random graphs of order n and size m. The set consists of $\binom{M}{n}$ elements, where $M = n \times (n-1)/2$. Each member is a classical random graph and has a $1/|G_{n.m}|$ chance of occurring. The $G_{n,p}$ model represents a set of all undirected, simple and labeled random graphs of order n in which every potential edge of the graph occurs with a probability of p. Bollobas shows that both $G_{n,m}$ and $G_{n,p}$ refer to the same graph when $m = p \times M$ [11].

Algorithm 1 can be used to generate a classical random graph. The algorithm is very straightforward. It starts with a graph with n isolated nodes. Every edge in the graph is introduced with a probability of p. The function **Rand**() generates a uniform random number between 0 and 1.

Let us now, look into some of the properties of these graphs. The degree distribution of these networks follows a binomial distribution. This can be explained as follows: Let v be a vertex picked uniformly at random from $G_{n,p}$ and let $\mathrm{P}(v)$ denote the probability distribution function of the random variable d_v. There are $(n - 1)$ vertices that could be adjacent to v with a probability of p, it follows that d_v would follow a binomial distribution with the parameters $(n - 1)$ and p, *i.e.*, $\mathrm{P}(k) = \binom{n}{k} p^k (1-p)^{n-k}$. This is in contrast to the power-law degree distributions observed in the real-world networks. The independence of the edges from one another, affects the overall transitivity of these graphs. This accounts for the low transitivity values in comparison to the real-world networks. One property that is in agreement with the real-world networks is the small-world property. It has been observed that for several values of p, $G_{n,p}$ satisfies the small-world property [12, 18, 25].

```
**ClassicalRandomGraph**($n$, $p$)

$V = \{v_1, v_2, \ldots, v_n\}$
$E' = \{e_1, e_2, \ldots, e_{nx(n-1)/2}\}$
$E = \phi$
for every edge $e_i \in E'$ do
        $r$ = Rand()
        if $r \leq p$ then
                $E = E + \{e_r\}$
        endif
endfor
```

Algorithm 1: To generate classical random graph.

1.4.2 *Watts and Strogatz model*

Watts and Strogatz realized that the connection topology in real-world networks is neither completely random nor completely regular, but lies somewhere in-between [92]. They came up with a network model that could be tuned to this middle ground. They achieved this by taking a regular network (such as a ring lattice) and rewiring its edges into disarray. The end result is a highly clustered network with small average path lengths. A *minimal lattice* is a path, bending this linear lattice and connecting the end vertices with an edge creates a *nearest-neighbor ring lattice* or cycle. This consists of n vertices and n edges, and every vertex is of degree 2. If one were to include edges to the nearest-neighbors and to the next nearest-neighbors, the ring lattice would consist of $2n$ edges and each vertex would be of degree 4. One could create denser ring lattices by connecting the vertex to more of its nearest neighbors.

Their rewiring procedure can be explained as follows: Start with a ring lattice, which has n vertices and k edges per vertex. For each edge e in the graph rewire one if its ends with a probability p, to another vertex, chosen uniformly at random.

Algorithm 2 can be used to create a Watts and Strogatz model random graph. The function **CreateRingLattice**(n, k) creates a new ring lattice graph $G(V, E)$, where in each vertex has k neighbors and the function **GetARandomNewNeighbor**(u) returns a vertex v such that $v \in V$ and initially $e_{uv} \notin E$. One could tune the value of p and construct graphs that are completely random (when $p = 1$) or completely regular (when $p = 0$). For values $0 < p < 1$, we get graphs with the desired properties. Figure 7 shows the graphs obtained by varying the value of p. The two properties that Watts and Strogatz were interested in particular were the average path length,

L_{avg}, which is a global property and clustering coefficient, C_{coeff}, which is a local property. They discovered that it is required that $n >> k >> \ln(n) >> 1$, where $k >> \ln(n)$ guarantees that the resulting graph will be connected. It was also noted that the regular lattice at $p = 0$ is a highly clustered, large world where L_{avg} grows linearly with n, whereas the random graph at $p = 1$ is poorly clustered, small world where L_{avg} grows logarithmically with n.

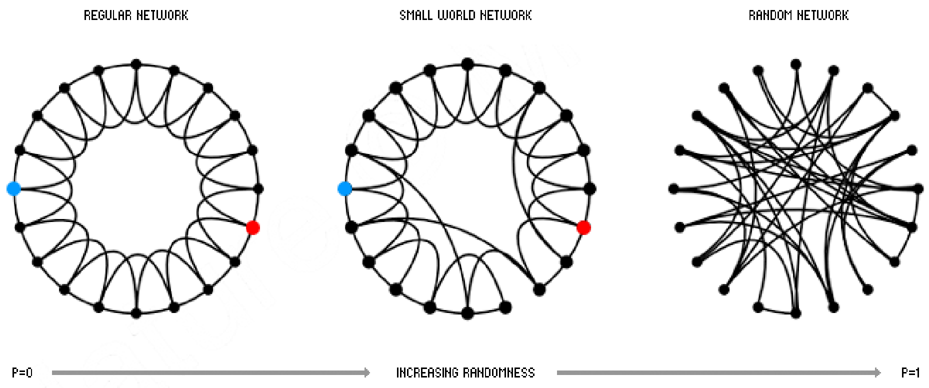


Figure 7: Watts and Strogatz models obtained by varying p between 0 and 1.

Though the Watts and Strogatz model was able to mimic the small world property found in real-world networks it failed to account for the power law degree distribution. As with the classical random graph model, the probability of finding nodes with large degree decreases exponentially.

```
WattsStrogratzGraph($n$, $k$, $p$)
$G(V, E)$ = CreateRingLattice($n$, $k$)
for $neighbor$ from 1 to $k$ do
    for each $c \in V$ do
        $r$ = Rand()
        if $r \leq p$ then
            $E = E - \{e_{i,i+k}\}$
            $j$ = GetARandomNewNeighbor($v_i$)
            $E = E + \{e_{i,j}\}$
        endif
    endfor
endfor
```

Algorithm 2: To generate Watts and Strogratz random graph.

1.4.3 *Preferential attachment model*

Barabàsi and Albert proposed the preferential attachment random graph model [7]. It modeled the *richer get richer* behavior typically observed in social networks. Unlike the previous two models which were static this is an evolving dynamic model, defined as a discrete time process $\{G_t(V_t, E_t)\}_{t > 1}$. Let $G_t(V_t, E_t)$ refer to the state of the graph at time t. For all values of $t > 1$, G_{t+1} can be obtained by applying some stochastic rules to the graph G_t. The process starts with some m isolated vertices at time $t = 0$. At every incremental time step, $t + 1$, one new vertex, v_{t+1} and m' ($m' < m$) edges are introduced into the graph. Each of the m new edges are connected to the vertex v_{t+1} at one end and at the other end they are connected to a vertex $v_i \in V_t$, that is chosen at random with some bias. The probability with which a new edge would be connected to the vertex v_i is directly proportional to its degree $d^t_{v_i}$.

$$P_{t+1}(v_i) = \frac{d^t_{v_i}}{\sum_j d^t_{v_j}} \text{ for } v_i \in V_t.$$

The main issue with this approach is that at time $t = 1$, $P_1(v_i) = 0$ for all $v_i \in V_0$. This is rectified by starting with a graph, G_0 that contains no isolated vertices [13]. After t time steps, the graph has $t + m$ nodes and $m' \times t + |E_0|$ edges.

Algorithm 3 can be used for generating a preferential attachment model random graph. The function **GenereteRandomGraph**(m) generates a connected classical random graph with m vertices and m edges and the function **GetPreferentialVertex**(G) returns at random the index of one of the vertices from the graph G with a probability proportional to its degree. The output of the algorithm would be a random graph G_t with $t + m'$ vertices and $m' \times (t + 1)$ edges.

The preferential attachment model addresses several shortcomings that were present in classical random graphs: (i) it is dynamic similar to most of the real-world networks, (ii) it has the small world property, and (iii) its degree distribution follows a power law. Kumar et al. while studying frequently occurring substructures in the web graph [57], found that it contains a large number of locally dense subgraphs, and none of the models discussed so far, including the preferential attachment model contain locally dense subgraphs.

1.4.4 *Copy model*

Kumar et al. came up with a directed dynamic random graph model [57]. They wanted to reproduce the large number of *bipartite cores* that occur in web graphs. To generate such a graph, one starts with a small random graph, G_0 with n_0 nodes at time $t = 0$. At each additional time step $t + 1$ a new vertex, v_{t+1} is introduced with m' outgoing edges. Now pick a node (uniformly at random) from the set of already existing vertices, and call it the prototype vertex, v_{pro}. For every i-th outgoing edge, with a probability α, connect it to one of the existing vertices selected uniformly at random, and with a probability $1 – α$, connect it to the vertex pointed by the i-th outgoing edge of v_{pro}.

Algorithm 4 can be used to generate a copy model random graph. The function **GetRandomVertex**(G) returns a vertex uniformly at random from the graph G and the function **GetJthOutlink**(v, j) returns the index of the vertex pointed by the j-th outlink of vertex v.

```
**PreferentialAttachmentGraph**($t$, $m'$)

$G_0(V_0, E_0)$= **GenereteRandomGraph**($m'$)

**for** $i$ from 1 to $t$ **do**

    $V_i = V_{i-1} + \{v_{i+m'}\}$
    **for** $j$ from 1 to $m'$ **do**

        $k$ = **GetPreferentialVertex**($G_{i-1}$)

        $E_i = E_{i-1} + \{e_{i+m',k}\}$

    **endfor**

**endfor**
```

Algorithm 3: To generate preferential attachment random graph.

The intuition behind the model is as follows: each time a new web page is created it would be on a particular topic, the author of the web page would include a few hyperlinks in his page that are already present in another web page that deals in the same topic and a few new hyperlinks are added to give his web page a new flavor. The Copy model apart from being scale free and having a power law degree distribution, contains many bipartite subgraphs (*bipartite cores*).

1.4.5 *Models with embedded communities*

Tawde, Oates and Glover proposed a web graph model, which has communities, embedded in them so as to satisfy the microscopic as well as macroscopic properties of the World Wide Web [88]. Unlike the previous two models, this one is not stochastic/dynamic.

They use a three-step procedure where, in the first step stand-alone communities are generated by using the preferential and random link distribution technique described by Pennock et al. in [77]. This would result in communities that have a few dangling links which would later be used to connect the community to the rest of the web. In the second step the communities generated above are combined by using the community interaction model suggested by Chakrabarthi et al. in [20]. This model is based on observed data for communities on the web. Chakrabarthi et al. experimented with 191 topics from DMoz[2] and generated a 191 x 191 matrix, which modeled the interaction among these communities.

[2] http://www.dmoz.com

```
CopyModelGraph( $t$, $m'$, $\alpha$)
$G_0(V_0, E_0)$= GenereteRandomGraph($m'$)

for $i$ from 1 to $t$ do

    $V_i = V_{i-1} + \{v_{i+m'}\}$
    for $j$ from 1 to $m'$ do

        $pro$ = GetRandomVertex($G_{i-1}$)

        $r$ = Rand()

        if r < $\alpha$ do

            $k$ = GetRandomVertex($G_{i-1}$)

            $E_i = E_{i-1} + \{e_{i,k}\}$

        else

            $k$ = GetJthOutlink($v_{pro}, j$)

            $E_i = E_{i-1} + \{e_{i,k}\}$

        endif

    endfor

endfor
```
Algorithm 4: To generate copy model random graph.

Element *i*, *j* of this matrix gives the empirical probability that an outlink, selected at random from a page in community *i* will link to a page in community *j*.The dangling links from the communities are used to perform this interconnection. Finally in the third step a benchmark link distribution of a web graph is used as a target and additional nodes and edges are added as required to achieve the link distribution of the target. Though this model is one of the very few which try to achieve an embedded community structure, it is to be noted that the model is not a purely evolving model like the ones mentioned previously.

2. EVOLUTION OF THE WORLD WIDE WEB

2.1 Importance of this work

This chapter throws some light on how the different properties of the World Wide Web evolve over time. Since its inception the World Wide Web has grown from a few thousand to several billion pages. Most of the work on real-world random networks has concentrated on their properties like degree distribution, average distance between node pairs, network transitivity, and community structure. The knowledge acquired was used to create synthetic models for these networks. Despite their dynamic nature, the above properties were studied over static instances of these networks. It would be interesting to analyze the evolution of such networks and observe how their properties change over time.

In this chapter we model the "ucf.edu" web domain as a graph and study its evolution and change in properties over a five-year period. The mapping for this model is as follows: every web page in the domain is a vertex and every hyperlink on a web page pointing to another webpage is a directed edge connecting the corresponding nodes.

There are several implications of this study: The trends and variation in properties could be used to design better models for web graphs and to evaluate existing ones. The data could be used for anomaly detection, for example, to identify spamming. Extrapolation of the collected data could help us predict properties of the network in future and also estimate its properties during a time frame when the data is not readily available. One could study the evolution of communities within these graphs and identify merging/partition of existing communities.

2.2 Related work

We are aware of very few publications discussing properties of real-world networks over time. In a recent paper Leskovec et al. [61] study edge density and diameter of citation graphs, affiliation graphs, and autonomous system graphs over time. The work showed some interesting results. First, they show that the number of edges grow super-linearly in the number of nodes and claim that most of these graphs densify over time, Second, they show that the diameter shrinks over time, in contrast to priori studies which state that this parameter would grow slowly as a function of the number of nodes. Redner [81] has analyzed the properties of citation networks using the datasets obtained from the journal Physical Review. The datasets used cover citations over a 110-year period. In another independent work Katz [53] discovered densification of power laws for citation networks. Grossman [47] has performed a more statistical observation of the mathematical research collaboration graphs obtained from the journal Mathematical Reviews. Toyoda and Kitsuregawa study the evolution of communities in the ".jp" web domain [89]. They show that the distribution of size of the communities follows a power law and its exponent does not change much over time.

2.3 Experimental setup

Studying graph properties at discreet time intervals has been difficult due to the lack suitable datasets. Another major draw back was storage requirements for such datasets. Web graphs are modeled based on crawls obtained by a web crawler. But these crawls can only provide us with the most recent version of the web pages. To study evolution in web graphs one needs web graph data over several years. With cost of storage decreasing, a number of organizations have started archiving their old web pages for future reference. The *Wayback Machine* is an online archive that collects and stores contents of the crawls by the Alexa crawler. It has stored copies of web pages dating back to the year 1996. It provides users with a web interface to obtain archived versions of web pages when queried with an URL and date.

To construct the web graph of the ucf.edu domain we designed a special purpose downloading module on top of the UcfBot crawler, designed by myself and Cami (see Section 7), that would query the Wayback Machine archives and obtain stored versions of the web pages. Each query to the Wayback Machine consists of a URL and a date stamp. Our crawler starts with a few seed URLs that are appended with a time stamp and converted into query form. These queries are now submitted to the Wayback Machine. The page retrieved is then parsed to extract hyperlinks in the page. Each hyperlink is then converted into a query and the process is repeated until all the web pages from the specified time period are obtained.

The crawl obtained is now modeled into a web graph with each web page, representing a vertex in the graph and each hyperlink on the page pointing to another page, representing a directed edge connecting the corresponding vertices. We obtain five such web graphs by performing separate crawls one for each year from 1997 to 2001. These web graphs represent samples of the ucf.edu web domain during a particular year. The validity of our datasets depends on how comprehensive the Alexa crawls were during the past years. Our experiments show that the obtained datasets cover a large portion of the indexable ucf.edu domain during the specific time periods.

2.4 Empirical observations

In this section we study the evolution of the “ucf.edu” domain by analyzing the properties of its web graphs at regularly spaced points in time. The *size* (number of edges) and *order* (number of nodes) of our datasets are given in Table 1.

Table 1: Size and order of the web graphs over the five-year period. t represents time.

t	*n*	*m*
1997	7260	15998
1998	10974	16795
1999	12444	31578
2000	31170	80940
2001	58957	185909

We analyze several properties of the web graphs like number of new nodes being introduced, average degree, edge density, in-degree and out-degree distributions, diameter, and average distance between nodes. Our observations are given below.

2.4.1 *New nodes*

Using the web graph data we compute the number of nodes that were introduced into the graph every year from 1998 to 2001. This number increases steadily every year. Figure 8 shows the increase in the number new nodes over time. The *x*-axis represents time in years and *y*-axis represents the number of new nodes being introduced.

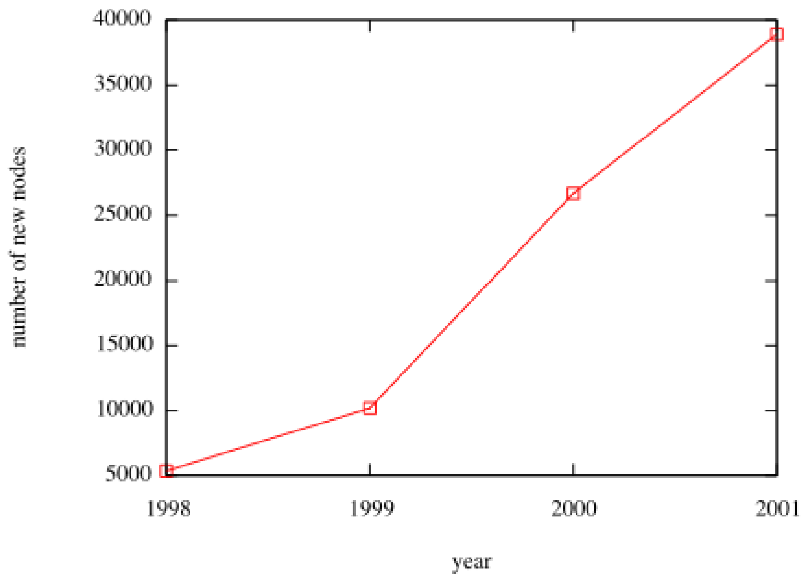


Figure 8: Number of new nodes introduced in the network over time.

2.4.2 *Densification power law and average out degree*

Most of the existing literature about evolution of real-world networks suggests that the number of edges grows linearly with the number of nodes and as a result the average degree remains a constant. Leskovec et al. [61] observe that the number of edges grow super-linearly in the number of nodes and this growth follows a power law pattern given by:

$$m_t \propto n_t^{\gamma}, \tag{1}$$

where m_t represent the number of edges and n_t the number of nodes in the network at time t, γ is the densification exponent and its value lies strictly between 1 and 2 for connected graphs. The graph is sparse when the value of γ is close to 1 and is dense when the value of γ is close to 2. Leskovec et al. call this *densification power law* and suggest that networks are becoming denser over time.

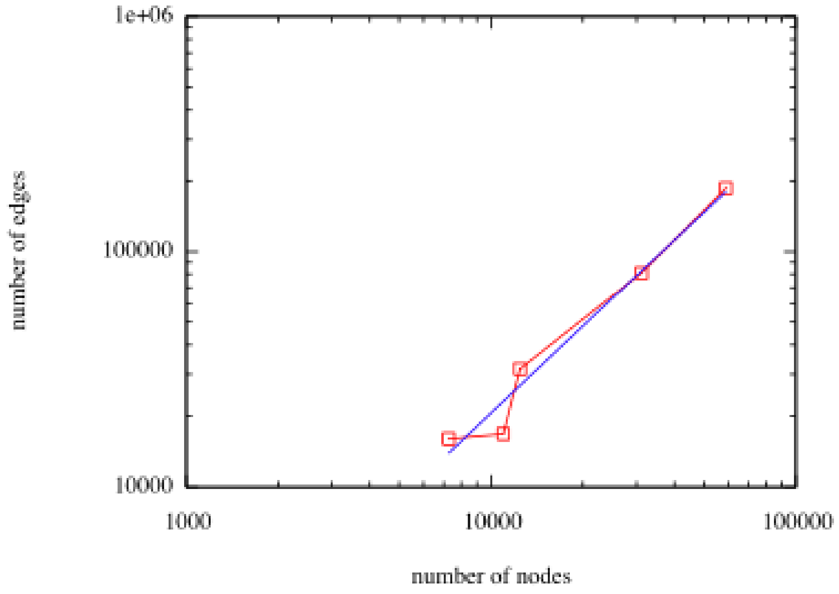


Figure 9: Number of edges m_t versus number of nodes n_t in log-log scales.

Figure 9 shows our observations from the ucf.edu datasets. The *x*-axis represents the number of nodes and *y*-axis the number of edges in log-log scale over the five-year period. Our results echo the results from [61]. The densification exponent γ in our case has a value of 1.22. The plot in Figure 10 shows the average node out degree over the five-year period. We see that the average out degree increases over time. Though our results about super-linear growth of edges and increase in average degree coincide with the results obtained by Leskovec et al., we see that the graphs are not getting denser over time. We explain this further in the upcoming section.

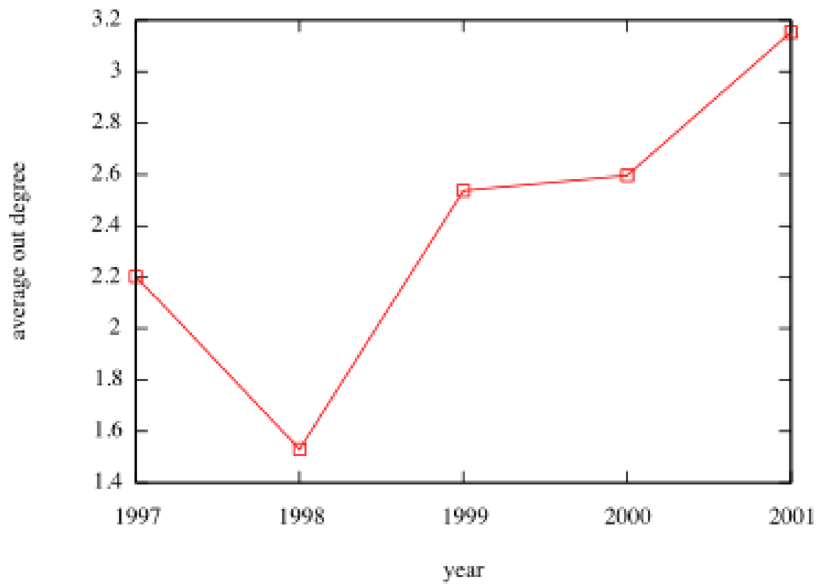


Figure 10: Average node out-degree over time.

2.4.3 *Edge density*

Edge density is the ratio of edges present in the graph to the maximum number of edges that could be present. For a directed graph the maximum number of edges is given by n (n - 1). The change in edge density over time is shown in Figure 11, x-axis represents time and y-axis edge density.

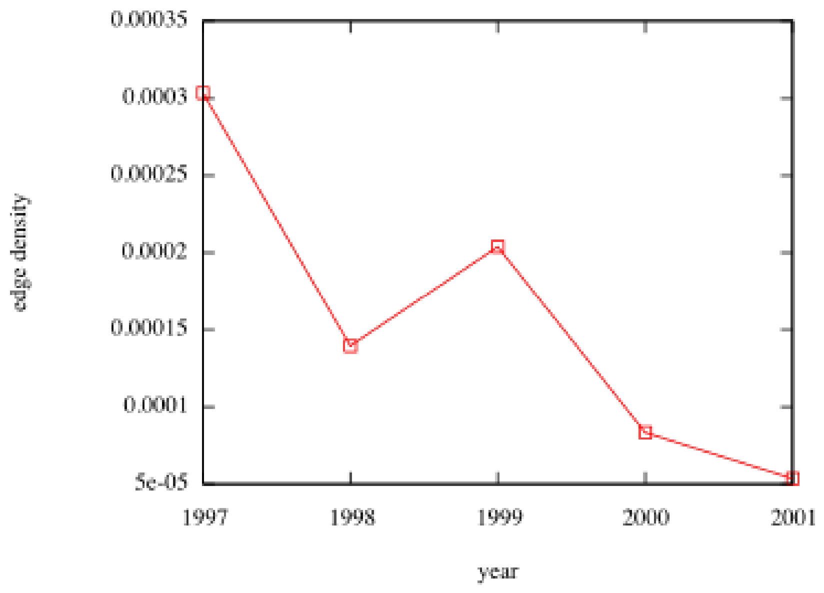


Figure 11: Edge density over time.

Our results suggest that the edge density of these web graphs decreases over time. This seems surprising considering the fact that the average degree is actually increasing. Now we reason about this strange behavior. Edge density of a directed graph is given by

$$\frac{m_t}{n_t(n_t - 1)},$$

where m_t represents the number of edges and n_t the number of nodes at time t. From equation 1 we get

$$\frac{m_t}{n_t(n_t - 1)} \propto \frac{n_t^{\gamma}}{n_t(n_t - 1)},$$

where γ is the densification exponent. To prove that the edge density decreases over time we need to show that

$$\frac{\dfrac{n_t^{\gamma}}{n_t(n_t-1)}}{\dfrac{n_{t+1}^{\gamma}}{n_{t+1}(n_{t+1}-1)}} > 1,$$

We know that $n_{t+1} = n_t + 1$.

$$\frac{n_t^{\gamma}}{n_t(n_t-1)} > \frac{(n_t+1)^{\gamma}}{n_t(n_t+1)},$$

$$\left(\frac{n_t}{n_t+1}\right)^{\gamma} > \frac{n_t-1}{n_t+1},$$

$$\left(\frac{n_t}{n_t+1}\right)^{\gamma} > \frac{n_t}{n_t+1} - \frac{1}{n_t+1},$$

When n_t is large $n_t + 1 \cong n_t$. Substituting this in the equation proves the inequality and shows that $\frac{n_t^{\gamma}}{n_t(n_t-1)}$ is a monotonically decreasing function. This explains the decreasing edge density over time.

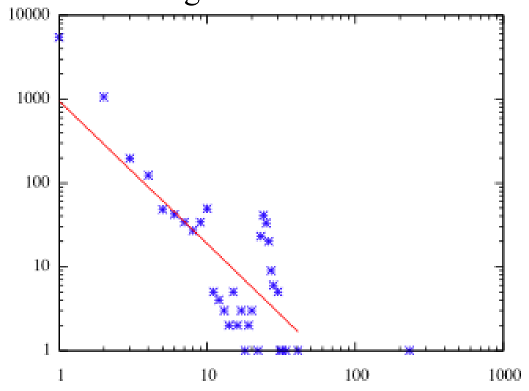
2.4.4 *Degree distributions*

Figure 12 shows the in-degree and out-degree distributions of our datasets during the five-year period. Each row represents a year from 1997 to 2001, the plots in the first column represent in-degree distributions and the ones in second column represent out-degree distributions. In each of the plots *x*-axis represents degrees and *y*-axis represents number of nodes. The plots for 1996 and 1997 seem to contain some noise but from 1998 the power law property is very evident. The presence of the power law, which is believed to be a basic web property, validates our data sets.

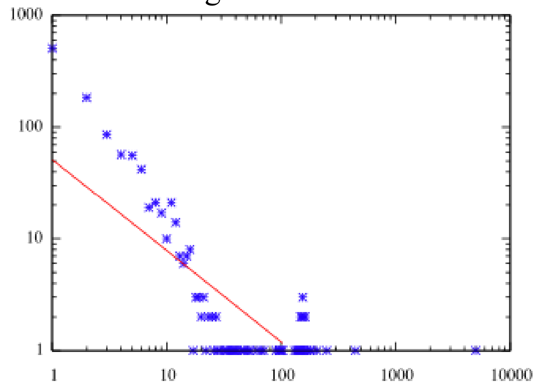
In-degree distributions

Out-degree distributions

1997

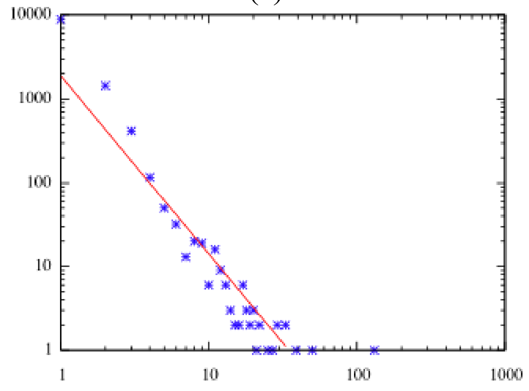


(a)

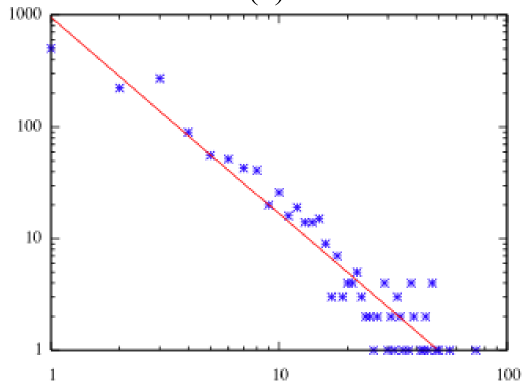


(b)

1998

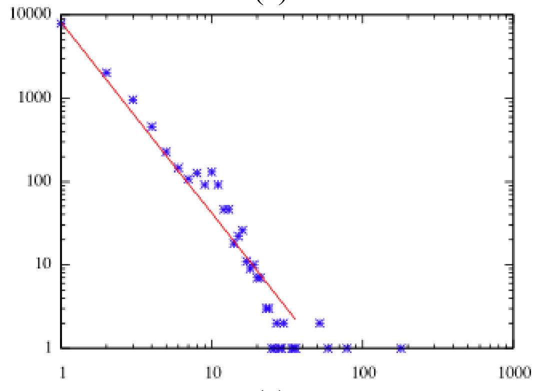


(c)

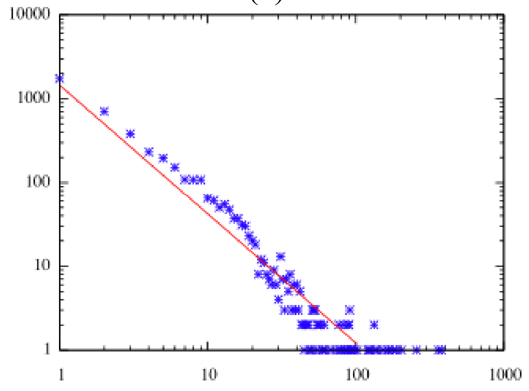


(d)

1999



(e)



(f)

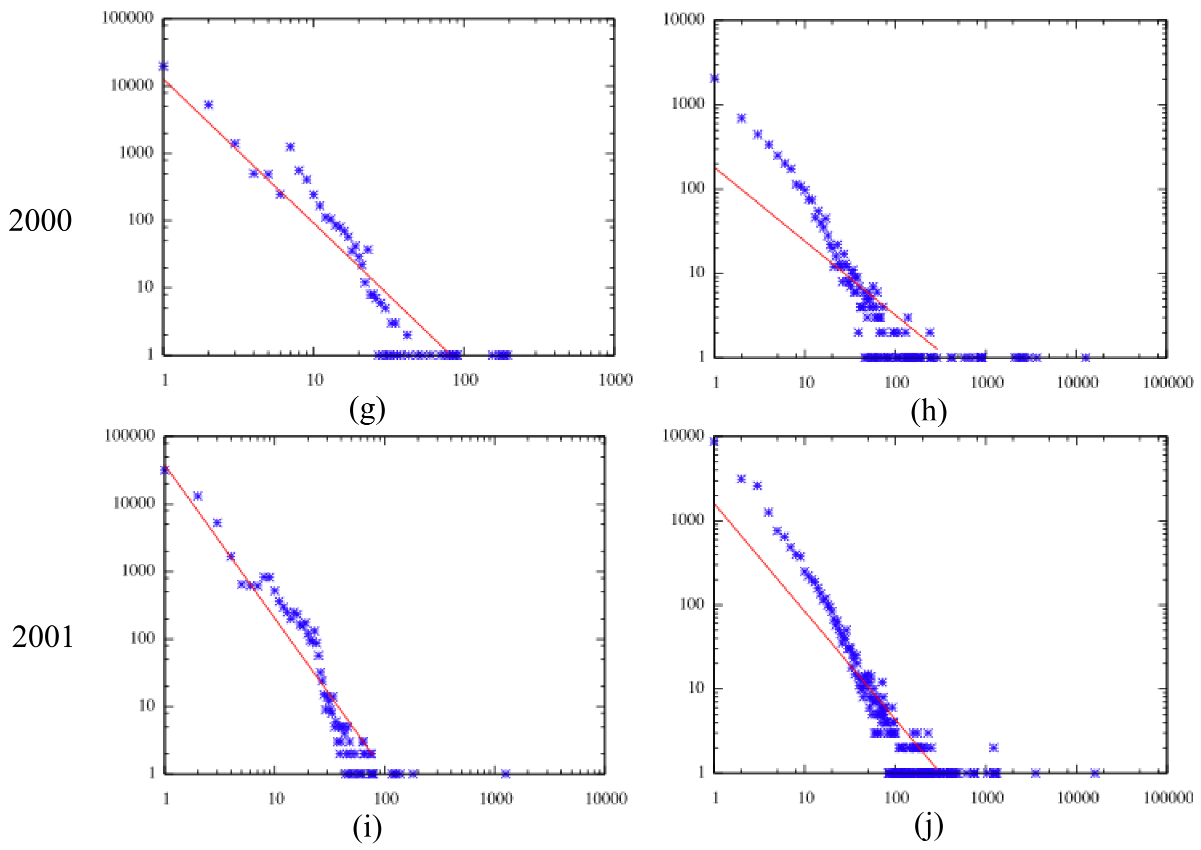


Figure 12: Degree distributions from 5 years of ucf.edu domain web graphs.

It is interesting to note that the initial segment of the out-degree distributions deviates significantly from the power law. This has been observed by Broder et al. in [16] and they suggest that pages with low out-degree follow a different (possibly Poisson or a combination of Poisson and power law) distribution. The power-law exponents of our datasets were between 1 and 2.5, which is a bit deviant from existing datasets. We believe this is due to the small order of our graphs compared to other web graphs, which consist of at least a few hundred thousand nodes. We did observe a slowly increasing trend in the value of the power-law exponent but we see the need to further validate this observation by experimenting with datasets over a larger time period.

2.4.5 *Diameter and average distance between node pairs*

Another property of interest is the distance between nodes. There are two metrics which are often used: diameter and average distance between nodes. *Diameter* of a graph is defined as the longest of the shortest distances between all node pairs. We compute the diameter of the five datasets and our results are show in Figure 13, *x*-axis represents time and *y*-axis diameter. Unlike the results obtained by Leskovec et al. we did not observe any decrease in the diameter of the graphs over time. However it has to be noted that Leskovec et al. use a different definition of diameter termed the effective diameter. *Effective diameter* of a graph is defined as a distance *d* such that at least 90% of the connected node pairs are at distance of at most *d*.

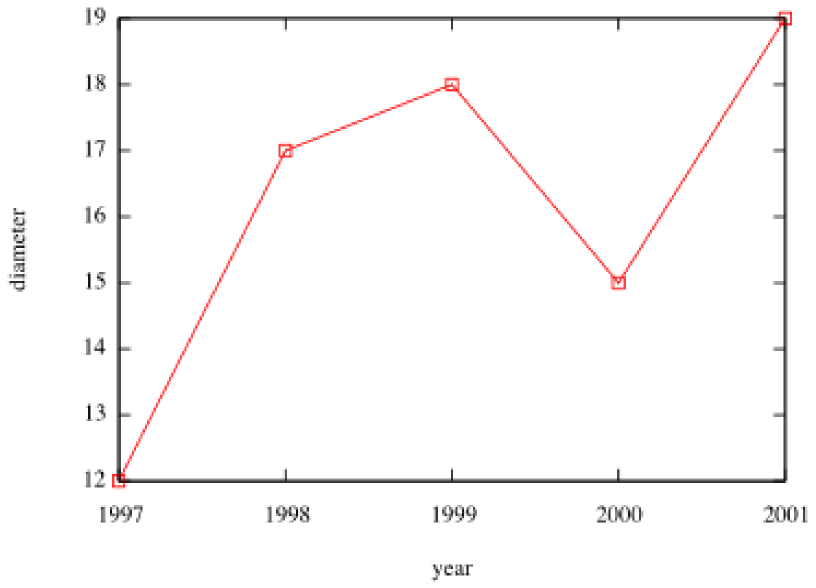


Figure 13: Diameter over time.

Average distance between node pairs: As stated in Section 1.2.1 this metric defines small world graphs. The average distance between nodes for the five datasets has been plotted in

Figure 14, x-axis represents time and y-axis average distance. Except for a major surge during the year 1999 the data seems to suggest that the average distance between nodes is growing at a slow rate.

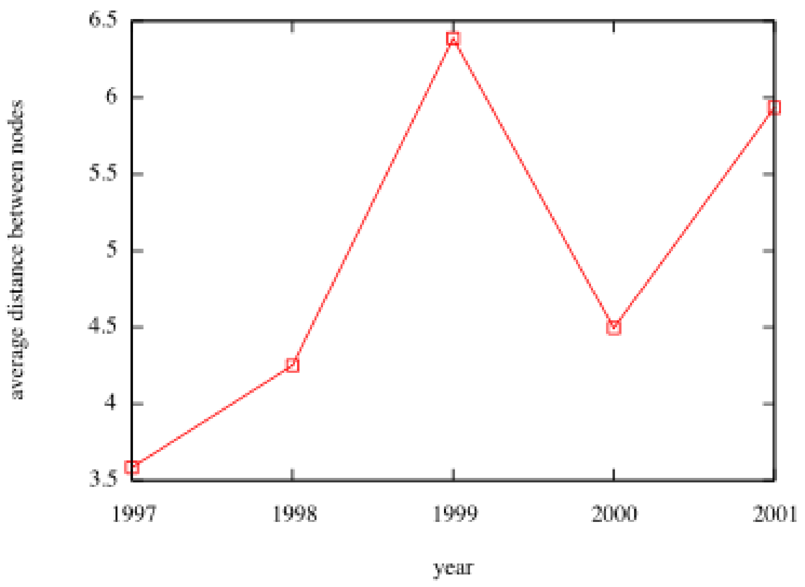


Figure 14: Average distance between node pairs.

2.5 Conclusion

The work proposes a new research track in the field of real-world networks. The knowledge obtained aids in better understanding the evolution of web graphs. Our results support the empirical results about super-linear growth of edges obtained by Leskovec et al. We also observe that the edge density reduces over time and explain this behavior. For future work, we would like to study, the growth and evolution of web communities; design new random graph models that posses the observed properties; and evaluate the validity of existing random graph models.

3. SURVEY OF EXISTING METHODS

3.1 Community related problems

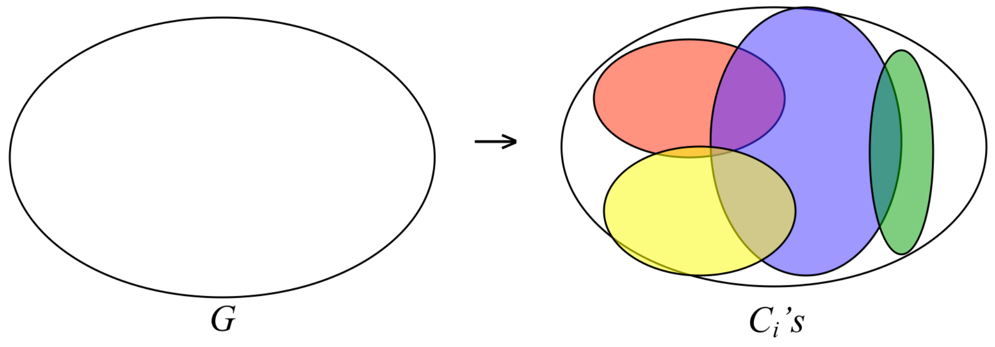


Figure 15: Community discovery.

Two problems that are of interest are community discovery and community identification. From a graph theoretic perspective *community discovery* is classifying vertices of a graph G into subsets $C_i \subseteq V$, $0 \leq i < k$, such that vertices belonging to a subset C_i are all closely related (classifying the nodes in a graph into different communities).

C*ommunity identification* is identifying the community C_i to which a set of nodes $S \subseteq V$ belong (identifying the community that contains the set of vertices S). In general community identification is considered an easier problem compared to community discovery, especially if the input graph is large.

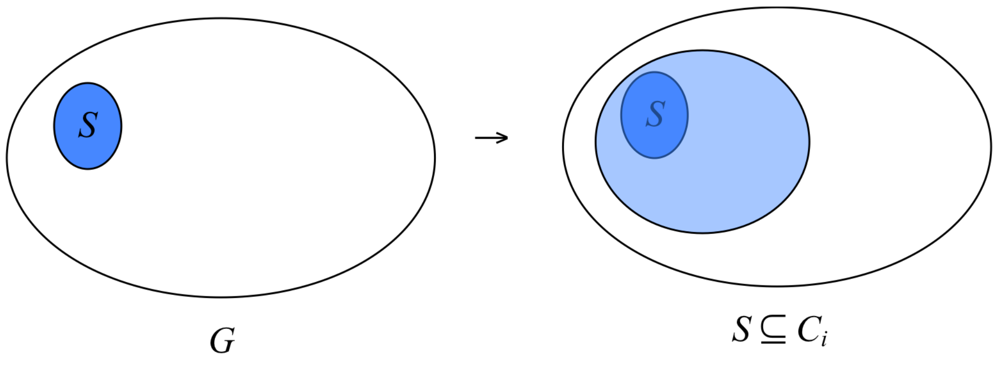


Figure 16: Community identification.

3.2 Existing algorithms

The majority of the existing community discovery algorithms employ hierarchical clustering techniques to extract communities. The hierarchical clustering algorithms work in two phases, at first, a similarity metric is defined to portray the strength of the relationship between a vertex pair. After which there are two possible ways of extracting communities using the defined metric (i) agglomerative and (ii) divisive.

The *agglomerative* algorithms begin by computing the similarity between every vertex pair. Initially each vertex is considered to be an individual community. During the course of the algorithm, closely related vertices are combined together to form bigger communities.

Divisive algorithms on the other hand, compute the similarity between adjacent vertex pairs. Initially the entire input graph is considered to be one large community. During the course of the algorithm, edges are removed between vertices that are least similar. This decomposes the graph into smaller but tighter communities.

Hierarchical clustering is preferred to other methods, because apart from extracting the communities it also provides their hierarchy. A general template for agglomerative and divisive algorithms is given in Algorithm 5 and Algorithm 6. S represents a matrix and its element $S_{i,j}$ represents how similar vertices i and j are, all its elements are initialized to Ø; the function **Similarity**(G, i, j) returns the similarity between vertices i and j; and the function **Max**(S) and **Min**(S) returns the index i, j of the largest and smallest element in the matrix S respectively.

The final for loop could be terminated after desired number of iterations and the components that are present in G' would represent the communities in the input graph G.

3.3 Similarity metrics

The following sections provide some insight into the existing measures used to discover communities.

3.3.1 *Node/Edge independent paths*

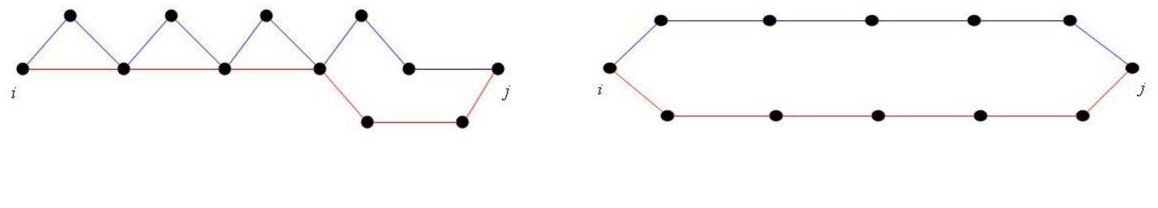


Figure 17: (a) edge independent paths, (b) node independent paths.

```
**AgglomerativeCommunityDiscovery**($G$)
**for all** vertex pairs $(u, v)$ in $G$ **do**
    $S_{u,v} =$ **Similarity**$(G , u, v)$
**endfor**
$V' = \{v_1, v_2, \ldots, v_n\}$
$E' = \{\varnothing\}$
**for** i from 1 to $\frac{n(n-1)}{2}$ **do**
    $u,v =$ **Max**$(S)$
    $E' = E' + \{e_{u,v}\}$
    $S_{u,v} = 0$
**endfor**
```

Algorithm 5: Template for agglomerative procedure.

```
**DivisiveCommunityDiscovery**($G$)
**for all** $e_{u,v} \in E$ **do**
    $S_{u,v} =$ **Similarity**$(G, u, v)$
**endfor**
$V' = V$
$E' = E$
**for** i from 1 to $|E|$ **do**
    $u,v =$ **Min**$(S)$
    $E' = E' - \{e_{u,v}\}$
    $S_{u,v} =$ NULL
**endfor**
```

Algorithm 6: Template for divisive procedure.

To the best of my knowledge there are no prior publications utilizing node/edge independent paths as a similarity metric. Girvan and Newman first suggested this concept in [44]. Two or more paths are vertex-independent (vertex-disjoint) if they don't share any vertex except maybe the initial and final vertices. Similarly, two or more paths are edge-independent (edge-disjoint) if they don't share any edges. Existence of a large number of vertex (edge) - independent paths between a vertex pair indicates better similarity.

The intuition behind this approach is that the number of vertex (edge)-independent paths between vertices in the same community is greater than the number of vertex (edge)-independent paths between vertices in different communities. It is know that the number of vertex (edge)-independent paths between a vertex pair i, j is equal to the minimum number of vertices (edges) that need to be deleted to disconnect i and j [62].

The number of edge independent paths can be obtained by using the max-flow algorithms. Each edge is assumed to posses a unit flow capacity. The cardinality of the min cut would give the number of edge disjoint paths. Algorithm 7 describes the procedure to obtain edge independent paths between a given pair of vertices.

```
**EdgeIndependentPathSimilarityMetric**($G, i, j$)
$s = i$
$t = j$
$G'(V', E') = G(V, E)$
**for all** $e_{uv} \in E'$ **do**
    $C_{uv} = 1$
**endfor**
$M =$ **MinCut**($G', C, s, t$)
```

Algorithm 7: Edge independent paths between *i* and *j*.

The algorithm first assigns one of the input vertices as a source and the other as a sink. Then each edge is assigned a unit flow capacity. The function **MinCut**(G, C, s, t) computes the min cut of the graph G with edge capacities specified by matrix C where s is the source and t is the sink. M would contain the set of edges that form the min cut and $|M|$ is the number of edge independent paths between i and j. If one were to use the Edmonds-Karp algorithm to compute the min cut then the complexity of the above approach would be $O(nm^2)$. For a sparse graph $m \approx n$ and the complexity is $O(n^3)$. Since this has to be computed for all vertex pairs, the overall complexity is $O(n^5)$.

3.3.2 *Edge betweenness*

Girvan and Newman [44] proposed the idea of edge betweenness as a similarity measure based on the concept of vertex betweenness introduced by Freeman [39]. *Vertex betweenness* of a vertex is defined as the number of shortest paths between pairs of vertices that pass through the given vertex. *Edge betweenness* of an edge is defined as the number of shortest paths between pairs of vertices that pass through the given edge. Betweenness in general is a centrality measure used to portray the importance of a vertex/edge within a graph. Centrality is discussed in detail in Section 4.1. This measure is only suited for divisive algorithms, as the value of edge betweenness for node pairs that are not adjacent is not defined.

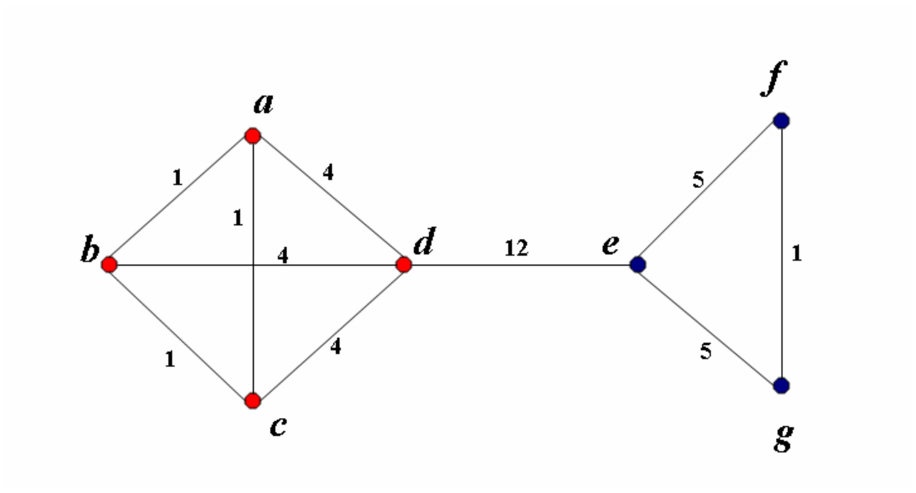


Figure 18: Edge betweenness of edges.

The edge betweenness of inter-community edges would be high, as the shortest paths between nodes in the two different communities would have to pass through them. After computing the edge betweenness of all the edges, one can remove the edges with high edge betweenness and expose the community structure. Figure 18 describes a graph where the weights of the edges represent their betweenness. Clearly the inter-community edge e_{de} has a higher value of betweenness than all the other edges.

The divisive algorithm proceeds as follows: first the edge betweenness of all the edges is computed, the edges with the highest value for edge betweenness is deleted and the edge betweenness of the remaining edges is re-computed. The whole process is repeated for desired number of iterations or until all the edges are removed. Re-computing the edge betweenness, after every iteration is necessary, without which the results were not very impressive.

Computing the shortest path between a vertex pair takes O(m) time. Since we need to compute the shortest path between n^2 vertex pairs, one might assume it would take O(mn^2) time to compute the betweenness of all the edges. In fact computing the shortest path from one vertex to the remaining $n - 1$ takes O(m) time. In effect computing all the shortest paths can be accomplished in O(mn) time. The algorithm requires the betweenness to be computed after removal of an edge. Hence the overall complexity of the algorithm is O(m^2n). For a sparse graph $m \approx n$ and the complexity is O(n^3).

Algorithm 8 can be used to compute the edge betweenness of all the edges in the graph. At first the shortest paths from vertex s to all other vertices is calculated. Then the number of shortest paths are computed by traversing the graph from a leaf node to the source. During the traversal we count the number of successor edges to each edge. This counts the number of shortest paths through an edge. Repeating these steps from every vertex s in the graph and then consolidating the sum of the counts provides us with required betweenness.

One of the main drawbacks of the divisive algorithms is that they fail in scenarios where the edges do not portray the community structure. For example a bipartite graph consists of two sets of vertices, and no edges exist between vertices in the same set. Each of these sets would constitute a community. Removal of edges in any order using a divisive algorithm would not reveal the two communities. The cubic complexity of the algorithm makes it impractical for graphs with a few thousand nodes.

```
EdgeBetweennessSimilarityMetric($G$)
for $s$ from 1 to $n$ do
    $s = 1$
    $D_s = 0$
    $W_s = 1$
    for all $i$ such that $e_{is} \in E$ do
        $D_i = D_s + 1$
        $W_s = 1$
        $W_i = 1$
    endfor
    do
        for all $j$ such that $e_{ij} \in E$ do
            if $D_j$ == NULL then
                $D_j = D_i + 1$
                $W_j = W_i$
            else if $D_j \neq$ NULL and $D_j = D_i + 1$ then
                $W_j = W_j + W_i$
            else if $D_j \neq$ NULL and $D_j < D_i + 1$ then
            endif
        endfor
    while we have no vertex $i$ such that $e_{ij} \in E$ and $D_i \neq$ NULL and $D_j$ == NULL
    for every leaf node $t$ do
        for all $i$ such that $e_{it} \in E$ do
            $B_{it} = B_{it} + W_i / W_t$
        endfor
    endfor
    for $e_{ij} \in E$ from the farthest $e_{ij}$ to closest $e_{ij}$ to node $s$ do
        for $e_{jk} \in E$ where $e_{jk}$ farther than $e_{ij}$ to node $s$ do
            $B_{ij} = B_{ij} + (B_{jk} + 1) * W_i / W_j$
        endfor
    endfor
endfor
```

Algorithm 8: Algorithm to compute edge betweenness of all edges.

3.3.3 *Edge clustering co-efficient*

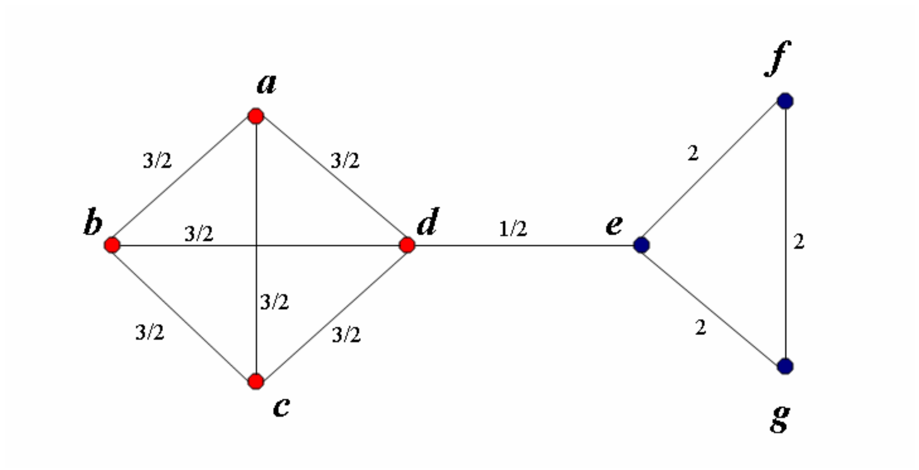


Figure 19: Edge clustering coefficient of edges.

Radicchi et al. came up with an algorithm that utilizes the local property of the graph rather than a global property like the edge betweenness or node/edge independent paths. They devised a new similarity measure called the edge clustering co-efficient [79]. The *Edge clustering co-efficient* of an edge is defined as:

$$\frac{T_{i,j}+1}{\min(d_i-1,d_j-1)},$$

where $T_{i,j}$ represents the number of triangles, K_3's, to which the edge *i, j* belongs to and d_i represents the degree of node *i*. The denominator $\min(d_i - 1, d_j - 1)$ indicates the maximum number of triangles to which the edge e_{ij} could belong. The "+ 1" in the numerator takes care of the degeneracy when $T_{i,j}$ is zero.

Algorithm 9 can be used to compute the edge clustering co-efficient of an edge (i, j). Once the clustering coefficient of all the edges is computed, the edge with the lowest value of clustering coefficient is removed. As in the edge betweenness algorithm, Radicchi et al. suggest re-computing the clustering coefficient of the set of edges that might be affected by removal of the edge. Deleting edges with low edge clustering coefficient value removes the inter-community edges and exposes the underlying communities. The complexity for computing the clustering coefficient for all the edges takes > $O(m^2)$ time. Clearly the algorithm would fail on all *triangle free* graphs.

The motivation behind this algorithm is as follows: the density of edges is higher inside the communities than along its boundaries. As a result edges connecting nodes in different communities are included in few or no triangles, on the other hand edges connecting nodes in the same community would be a part of many triangles and consequently have a high clustering co-efficient value. One could repeat the above algorithm by counting the number cycles of length four (squares) or cycles of higher order, which increases the complexity of the algorithm.

3.3.4 *Random walks*

Girvan and Newman propose a similarity measure based on random walks [71]. To compute the similarity one has to compute the expected number of times random walks between a pair of nodes would pass down a particular edge and sum over all node pairs. A large count indicates better similarity. Let the random walks start at node s, pass down a particular edge (u, v) before ending at a target node t. Let A represent the adjacency matrix of the graph with $A_{i,j} = 1$

if there exists an edge between nodes i and j and $A_{i,j} = 0$ otherwise. The random walk on each step would decide uniformly between the neighbors of the current node j and move to one of them. The probability of transition from node j to i is $A_{i,j} / d_j$. The value $A_{i,j} / d_j$ is a element of the matrix $M = A \times D^{-1}$, where D is the diagonal matrix with $D_{i,i} = d_i$. All the random walks are to terminate at node t (absorbing state) to achieve this; the column corresponding to node t is made all zeros. Let $\underline{M} = \underline{A} \times \underline{D}^{-1}$ be the matrix M with the t^{th} column removed (and similarly for $\underline{A}$ and $\underline{D}$). The probability that a walk starts at s, takes k steps, and ends at some node i (other than t), is given by the i,s element of matrix $\underline{M}^k$ (denoted by $\underline{M}^k_{i,s}$). In particular, walks end up at nodes i and j with probabilities $\underline{M}^k_{i,s}$ and $\underline{M}^k_{j,s}$, and of those a fraction $1/d_u$ and $1/d_v$ respectively pass over the edge (i, j) in one direction or the other. Summing over all k, the mean number of times a walk from s to t traverses the edge from i to j is

$$\frac{\left[\left(I - \underline{M}\right)^{-1}\right]_{i,s}}{d_v}$$

```
**EdgeClusteringCoefficientSimilarityMetric**($G$, $i$, $j$)
$d_i = 0$
$d_j = 0$
$T_{i,j} = 0$
for all $k \in V$ such that $e_{ik} \in E$ do
    $d_i = d_i + 1$
endfor
for all $k \in V$ such that $e_{jk} \in E$ do
    $d_j = d_j + 1$
endfor
for all $k \in V$ such that $e_{jk} \in E$ and $e_{ik} \in E$ do
    $T_{i,j} = T_{i,j} + 1$
endfor
$T_{i,j} = \frac{T_{i,j} + 1}{\min(d_i - 1, d_j - 1)}$
```

Algorithm 9: Edge clustering coefficient of edge, $e_{i,j}$.

and similarly for all walks from *v* to *u*, which would be an element of the matrix

$$E = D^{-1} \times \left(I - \underline{M}\right)^{-1} \times S = \left(\underline{D} - \underline{A}\right)^{-1} \times S\ .$$

Here *I* is the identity matrix and *S* is a vector whose components are all 0 except for a single 1 in the position corresponding to the source node *s*. Element *i*, *j* of the matrix *E* represent the expected number of times a random walk between nodes *s* and *t* would pass the edge (*i*, *j*) summing this over all pairs of *s* and *t*, we can compute the random walk betweenness of any edge (*i*, *j*).

Algorithm 10 can be used to compute the random walk betweenness of all the edges in a graph. The input to the algorithms is the adjacency matrix of the graph. Matrix *R*, which stores the random walk betweenness of all the edges, is initialized to all zeros. Element $R_{i,j}$ would give the betweenness of edge (*i*, *j*). The Newman and Girvan claim that this method produces poor results and hence random walk betweenness is not a very good similarity measure [70].

3.3.5 *Current betweenness*

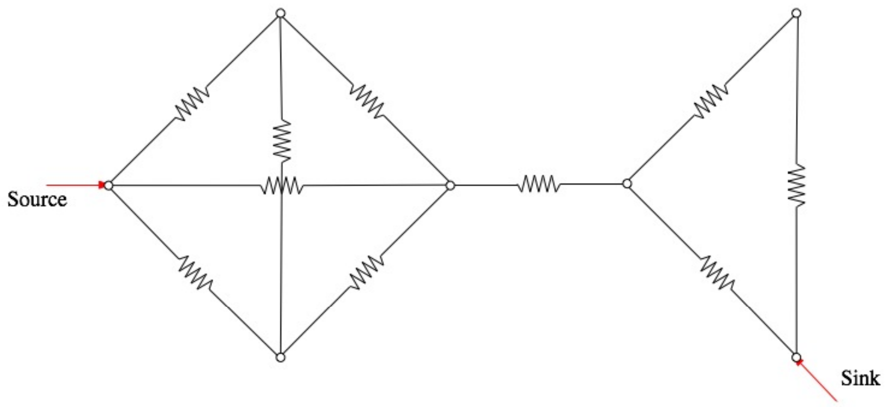


Figure 20: Example resistor network.

```
**RandomWalkBetweenessSimilarityMetric**($A$)
**for** all $i$ from 1 to $n$ **do**
    **for** all $j$ from 1 to $n$ **do**
        $R_{i,j} = 0$
        $D'_{i,j} = 0$
    **endfor**
**endfor**
for all node pairs $s, t$
**for** all $s$ from 1 to $n$ **do**
    **for** all $t$ from 1 to $n$ **do**
        $D = D'$
        **for** all $i$ from 1 to $n$ **do**
            **for** all $j$ from 1 to $n$ **do**
                **if** $i \neq t$ **do**
                    $D_{i,i} = D_{i,i} + A_{i,j}$
                **endif**
                **if** $i \neq t$ **do**
                    $D_{j,j} = 0$
                **endif**
            **endfor**
        **endfor**
    **endfor**
**endfor**
**for** all $i$ from 1 to $n$ **do**
    $A_{t,i} = 0$
    **if** $i = s$ **do**
        $S_i = 1$
    **endif**
**endfor**
$E = (D - A)^{-1} \times S$
$R = R + E$
```

Algorithm 10: Random walk betweenness of all edges.

The resistor network based approach introduced by Girvan and Newman in [71] is motivated by ideas from elementary circuit theory. The input graph is considered as a resistor network with edges being substituted with a resistor of unit resistance and a unit voltage is applied between a *source node* and a *sink node*. Current would flow from the source to the sink via a number of paths and the paths with least resistance would carry the greatest fraction of the current. The amount of current flowing through the edge is termed as *current betweenness* of the edge. These paths can be identified by solving Kirchoff's equations. Let A represent the adjacency matrix of the input graph and let the V_i represent the voltage at vertex, v_i. D is a diagonal matrix with $D_{i,i} = \sum_j A_{ij}$. Now $(D - A) \bullet V = S$. Where S is the source vector with the following components

$$S_i = \begin{cases} +1 & \text{for } i = s \\ -1 & \text{for } i = t \\ 0 & \text{otherwise.} \end{cases}$$

One could now compute the voltage across all the vertices by solving the equation.

$$V = (D - A)^{-1} \bullet S$$

Since the edges contain a unit resistance, the current flowing across an edge e_{ij} would be equal to the difference in voltage $(V_i - V_j)$ along vertices v_i and v_j. Removal of the edges with low current flow would disconnect the graph into components each representing a community. The matrix inversion would require $O(n^3)$ time and the subsequent calculations to compute the betweenness require $O(mn^2)$ time. Computing the betweenness of all the edges, including the recalculation step as with the previous algorithms would take $O((n + m)\, mn^2)$ time. For a sparse graph this is of O(n^4). Girvan and Newman prove that the current betweenness is numerically equal to the random walk betweenness [70].

3.3.6 *Euclidean distance*

The Euclidean distance similarity measure is based on the structural equivalence between vertex pairs. Two vertices are said to be structurally equivalent if they have the same set of neighbors (other than each other, if they are connected). The *Euclidean distance* similarity measure [17, 91] between two nodes u and v is

$$\delta_{u,v}^{Eucl} = \sqrt{\sum_{k \neq u,v} (A_{u,k} - A_{k,v})^2}$$

where $A_{i,j}$ is the element of the adjacency matrix for the vertices i and j. The Euclidian distance is actually a dissimilarity measure. A large value of this distance between a vertex pair represents low similarity and a small value represents high similarity. A value of zero is given to nodes that are structurally equivalent. Unlike the previous measures, one could use this measure with an agglomerative or divisive algorithm to extract the communities. Another interesting feature is that, the measure does not depend on the existence of an edge between vertices. Hence, two vertices that are not connected might still be structurally equivalent. Such measures are useful to deduce the community structure in graphs like the bipartite graphs where existence of an edge between a vertex pair does not necessarily convey similarity.

3.3.7 *Pearson correlation coefficient*

Another commonly used similarity measure in social networks is the Pearson correlation between columns (or rows) of the adjacency matrix [91]. This measure defines the mean and variances of columns of the adjacency matrix as follows:

$$\mu_u = \frac{1}{n}\sum_v A_{u,v}, \qquad \sigma_u^2 = \frac{1}{n}\sum_v (A_{u,v} - \mu_u)^2 ,$$

where A is the adjacency matrix of the graph. The correlation coefficient of an edge (u, v)

$$\frac{\frac{1}{n}\sum_k (A_{u,k} - \mu_i)(A_{v,k} - \mu_v)}{\sigma_u \times \sigma_v} .$$

Vertex pairs with high degree of structural equivalence will have high correlation coefficient value and those do not will have low values.

3.3.8 *Flow network based*

For the flow based approach the entire network is considered as a flow network with the edges representing pipes. If one were to introduce a flow into this network via a *source node* then drain the network via a *sink node* then edges between the communities would act as a bottle neck controlling the amount of the flow. Using a *max flow-min cut* algorithm, one can easily identify these edges. Removal of these edges would result in the bisection of the graph into two communities. One could further bisect these communities to obtain smaller communities in the similar fashion. For the flow based approach to produce good results, the sink node and the source node should be in two different communities, this turns out to be a draw back because it requires priori information. For the sake of clarity we assume that the graph G is undirected. Let $c(u, v)$ be a function that denotes the capacity of the edge (u, v), $c(u, v) = 0$ if $(u, v) \notin E$. Similarly let $f(u, v)$ be a function that denotes the flow along the edge (u, v) and $f(u, v) \leq c(u, v)$. Given two nodes u and v, the maximum flow that can be routed from s to t is identical to the value of the minimum cut that separates s and t [37]. Removal of the edges corresponding to the minimal cut would disconnect the graph into two components with one that contains node s and the other that contains node t. There are several algorithms for solving the *max flow min cut* problem; refer to [1, 31] for more information on this topic.

Flake, Lawrence and Giles came up with a modified flow based approach to identify communities in web graphs [36]. They define communities as subsets of nodes in a graph that have more links (in either direction) to members of the community than to non-members. The goal of their algorithm is to bisect the graph into two components using flow techniques such that one of the components represents the community. The algorithm is designed to work on directed graphs, and starts by exploring the nodes adjacent to a small set of nodes called the seed B. Let the nodes adjacent to the seed belong to set C. Now the nodes adjacent to each node in the set C are obtained (a few of these may be already present in the set B and can be omitted), let these new nodes belong to set D. The nodes in set D are treated as a composite sink node by connection to a virtual sink node and all the nodes in the seed set B are connected to a virtual source s, as shown in Figure 21. All edges that start and end at nodes that belong to set B or C are made bidirectional.

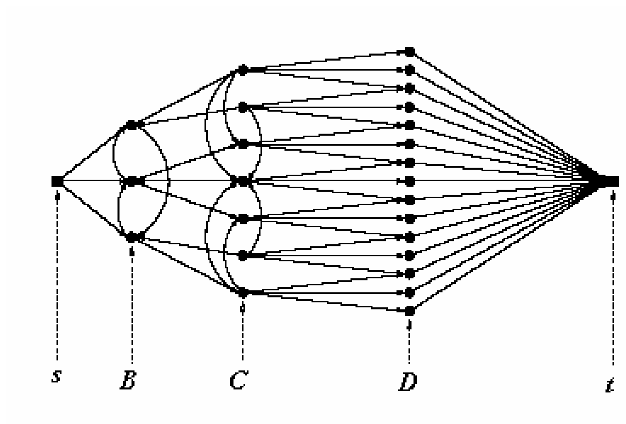


Figure 21: Graph obtained by using the algorithm in [36].

Now the edge capacities of all the edges in the graph are multiplied by a constant factor *k* except for the edges that are incident on the virtual sink, these are of unit capacity. *A* min cut of the graph thus obtained would bisect the graph and the component containing the seed set *B* would represent the community. The size of the community is further increased by adding to the seed set a node u^* from the community, that has the highest in-degree relative to the graph and all other nodes u_i that belong to the community and have the same in-degree as u^*. The seed set is further increased by adding a node v^* from the community that has the highest out-degree relative to the graph and all other nodes v_i from the community that have the same out-degree as v^*. In Algorithm 11, *G* represents the graph obtained from a given seed set as described above. This algorithm differs from the remaining algorithms in several ways; first its objective is community identification and not community discovery; second it does not require knowledge of the entire graph, which suits networks like the web graph where obtaining the entire graph is a difficult task.

3.4 Spectral methods

This section provides a brief survey of linear algebra and eigenvectors. Any non-defective matrix *M*, can be represented as a summation of vector outer products.

$$M = \sum_{i=1}^{k} \lambda_i \cdot r_i \cdot l_i^T$$

where l_i and r_i are the *i*th left and right eigenvectors of matrix *M*. λ_i is the *i*th eigenvalue of *M,* and *M* has the following properties with respect to its eigenvectors and eigenvalues:

$$\begin{aligned}
&\lambda_i \cdot r_i = M \cdot r_i, \\
&\lambda_i \cdot l_i = M^T l_i, \\
&l_i^T \cdot l_i = r_i^T \cdot r_i = r_i^T l_i = 1, \text{ for all } i, \\
&l_i^T \cdot r_j = 0, \text{ for } i \neq j, \text{ and} \\
&\lambda_1 \geq \lambda_{i+1} \text{ for all } i.
\end{aligned}$$

The eigenvalues and eigenvectors form the *spectrum* of the matrix; and algorithms employing the eigenvectors and the eigenvalues are referred to as *spectral algorithms*. If the spectrum of the matrix consists of *n* distinct eigenvectors, then either the left or right eigenvectors can be used as a basis to express any *n*-dimensional vector. For any symmetric matrix the left and right eigenvectors are identical and for any asymmetric matrix the left and right eigenvectors form a *contravariant basis* with respect to each other, provided that all the eigenvectors and eigenvalues are real. The key intuition behind the eigen-decomposition of a matrix is that it yields a procedure for compressing a matrix into $k \leq n$ outer-products, and for expressing matrix-vector products as a summation of inner products. When $M = A^T \times A$ for some choice of *A*, then the spectral decomposition of *M* is closely related to the singular value decomposition of *A* [87].

```
CommunityIdentificationUsingNetworkFlows($G, s, t, q$)
$S = \{s\}$
$k = 0$
while $k < q$ do
        $C$ = MaxFlowAnalysis($G$)
        $v_v = v_i$ such that $v_i \in C$ and $d^{in}_{v_i} = \max(\sum_j v_{j,i})$, relative to $G$
for all $v_u \in C$ such that $d^{in}_{v_u} = d^{in}_{v_v}$ do
                $S = S + \{v_u\}$
                $E = E + \{e_{s,v_u}\}$
endfor
$v_u = v_i$ such that $d^{out}_{v_i} = \max(\sum_j v_{i,j})$,relative to $G$
for all $v_u \in C$ such that $d^{out}_{v_u} = d^{out}_{v_v}$ do
                $S = S + \{v_u\}$
                $E = E + \{e_{s,v_u}\}$
endfor
$G$ = Crawl($G, S$)
endwhile
```

Algorithm 11: Community identification using flow algorithms.

3.4.1 *HITS*

Kleinberg's HITS algorithm (which stands for *hyperlink-induced topic search*) takes a subset of the Web graph and generates two weights for each page in the subset [56]. The weights are usually referred to as the *hub* and *authority* score, respectively, and they intimately relate to the spectral properties of the portion of the Web graph for which the algorithm is being used. Conceptually, a hub is a Web page that links to many authorities, and an authority is a Web page that is linked by many hubs. The two scores for each page characterize to what degree a page obeys the respective property.

HITS algorithm works in two stages. The first is a preprocessing step used to select the subset of the Web graph to be used, while the second part is an iterative numerical procedure. The first part usually proceeds as follows:

1. Send a query of interest to the search engine of your choice.
2. Take the top 200 or so results from the search engine.
3. Identify all Web pages that are one or two links away (in either direction) from the results gathered in the previous step.

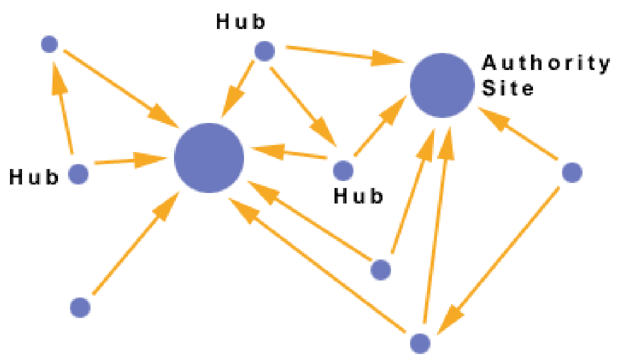


Figure 22: Hubs and Authorities.

All told, the first part generates a base set of Web pages that either contain the original query of interest, or are within two links away from a page that does. Of course, other heuristic constraints need to be used, such as limiting the total number of pages, only considering inter-domain hyperlinks, and changing the size of the initial result set and/or the diameter of the base set. With the base set of pages being generated, let $G = (V, E)$ refer to this subset of the Web graph (with intra-domain hyperlinks removed) and let A be this graph's adjacency matrix. The iterative numerical part of HITS updates two $n \times 1$ dimensional vectors, h and a, as follows, with the initial values of both vectors being set to unity:

$$\begin{aligned} a &= A^T \cdot h \\ h &= A \cdot a \\ a &= a / \| a \|^2 \\ h &= h / \| h \|^2 \end{aligned}$$

The equation $a = A^T \cdot h$ sets a page's authority score be equal to the sum of the hub scores of the pages that link to it, while the equation $h = A \cdot a$ sets a page's hub score be equal to the sum of the authority scores that it links to. The remaining equations enforce h and a to maintain unit length. After iterating the equations, we select the authority pages to be those with the largest corresponding value in *a*, and the hub pages to be the ones with the largest corresponding value in *h*. HITS is a close to the power method in [87] for calculating the eigenvector of a matrix with the largest eigenvalue (the maximal eigenvector). Both procedures converge to a solution in *k* iterations with error proportional to $O(|\lambda_2/\lambda_1|^k)$, where λ_2 and λ_1 are the first and second eigenvectors. Hence, the procedure can be slow, but it tends to be fast for power law graphs which often have the property that $\lambda_1 >> \lambda_2$ [27]. With minimal substitution, we can see that *h* and *a* converge to the maximal eigenvectors of $A \cdot A^T$ and $A^T \cdot A$. Thus, HITS produces a rank one approximation to the raw bibliographic and co-citation coupling matrices.

3.4.2 *HITS Communities*

We saw earlier, that a matrix can be rewritten as a summation of outer products. Because both of the matrix products $A^T \cdot A$ and $A \cdot A^T$ are symmetric and positive definite, each will have the property that the left and right eigenvectors will be identical (because of symmetry) and that the first eigenvector will have all positive components (with positive eigenvalue). All other eigenvectors for these matrices can be heterogeneous in that their elements can have mixed signs. These subsequent eigenvectors can be used to separate pages into different communities in a manner related to more classical spectral graph partitioning [26], or in a manner that is related to principal component analysis [52]. Kleinberg [56] and his collaborators [42] have found that the non-maximal eigenvectors can be used to split pages from a base set into multiple communities that contain similar text but are dissimilar in meaning.

In this manner, HITS, can be adapted for community identification. The main caveat to spectral approaches is that as the sizes of the communities get smaller, the less significant eigenvectors can be dominated by noise and confused by paths of longer length. Nevertheless, the approach has considerable power and spectral methods offer a very elegant mathematical derivation.

3.5 Conclusion

This chapter provides a comparison of the various algorithms for community discovery in terms of speed and sensitivity. A direct comparison of all the methods might not be possible, as they are varied both conceptually and in their application.

Table 2 provides us with the computational cost associated with the various algorithms in terms of the size and order of the input graphs. Some algorithms require certain extra parameters.

Agglomerative algorithms are very suitable in cases where the input graphs do not exhibit a strong community structure, e.g. in the case of bipartite graphs. Divisive algorithms on the other hand are useful to quickly deduce the community structure of graphs that exhibit a strong community structure. The high sensitivity of the agglomerative algorithms is attributed to the fact that these algorithms can compute the similarity between vertex pairs that are not adjacent where as for the divisive algorithms one requires that the vertex pair be adjacent. This explains the lower time complexity of the divisive algorithms as they compute the similarities between just the adjacent vertices in the graph, m pairs to be precise. The agglomerative algorithms need to compute the similarities between all the vertex pairs in the graph, that's n^2 pairs.

Comparing the different similarity measures one could say that the measure utilizing the local properties of the graph, e.g., edge clustering coefficient are better measures as they do not require the entire input graph to compute the similarity between a given vertex pair, whereas measures that depend on global properties like edge betweenness require the entire input graph to compute the similarity between a given pair. Local properties are also useful if one were to compute the similarities online while performing a web crawl for example where the complete structure of the input graph is not available.

Table 2: Comparing the complexity of the various algorithms.

Reference	Measure	Complexity	
		Agglomerative	Divisive
[44]	Node/edge independent paths	$O(n^5)$	$O(n^4 m)$
[44]	Edge betweenness	-	$O(n^3)$
[79]	Edge clustering co-efficient	-	$O(n^4)$
[71]	Random Walks	-	$O(n^4)$
[71]	Current betweenness	-	$O(n^4)$

4. PROPOSED CENTRALITY BASED COMMUNITY DISCOVERY

4.1 Centrality metrics

Centrality as a tool has been widely used in social network analysis. It portrays how central a node is to a given graph. For example, in the telecommunication industry, one could mine the call records of telephone operators and evaluate the customer network or determine the influence of different people in such a network. Such data would be of great value in viral marketing and advertising. Similar techniques could be used to identify popular web pages in the World Wide Web. Though centrality metrics were initially used to investigate the role and identity of individual nodes in a network, now the emphasis is more shifted to the distribution of centrality values through all vertices. The upcoming sections discuss the different centrality metrics.

4.1.1 *Degree centrality*

Degree centrality C^D, is the simplest form of vertex centrality. The simplicity aids in understanding the concepts behind centrality. It is based on the idea that important vertices have a large number of ties to other vertices in the graph. The *degree centrality* of a node *i* in a graph is defined as [40, 91]:

$$C_i^D = \frac{\sum_{j \in G} a_{i,j}}{n-1} = \frac{d_i}{n-1}$$

where d_i is the degree of node i, i.e. the number of nodes adjacent to i. Figure 23 shows the degree centrality of the vertices in a star, $K_{1,8}$ and a cycle, C_8. It is clear that the center of the star which has the highest degree of all the vertices is the most influential vertex in the star. On the other hand in a cycle all nodes have the same degree and hence the same amount of influence.

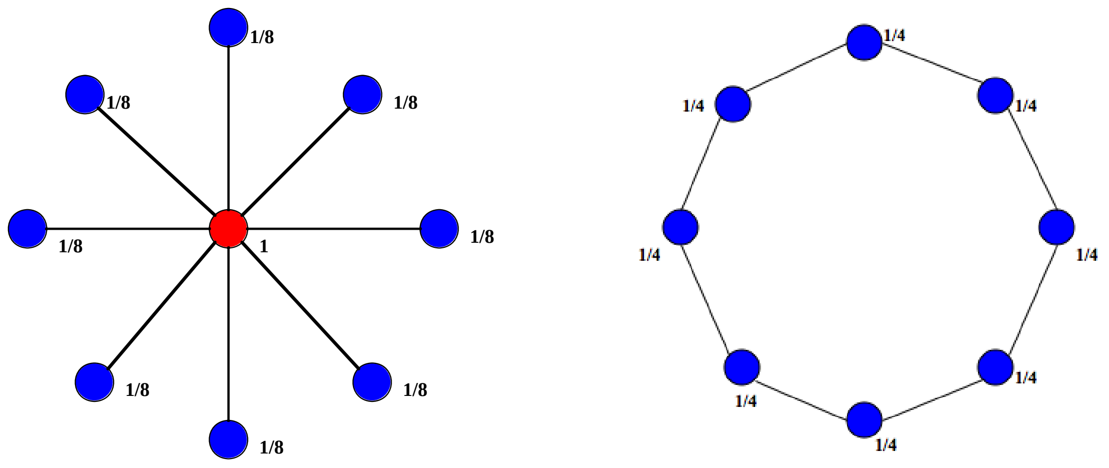


Figure 23: Degree centrality of a star, $K_{1,8}$, and a cycle C_8.

4.1.2 *Closeness centrality*

If one needs to broadcast a message to the entire network in the shortest amount of time, then we need to look for a vertex whose average of the shortest paths to all the remaining vertices is minimum. The closeness centrality value of a vertex defines such a measure. *Closeness centrality* C^C*,* measures to which extent a vertex i is near to all the other nodes along the shortest paths, and is defined as [40, 84, 91]:

$$C_i^C = \frac{n-1}{\sum_{\substack{j \in G \\ i \neq j}} \delta_{i,j}}$$

where $\delta_{i,j}$ is the shortest path length between i and j, defined, in a valued graph, as the smallest sum of the edge lengths throughout all the possible paths in the graph between i and j.

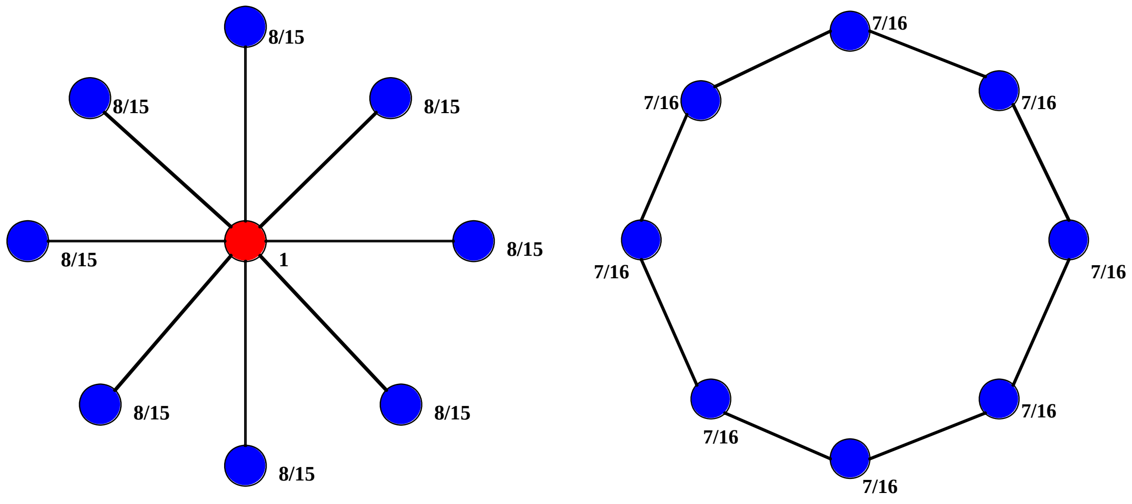


Figure 24: Closeness centrality of a star, $K_{1,8}$, and a cycle C_8.

Figure 24 shows the closeness centrality values for the vertices in a star, $K_{1,8}$, and a cycle C_8. It's clear that the center of the star would be the best candidate to broadcast any message to the star in the shortest amount of time. In the cycle however all the nodes have the same value of closeness and it would take the same amount of time to broadcast the message irrespective of which vertex is broadcasting.

4.1.3 ***Betweenness centrality***

Betweenness centrality is to vertices as edge betweenness (refer to Section 3.3.2) is to edges. If one were to place an agent to monitor data flow in a network, then the agent has to be placed on a vertex that is on several shortest paths between vertex pairs. Betweenness centrality can aid us in selecting one such vertex. Betweenness centrality C^B, is based on the idea that a vertex is central if it lies between many other vertex pairs, in the sense that it is traversed by many of the shortest paths connecting the vertex pairs. The betweenness centrality of vertex *i* is [40]:

$$C_i^B = \frac{1}{(n-1)(n-2)} \cdot \sum_{\substack{j,k \in G \\ j \neq k \neq i}} \frac{n_{j,k}(i)}{n_{j,k}}$$

where $n_{j,k}$ is the number of shortest paths between *j* and *k*, and $n_{j,k}$ (i) is the number of shortest paths between *j* and *k* that contain node *i*.

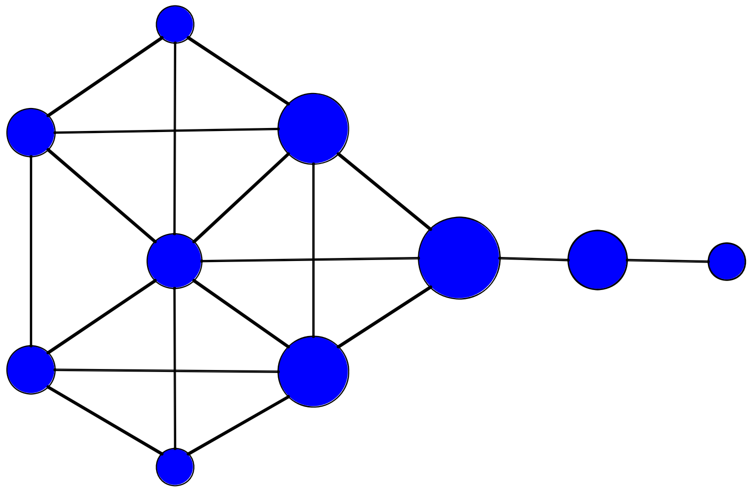


Figure 25: An example for betweenness centrality.

Figure 25 shows an example graph where in the size of the vertices represent their betweenness centrality.

4.1.4 ***Straightness centrality***

Straightness centrality C^S, originates from the idea that the efficiency in the communication between two nodes *i* and *j* is equal to the inverse of the shortest path length $\delta_{i,j}$ [58]. The straightness centrality of node *i* is defined as:

$$C_i^S = \frac{\sum_{\substack{j \in G \\ j \neq i}} \frac{\delta_{i,j}^{Eucl}}{\delta_{i,j}}}{n-1}$$

where $\delta_{i,j}^{Eucl}$ is the Euclidean distance between nodes *i* and *j* along a straight line, and it adopts a normalization recently proposed for geographic networks [90]. This measure captures the extent to which the connecting route between nodes *i* and *j* deviates from the virtual straight route.

4.1.5 ***Information centrality***

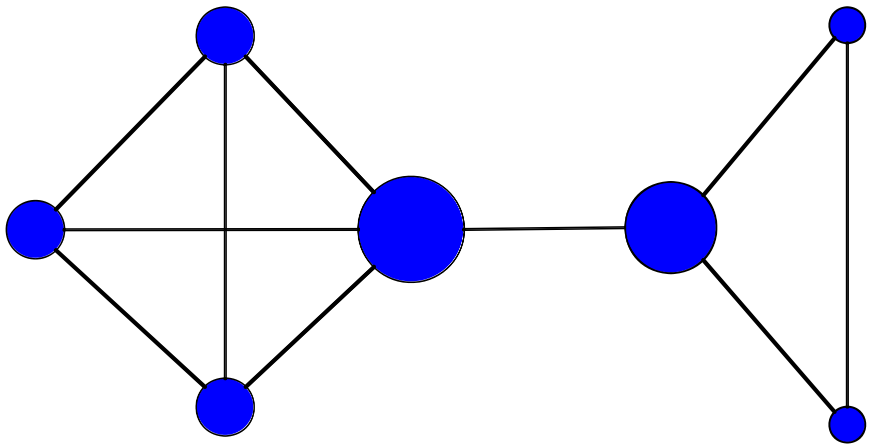


Figure 26: An example for information centrality.

Flow of information in a graph depends on how efficiently its vertices transfer information. It is believed that information/communication in a network always flows along the shortest path. The efficiency of communication between two vertices, say *i* and *j*, is inversely proportional to the shortest distance, $\delta_{i,j}$, between them. The communication efficiency of the entire graph is defined as the average of efficiency values for all vertex pairs in the graph:

$$E(G) = \frac{\sum_{i \neq j \in V} \frac{1}{\delta_{i,j}}}{n \cdot (n-1)}.$$

The value of $E(G)$ is in the range [0, 1]. The information centrality measure is based on the concept of communication efficiency. It was introduced in [59], and relates the importance of a vertex to the ability of the network to respond to the deactivation of the vertex. The network performance, before and after a certain vertex is deactivated, is measured by the efficiency of the graph *G* [58, 60]. The information centrality of vertex *i* is defined as the relative drop in the network efficiency caused by the removal from *G* of the edges incident in *i*:

$$C_i^I = \frac{\Delta E}{E} = \frac{E(G) - E(G')}{E(G)},$$

where *G′* is the network with *n* vertices and $m\text{-}d_i$ edges obtained by removing from *G* the edges incident in vertex *i*. Notice that $E(G)$ is finite even for a non-connected graph. Figure 26 shows a graph where the vertex size represents information centrality.

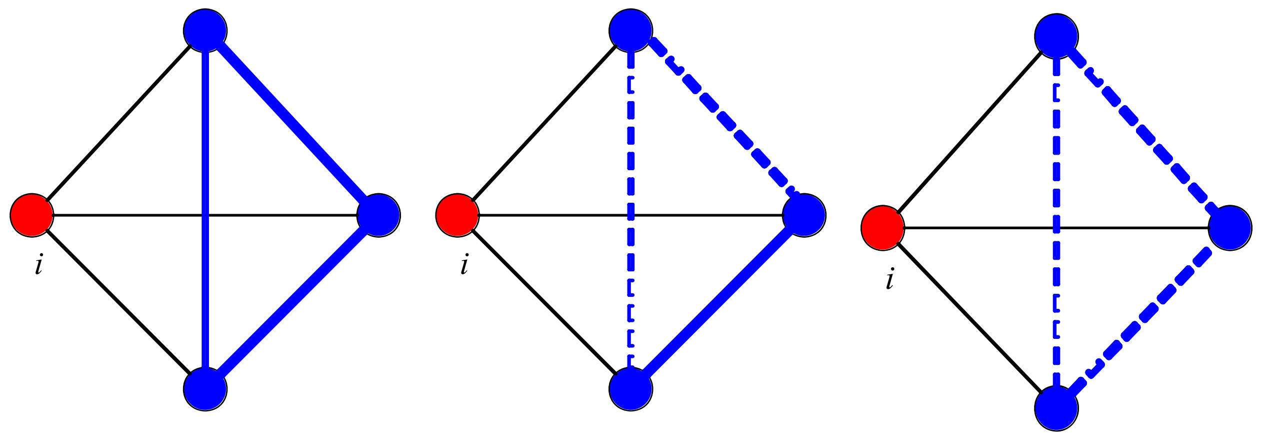
4.1.6 *Clustering co-efficient*

Watts and Strogatz [92] introduced this measure of centrality, as a method to determine whether or not a graph is a small world network. It has gained a lot of importance in recent years.

Another similar measure, transitivity (refer 1.2.4) was introduced as an alternative to clustering co-efficient by Newman along with Watts, and Strogatz [72]. Clustering co-efficient is used to measure the cliquishness of the neighborhood of a vertex. The clustering coefficient of a vertex *i* is defined as the proportion of edges that exist between the vertices within its neighborhood divided by the number of edges that could possibly exist between them. This can be formalized as:

$$C_i^{cc} = \frac{|E_i|}{d_i(d_i - 1)/2},$$
$$E_i = \{(j,k) : (j,k) \in E \ \& j,k \in N(i)\}$$

E_i represents the set of edges (j, k) such that both nodes j and k are adjacent to node i.



$C_i^{CC} = \frac{3}{3} = 1$ (a) $C_i^{CC} = \frac{1}{3}$ (b) $C_i^{CC} = 0$ (c)

Figure 27: An example for clustering coefficient.

Figure 30 shows the clustering coefficient of the vertex *i*, with varying configurations. The blue vertices are neighbors of node *i*. The solid blue edges represent the edges between the neighbors of *i* and the dotted blue edges indicate possible edges between the neighbors of *i*.

4.2 Centrality of a graph

For all the centrality metrics described in the previous section, the centrality of the entire graph is defined as the mean of the centrality measures of all its nodes:

$$C(G) = \frac{\sum_{v \in V} C_i}{n}.$$

4.3 Community discovery using centrality measures

In this section we propose a novel approach to discover communities by using the centrality values of the nodes. The centrality measures portray the importance or ranking of the vertices, whereas the community discovery algorithms deal with the importance or rank of edges. There have been a few algorithms where in a centrality metric for the vertices has been translated into an equivalent metric for the edges. One such approach is Newman's edge betweenness metric (refer 3.3.2) derived from vertex betweenness. A recent paper by Fotunato [38] defines a new centrality metric for edges based on information centrality. The information centrality, $C^I_{i,j}$ of an edge, e_{ij} is defined as the relative drop in network efficiency caused by the removal of e_{ij} from the graph. Let $\delta_{i,j}$ represents the shortest distance between vertices *i* and *j* then $\frac{1}{\delta_{i,j}}$ represents the efficiency in communication between the two vertices and

$$E(G) = \frac{\sum_{i \neq j \in V} \frac{1}{\delta_{i,j}}}{n \cdot (n-1)},$$

represents the efficiency of the entire graph. Now the information centrality of the edge $e_{i,j}$ is given by

$$C^{I}_{i,j} = \frac{\Delta E}{E} = \frac{E(G) - E(G')}{E(G)},$$

where, G' represents the graph G with the edge $e_{i,j}$ removed. They use a divisive approach to extract the communities. The algorithm has a complexity of $O(n^4)$ for sparse graphs.

4.3.1 ***Proposed approach***

Instead of translating the centrality metric to reflect the measure for an edge, I propose a method of utilizing the existing centrality definition to compute the importance of an edge in a given graph. It is evident that every edge contributes to the overall centrality value of the graph. This role of the edge is quantified as the importance of the edge. The importance of an edge, e_{ij} can be defined as the drop in centrality value caused by removal of the edge from the graph. The centrality of an edge, e_{ij} is given by

$$C_{i,j} = C(G) - C(G'),$$

where $C_{i,j}$ is the centrality of the edge, $e_{i,j}$, $C(G)$ is the centrality of the graph, G' represents the graph with the edge $e_{i,j}$ removed and $C(G')$ is the centrality of the graph after removal of the edge $e_{i,j}$. One could further normalize this value by considering the relative drop in efficiency caused by removal of the edge from the graph:

$$C_{i,j} = \frac{C(G) - C(G')}{C(G)}.$$

Now we compute the centrality values for all the edges in the graph. We use a divisive algorithm and remove the edges in ascending or descending order of their centrality values. The components formed by removal of edges represent the underlying communities. Algorithm 12 represents a general template for an algorithm used to compute the centrality of the edges in a graph. A is a adjacency matrix of size $n \times n$ with $A_{i,j} = 1$ when $e_{i,,j} \in E$ and $A_{i,j} = 0$ when $e_{i,j} \notin E$. The function **NodeCentrality**(A) computes the centrality values of the vertices of the adjacency matrix A and returns a one dimensional array C, of size n where C_i represents the centrality of vertex i. The function **Mean**(C) computes the mean value of the one the one dimensional array C. E^C is an adjacency matrix with $E^C_{i,j}$ representing the centrality value of edge $e_{i,j}$.

The method is not suitable for all centrality metrics, e.g., degree centrality cannot be used to obtain edge centrality values by using our approach. This is explained below:

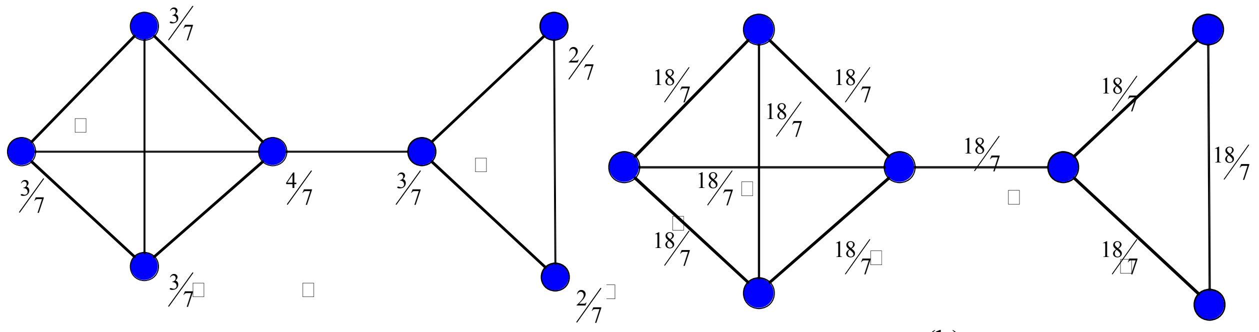


Figure 28: (a) degree centrality of the nodes in the graph, (b) corresponding edge centralities.

```
**EdgeCentrality**($A$)
$C$ = **NodeCentrality**($A$)
$c$ = **Mean**($C$);
**for** $i$ from 1 to $n$ **do**
    **for** $j$ from 1 to $n$ **do**
        **if** $A_{i,j} == 0$ **do**
            $E^{C}_{i,j} = 0$;
        **else**
            $A' = A$;
            $A'_{i,j} = 0$
            $A'_{j,i} = 0$
            $C'$ = **NodeCentrality**($A'$)
            $c'$ = **Mean**($C'$);
            $E^{C}_{i,j} = c - c'$;
        **endif**
    **endfor**
**endfor**
```

Algorithm 12: Computing centrality of the edges in a graph.

Every edge brings in 2 degrees to the graph and irrespective of the edge being removed the graph as a whole would loose 2 degrees. So the drop in centrality caused by the removal of any edge would be the same and the centrality of all the edges would be the same.

We test the approach selectively on the following centrality metrics:

1. Closeness centrality
2. Straightness centrality
3. Clustering coefficient
4. Betweenness centrality

As a benchmark synthetic graphs were generated as described by Girvan and Newman in [44]. Each of the generated graphs consists of 128 vertices divided into 4 communities of equal size. Edges were placed uniformly at random, such that each vertex on average has z_{in} neighbors in the same community and z_{out} neighbors outside. The average degree of the graph is kept close to 16. For the selected centrality metrics, the fraction of vertices that were classified correctly was measured by varying the number of inter-community edges per vertex from 0 to 16 while keeping the average degree constant. Each point on the graph is an average of 10 runs of the algorithm with the given specifications. Figure 29 shows the results obtained. Edge centrality measures obtained by using clustering coefficient were very efficient in detecting the communities. The algorithm was able to classify more than 95% of the vertices for $z_{in} \geq 5$, after which the accuracy reduces and only 40% of the vertices were correctly classified when $z_{in} = z_{out} = 8$. The edge centrality measures obtained using straightness centrality were next best in performance followed by the measures obtained using closeness and betweenness centrality.

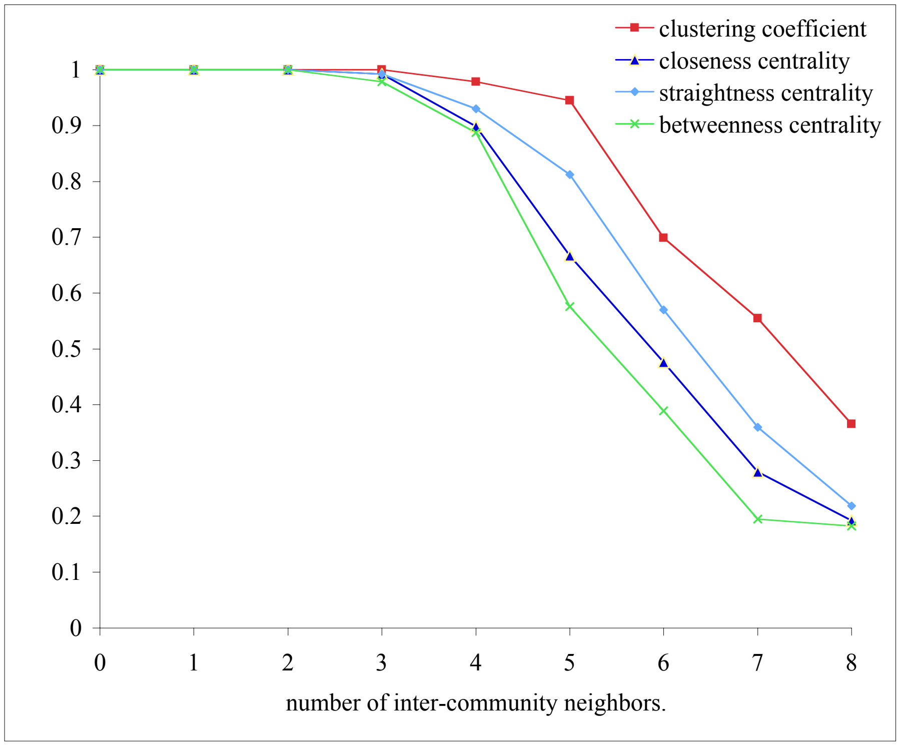


Figure 29: Performance of community discovery using different centrality measures.

5. BIBLIOMETRIC APPROACH

5.1 Inspiration

The motivation for the current work is from bibliographic metrics, which are used to determine similarity between publications. There are two measures that have been used: bibliographic coupling and co-citation coupling. Given two documents, *bibliographic coupling* is defined as the number of publications that are cited by both the given documents [54] and *co-citation coupling* is defined as the number of publications that cite both the given documents [85]. Combining the above two measures we obtain a unified metric that can be used to determine similarity between two nodes in a graph.

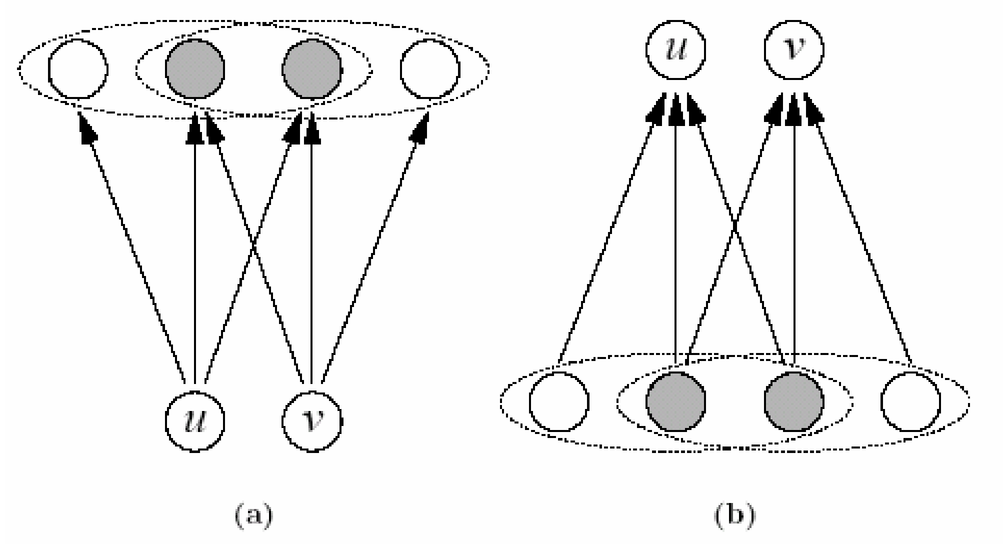


Figure 30: (a) bibliographic coupling (b) co-citation coupling.

5.2 Bibliometric similarity

The bibliometric similarity between two nodes *u* and *v* in a graph *G* is given by:

$$\frac{|N[u] \cap N[v]|}{\min(d_u, d_v) + 1},$$

where $N[u]$ refers to the closed neighborhood of node u and d_u refers to its degree. In simple terms we rate the similarity between two nodes in a network by the number of common neighbors they share. The more the number of common neighbors, the better the similarity.

Algorithm 13 can be used to compute the bibliometric similarity between all pairs of nodes in the graph. The input to the algorithm is a graph G. $N_{u,v}$ represents the overlap in the closed neighborhood of nodes u and v and d_u represents the degree of the node u. The output of the algorithm is a matrix S, where an entry $S_{u,v}$ would represent the similarity between nodes u and v. Given a graph G of order n we compute the measure of similarity between every pair of nodes in the graph using the above approach. Once the similarity between pairs of nodes in the graph has been defined we can use an agglomerative or divisive approach to extract the communities in the network.

One of the main drawbacks of the agglomerative algorithms developed so far is that they classify pendent nodes as separate communities [44]. This is because the similarity metric used is some global property like number of paths or number of node/edge independent paths between node pairs. As a result this value is low for edges connecting pendent nodes to the rest of the graph. This drawback could be overcome by using local measures of similarity like the one introduced above. Also, by using an agglomerative approach rather than a divisive one, we would be able to recognize communities in graphs like bipartite graphs where there are no edges between nodes of the same community.

```
BibliometricSimilarity($G$)
for $u$ from 1 to $n$ do
    for $v$ from 1 to $n$ do
        $N_{u,v} = 0$
    endfor
endfor
for $i$ from 1 to $n$ do
    $d_i = 0$
    for $j$ such that $e_{i,j} \in E$ do
        $d_i = d_i + 1$
        $\mathrm{N}_{\mathrm{i,j}} = \mathrm{N}_{\mathrm{i,j}} + 2$
    endfor
endfor
for $u$ from 1 to $n$ do
    for $v$ from 1 to $n$ do
        for $w$ such that $e_{u,w} \in E$ and $e_{v,w} \in E$ do
            $N_{u,v} = N_{u,v} + 1$
            $S_{u,v} = \frac{N_{u,v}}{\min(d_u, d_v) + 1}$
        endfor
    endfor
endfor
```

Algorithm 13: Bibliometric similarity between all pairs of nodes.

Next we test our algorithm on a number of computer generated and real-world graphs that are considered as benchmarks.

5.3 Performance on computer generated models

Computer-generated networks: Graphs with known community structure were generated as described by Girvan and Newman in [44]. Each of the generated graphs consists of 128 vertices divided into 4 communities of equal size. Edges were placed uniformly at random, such that each vertex on average has z_{in} neighbors in the same community and z_{out} neighbors outside. The average degree of the graph is kept close to 16. Our algorithm was tested on these graphs and the fraction of vertices that were classified correctly was measured by varying the number of inter-community edges per node from 0 to 16. The algorithm correctly classified up to 90% of the vertices in graphs with $z_{out} \leq 6$ and close to 70% of the vertices in graphs with $6 < z_{out} \leq 8$. For graphs with $z_{out} > 8$, each vertex on average has more neighbors outside the community than inside and the graphs no longer posses a well defined community structure. Our results are summarized in Figure 31.

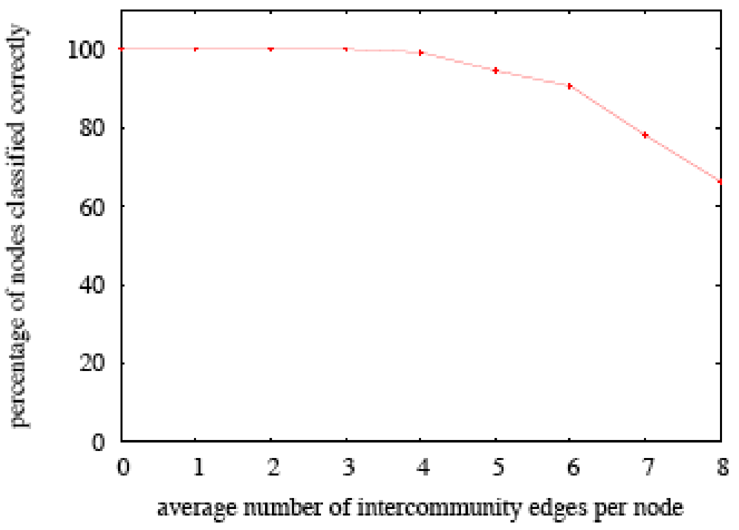


Figure 31 : Performance on computer-generated models.

5.4 Performance on real-world networks

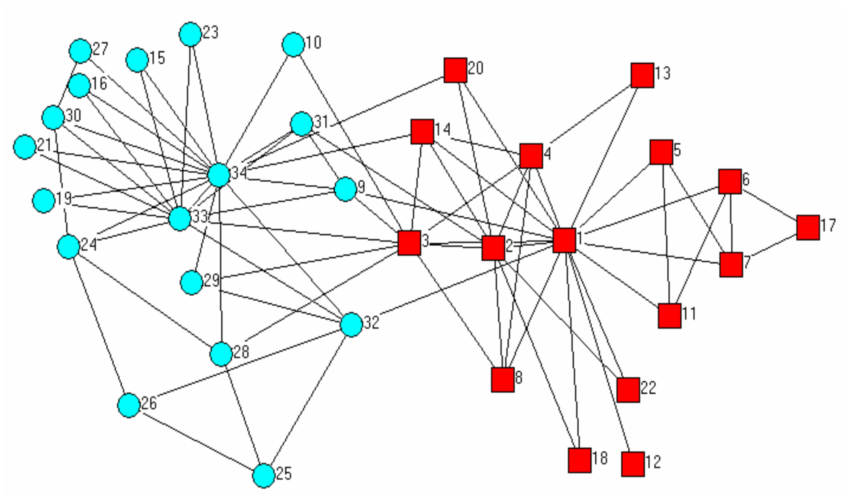


Figure 32: The Zachary Karate Club Network.

The Zachary Karate Club network: The karate club network is a social network consisting of 34 vertices representing people from a karate club at an American university and edges representing friendships between them. Zachary [96] who was studying the social interactions between the members of the club compiled this network. During the course of the study a dispute between the administrator of the club and the instructor of the club resulted in the split of the club into two. The instructor opened another club with about half the members from the original club. The karate club network is shown in Figure 32, the square vertices indicate the instructors group and the round vertices indicate the administrators group. We apply our bibliometric approach to the karate club network to identify the factions in the club. The dendrogram in Figure 33 shows the communities as discovered by our algorithm. All the nodes in the two groups were classified correctly except for the nodes, 10 and 28 which were not classified into any community.

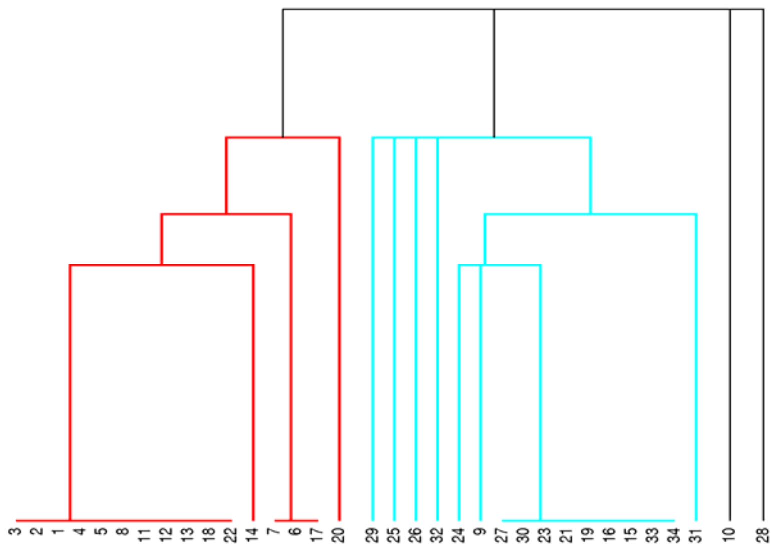


Figure 33: Communities identified in the Zachary Karate Club Network.

The Football team network: Girvan and Newman in [44] first studied the community structure of the football team network. This network represents the regular schedule of the Division I college football games for the year 2000 and consists of 115 nodes. The nodes in the network represent teams and the edges represent matches between them. Out of the 115 teams 110 were divided into 11 conferences and the remaining 5 were not classified into any conference. Each team on average played seven intra-conference games and four inter-conference games during the season. Moreover the inter-conference games were not uniformly distributed with more games being played between teams that are geographically close to one another than with teams that are further apart. Applying our algorithm to this network we were able to extract the conference structure of the network with high precision. Our results are shown in Figure 34 by means of a dendrogram. Labels in the dendrogram represent the name of the team followed by the conference number to which they belong. The teams that did not belong to any conference (represented by conference number 5 in the Figure 34) ended up with the conferences with which they were closely associated. For certain teams the network structure did not portray the conference structure and these teams ended up being misclassified, which was anticipated. For example the Texas Christian team belonging to conference 4 played majority of their games with teams belonging to conference 11.

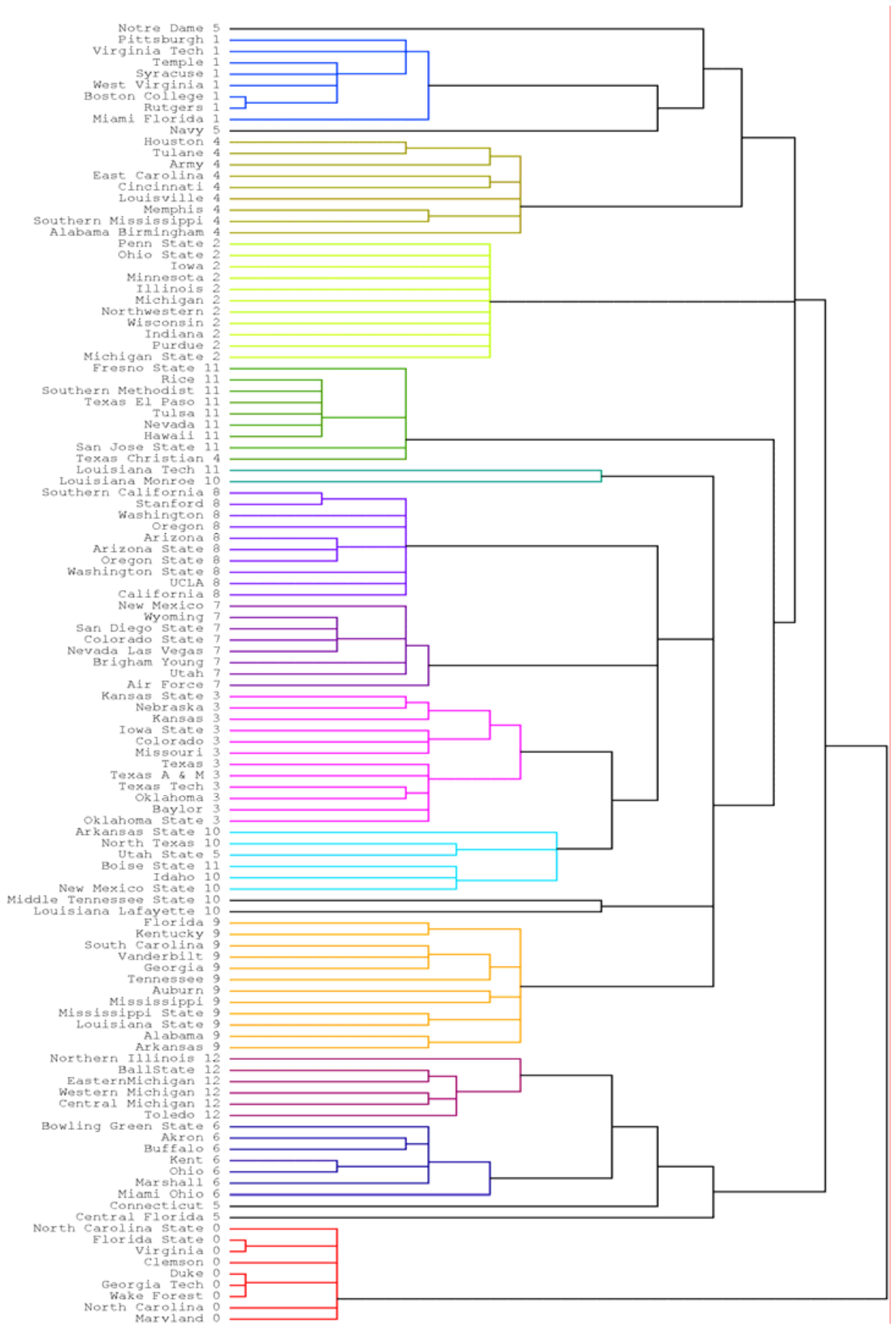


Figure 34: Communities identified in the College Football Network.

The Santa Fe Institute collaboration network: Next we test out algorithm on a scientific collaboration network consisting of scientists from different disciplines at the Santa Fe Institute. Girvan and Newman in [44] studied the community structure of this network. The vertices of the network represent scientists from the Santa Fe Institute and an edge is drawn between two vertices if the corresponding scientists have coauthored at least one publication during the calendar year 1999 or 2000. On average each scientist coauthored articles with approximately five others. The actual network consists of 271 vertices, but here we study the largest component of the network consisting of 117 vertices as the community structure of the former was not available for verification of results. Figure 35 shows the structure of the collaboration network with different vertex shapes indicating different disciplines of research. The entire network could be broken down into four major components and a few of these could be further divided. The vertices represented by squares represent the community of scientists working primarily on the structure of RNA. The vertices represented by triangles, inverted triangles, crossed and circled squares represent scientists working on statistical physics and can be further subdivided. The vertices in diamonds represent the scientists working on the mathematical models in ecology and the ones in dotted diamonds represents the group of scientists using agent-based models to study problems in economics and traffic flow. Application of our algorithm to this collaboration network identifies all the major communities in the network. Our results are shown by means of a dendrogram in Figure 37. The communities representing scientists using agent-based models and the ones working on mathematical ecology seem to be classified as a single community. Further divisions within the scientists working on statistical physics are also visible.

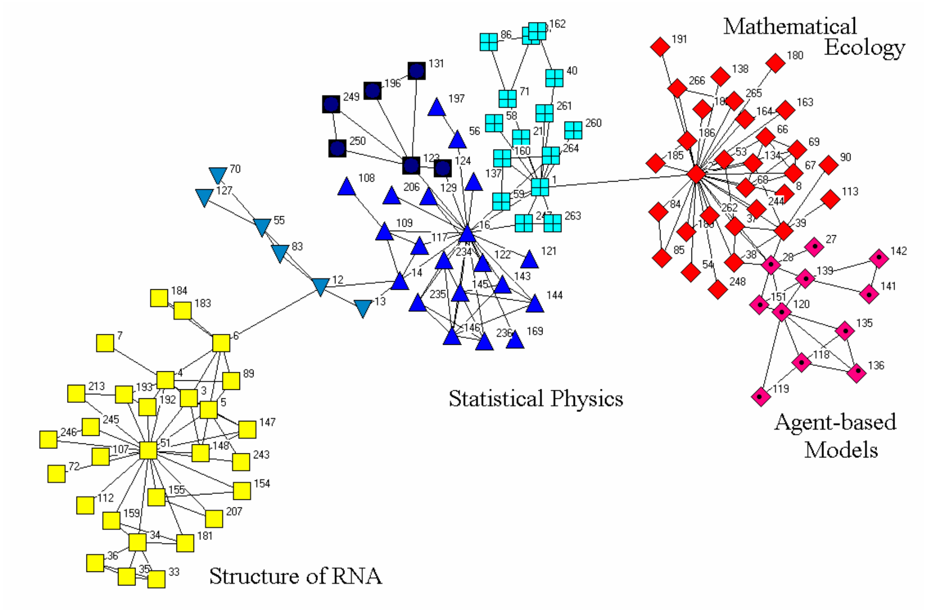


Figure 35: The largest component of the Santa Fe Institute collaboration network.

Roget's thesaurus: To put the algorithm to further testing we test our approach on the Roget's thesaurus network, which consists of 1022 vertices each representing one category in the 1879 edition of Peter Mark Roget's *Thesaurus of English Words and Phrases*, edited by John Lewis Roget [83]. A directed edge is drawn between two vertices *u* and *v*, if Roget gave a reference to the category represented by vertex *v* among the words and phrases of category represented by vertex *u*.

Applying our algorithm to this lexical network resulted in division of the entire network into a number of small communities. Each community consisted of words that were closely related. A few of these communities have been listed in Figure 36.

uniformity	velocity	heat
agreement	haste	thermometer
conformity	earliness	furnace
concurrence	instantaneity	refrigeratory
cooperation	transientness	refrigeration
concord	present-time	cold
peace	different-time	calefaction
Assent	time	investment
	period…	covering…
untruth	clergy	color
falsehood	churchdom	ugliness
misteaching	belif	ornamentation
deception	thoelogy	deterioration
cunning	orthodoxy	blemish
misinterpretation	irreligion	beauty
ambush…	idolatory…	simplicity…

Figure 36: A few communities identified in the Roget's thesaurus network.

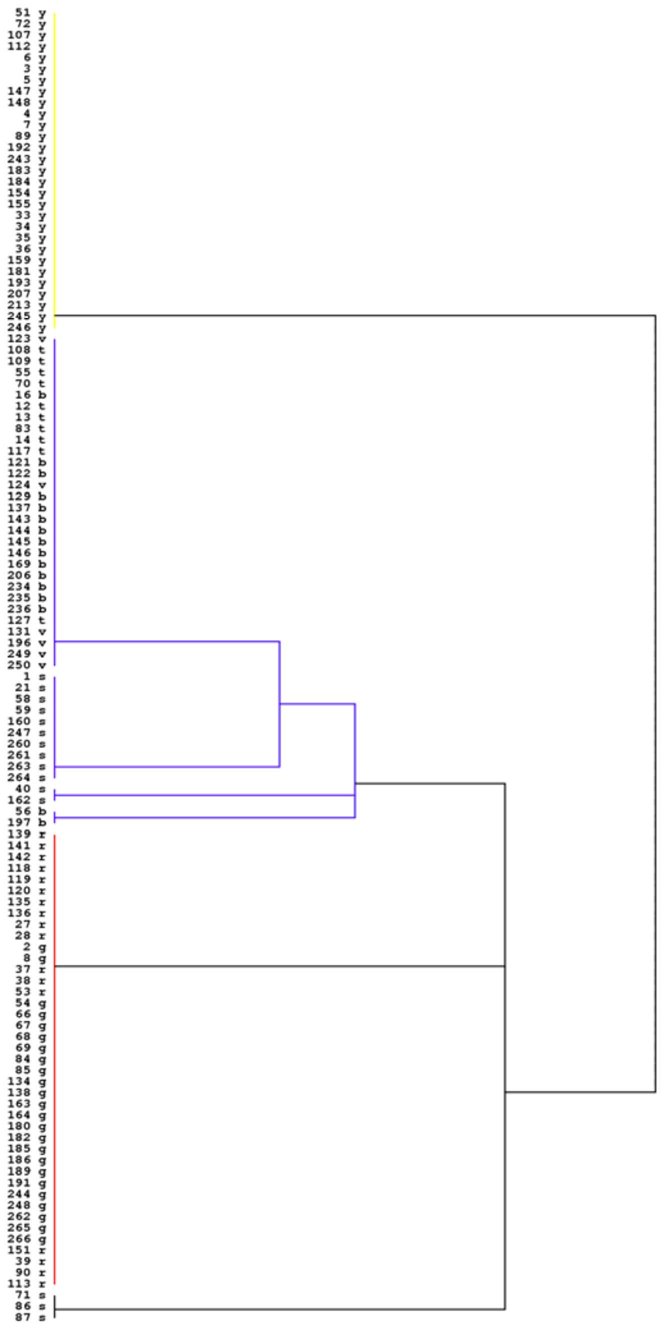


Figure 37: Communities identified in the largest component of the Santa Fe Institute collaboration network.

6. COMMUNITIES IN LARGE NETWORKS

6.1 Requirement

Most of the real world networks of interest e.g. World Wide Web, Internet, social networks, etc. are enormous in size, varying from a few thousand vertices to a few billion. Extracting communities in such networks would help us in mining valuable data concealed by their topology. Algorithms seen so far are computationally expensive with cubic and quartic time complexities. Scaling the existing community discovery algorithms to the size of these networks is a very challenging task. There have been very few publications in this direction. Clauset et al. came up with a hierarchical agglomerative algorithm specifically for large graphs [28]. Their algorithm has a runtime complexity of $O(n \log^2 n)$. Wu and Huberman came up with a modified resistor network algorithm with a complexity of $O(n \log n)$ [95]. However, the algorithm requires apriori knowledge about the network and the output contains only a bisection of the input graph into roughly equal size communities. Newman suggests a $O(n^2)$ algorithm which is based on modularity maximization [68]. Modularity is a value that rates the goodness of the extracted communities (refer Section 8.3).

A part from scaling to the large size of these networks the algorithm has to satisfy a few other properties:

- **Low storage requirements**: The algorithm should employ data structures that require roughly $O(m)$ storage.

- **Adapts to the available memory**: The algorithm should make effective use of the available memory.
- **Parallelizable**: For massive graphs, one must be able to distribute work among several processors with low overheads.

6.2 Proposed algorithm

In this section we propose a new approach built on the earlier bibliometric algorithm. We use a slightly modified adjacency list to satisfy the low storage requirements. The first member of the adjacency list of every vertex is the vertex itself. This helps us in computing the intersection of the closed neighborhood of the vertices as required by the bibliometric approach. One option programmer's use very often while dealing with sets is to use a bit array of size *n*, where setting the i^{th} bit marks its membership. The intersection of two sets could now be obtained by computing the bitwise-AND of the two bit vectors. This method is very effective while dealing with small values of *n*. Unfortunately this method requires O(n^2) space. We instead use a hash table to compute the intersection. The set of vertices belonging to the closed neighborhood of the first vertex are at first inserted into the hash table. The vertices belonging to the closed neighborhood of the second vertex are then looked up in the hash table. If the result of the lookup indicates the presence of a node, we increment a counter that counts the size of the intersection. One could also compute the degree of the vertices in the same iterative loop while performing insertions and lookups.

Repeating this operation with the end vertices of all the edges in the graph helps us compute the bibliometric similarity of all adjacent vertex pairs. The algorithm requires O(m) storage for its adjacency list. Every iterative loop computing the intersection would require a maximum of O(n) storage depending on the implementation of the hash function. The algorithm is also easily parallelizable to deal with very large graphs.

Algorithm 14 gives a more detailed description of the approach. After computing the similarities, using the divisive approach to extract the communities would require a complexity of O(m log m). So we use a different approach with a lower time complexity. After obtaining the bibliometric similarity of all the edges we use a predefined threshold value, τ where $0 \leq \tau \leq 1$, and eliminate all the edges from the graph with similarity value lower than the threshold, τ. Now the components formed by the remaining edges in the graph constitute the extracted communities. Determining the right threshold value would be a challenge, but it has been observed that the threshold values that produce good results remain more or less constant for a given type of the input network. This result is yet to be verified.

```
FasterBibliometricSimilarity($G$)
for every $e_{u,v} \in E$ do
    $d_u = 0$
    $d_v = 0$
    $N_{u,v} = 0$
    hashTable.initialize()
    for every $k$ such that $e_{u,k} \in E$ do
        $d_u = d_u + 1$
        hashTable.insert($k$)
    endfor
    for every $k$ such that $e_{v,k} \in E$ do
        $d_u = d_u + 1$
        if hashTable.lookup($k$) do
            $N_{u,v} = N_{u,v} + 1$
        endif
    endfor
    $S_{u,v} = \frac{N_{u,v}}{\min(d_u, d_v) + 1}$
endfor
```

Algorithm 14: Faster algorithm to compute bibliometric similarity.

6.2.1 *Complexity analysis*

This section analyzes the computational cost of the algorithm. The algorithm has two main loops. One loop iterates over all the edges in the graph and the other iterates over the vertices in the closed neighborhood of a vertex. The second loop computes the size of the intersection of the closed neighborhoods by performing operations on a hash table. The hash table was chosen because it allows for insertion and lookup in constant time. Computing the size of the intersection first involves inserting the vertices from the closed neighborhood of one of the end points of the edge into the graph. There can a maximum of Δ vertices in the closed neighborhood and with constant time insertions, this would take $O(\Delta)$ time. The next step involves looking up the hash table with vertices in the closed neighborhood of the other end of the edge. There can be a maximum of Δ such lookups and with constant time lookups, this would take $O(\Delta)$ time. The outer loop iterates over all the edges in the graph, hence has a complexity of $O(m)$. The overall complexity of the algorithm is

$$= O(m\,(\Delta + \Delta)),$$

$$= O(m\,\Delta).$$

For a sparse graph ($m \approx n$) computing the bibliometric similarities of all the edges can be done in $O(n\,\Delta)$ time.

6.3 Results

We test our algorithm on Newman's synthetic graph with 16384 vertices divided into 32 communities of equal size with $z_{in} = 32$ and $z_{out} = 8$. Figure 38 shows the outputs obtained for various τ

values. For $\tau = 0.125$ close to 92% of the nodes were correctly classified. The main drawback of this algorithm is that it requires priory knowledge about the good τ value for a given network.

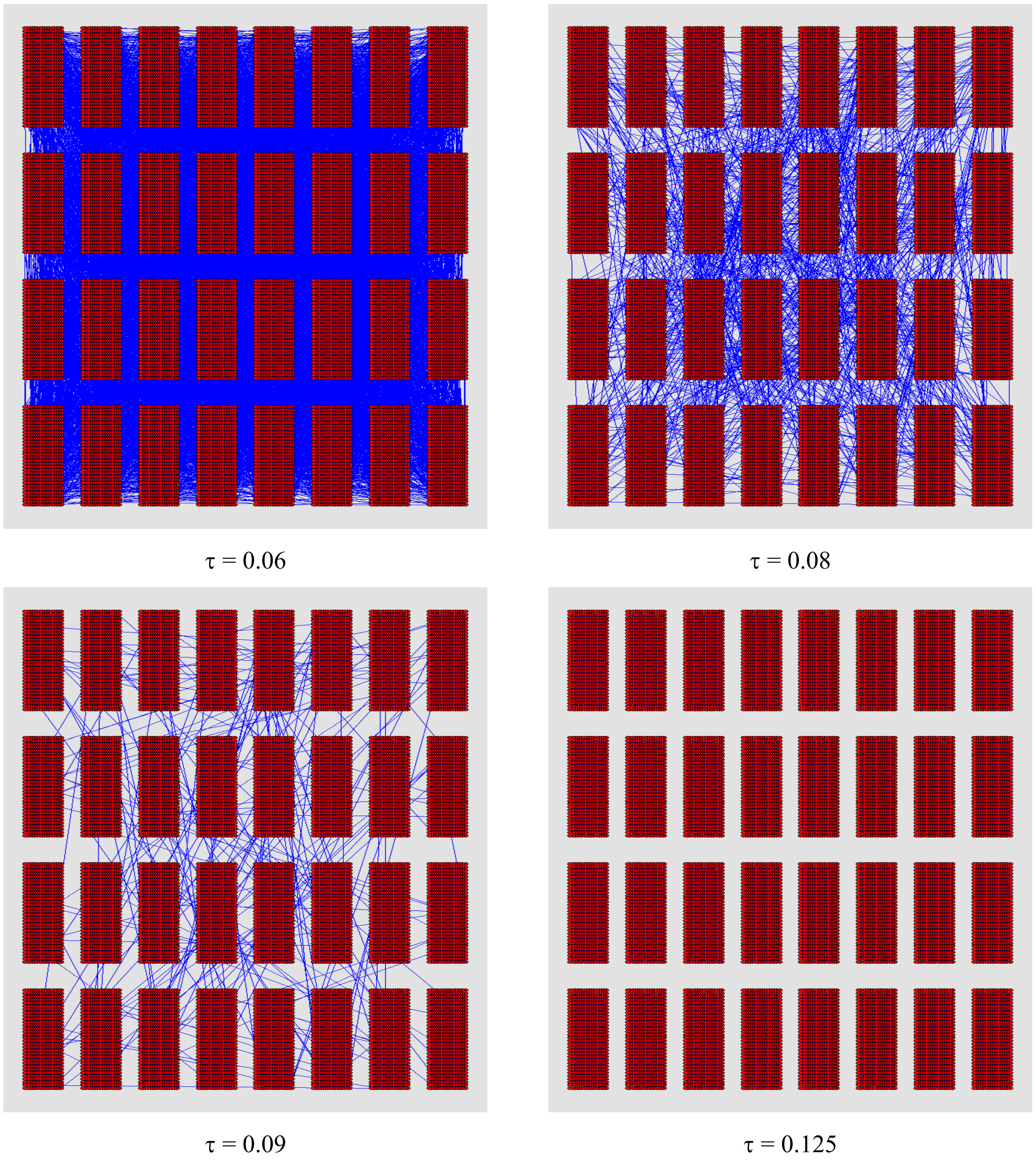


Figure 38: Output for various threshold values.

7. UCFBOT CRAWLER

7.1 Introduction

In this chapter we describe the architecture and implementation details of the robust, fault tolerant UcfBot crawler developed by myself and Cami. This crawler is the part of a project aimed at collecting a huge and suitable data set for our experiments. By definition, a *web crawler*, also known as a *spider*, *wanderer*, *bot*, or *robot*, is a program that downloads a seed page from the World Wide Web, extracts the hyperlinks contained in that page, and then downloads the pages those hyperlinks refer to, extracts the hyperlinks in those pages, and repeats this process recursively. Crawlers are most commonly used for web indexing, information retrieval, HTML validation, hyperlink validation, monitoring for changes in a website, mirroring, *etc*. The initial crawlers had to deal with a considerably smaller size of the web; however today the web spans a few billion pages and all the existing search engines put together index about 50% of the web [ ].

The UcfBot crawler is designed to capture the link structure of the World Wide Web in the form of a graph although it could be easily extended to perform other operations. During its implementation emphasis was paid to make the crawler efficient and fault tolerant. In the case of a crash, one can restart the crawl without losing considerable amounts of accumulated data. The current version of the crawler includes a configuration file, which is primarily used to specify options like the seed set, pages not to crawl, domains to avoid, time interval between successive requests to the same web server, *etc*.

Let's go over a few terminologies and definitions that define the characteristics of a crawler: a *batch crawler* is one which crawls for pages periodically, an *incremental crawler* is one which crawls for pages all the time and tries to keep the collection of pages fresh by frequently crawling pages that change more often, a *scalable crawler* is one whose speed and performance is unaffected if the size of the web increases, a *polite crawler* is one that does not overwhelm the document server with requests and avoids indexing certain links which are not supposed to be indexed, a *fault tolerant crawler* is one which can withstand / overcome a system crash, an *extensible crawler* is one which can be extended to perform other functions apart from what it has been designed for and a *portable crawler* is one which can be run on many different platforms.

7.2 The Crawling Algorithm

A very simple crawler can be written with a few lines of code. However, when one needs to crawl a significant portion of the Web—*i.e.,* hundreds of millions of web pages—the task becomes challenging. Let's look at a straightforward crawling procedure:

```
Crawl(S)
urlsEncountered = S;
urlsToVisit = S;
while ! urlsToVisit.isEmpty() do
        url = urlsToVisit.getNext();
        ip = DNSlookup(url.getHostname());
        page = downloadPage( ip, url.getPath());
        newUrls = parseForHyperLinks(page);
        for all newUrls do
                if newUrl.isRelative() then
                        newUrl = newUrl.makeAbsolute();
                end if
                if ! urlsEncountered.contains(newUrl) then
                        urlsToVisit.insert(newUrl);
                        urlsEncountered.insert(newUrl);
                end if
        end for
end while
```

Algorithm 15: Template for a crawling algorithm.

This procedure begins by initializing the set urlsEncountered (URLs that are known to the crawler) and the set urlsToVisit (URLs that are yet to be crawled) to the seed, S. The seed would preferably be the URL of a web page which contains many hyperlinks—*e.g.,* the home page of an organization—or it could also be a set of URLs. Next, a URL from the set urlsToVisit is extracted using the getNext() function. In order to download the web page being pointed to by this URL, one has to obtain the IP address of the server which holds the document. This is done by contacting a server running the *Domain Name Service* (DNS) which would convert the domain name to an IP address. Now, the web server where the page resides can be contacted and the web page requested for download. After the web page is downloaded, it is parsed in order to extract the hyperlinks pointing to other web pages. The extracted URLs which are found to be relative, *e.g.,* ../../somepage.html, are converted into absolute format, *e.g.,* http://somedomain/somepath/page.html. Afterwards, the procedure checks whether the web pages being pointed to by the newly discovered URLs have been previously crawled. This is achieved by searching the set urlsEncountered. If a URL is found in urlsEncountered, it is discarded, if not it is added to the urlsToVisit and urlsEncountered data structures. This process is repeated until the set urlsToVisit becomes empty or the crawler stops based on some other condition.

The just described procedure is deceptively simple. As we will see in the following sections, numerous issues arise when trying to scale up.

7.3 Related Work

Though there has been considerable amount of work in this area the most efficient crawlers that are built are used by commercial organizations and the details of their implementation are not readily available. However the design of a few of these crawlers has been made public. Web crawlers are as old as the web itself, the first web crawler was written by Matthew Gray in 1993 [46] for measuring the growth rate of the web. Most of the recent high performance crawlers are used by search engines to build large search indexes. These crawlers use multiple machines to scale up to the size of the web; based on this aspect the crawlers could be classified to be *homogeneous* and *heterogeneous*. A homogeneous crawler is one in which all the parallel subsystems of the crawler perform the same task, such crawlers can be easily scaled up, on the other hand heterogeneous crawlers are the ones in which each of the parallel subsystems performs a different task. Let us briefly look into the designs of a few of these crawlers in chronological order.

GoogleBot: Was written by Brin and Page as a part of their new search engine "Backrub". Although the design and working of the most recent Google crawler is a well-kept secret, their initial crawler has been described in [14]. The crawler was written in C++/Phython and performed both crawling and indexing of web pages simultaneously. The main purpose of this project was to introduce a new ranking strategy for web pages which would determine the order in which the search results would be displayed. The GoogleBot possesses a heterogeneous architecture. Apart from crawling and indexing the web the GoogleBot also stored compressed form of web pages in its huge database.

Incremental Crawler: The incremental crawler was written by Cho and Garcia-Molina and its implementation details can be found in [22]. As the name implies the main goal of this crawler was to download a copy of the web and then try to keep this copy fresh by downloading only pages which would change over the given time. This reduces the number of pages that required to be downloaded. Cho and Garcia-Molina along with Page have also worked on scheduling algorithms that determine the best order in which the pages have to be crawled to obtain the important pages first [23].

Mercator: The Mercator crawler is another high performance crawler written by Najork and Heydon and its implementation is described in [65]. This crawler was later used by the AltaVista search engine. The crawler was written entirely in JAVA and has a homogeneous architecture. The main features of Mercator are that it's distributed, scalable, efficient, polite and portable.

Webfountain: Developed by Edward, McCurley and Tomlin for IBM [33].

All the above mentioned crawlers use multiple machines and try to scale up to the current size of the web. Cho and Gracia-Molina [24] describe the issues that need to be addressed in parallel crawling.

7.4 UcfBot Architecture

In order to be of any use, a crawler has to deal with numerous issues arising from the enormousness of the Web. First, it has to be very fast. For instance, it is believed that Google—which indexed more than 8 billion web pages as of 2005—indexes the web once every month (a process known as the Google "dance" [86]), *i.e.,* at the rate of approximately 3000 pages per second. Apart for achieving such high speed, crawlers are expected to follow a certain etiquette and be polite during the crawl. These and other issues make crawling a complex endeavor.

Next, we describe the architecture of UcfBot, focusing mainly on the performance issues mentioned above. UcfBot crawler consists of 6 main components: (1) the urlsToVisit data structure; (2) the DNS lookup module for obtaining IP addresses; (3) the *pageFetch* module for downloading pages; (3) the *Parser* module for extracting hyperlinks; (4) the urlsEncountered data structure; and (6) Checkpointing module for restoration of the system in the event of a failure. **Figure 39**, shows these components and interactions among them.

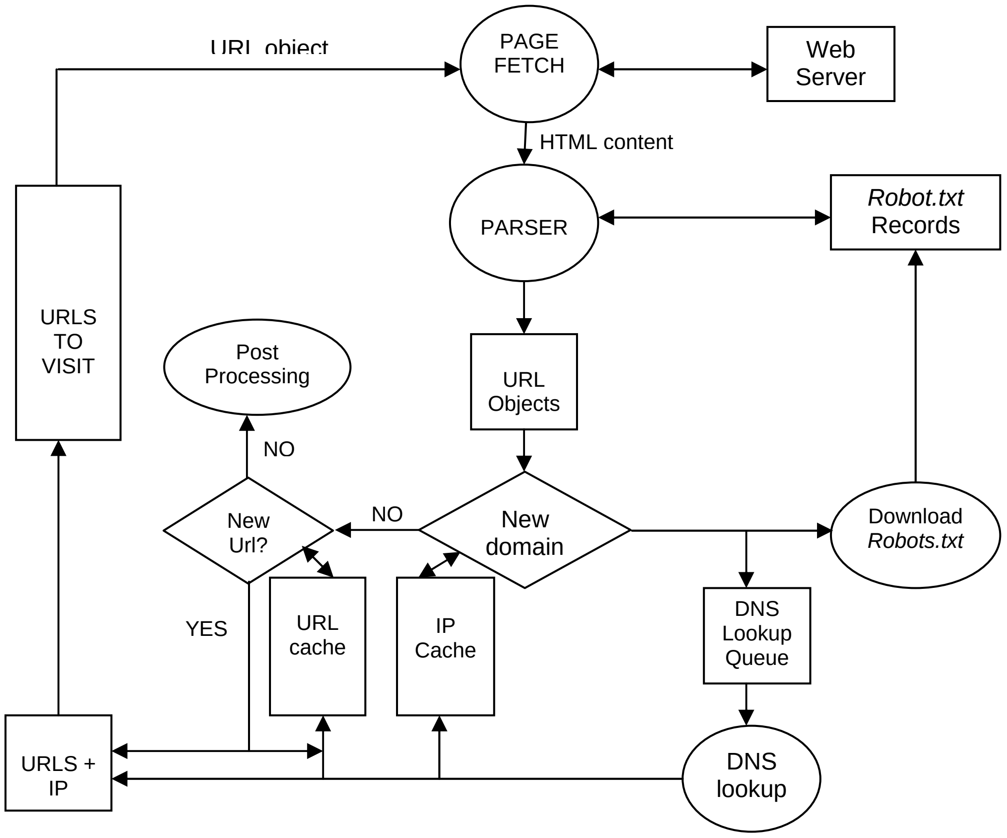


Figure 39: The architecture of UcfBot.

Now, we describe each of these components in detail.

7.5 urlsToVisit Data Structure

The urlsToVisit data structure holds the set of pages that are yet to be crawled. It provides 2 important functions urlsToVisit.insert(URL) and urlsToVisit.getNext() for inserting newly found hyperlinks and obtaining the next URL to crawl. Initially it would hold the seed S, during each step of the crawl one URL is removed from the set using the getNext() function. The getNext() function determines the order in which the URLs would be crawled. If one wants to crawl the entire web without any bias this order would not matter but if one wants to retrieve a subset of the web and wants to do it fast, a number of options are available, for example:

- Breadth first traversal
- Depth first traversal
- Best first traversal

Breadth first traversal and depth first traversal would not require much explanation. If breadth first traversal is employed then the urlsToVisit data structure would be implemented as a queue and if depth first traversal is employed then the urlsToVisit data structure would be implemented as a stack. For topic-based crawling breadth first traversal is said to yield better results [64]. The best first strategy refers to the ordering of the URLs based on some priority scheme, for example, using Page Rank as described by Google [75], or based on the number of hyperlinks coming out/pointing to the page. The data structure used is a priority queue. The urlsToVisit.insert() function would determine the priority of the URL and insert it at the appropriate position in the queue where as urlsToVisit.getNext() function would remove the first URL from the queue.

Though the above ordering schemes are straightforward, we need to make sure that the crawler performs the crawl in a polite manner. By polite we mean not bombarding a server with requests. Placing one request every 4 seconds to the same server is considered to be polite, so the urlsToVisit.getNext() function should return URLs from different domains during consecutive calls. One way of achieving this is to use multiple queues, as shown in Figure 40.

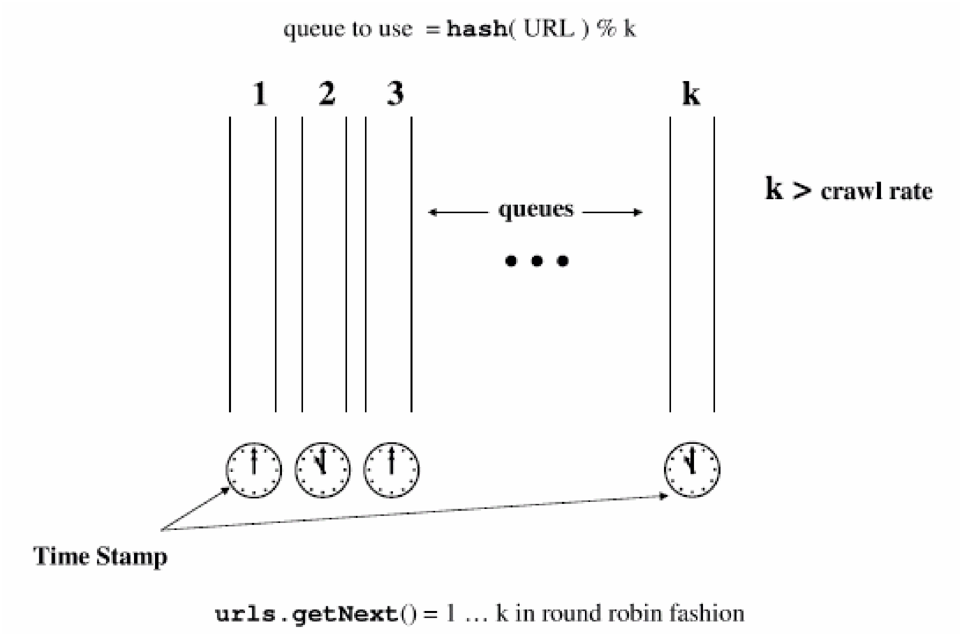

queue to use = hash(URL) % k
1
2
3
k
queues
k > crawl rate
Time Stamp
urls.getNext() = 1 … k in round robin fashion

Figure 40: Implementation of the urlsToVisit data structure.

Number of queues used k > 4 × crawling rate. The urlsToVisit.insert() function would insert the set of URLs from the same domain into the same queue by using a hash function or a checksum on the domain names and the urlsToVisit.getNext() function would remove one URL from each queue in a round robin fashion.

```
urlsToVisit.insert(URL)
queue = URL.hashDomain()% k
insertin(queue, URL)
```

Algorithm 16: To insert an URL to the queue.

```
urlsToVisit.getNext()
counter++
queue = counter % k
while getTimeStampOf(queue) + 4 < currentTime &&
  getTimeStampOf(queue) != ∅ do
        queue = (queue + 1) % k
endwhile
URL = getNextFrom(queue);
if getNextDomain(queue) == URL.getDomainName()  then
        timeStamp(queue) = currentTime
else
        timeStamp(queue) = ∅
endif
return URL;
```

Algorithm 17: To obtain the next URL from the queue.

Each queue has a time stamp associated with it which is updated when a URL is removed from the queue and the next URL from the queue is from the same domain as the current URL, if not the time stamp is made null. For each fetch from a queue the urlsToVisit.getNext() function checks the time stamp to see if its null or the previous fetch from this queue was done before 4 seconds. Due to high domain locality, URLs being extracted from a page would be of the same domain and would end up in the same queue but, as the urlsToVisit.getNext() function would access different queues for each step, the above requirement is met. This method is similar to what was implemented on the Mercator [50]. Castillo et al. survey various scheduling algorithms for crawling in [19]. Any computer system has finite amount of memory and the urlsToVist data structure would grow at a high rate that would require storing a major portion of the queue on disk. Only a fixed portion of the queue would be held in memory. Inventory is performed at regular intervals and data is retrieved from disk or the tail portion of the queue, which was accumulated, is written to disk as required.

7.6 DNS Lookup module

The *UrlsToVisit* data structure contains the URLs that are to be crawled. But in order to request from a *web server* the web page that is pointed to by an URL, one has to convert the domain name encapsulated in the URL into an IP address. This is the job of the *Domain Name Service* (DNS) *Lookup* module.

Every Internet Service Provider (ISP) provides a DNS server to its customers. Using this server would require a DNS lookup to be first sent to the DNS server and resolved before a page may be requested. For browsing the Web this might not be a big delay but for a crawler this ends up being a bottleneck. The DNS system is not very complex and can be run on a moderate PC with 256 MB of RAM and a disk cache of few gigabytes. Running the DNS on a machine in the local network would reduce considerably the amount of time needed for the DNS lookup. However, most of the available implementations of DNS clients, *e.g.* BIND[3], gethostbyname() (the DNS client API), are *synchronous,* implying that only one call can be placed to the DNS server at a given instance; the next call could be placed only after the results of the previous call have been obtained. Needless to say, synchronous access to a DNS server is prohibitively time consuming for a crawler. Most of the existing crawlers implement their own custom client for DNS resolution, capable of handling parallel calls asynchronously.

The DNS lookup module of UcfBot has asynchronous access capability achieved through adns—a DNS client library for the C/C++ languages [51]. This would enable the pre-fetching of the IP addresses corresponding to the URLs waiting to be visited, by submitting multiple DNS requests in parallel.

7.7 Page Fetch

After resolving the ip address of a server a socket connection is open with the http server to obtain a page. Each server would respond to this request in a different manner at various speeds. A few of these servers could be nonexistent or be very slow to replying. Performing this operation *synchronously* (waiting for completion of one request before placing the next) could seriously reduce the speed of the crawler. This is overcome by using multiple threads or processes. Each thread opens a socket connection to one http server and is responsible for

[3]See http://www.isc.org

obtaining one page. The requests could be blocking or non-blocking. Managing the threads becomes difficult if one were to allow infinite amount of threads, instead most crawlers have fixed number of threads, which obtain individual URLs to crawl from the urlsToVisit module. Some people prefer using processes instead of threads because failure of one process would not affect the state of other processes. The drawbacks of the above approach are: need for mutual exclusion and concurrent access to data structures which is a performance penalty and the threads/processes cause random input-output on disk resulting in slow disk seeks.

7.8 Parser

The Parser module takes as input an HTML file generated from the PageFetch module, It extracts from it, all the hyperlinks, filters them based on the specifications given in a configuration file. An URL object is created for each hyperlink that passes through filtering and returns these URL objects to the main thread of the control.

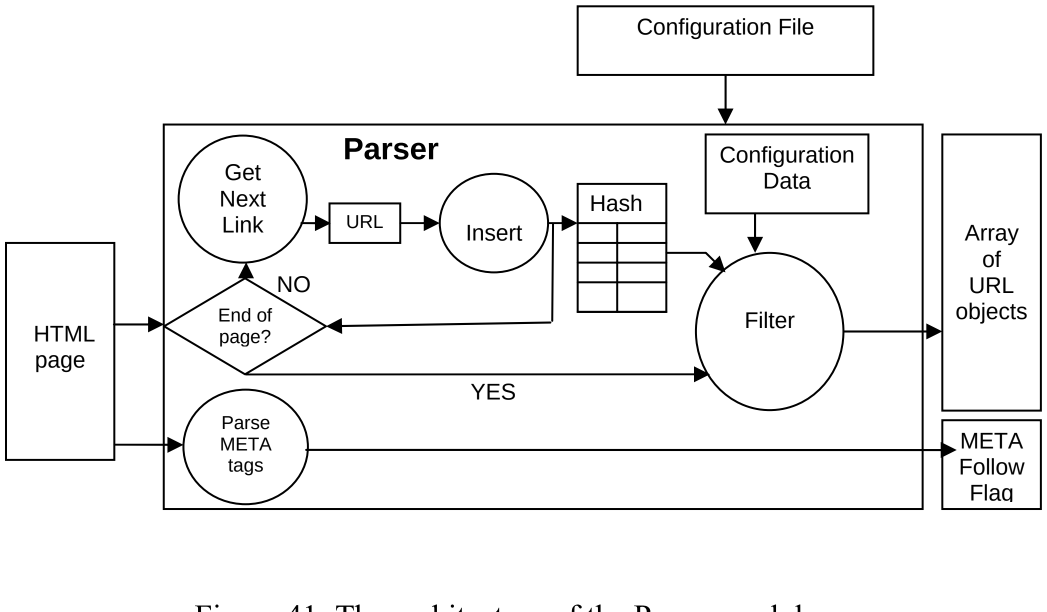


Figure 41: The architecture of the Parser module.

Figure 41 is a schematic representation of the Parser module. Two options are available when implementing a parser module with the functionality described above. First, one may develop a parser from scratch, possibly using tools such as Lex and Yacc. An example of such a parser is the custom-built parser of the Google bot, which is developed on top of a lexical analyzer [15] (it has been claimed that although the parser is very effective, its development was a very time-consuming effort). Second, one could build the parser based on one of several existing of-the-shelf HTML parsers.

Following the latter approach, the UcfBot parser is built on top of the libwww library—a large C/C++ library developed by the World Wide Web Consortium (W3C). The main advantage of using libwww is that this library has been tested in numerous real applications and its parsing engine is very fast. The main shortcoming of this library is the lack of proper documentation, which makes the integration of the library with the rest of the crawler a challenging task.

Next, we describe briefly the functionality of libwww that is utilized by UcfBot and give additional details on the various parts of the parser.

7.8.1 *libwww library*

In the web site of the libwww library [73] it is stated that:

> "…libwww is a highly modular, general-purpose client side Web API written in C for Unix and Windows (Win32). It's well suited for both small and large applications, like browser/editors, robots, batch tools, *etc*. Pluggable modules provided with libwww include complete HTTP/1.1 (with caching, pipelining, PUT, POST, Digest Authentication, deflate, *etc*.,) MySQL logging, FTP, HTML/4, XML (expat), RDF

(SiRPAC), WebDAV, and much more. The purpose of libwww is to serve as a testbed for protocol experiments…”

As stated in this excerpt, libwww is a versatile library providing functionality than goes much beyond parsing. The programming model of this library is *event-based*: whenever a hyperlink is detected in the input HTML stream, a user-defined function that takes this hyperlink as a parameter, is called. The parser of UcfBot uses several utility functions provided by the libwww API to perform post-processing on the extracted hyperlinks: First, all relative links are converted to the absolute format. Second, each absolute hyperlink is broken into the following parts: (1) the access method, such as http, https, *etc*.; (2) the domain name, *e.g.,* www.ucf.edu; (3) port number, *e.g.,* :80; and (4) path name, *e.g.,* somefile.html. All these different fields are packed into a URL object and the list of all such objects is passes to the filtering procedure described below. The part of the libwww API utilized by the UcfBot is represented inFigure 39 by the GetNextLink procedure.

7.8.2 *Removing duplicates*

All URL objects returned by the GetNextLink procedure are inserted into a hash table in order to ensure that there are no duplicates. This operation is represented by the *Insert* procedure in Figure 39.

7.8.3 *Robots META Element*

The META element of an HTML file provides metadata, such as a document's keywords, description, author information, *etc.* The Head of an HTML document may contain any number of META elements.

The Robots META element is used to provide directives to robots crawling an HTML page. These directives can tell a robot whether this HTML page is to be indexed and whether its hyperlinks may be followed. For instance, the element:

<META NAME=”robots” CONTENT=”noindex,follow”>

specifies that the page may not be indexed, but its hyperlinks may be followed.

As mentioned earlier, UcfBot does not do any indexing of the visited web pages and thus its parser searches only for the follow/nofollow directives. This information is returned to the main thread of control through a flag, represented by the “META Follow Flag” in Figure 39.

7.8.4 *Filtering*

Only a fraction of the hyperlinks found in a web page would point to other web pages. Another fraction of these hyperlinks may point to scripts such as *.cgi, .asp,* and *.exe* files, to application-specific files such as *.doc*, *.pdf,* or *.ppt* files, to images, *etc.* The rest of the hyperlinks might contain access schemes such as *mailto* and *gopher*, which are not of interest when extracting the graph structure of the Web.

The filtering procedure aims to detect and filter out all the hyperlinks that don’t point to HTML files. This function of this procedure is controlled by a configuration file, which has the format shown below:

```
#ALLOWED EXTENSIONS:
htm html asp aspx php php3 php4 pl
#FORBIDDEN EXTENSIONS:
js jsp dwt lbi css cfm jsp png cfml mspx gif avi jpeg jpg tiff tif mpeg midi pdf
```

At the onset of each crawl, the parser reads the information contained in this configuration file into a data structure. This filtering information is subsequently used during the whole crawling process. The list of URLs that pass through the filter is passed to the main thread of control, as shown in Figure 39.

7.9 urlsEncountered Data Structure

The list of URLs that have been crawled is stored in the urlsEncountered data structure. It provides 2 main functions urlsEncountered.insert(URL) and urlsVisited.contains(URL) for inserting URLs and checking for duplicate URLs. One could store the entire hyperlink or just its checksum/hash value. As different URLs are of different sizes it would make sense if we could just store the checksum/hash values, but if we are required to crawl the same page twice (for refreshing) then we have no other choice but to save the entire URL. The main purpose of this data structure is to avoid crawling the same page over and over again, once a hyperlink is found its URL has to be compared to all the URLs that have been already encountered to avoid duplication. A real bad implementation would store the entire URL strings in a list and compare the new URL sequentially with this list. This might be time consuming, one improvement would be to use a sorted list and perform binary search but a still better implementation would store just the checksum/hash values of the URL and when a new URL is encountered we can perform a binary search using the checksum/hash value of the new URL. The MD5 hash algorithm would again be of great use at this point. As explained before we can store a list of a billion URLs on 16 GB of memory. As this list has to be kept in a sorted manner, each urlsEncountered.insert(URL) function call would take O(*log n*) time, which is not a lot considering that each urlsEncountered.contains(URL) function call would also take the same time. Using a small buffer to hold the most recently crawled URLs in main memory we can first try to check if a newly found URL is present in the buffer; if found the new URL is discarded, if not we can store all such URLs in another temporary sorted list until it grows to a considerable size and then try to check for duplicates (this is effective because of the high domain locality of the URLs). This kind of batch processing can be well tuned by trying different buffer sizes for different size of the temporary list.

7.10 Checkpointing

A typical crawl of the web usually lasts for days, if the system were to crash, say after 20 days of operation then all the data collected till then would be lost and the crawler has to start its crawl from the seed again. Checkpointing is a way of storing the data that has been collected along with the current state of the system onto the disk. Checkpointing could be done 2 or 3 times per day and in the event of a crash the system can be restored back to a state, as during it last checkpoint. Since we have stored the state of the crawler on disk this information can be used to start the crawler from that point onwards. Though checkpointing would reduce the speed of the crawler momentarily it would be of great help in the event of a system failure.

The checkpointing operation involves backing up of the two main data structures the urlsEncountered which stores the list of URL's that are known to the crawler and the urlsToVisit which stores the list of URL's that are to be crawled next. The checkpointing operation is performed after crawling some fixed number of pages. During checkpointing the regular crawling is paused momentarily while the data is backed up on disk. The checkpointing is done as an incremental operation in the case of urlsEncountered data structure. By incremental we mean that the current checkpointing operation would backup data that has been freshly obtained after the previous one. The incremental backup is not of much help in the case of urlsToVisit datastructure because of its dynamic nature. The data once backed up is also compressed using gzip.

In the event of a crash the crawler config file can be used to restore the crawler to the last checkpoint, that was performed.

7.11 A Polite Crawler

Unsupervised crawling raises numerous ethical issues. UcfBot abides by the generally accepted ethics for crawlers. First, it identifies itself to each Web server it visits, so that the administrator of that server would know whom to contact in the event of a server crash. Furthermore, it takes care to avoid denial-of-service (DOS) attacks and to follow the robot exclusion protocol.

7.11.1 *Avoiding DOS attacks*

Frequently, the number of simultaneous connections to a web site is limited. Requesting a large number of documents from the same server in a very short time span might lead to a denial-of-service attack. A widely accepted rule aiming to prevent such behavior on the part of the crawler is to allow a time lapse of at least 4 seconds between successive requests to the same server. UcfBot implements this delay mechanism through the round-robin access to the queues that make up the urlsToVisit data structure, as explained in Section 7.5.

7.11.2 *Robot Exclusion Protocol and META elements*

The robots.txt[4] file is used by web site owners to indicate certain parts of a web site that should not be crawled or indexed. This file may also indicate the names of certain crawlers to whom the domain is not open.

When it accesses a new domain, say http://www.ucf.edu, UcfBot first attempts to access the page http://www.ucf.edu/robots.txt, which would be the location of robots.txt file for this domain. If such a file exists it is parsed into an array of records, which is then passed to the Parser module (see Figure 39).

7.12 Mirroring

The web contains many pages, which have the same content but different names. These duplicates are known as mirrors. Likewise, some organizations have several domains mapped to the same machine or several machines—each with a different IP address—mapped to a single

[4]`http://www.robotstxt.org/wc/robots.html`

domain. In order to avoid indexing duplicates, it is desirable that a crawler has the capability of recognizing mirrors.

A general solution to this problem of recognizing two pages that have the same content is to hash their respective contents and compare the hash values. The Message Digest 5 (MD5) [82] hash function is considered to be very effective for this purpose. Firstly, it can work on any arbitrary size string (in our case HTML source of a page) and return a 128 bit string. It is fast and claims that no two distinct strings can be mapped to the same hash value. The main advantage is that instead of storing the contents of the entire web page each of which could be several KB, we just store a 128 bit value to represent its content. That is to store the content of say a billion pages all we would require is 16 GB of memory space. By storing a part of this on main memory and the remainder on disk we can perform this operation efficiently. Bharat et al. [10] describe several ways of identifying mirror sites on the web. Currently, UcfBot does not implement any mechanism for detecting mirror web sites.

7.13 Conclusion

The current version of UcfBot was written in C++ and is multithreaded in operation. UcfBot along with a few extensions was used to collect the data used for studying the evolution of the www.ucf.edu web domain (see Chapter 2). The project has helped me realize the problems that are associated with crawling. The knowledge gained would be used in coding the next version of UcfBot which would be a parallel, high performance crawler coded in C++ capable of crawling hundreds of pages every second.

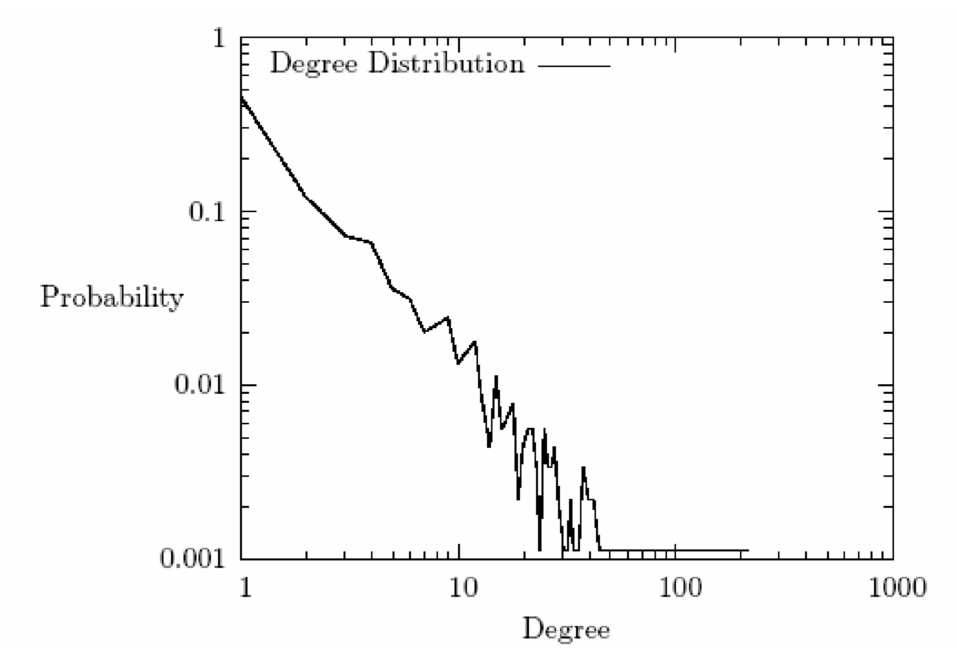


Figure 42: Degree distribution of a small portion of www.ucf.edu domain.

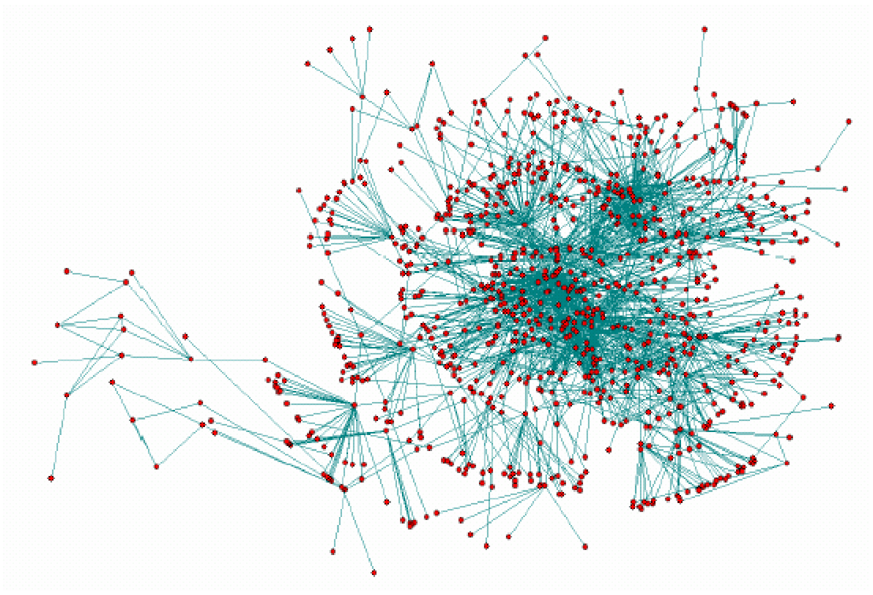


Figure 43: A snapshot of the network of the www.ucf.edu domain.

8. CONCLUSION AND FUTURE DIRECTIONS

The primary objective of the current research is to devise fast algorithms for identifying communities in large real-world networks. There are several other problems associated with the community structure of networks, which need to be addressed. This section provides a summary of the current work and future work related to this area.

8.1 Overlapping communities

The existing methods for discovering/identifying communities in large networks find disjoint communities, while the actual networks are made of overlapping cohesive groups of vertices. For example a large fraction of proteins belong to several protein complexes simultaneously. Little effort has been made in this direction [8, 76].

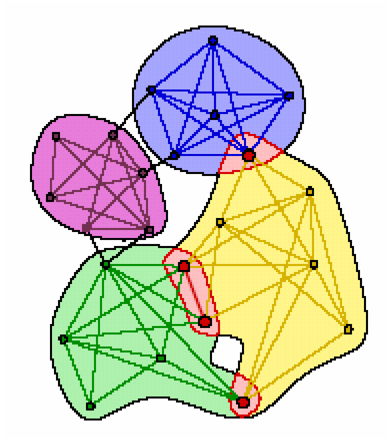


Figure 44: Overlap in communities.

8.2 New definition

Section 1.3 provided insight into the existing definitions for *community*, the main reason for this ambiguity is due to the fact that none of the definitions capture the notion of a true community. For example the definition based on edge density, would fail to account for the two communities in bipartite graphs, both of which share no edges with members of their community. None of the definitions in Section 1.3 addresses the possibility of overlap among communities. We also believe that there exists a central core of nodes within each community that are true members of the community.

To overcome these shortcomings, we propose defining a community of a node to be a vector of size $|C|$ where C is the set of communities in the graph. The community of a node u, $C_u = \{c_1, c_2, \ldots, c_{|C|}\}$, where c_i is a value between 0 and 1 that represents how much a node belongs to the community i and $\sum_i c_i = 1$. A value of $c_i = 1$ indicates total belonging of the node to community i, and a value of $c_i = 0$ indicates that the node does not belong to the community i. Each community i would consist of at least one node with its $c_i = 1$ and a central core of nodes with $c_i \approx 1$.

8.3 Quality of communities

Though we have numerous community-discovery algorithms we do not have a metric to measure the quality of communities produced. Without apriori knowledge of the true community structure of input networks, the goodness of communities obtained by a community discovery/identification algorithm cannot be judged. Newman and Girvan define a measure of the quality of a particular division of a network called *modularity* [70]. This measure is based on a previous measure of assortative mixing proposed by Newman [66]. Consider a particular division of a network into k communities. Let us define a $k \times k$ symmetric matrix $\check{C}$ whose element $\check{C}_{ij}$ is the fraction of all edges in the network that link vertices in community i to vertices in community j. (Here they consider all edges in the original network—even after edges have been removed by the community structure algorithm their modularity measure is calculated using the full network.).

The trace of the matrix $T(\check{C}) = \sum_i \breve{C}_{ii}$ gives the fraction of edges in the network that connect nodes in the same community, and clearly a good division into communities should have a high value of this trace. The trace on its own, however, is not a good indicator of the quality of the division since, for example, placing all vertices in a single community would give the maximal value of $T(\check{C}) = 1$ while giving no information about community structure at all. To deal with this, Newman and Girvan define the row (or column) sums $a_i = \sum_j \breve{C}_{ij}$, which represent the fraction of edges that connect to vertices in community i. In a network in which edges fall between vertices without regard for the communities they belong to, one would have $e_{ij} = a_i a_j$. Now, the modularity measure

$$Q = \sum_i (\breve{C}_{ii} - a_i^2) = T(\breve{C}) - \| \breve{C}^2 \| ,$$

where $\| X \|$ indicates the sum of the elements of the matrix X. This quantity measures the fraction of the edges in the network that connect vertices of the same type (i.e., within-community edges) minus the expected value of the same quantity in a network with the same community divisions but random connections between the vertices. If the number of within-community edges is no better than random, we will get $Q = 0$. Values approaching $Q = 1$, which is the maximum, indicate strong community structure. In practice, values for such networks typically fall in the range from about 0.3 to 0.7. Higher values are rare.

The main drawback for this approach is that one requires the entire graph and all the divisions of the graph to compute its modularity and this makes it not suitable for performing community identification. The definition also does not address overlapping of communities.

8.4 Algorithms for community identification

Most of the existing community structure algorithms do not perform community identification. Community identification is an easier problem than community discovery especially if the input graph is very large or if the entire graph is unavailable for input. This would be well suited for web graphs where obtaining the entire graph is a daunting task. We believe community identification is well suited for online algorithms and for web graphs can be performed while crawling. Another interesting task would be to adapt the existing community discovery algorithms to perform community identification. Improvising the existing algorithms and coming up with approximate algorithms of reduced complexity that can scale up to the large size of some of the real-world random networks.

8.5 Centrality based community identification

Section 4.1 provided some insight in the concept of centrality measures and utilizing them to discover communities. In an effort to perform community identification using centrality metrics we would like to try a new approach as described below:

Start with a small subgraph representing the community then at each step a new node from the neighborhood of the subgraph is included in the subgraph. The node selected at each stage is the one that increases the centrality of the subgraph on a whole. One could stop at any stage and the subgraph formed represents the community that has been identified. The choice of the subgraph to start with decides the quality of the community produced. One obvious choice is to start with a subgraph composed of the seed nodes.

8.6 Parallel algorithms

The algorithms discussed in Section 3.2 have a major drawback, their time complexity is either cubic or quartic. This brings in the scalability issue, considering the large size of the real-world networks. One solution to this is to use parallel versions of the algorithms [74, 94] to scale up to the order of the networks. The parallel algorithms have to deal a number of issues a few of which are:

a. **Distribution of the dataset:** The enormous size of the input data set makes it infeasible for every machine to store a local copy. A solution to this problem could be to either distribute the entire data set among the parallel machines which would lead to a communication overhead or to use a central server to store the entire data set in which case this server would become a bottleneck.

b. **Division of the problem:** There are two popular approaches homogeneous and heterogeneous. *Homogeneous* division would imply that all the parallel machines would perform the same operation on different parts of the data set whereas *heterogeneous* division would imply different operations being performed by different machines on a single dataset.

c. **Quality of the communities:** While scalability and the speed of the algorithms are the main parameters of the study, importance is also provided to the quality of the communities being produced.

d. **Gathering of the results:** Once the communities have been identified by the parallel algorithm the result has to be accumulated in a form that can be put to further use.

Efforts will be made to parallelize a few of the existing algorithms for community identification and to come up with newer algorithms that are more efficient

8.7 Applications

Though there are numerous applications for community mining we concentrated on the applications related to the web graph. A few of these applications have been described below.

8.7.1 *Visualization of search results*

A user searching the Web can be overwhelmed by thousands, or even millions, of results returned by a search engine. Besides that, queries are often prone to ambiguity, and sometimes only few of the returned results are relevant. The PageRank [75] ranking algorithm took a first step to remedy these issues by assigning a prestige value to each website and sorting the responses by the prestige value before returning them. Obviously, more needs to be done. For instance, a user looking for information about the Amazon rain forest would type the word “Amazon” as his search query but any search engine employing ranking techniques would return information about the online bookseller www.amazon.com due to its high rank. To overcome this ambiguity search engines can group their results based on content. For example, for the above query if the search engine were to cluster the search results into topics like "Rain forest", "Bookseller", "Female warriors" etc. then the user may selectively view the group of pages that he is interested in.

8.7.2 *Automated web directory generation*

Web directories like http://www.google.com and http://directory.yahoo.com have been created to organize and catalogue web pages. Web directories provide an alternative to search engines while looking for information. Though very affective, creating and maintaining such web directories to a large extent is still done manually. If a new page has to be added to the directory the user is required to submit his web URL along with some content related information, which is again manually verified and categorized appropriately. This process can be done without human intervention by using a topical crawler. Initial work on mining web communities was done by employing text based [20, 21, 45] document clustering techniques, where in each web page was converted into a vector of strings and then compared to other pages for similarity. This approach though very effective is time consuming and is not scalable to the size of the Web. Moreover these document-clustering techniques are prone to spamming. Lately people have been employing graph theoretic techniques to mine Web communities.

8.7.3 *Focused crawling*

Many experts agree [2] that exhaustive crawling is becoming increasingly unattainable due to the huge size and dynamic content of the Web. A potential solution to this problem is the design of focused (or, topical) crawlers [30] which selectively seek out pages that are relevant to a pre-defined topic. Of course, underlying such a focused crawler there must be an algorithm that directs the crawl towards the valuable pages, i.e., an algorithm for mining Web communities.

8.7.4 *Network security enhancement*

Cyber attacks have become increasingly severe, as a result of faster and more malicious propagation techniques. Current network-security solutions, such as antivirus software and firewalls offer local protection from known cyber attacks only. A more promising approach would be to detect these malicious worms by monitoring their propagation pattern. A new approach for controlling the propagation of network worms, based on the game of cops and robbers [29] has been suggested in [80]. Copies of a worm can be viewed as robbers traversing a cyber-graph (a graph that models a network of interest, such as, the Web, the Internet, an e-mail network, etc.). To block robbers' propagation, software agents (cops) can be deployed at some nodes and moved along the edges of the graph. Understanding the nature and structure of computer networks would aid us in determining the spread of computer viruses and worms over the network. Characterization of the community structure of a cyber-graph can potentially be utilized to design efficient quarantining strategies, for example by placing the cops in the sparse regions of the cyber-graph.

8.7.5 *Other applications*

Other interesting applications would include Web filtering (*e.g.,* identification of hate or pornographic websites) [35], selective advertising, assisting search engines in handling Web spamming, *etc*. Apart from the web, community mining is of great interest in social and biological networks where it could be used to study the interaction between people or animals or the spread of diseases. It can be employed in gene clustering which aids in medical diagnostics and can also reveal insights into functional genomics or in image segmentation to separate the background of the image from its foreground.

8.8 Conclusion

In this dissertation we investigated the community structure of real-world random networks and ways of mining these structures. At first we study properties of real-world networks and their existing models. We perform an analysis of the evolution of the web graph and study changes in their properties over time. Next we perform an in-depth survey of the existing community discovery algorithms. We then propose a new bibliographic metric, which can be utilized to extract communities from real-world networks very effectively. We have also proposed a new faster algorithm for extracting communities in very large graphs and finally state several open problems for future direction in this topic.

LIST OF REFERENCES

1. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network flows: Theory, algorithms and applications,* Prentice Hall, NJ, 1993.

2. M. Aigner , and M. Fromme, “A game of cops and robbers,” *Discrete Applied Mathematics*, 8, pp. 1-12, 1984.

3. R. Albert, and A. L. Barabasi, “Statistical mechanics of complex networks,” *Review of modern physics,* 74, pp. 47-97, 2002.

4. R. Albert, H. Jeong, and A. L. Barabasi, “Diameter of the World Wide Web,” *Nature*, 401, pp. 130-131, 1999.

5. G. Alexanderson, “Euler and Königsberg's bridges: a historical view,” *Bulletin of the American Mathematical Society*, 2006.

6. K. Appel, and W. Haken, “Solution of the Four Color Map Problem,” *Scientific American*, 237, pp. 108-121, 1977.

7. A. L. Barabasi, and R. Albert, “Emergence of Scaling in Random Networks,” *Science*, 286, pp. 509-512, 1999.

8. J. Baumes, M. Goldberg, and M. Magdon-Ismail, “Efficient Identification of Overlapping Communities,” Atlanta, 2005,

9. A. Ben-Dor, R. Shamir, and Z. Yakhini, “Clustering Gene Expression Patterns,” *Journal of Computational Biology*, 6, pp. 281-297, 1999.

10. K. Bharat, A. Broder, J. Dean, and M. R. Henzinger, “A comparison of Techniques to Find Mirrored Hosts on the WWW,” *Journal of the American Society for Information Science*, 51, pp. 1114-1122, 2000.

11. B. Bollobas, *Graph theory: an introductory course,* New York: Springer Verlag, 1979.

12. B. Bollobas, “The diameter of random graphs,” *IEEE transactions on information theory*, 36, pp. 285-288, 1990.

13. B. Bollobas, O. Riordan, J. Spencer, and G. E. Tusnady, “The degree sequence of a scale-free random graph process,” *Random Structures and Algorithms 18(3), 279-290*, 2001.

14. S. Brin, and L. Page, “The anatomy of a large-scale hypertextual Web search engine,” *Computer Networks and ISDN Systems*, 30, pp. 107-117, 1998.

15. S. Brin, and L. Page, “The anatomy of a large-scale hypertextual Web search engine,” *Computer Networks and ISDN Systems*, 30, pp. 107-117, 1998.

16. A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, “Graph Structure in the Web,” International World Wide Web Conference, pp. 33-31, 2000.

17. R. S. Burt, “Positions in networks,” Social Forces, 55, pp. 93-122, 1976.

18. J. D. Burtin, “Extremal metric charecteristics of a random graph,” Theo. Verojatnost. i Primenen, 19, pp. 740-754, 1974.

19. C. Castillo, M. Marin, A. Rodriguez, and R. Baeza-Yates, “Scheduling algorithms for Web crawling,” 2004, pp. 10-17.

20. S. Chakrabarti, M. Joshi, K. Punera, and D. M. Pennock, "The structure of broad topics in the Web," 2002,

21. S. Chakrabarti, B. Dom, R. Agrawal, and P. Raghavan, "Scalable feature selection, classification and signature generation for organizing large text databases into hierarchical topic taxonomies," *The VLDB Journal*, 7, pp. 163-178, 1998.

22. J. Cho, and H. Garcia-Molina, "The Evolution of the Web and Implications for an Incremental Crawler," 2000,

23. J. Cho, H. Garcia-Molina, and L. Page, "Efficient crawling through URL ordering," *Computer Networks and ISDN Systems*, 30, pp. 161-172, 1998.

24. J. Cho, and H. Garcia-Molina, "Parallel crawlers," 2002,

25. F. Chung, and L. Lu, "The diameter of sparse random graphs," *Adv. in Appl. Math.,*, 26, pp. 257-279, 2001.

26. F. R. K. Chung, "Spectral Graph Theory," *Regional conference series in mathematics*, 92, American Mathematics Society, 1996.

27. F. R. K. Chung, and L. Lu, "The average distance in random graphs with given expected degrees," *Proc. Natl. Acad. Sci. 99, pp 15879-15882*, 2002.

28. A. Clauset, M. E. J. Newman, and C. Moore, "Finding community structure in very large networks," *Physical Review E*, 70, pp. 066111-066116, 2004.

29. L. Denoyer, J. N. Vittaut, P. Gallinari, S. Brunessaux, and S. Brunessaux, "Structured multimedia document classification," 2003, pp. 153-160.

30. N. Deo, and Z. Nikoloski, “The game of cops and robbers on graphs: a model for quarantining cyber attacks,” *Congressus Numerantium*, 162, pp. 193-215, 2003.

31. Deo. N, *Graph theory with applications to engineering and computer science,* Prentice-Hall, 2002.

32. S. N. Dorogovtsev, and J. F. F. Mendes, “Evolution of Networks,” *Advances in Physics*, 51, pp. 1079-1187, 2002.

33. J. Edwards, K. S. McCurley, and J. A. Tomlin, “An adaptive model for optimizing performance of an incremental web crawler,” 2001, pp. 106-113.

34. P. Erdos, and A. Renyi, “On Random Graphs I,” *Publicationes Matematicae Debrecen 5, 290-297*, 1959.

35. D. Fetterly, M. Manasse, and M. Najork, “Spam, damn spam, and statistics: using statistical analysis to locate spam web pages,” 2004,

36. G. W. Flake, S. Lawrence, and C. L. Giles, “Efficient Identification of Web Communities,” 2000, pp. 150-160.

37. L. R. Ford Jr, and D. R. Fulkerson, “Maximal flow through a network,” *Canadian Journal of Mathematics*, 8, pp. 399-404, 1956.

38. S. Fortunato, V. Latora, and M. Marchiori, “Method to find community structures based on information centrality,” *Physical Review E*, 70, pp. 0561041-05610413, 2004.

39. L. C. Freeman, “A set of measures of similarity based on betweenness,” *Sociometry*, 40, pp. 35-41, 1977.

40. L. C. Freeman, “Centrality in social networks: Conceptual clarification,” *Social Networks*, 1, pp. 215-239, 1979.

41. E. Garfield, I. H. Sher, and R. J. Torpie, *The Use of Citation Data In Writing the History of Science,* Philadelphia: Institute for Scientific Information, 1964.

42. D. Gibson, J. M. Kleinberg, and P. Raghavan, “Inferring Web Communities from Link Topology,” Pittsburgh, PA, 1998, pp. 225-234.

43. E. N. Gilbert, “Random Graphs,” *Annals of Mathematical Statistics 30*, 1959.

44. 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, pp. 7821-7826, 2002.

45. E. J. Glover, G. W. Flake, S. Lawrence, W. P. Birmingham, A. Kruger, C. L. Giles, and D. M. Pennock, “Improving category specific Web search by learning query modifications",” 2001,

46. M. Gray, *Measuring the growth of the web,* 1993.

47. J. W. Grossman, “The evolution of the mathematical research collaboration graph,” *Congressus Numerantium*, 158, pp. 201-212, 2002.

48. R. Guimerà, S. Mossa, A. Turtschi, and L. A. N. Amaral, “The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles,” *Proceedings of the National Academy of Sciences*, 102, pp. 7794-7799, 2005.

49. S. M. Hedetniemi, S. T. Hedetniemi, and P. Kristiansen, “Alliances in Graphs,” *Journal of Combinatorial Mathematics and Combinatorial Computing*, 48, pp. 157-177, 2004.

50. A. Heydon, and M. Najork, “Mercator: A Scalable, Extensible Web Crawler,” *World Wide Web*, 2, pp. 219-229, 1999.

51. I. Jackson, “GNU adns,” *http://www.chiark.greenend.org.uk/~ian/adns/*,

52. I. T. Jolliffe, “Principal Component Analysis,” Springer Verlag, 1986.

53. J. S. Katz, “Scale independent bibliometric indicators,” *Measurement: Interdisciplinary Research and Perspectives*, 3, pp. 24-28, 2005.

54. M. Kessler, “Bibliographic coupling between scientific papers,” *American Documentation*, 14, pp. 10-25, 1963.

55. J. Kleinberg, S. R. Kumar, P. Raghavan, and S. a. T. A. Rajagopalan, “The Web as a Graph: Measurements, Models, and Methods,” 1999,

56. J. M. Kleinberg, “Authoritative Sources in a Hyperlinked environment,” *Journal of the ACM*, 46, pp. 604-632, 1999.

57. S. R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins, “Extracting Large-Scale Knowledge Bases from the Web,” 1999, pp. 639-650.

58. V. Latora, and M. Marchiori, “Efficient Behavior of Small-World Networks,” *Physical Review Letters*, 87, pp. 1987011-1987014, 2001.

59. V. Latora, and M. Marchiori, “A measure of centrality based on network efficiency,” 2004.

60. V. Latora, and M. Marchiori, “Vulnerability and Protection of Critical Infrastructures,” *Physical Review E*, 71, pp. 0151031-0151034, 2005.

61. J. Leskovec, J. Kleinberg, and C. Faloutsos, “Graphs over time: densification laws, shrinking diameters and possible explanations,” Chicago, Illinois, USA, 2005, pp. 177-187.

62. K. Menger, and Z. Kurventheorie, *Fundamenta Mathematicae,* 10, 1927.

63. S. Milgram, “The small world problem,” *Psychology Today*, pp. 60-67, 1967.

64. M. Najork, and J. L. Wiener, “Breadth-first crawling yields high-quality pages,” 2001, pp. 114-118.

65. M. Najork, and A. Heydon, “High-performance Web crawling,” *Technical Report*, SRC-RR-173, 2001.

66. M. E. J. Newman, “Mixing Patterns in Networks,” *Phys. Rev. E, 67, art. no. 026126*, 2003.

67. M. E. J. Newman, “The Structure and Function of Complex Networks,” *SIAM Review, Vol 45, No 2, pp. 167-256*, 2003.

68. M. E. J. Newman, “Fast Algorithm for Detecting Community Structure in Networks,” *Physical Review E*, 69, pp. 066133-1, 2004.

69. M. E. J. Newman, “High school friendship network,” 2004.

70. M. E. J. Newman, and M. Girvan, “Finding and Evaluating Community Structure in Networks,” *Physical Review E*, 69, pp. 026113-1, 2004.

71. M. E. J. Newman, and M. Girvan, “Finding and Evaluating Community Structure in Networks,” *Physical Review E*, 69, pp. 026113-1, 2004.

72. M. E. J. Newman, S. H. Strogatz, and D. J. Watts, “Random Graphs with Arbitrary Degree Distributions and Their Applications,” *Physical Review E*, 64, pp. 026118.1-026118.19, 2001.

73. H. F. Nielsen, “Libwww - the W3C Protocol Library,” *http://www.w3.org/Library/*,

74. C. F. Olson, “Parallel algorithms for hierarchical clustering,” *Parallel Computing*, 21, pp. 1313-1325, 1995.

75. L. Page, S. Brin, R. Motwani, and T. Winograd, “The PageRank citation ranking: Bringing order to the Web,” *Technical Report*, 1998.

76. G. Palla, I. Derenyi, I. Farkas, and T. Vicsek, “Uncovering the overlapping community structure of complex networks in nature and society,” *Nature*, 435, pp. 814-818, 2005.

77. D. M. Pennock, G. W. Flake, S. Lawrence, C. L. Giles, and E. J. Glover, “Winners don't take all: Charecterizing the competition for links on the Web,” 2002,

78. M. R. Quillian, “Semantic memory,” *Semantic Information Processing*, pp. 216-270, 1968.

79. F. Radicchi, C. Castellano, F. Cecconi, V. Loreto, and D. Parisi, “Defining and Identifying Communities in Networks,” *Proceedings of the National Academy of Sciences*, 101, pp. 2658-2663, 2004.

80. A. Ranganathan, and R. H. Campbell, “Advertising in a pervasive computing environment,” *International Workshop on Mobile Commerce*, 2002.

81. S. Redner, “Citation statistics from more than one a century of physical review,” *Technical Report*, 2004.

82. R. Rivest, “RFC 1321 - the MD5 message-digest algorithm,” *MIT Laboratory for Computer Science and RSA Data Security*, 1992.

83. P. M. Roget, *Roget's Thesaurus of English Words and Phrases,* 1879.

84. G. Sabidussi, “The centrality index of a graph,” *Psychometrika*, 31, pp. 581-603, 1966.

85. H. Small, “Co-citation in the scientific literature: A new measure of the relationship between two documents,” *Journal of American Society for Information Science*, 24, pp. 265-269, 1973.

86. M. Sobek, “Google Dance - The Index Update of the Google Search Engine,” *Electronic report*, 2003.

87. Strang. G, *Linear algebra and its applications,* Harcourt Brace Jovanovich, San Diego, 1980.

88. V. Tawde, T. Oates, and E. Glover, “Generating Web Graphs with Embedded Communities,” 2004,

89. M. Toyoda, and M. Kitsuregawa, “Extracting Evolution of Web Communities from a Series of Web Archives,” Nottingham, UK, 2003, pp. 28-37.

90. I. Vragovìc, E. Louis, and A. Dìaz-Guilera, “Efficiency of informational transfer in regular and complex networks,” 2004.

91. S. Wasserman, and K. Faust, *Social Network Analysis,* Cambridge University Press, Cambridge, 1994.

92. D. J. Watts, and S. H. Strogatz, “Collective Dynamics of Small World Networks,” *Nature, 393, (440-442)*, 1998.

93. J. G. White, E. Southgate, J. N. Thomson, and S. Brenner, “The structure of the nervous system of the nematode C. elegans,” *Philosophical Transactions: Biological Sciences*, 314, pp. 1-340, 1986.

94. C. H. Wu, S. J. Horng, and H. R. Tsai, “Efficient Parallel Algorithms for Hierarchical Clustering on Arrays with Reconfigurable Optical Buses,” *Journal of Parallel and Distributed Computing*, 60, pp. 1137-1153, 2000.

95. F. Wu, and B. A. Huberman, “Finding Communities in Linear Time: a Physics Approach,” *The European Physics Journal B*, 38, pp. 331-338, 2004.

96. W. W. Zachary, “An information flow model for conflict and fission in small groups,” *Journal of Anthropological Research*, 33, pp. 452-473, 1977.