SuperBatCatv2.0

Student Name: Charupriya Sharma
Roll Number: 2011037

Student Name: Prateek Mehra
Roll Number: 2011078

Student Name: Utkarsh Gupta
Roll Number: 2011117

IIIT-D-BTech-CSE-2011

BTP report submitted in partial fulfillment of the requirements
for the Degree of B.Tech. in Computer Science & Engineering
on 23rd April, 2014

BTP Track: Research

BTP Advisor
Dr. Srikanta Bedathur
Dr. Rajiv Raman

Indraprastha Institute of Information Technology
New Delhi

Student's Declaration

We hereby declare that the work presented in the report entitled **"SuperBatCat"** submitted by us for the partial fulfillment of the requirements for the degree of *Bachelor of Technology* in *Computer Science & Engineering* at Indraprastha Institute of Information Technology, Delhi, is an authentic record of our work carried out under guidance of **Dr. Srikanta Bedathur and Dr. Rajiv Raman**. Due acknowledgements have been given in the report to all material used. This work has not been submitted anywhere else for the reward of any other degree.

..... **Place & Date: New Delhi, 23rd April 2014**
(Charupriya Sharma)

..... **Place & Date: New Delhi, 23rd April 2014**
(Prateek Mehra)

..... **Place & Date: New Delhi, 23rd April 2014**
(Utkarsh Gupta)

Certificate

This is to certify that the above statement made by the candidate is correct to the best of our knowledge.

..... **Place & Date: New Delhi, 23rd April 2014**
(Dr. Srikanta Bedathur)

..... **Place & Date: New Delhi, 23rd April 2014**
(Dr. Rajiv Raman)

Abstract

There is a lot of work done in the field of graph visualisation, the field is not new and has been around since a long time. There are a lot of limitations in the state of art systems like that of scalability and smooth rendering. We introduce a new method for community detection within graphs and present a new tool for visualisation and exploring of graphs. The tool has been tested for graphs upto 70k nodes and has been found to render graphs smoothly. We have also introduced a heuristic for calculating shortest path in between communities.

Keywords: Large Graphs, Visualisation, EigenVectors, EigenValues, Community Detection, Shortest Path

Acknowledgments

We would like to thank Dr. Bedathur and Dr. Raman for giving us such an interesting problem to work with. We would especially like to thank Dr. Raman who joined the team at a later stage but his guidance has been invaluable.

We would also like to thank Dr. Marc Plantevit, Associate Professor, University of Lyon for providing us with the DBLP dataset.

Our thanks also goes to the authors of SNAP, TinkerPop, Neo4j, Metis and iGraph.

Contents

Chapter 1

Introduction

Data can be represented as a set of entities and relations between them. This enables us to represent everything as graphs, since they can be directed or undirected, connected or complete, therefore they are a powerful tool to model data. Study of graphs involves study of network features. One such feature is community structure i.e. grouping of vertices with high density of intra community edges. Another is the small world effect [10], which is the name given to the finding that the average distance between vertices in a network is short.

A graph visualisation tool can help in exploiting these features effectively in real time to ensure exploration of the network structure in a more meaningful manner than statistical measures like clustering coefficients, diameter and degree distribution.

1.1 Problem Statement

Currently in its development stage, SuperBatCat is an application which aims at visualising large graphs by exploiting the natural community structure of real world data.

We also aim at creating a heuristic for finding shortest paths from one community to another which as explained later can get very inefficient computationally by naive Dijkstra.

1.2 Approach taken

Since the major motivation of the project is to visualise large graphs, we face major issues regarding hardware limitations of modern day systems regarding the memory. To overcome this problem, we hierarchically cluster the graph so that we have fewer number of nodes to visualize at each level of clustering. For clustering, we ran community detection algorithms so that the

clustering that we get is meaningful, so that even the community structure can be exploited and the graph can be represented in a more meaningful way.

Also, we have defined a heuristic for finding out the shortest path across different communities.

1.2.1 Network Structure

Real world like the World Wide Web links networks are often scale free [1], i.e. characterised by power law distribution of vertex degrees, high clustering coefficient and short average path lengths. The performance of a graph visualisation tool can be improved significantly by exploiting these features effectively. A good visualisation tool should have user interaction to enable an efficient exploration process.

Our solution is to implement a community detection algorithm that has a recursive nature. Communities detected at each level would have decreasing abstraction as we go down the hierarchy. A simple example is a graph studying interactions between people in the academic world. At the top level, we would expect universities. In a university, communities could consist of departments, which in turn could consist of a set of interacting research groups.

1.2.2 Shortest Path Queries

Hierarchical community detection can be exploited more effectively by shortest path queries that account for structure of the graph. In our academia example, one might want to know the best way to reach the computer science department of MIT from IIIT-Delhi. This basically means a query across the levels of our clustering tree hierarchy.

Since we are clustering the data, we can exploit that to compute our shortest path faster. Our system should implement a heuristic that can account for structural properties of the graph to compute the shortest path between nodes and clusters.

Chapter 2

Related Work

There has been extensive work done in community detection and graph visualisation of graphs. In this section we review some of these existing methods and discuss the ways in which these approaches may fail, before describing our own system, which avoids some of the shortcomings of the these techniques.

2.1 Traditional Techniques in Community Detection

2.1.1 SNAP

Stanford Network Analysis Platform (SNAP) is a general purpose network analysis and graph mining library. We used SNAP because the documentation claimed that it scaled well for millions of nodes.

SNAP had two applications for community detection : community and bigclam

1. **Community:**
   community gives an option of either of the two network community detection algorithms: Girvan-Newman and Clauset-Newman-Moore

   (a) Girvan-Newman
   This algorithm [8] constructs communities by progressively removing edges from the original graph. It focuses on those edges which are least central, the edges which are most "between" communities.
   Edge betweenness of an edge is defined as the number of shortest paths between pairs of vertices that use it. Communities should ideally be connected loosely by a few

edges, and shortest paths between communities should use these edges. Hence, the algorithm works by removing edges of high betweenness.

The drawbacks to using this algorithm was its high complexity of $O(n^3)$ and space requirements. For our Enron graph, the order was prohibitive.

(b) Clauset-Newman-Moore
This algorithm [4] works by optimizing a modularity function, Q. Modularity is the ratio of the number of edges within each community to the number of edges between each community, minus the same ratio from a random network.

The operation of the algorithm involves finding the changes in Q that would result from the amalgamation of each pair of communities, choosing the largest of them, and performing the corresponding amalgamation.

The time complexity of this algorithm is $O(m \log^2 n)$, where m is the number of edges in the network. While this is a significant improvement from Girvan-Newman, this algorithm, the massive space requirements of the SNAP implementation made it difficult to use for large graphs.

2. **bigclam:**
This program formulates community detection problems into non-negative matrix factorization and discovers community membership factors of nodes by maximum likelihood estimation [13].
While this program was specifically designed for large networks, it failed to execute on the Enron corpus on a 4GB RAM. The makers of SNAP recommended at least 16 B of RAM for Enron, and hence we could not use bigclam for our system if we had to scale it for graphs with millions of nodes.

2.1.2 METIS

Another algorithm that we spent time on was METIS [9]from Karypsis Labs. The algorithm took as input k the number of partitions the graph is to be divided in. The algorithm is extremely fast and the implementation quite good, we were able to partition the Wiki Talk corpus into parts in less than a few seconds. But since the algorithm took as input the number of communities, the natural community structure of the graph was lost. Thus, after trying to attempt to play with METIS, we had to move away from METIS.

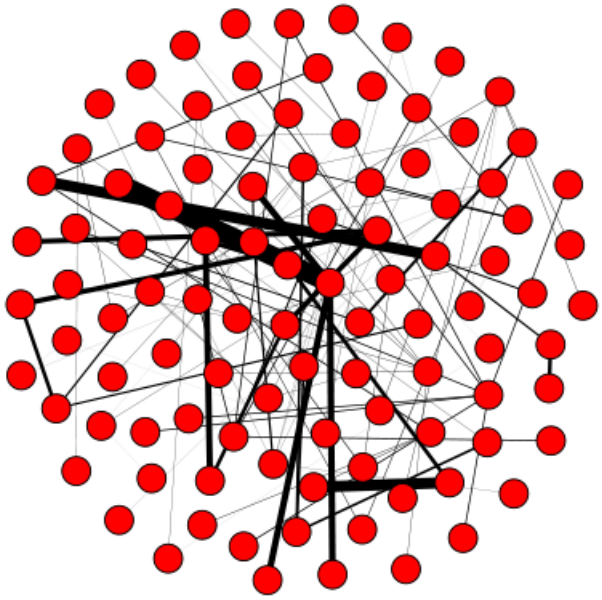

Figure 2.1: METIS Visualsation Top Level for Enron

2.1.3 iGraph

iGraph [5] is an R library that provides methods for creating and manipulating graphs. It has implementations for classic graph theory problems like minimum spanning trees and network flow, and also implements algorithms for network analysis. Community detection in Igraph can be done by a number of different algorithms.

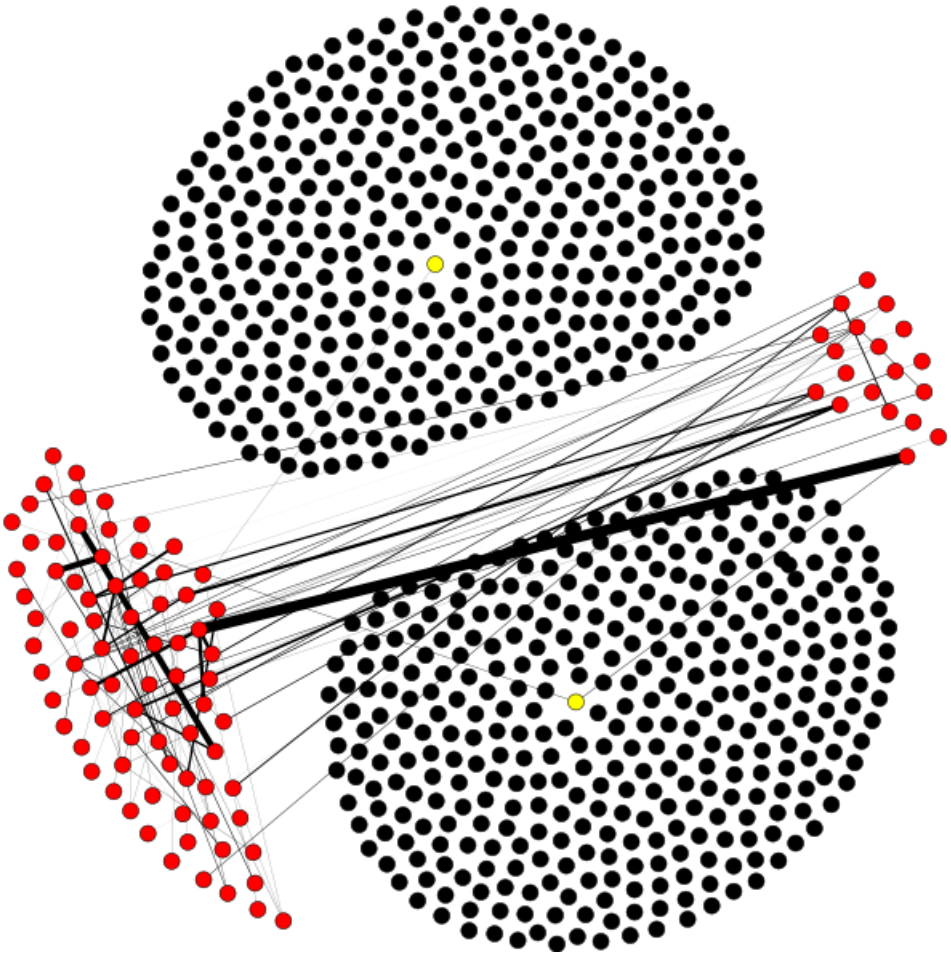

Figure 2.2: METIS Visualsation with Open Clusters for Enron

1. **Leading Eigenvector:**
   This algorithm [11] also focuses on modularity. It recursively divides a graph into two communites while the modularity can be maximized. The modularity Q is defined as
   $Q = \frac{1}{4m} \sum_{ij} [A_{ij} - Pij] s_i s_j$

s_i is 1 if vertex i belongs to community 1, and -1 if it belongs to community 2

P is the adjaceny matrix of a random graph with the number of nodes equal to the number of nodes in our input graph. The edges are random, but subject to the constraint
$\sum_j P_{ij} = k_i$
where k_i is degree of vertex i.

Hence
$\sum_{ij} P_{ij} = \sum_{ij} A_{ij} = 2m$
Where A is the adjacency matrix, m is the number of edges in the graph.

The algorithm defines a modularity matrix B such that
$B_{ij} = A_{ij} - P_{ij}$

By definition of P
$\sum_j B_{ij} = \sum_j P_{ij} - \sum_j A_{ij} = k_i - k_i = 0$

$Vector(1, 1, 1,)$ is always an eigenvector of B with eigenvalue zero. In practice, eigenvalues can either be positive or negative. This observation is used to deduce the community structure of the graph

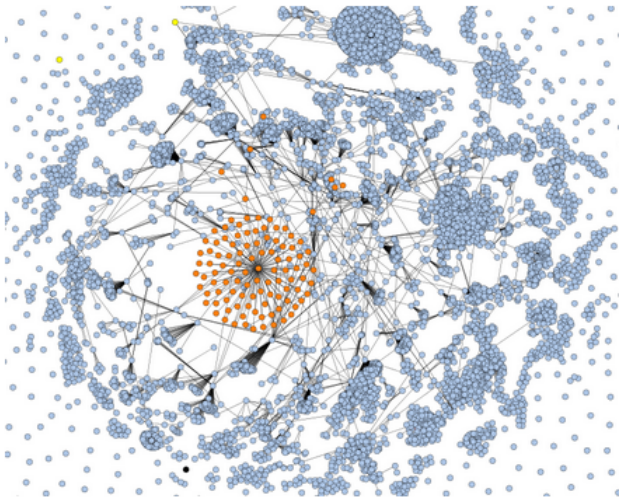

Figure 2.3: Community Detection in Enron with Leading EigenVector from Igraph

We write s as the linear combination of the normalized eigenvectors u_i of B

$$s = \sum_{n}^{i=1} a_i u_i$$

$$a_i = u_i^T s$$

The modularity Q now becomes

$$Q = \frac{1}{4m} \sum_i a_i^2 \beta_i$$

where β_i is the eigenvalue of B corresponding to the eigenvector u_i. Eigenvalues are assumed to be labeled in decreasing order $\beta_1 \leq \beta_2 \leq ... \leq \beta_n$ and the task of maximizing Q is one of choosing the quantities a_i^2 so as to place as much as possible of the weight in Q in the terms corresponding to the largest eigenvalues.

For this purpose, s is chosen to be parallel to u

$$s_i = \begin{cases} 1 & u_i^{(2)} \geq 0 \\ -1 & u_i^{(2)} \leq 0 \end{cases}$$

The algorithm basically finds the eigenvector corresponding to the most positive eigenvalue of the modularity matrix and divides the network into two communities according to the signs of the elements of this vector.

The drawbacks to using this approach was simply that a community given by this algorithm could be a disconnected one. For the purposes of shortest path features in our system, allowing disconnected communities would leave the shortest path open to a lot of errors. Hence, this algorithm could not be used.

2. **Edge-Betweenness:**
   This is an implementation of the Girvan-Newman [8] algorithm discussed in 1a

3. **Fast Greedy:**
   This is an implementation of the Clauset-Newman-Moore [4] algorithm discussed in 1b

2.2 Visualisations

In this section we discuss the graph visualisation techniques that already exist. We see what they can do, and what their limitations are and how our system can overcome these.

2.2.1 Node XL

NodeXL [12] is a graph visualisation tool by Microsoft. This works with Microsoft Excel which supports various graph formats like network edge list. They have the following features as said on their website:

1. **Flexible import and export:** Can import and export graphs in GraphML, Pajek, UCINet, and matrix formats.

   SuperBatCat imports the data from Neo4j [6] or potentially any graph database to give it as input to our system.

2. **Direct Connections to Social Networks:** Import social networks directly from Twitter, YouTube, Flickr and email, or use one of several available plug-ins to get networks from Facebook, Exchange, Wikis and WWW hyperlinks.

   Again, SuperBatCat retreives its input from any graph database as input.

3. **Zoom and Scale:** Can zoom into areas of interest, and scale the graph's vertices to reduce clutter.

   Along with zoom, SuperBatCat allows the user to pan the graph so that the graph can be easily moved around for a better experience.

4. **Flexible Layout:** Can use one of several "force-directed" algorithms to lay out the graph, or drag vertices around with the mouse. Have NodeXL move all of the graph's smaller connected components to the bottom of the graph to focus on what's important.

   SuperBatCat also uses force-directed algorithms to layout the graph. The placement of graph entities is one areas which we need more work on and we will improve it in the future versions of the system.

5. **Easily Adjusted Appearance:** Can set the color, shape, size, label, and opacity of individual vertices based on vertex attributes such as degree, betweenness centrality or PageRank.

   SuperBatCat uses pre-defined colors, shapes etc. This is one thing that can improve when the whole system is in place.

6. **Dynamic Filtering:**Can instantly hide vertices and edges using a set of slidershide all vertices with degree less than five, for example.

   SuperBatCat does not have a feature like this right now, but can be eaily incorporated in future versions.

7. **Powerful Vertex Grouping:** Can group the graph's vertices by common attributes, calculate their connectedness and automatically group them into clusters.

   SuperBatCat performs community detection with an algorithm that we ourseleves created. It supports exploring communities by opening communities and closing them.

8. **Graph Metric Calculations:** Can calculate degree, betweenness centrality, closeness centrality, eigenvector centrality, PageRank, clustering coefficient, graph density and more.

   SuperBatCat does not support this right now, but can be easily incorporated.

2.2.2 Gephi

Gephi [7] is a software for visualizing and analyzing large networks. It has a large number of available layout algorithms as well as functionality to calculate degree, clustering coefficient, diameter etc. On input of the Enron graph, Gephi failed to settle the layout after 3 hours of execution. Moreover, there is no mechanism for community detection in Gephi. Even for graphs of a few hundred nodes, it is very difficult to explore the structure of the network. The main strength of Gephi is in statistical analysis, which our system does not provide currently, but due to integration of Neo4j, providing these features should be a relatively simple task.

Chapter 3

Algorithm

3.1 Community Detection with Influential Nodes

This algorithm is inspired from the fact that real world communities often form around a set of influential nodes (nodes of high degree). The community detection algorithm approximates this influential set by filtering a set of nodes with a user defined threshold degree (during testing, this was taken to be half of the maximum degree). This set was removed, and the result was a set of connected components, each that had some edges to the set of large degree nodes. These were taken as communities.

Algorithm 1: Community Detection

input : Graph, G(V,E) in edgelist format, and $\alpha \in [0, 1]$

output: A hierarchical community detection on nodes of the graph

1. Remove the nodes with degree $> k = \alpha * max(degree(G))$;
2. The graph splits into connected components. Each connected component is a natural cluster. Collapse each connected components into super-nodes;
3. This gives us a new graph H - which consists of the large degree nodes and the super-nodes;
4. If any super-nodes has degree 1, collapse it into its large degree neighbour;
5. Recursively apply algorithm to each connected component ;

3.1.1 Recurrence Relation

$$T(n) \leq \sum_{i=1}^{k} T(n_i) + O(n^2)$$

Here, k is the number of large clusters, and n_i is the number of nodes in the i^{th} cluster. The cost of $O(n^2)$ comes from an implementation of Union-Find data structure for computing the connected components. Our current implementation uses BFS to compute the connected component, but we plan to optimize this in future work.

3.1.2 Terminating Condition

The algorithm returns when either of the two following conditions hold true :-

1. When only one cluster was detected
2. When a graph has only one node

To prove that the algorithm will always terminate, we need to show that these conditions can always be reached.
If at step 2, removal of large degree nodes results in deletion of the graph (in cases of k-connected graphs), then we terminate under the first condition. Otherwise, we have atleast one connected component to proceed to step 3. If at step 4, all supernodes are absorbed into the set of large degree nodes, we have only one cluster, and we terminate under the first condition. Otherwise, we have at least two clusters, each with an associated induced subgraph.

3.2 Randomized BFS

Algorithm 2: Randomized BFS

input : Subgraph G(V,E) of top level of clustering given by Algorithm 1, BFS cutoff threshold, α

output: community detection on nodes of the graph

```
1 Compute degree distribution of G;
2 while there is a node not assigned to a community do
3 |   Sample a node with probability proportional to its degree;
4 |   Run BFS from that node until depth β ;
5 |   All nodes in BFS are assigned to a new committee ;
6 end
```

3.2.1 Determination of BFS Cutoff Threshold

We used the multiple community modularity function [11]
A matrix $S = (s_1|s_2|...|s_c)$ of size $n \times c$ is taken, where c is the number of communities. Each column is a binary vector

$$S_{ij} = \begin{cases} 1 & \text{vertex i belongs to community j} \\ 0 & \text{otherwise} \end{cases}$$

The columns of S are mutually orthogonal, and the rows each sum to unity, and that the matrix satisfies the normalization condition $Tr(S^T S) = n$

The modularity now becomes
$Q = \sum_{i,j=1}^{n} \sum_{k=1}^{c} Tr(S^T B S)$

We run experiments to determine the threshold that has the maximum modularity. Note that use of Algorithm 2 is necessary only when the top level of clustering has a large number of clusters. Neither DBLP nor Enron exceeded 70 clusters at top level. Use of this algorithm to find community structure in the original graph was also explored. The threshold was tested for numbers close to half of the diameter of the graph. We used the largest connected component in Enron corpus for testing. This had a diameter of 13. However, this did not result in significant community structure detection, and hence is now used only on only the top level of clustering (if required) to make the visualisation more compact.

Chapter 4

Experiments

4.1 Data Sets

Everyday, tons of data is produced, Google, Facebook, Wikipedia add hundreds of users to their database, thus making their databases larger and their relationship graphs more complex. Since, naturally all data can be modelled as a type of graph, the problem becomes more interesting and the solution, more usable. This also means that studying the community structure in these type of graphs becomes a much more interesting problem and visualising them becomes much harder. Though our ultimate aim is to be able to visualise the Wikipedia Article relationship graph (Wiki Talk) which has around 23,94,385 nodes and around 50,21,410 edges, we started off with something smaller the Enron-Email dataset and the DBLP graph. We assume that all datasets that we use are undirected graphs as most of the community detection algorithms do not work for directed graphs.

4.1.1 Enron-Email

The Enron-Email is the email communication network between the employees of Enron Corporation which was an US based energy, commodities and services based company. The data set has 36,692 nodes and 1,83,831 edges. The dataset like all other datasets is undirected. The size of largest connected component is 33,696 nodes and 1,80,810 edges. The degree distribution of the graph as shown in 4.1 follows power law distribution, which is true for most of the real world data.

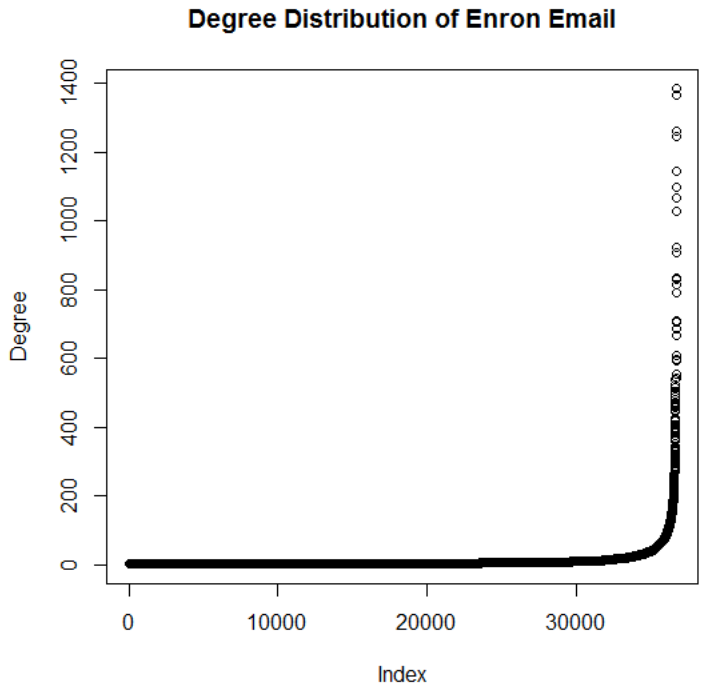

Figure 4.1: Enron Degree Distribution

4.1.2 DBLP

We also used the DBLP co-authorship graph constructed by Marc Plantevit, University of Lyon. This dataset provides the co-authorship network of authors (as nodes) who published papers in at least one major conference or a journal pertaining to Data Mining and Databases during the period January 1990 to Febraury 2011. The edges, as the name suggests represents collaborations. The dataset had information about 42,252 authors and had 21,03,20 edges. The largest connected component of the same has 39,767 authors and 20,8767 edges. The degree distribution looks like the one shown in 4.2.

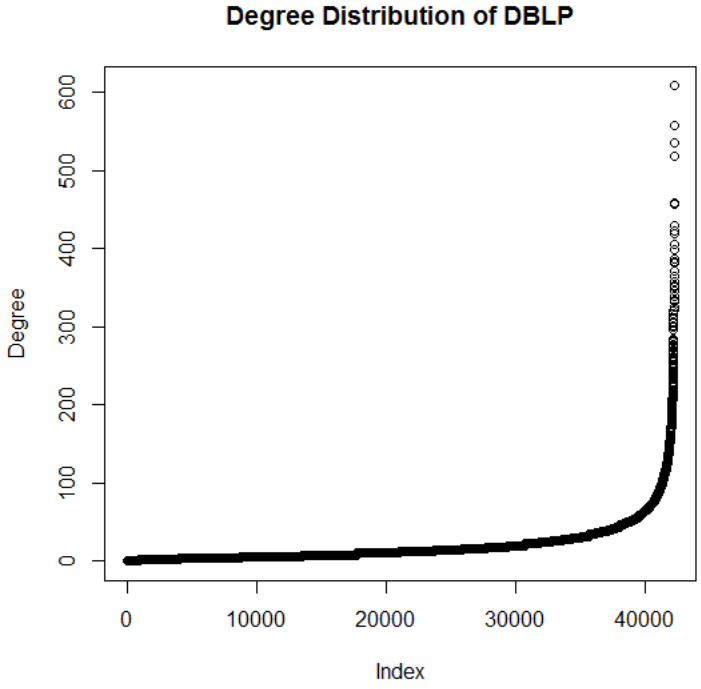

Figure 4.2: DBLP Degree Distribution

4.1.3 Wiki Talk

Wiki Talk, is a graph which is the communication network about the updation of various Wikipedia articles of the registered Wikipedia users. The dataset contains all communication data starting from the inception of Wikipedia till January 2008. The number of nodes is 23,94,385 and the number of edges is 50,21,410. Running community detection algorithm and visualising this dataset is one of the final outcomes of our B.Tech. Project.

4.1.4 Others

We also ran the visualisation and the community detection algorithms on various small datasets like the Dr. Who dataset (1,059 nodes, 2,285 edges), SlashDot (77,360 nodes, 9,05,468 edges), WikiVote (7,115 nodes, 1,03,689 edges). We also generated some random Kronecker graphs of varying sizes. We experimented with graphs of sizes 2^{13}, 2^{14} and 2^{15}.

4.2 Experiments on Community Detection

4.2.1 Largest Connected Component in Enron

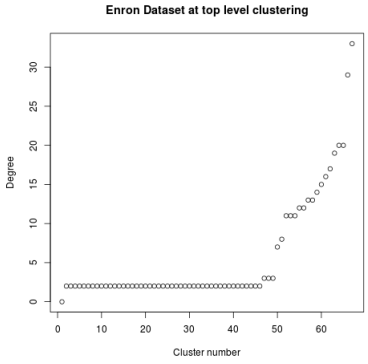

Figure 4.3: Community Detection in Enron at Top Level

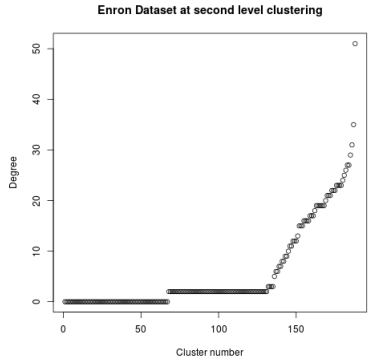

Figure 4.4: Community Detection in Enron at Depth 2

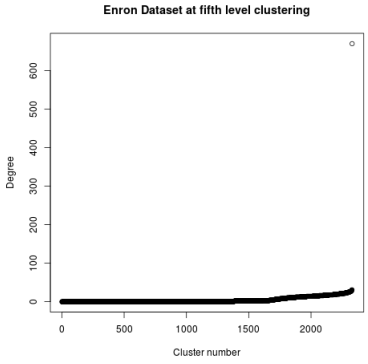

Figure 4.5: Community Detection in Enron at Depth 5

The algorithm ran for a maximum recursion depth of 11 in 25 minutes on a machine with 4 GB RAM.

4.2.2 Largest Connected Component in DBLP

The algorithm ran for a maximum recursion depth of 12 in 30 minutes on a machine with 4 GB RAM.

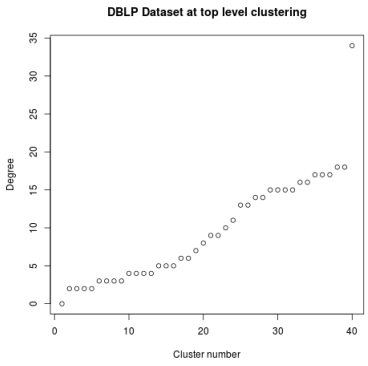

Figure 4.6: Community Detection in DBLP at Top Level

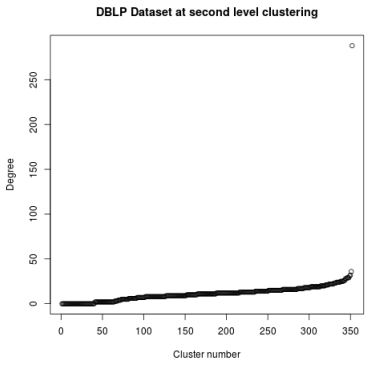

Figure 4.7: Community Detection in DBLP at Depth 2

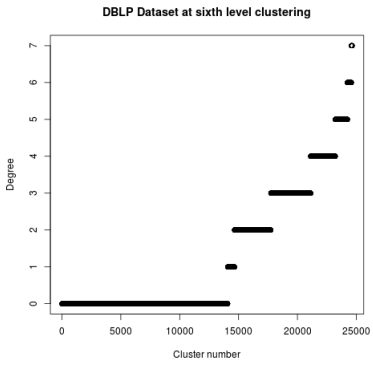

Figure 4.8: Community Detection in DBLP at Depth 6

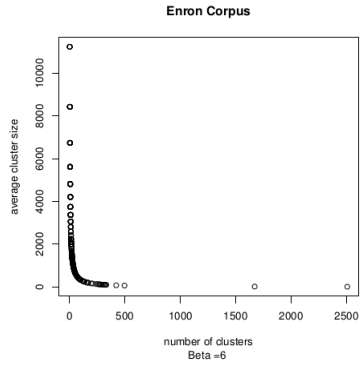

Figure 4.9: Biased Sampling for Beta=6

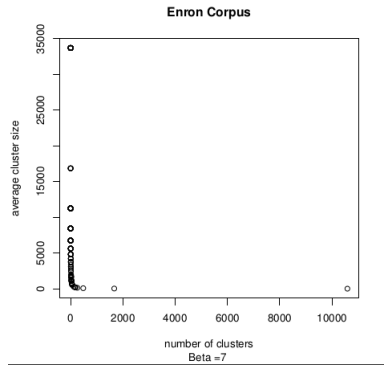

Figure 4.10: Biased Sampling for Beta=7

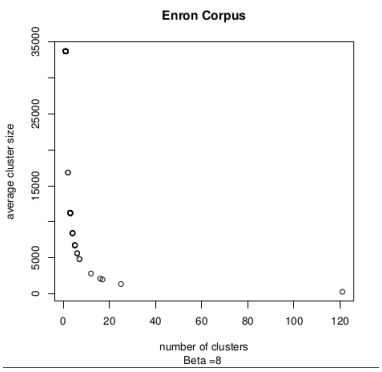

Figure 4.11: Biased Sampling for Beta=8

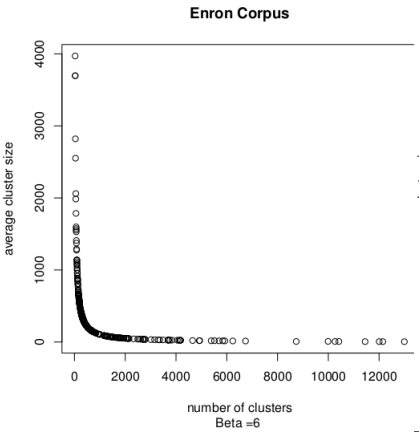

Figure 4.12: Uniform at Random Sampling for Beta=6

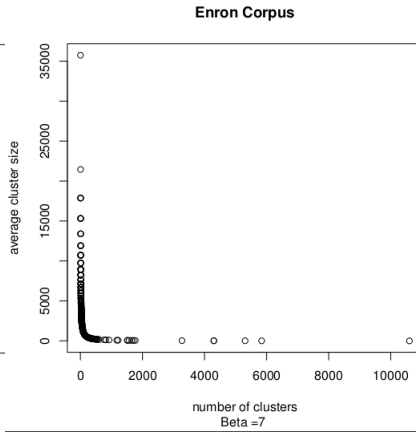

Figure 4.13: Uniform at Random Sampling for Beta=7

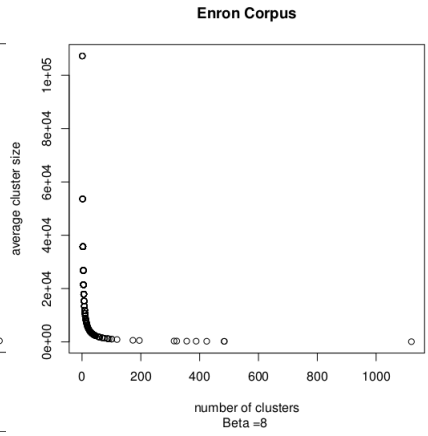

Figure 4.14: Uniform at Random Sampling for Beta=8

4.3 Experiments with Randomized BFS

4.3.1 Largest Connected Component in Enron

We sampled nodes for BFS in both random at uniform and randomly proportional to the degree. The graph had a diameter of 13. At the threshold of 6, the largest cluster had almost all of the nodes in biased sampling. This trend became visible in uniform at random sampling at a threshold of 7. These graphs were plotted for data obtained in more than 400 runs of the algorithm for each sampling method.

Chapter 5

Visualisation

Since we are working to visualize very large datasets, we needed the simplest engines to render nodes and edges of the graph. Existing visualization tool visualize attributed graphs very well but none of them are aimed at achieving scalability, which was our primary motive. To suit our requirements we not only needed this but also required that the visualization could space out nodes and place them to bring out the community structure hidden underneath.

We developed a web browser based tool that used the force directed layout in D3.js to utilize the browser's display engines and visualize graphs as a set of hierarchical clusters, which could be clicked on to expand into connected clusters of successively lower depths, until the lowest level in the hierarchy (original nodes in the graph dataset) is reached.

D3.js [3] is a JavaScript library for manipulating documents based on data. D3 uses HTML, SVG and CSS to visualize data. D3s emphasis on web standards gives you the full capabilities of modern browsers without tying yourself to a proprietary framework, combining powerful visualization components and a data-driven approach to DOM manipulation. We used D3.js because it's extremely scalable, can be extended and customized by making use of the parameters and is extremely well documented .

D3.js uses a flexible force-directed graph layout implementation using position Verlet integration to allow simple constraints and uses a quad tree to accelerate charge interaction using the Barnes-Hut [2] approximation. Apart from the repulsive charge force, the force directed layout implementation also uses a pseudo-gravity force to keep nodes centered in the visual area and avoid expulsion of disconnected subgraphs. For our visualization, nodes are mapped to SVG

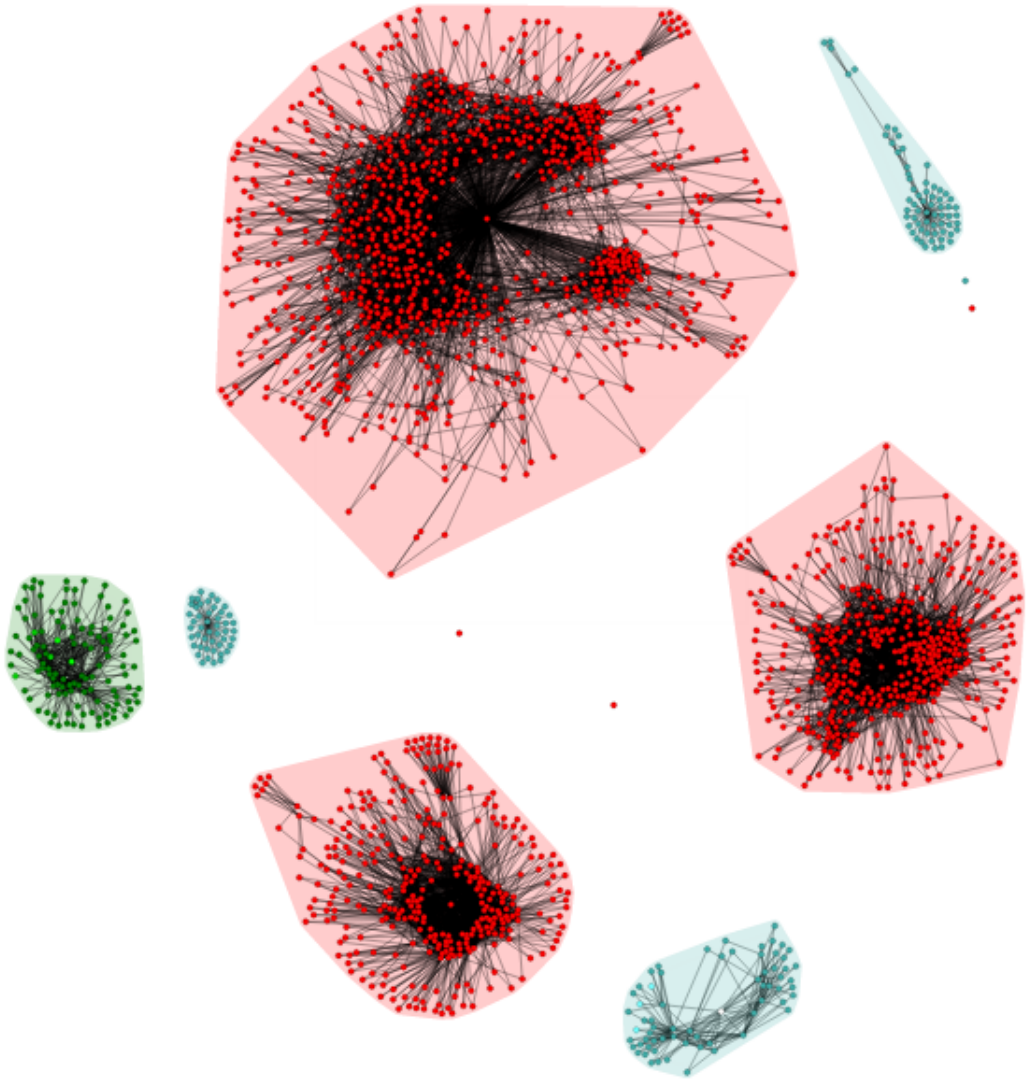

Figure 5.1: Clusters in Enron

circles and links to SVG lines.

The force direction layout allows one to modify a set of parameters to improve the visualization. We modified the following parameters to improve our visualization:

force.**size**
It sets the size of the 2 dimensional SVG layout that would be used to display the nodes and edges. The default size is 1X1 It takes as argument a two-element array of numbers describing x and y.

force.**linkDistance**
It sets the target distance between linked nodes to the specified number. The default value is 20. The argument can also be a function evaluating for each link and its index being passed. The distance is typically specified in pixels.

Links are not implemented as spring-forces but weak geometric constraints. The method of constraints relaxation on top of Verlet integration allows for more flexible implementation of other constraints such as hierarchical layering and proves to be much more stable than methods using spring forces.

force.**friction**
It sets the friction coefficient to the specified value, which is not the coefficient of friction in standard physics but more closely approximates velocity decay at each tick of the simulation, i.e, at every instance when the visualization is refreshed after being called for update, the velocity of the particles scales down by a factor which is specified by friction. The default value is 0.9 with the range [0,1]. Thus, a value of 1 corresponds to a frictionless environment and 0 freezes all particles in place.

force.**charge**
It sets the charge for each node in the layout which decides the repulsive force that this node exerts towards other nodes. The default value is -30. A negative charge implies repulsion while a positive charge implies attraction. A function in the nodes and their index can also be passed to set different value of charge per node. Setting charge to 0 exempts the quad tree computation which noticeably improves performance if one doesn't need n-body forces.

force.**gravity**
It sets the gravitational strength. Gravity is implemented as weak geometric constraint similar

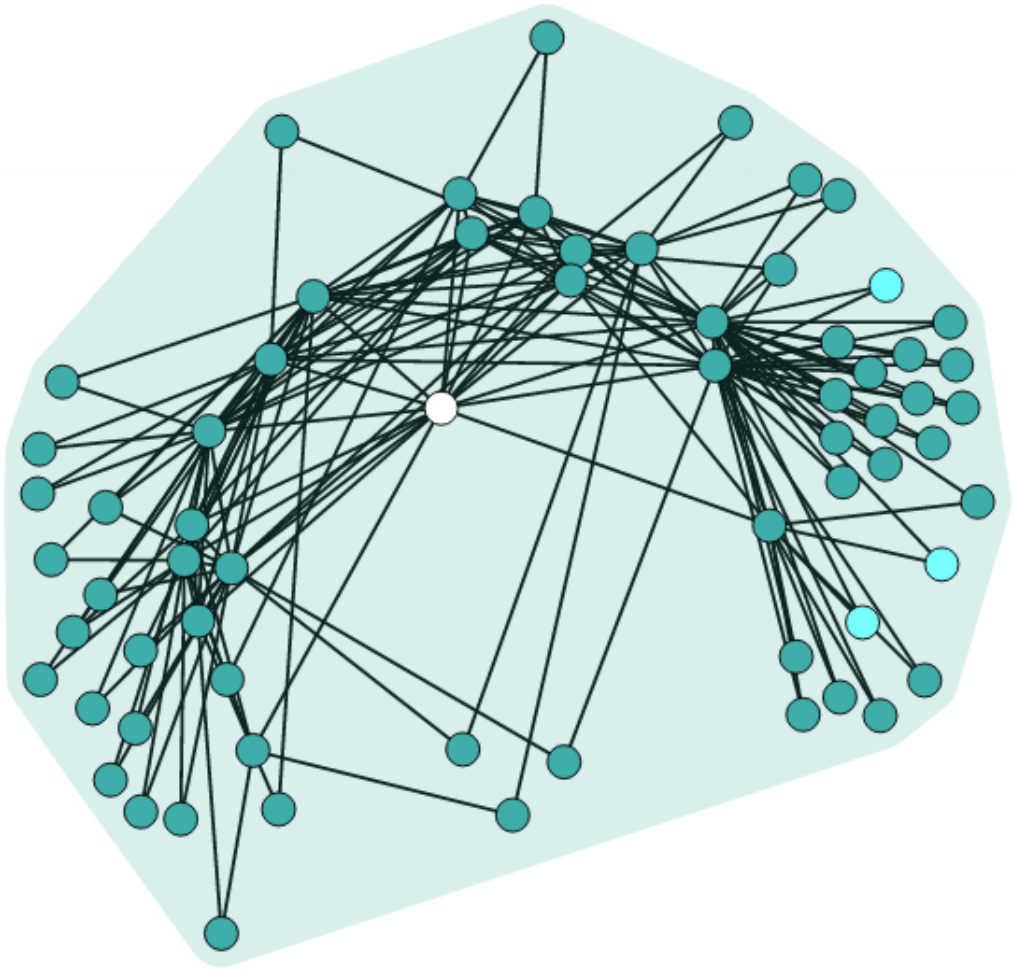

Figure 5.2: Top Level of Communities in Enron

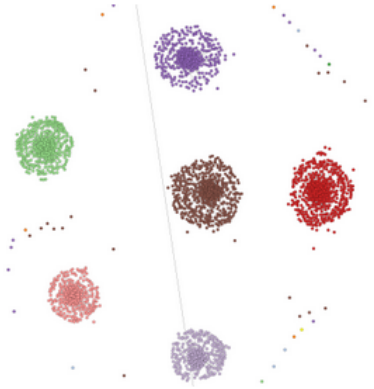

Figure 5.3: Community Detection in Wiki-Vote with EigenVector from Igraph

to virtual spring connecting each node to the centre of the layout's size. Gravitation strength is linearly proportional to the distance from the centre and is 0 at the centre, thus avoiding distortion of the layout.

Other parameters that can be set in the layout are **theta**, **chargeDistance**, **linkStrength** and **alpha**.

The best part is that the visualization computes the best possible placement for each of the nodes to give least overlapping and crossing nodes and edges respectively. The visualization is zoomable and can be panned to see overflowing nodes and links, making it easier to see through overlapping due to overcrowding. The zoomable nature of the visualization is achieved by the utilizing the SVG properties that associate an SVG object with each node and edge and redraws it every time the zoom level is changed. This is the simplest it could get.

Clusters are color coded according to hierarchy, i.e. all of the children of a cluster are given the same color. Labels appear on mouseover, and they indicate unique identifiers for the nodes and the for links they indicate the source and target node identifiers.

Our visualisation has also explored the use of convex hulls for limiting scattering of nodes in a cluster. Hulls bind a cluster to a highlighted space. This helps in better identification of structure in a cluster. The hulls also highlight the lowest abstraction level being displayed at the moment so that it is easier to identify these nodes.

The visualization is also capable of displaying shortest paths over graphs which get highlighted on supplying the source and the target nodes.

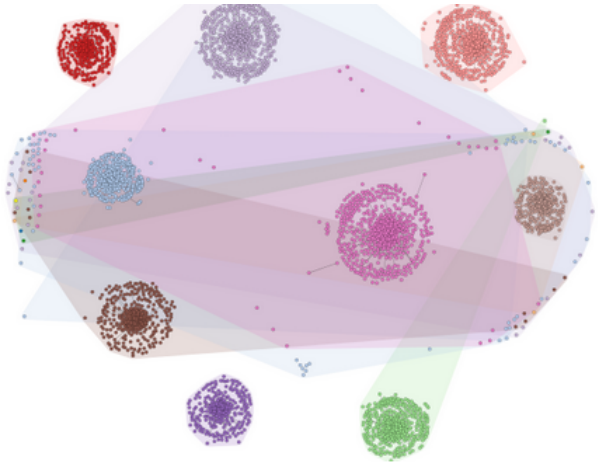

Figure 5.4: Use of Convex Hulls in Visualising Clusters

Chapter 6

Shortest Path

Another important aim of the project was to run some shortest path algorithms on the clustered graph. This would make exploring the graph much easier and in most graphs it is interesting to study the reachability metrics like the shortest path. We started off with implementing Shortest Path. We will illustrate the use of such a feature with an example. Suppose we have a graph which shows us the connection of IIIT-Delhi students with students at GaTech. Now, we have detected the communities in this graph, that means that we have separated the theory group of IIIT-Delhi and the theory group of GaTech. We want to find out how one can reach from the theory group of IIIT-Delhi to that of GaTech, so could just compute the Shortest Path considering the source to be the theory group community of IIIT-Delhi and the target to be the theory group community of GaTech. This is just one of the use of the many possible uses of such a feature. Implementing such a feature poses major challenges, and is one of the things because of which we needed to change our algorithm (as explained in the previous chapter) several times.

6.1 Requirements

Before stating anything on how we implemented this, the requirements of the system should be very clear. Since the whole graph can be imagined as a forest of communities, wherein each tree in the forest could have a different depth. Taking an example, the IIIT-Delhi community may say have depth 2, the whole community and the various research groups but the GaTech community can have depth more than 2 which may refer to the specialised groups within each research group and so on.

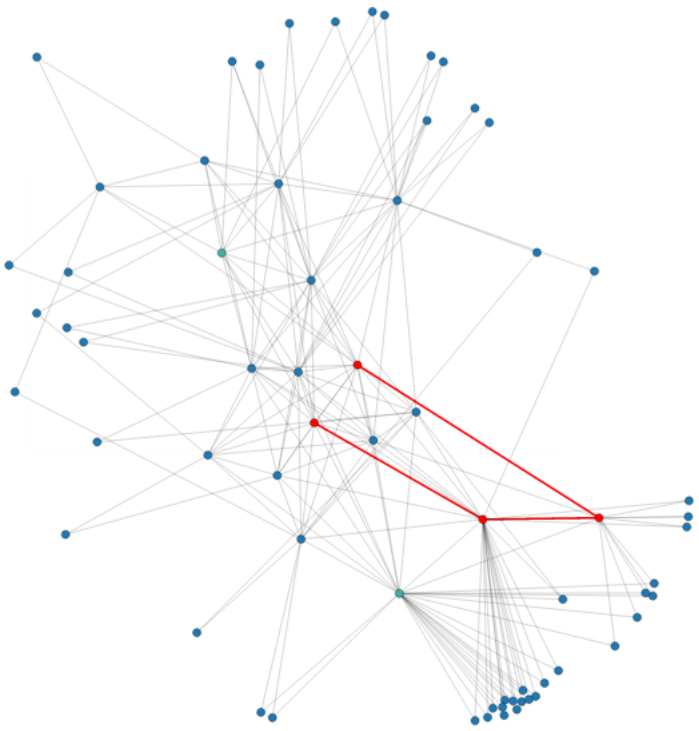

Figure 6.1: Shortest Path in Enron

6.1.1 Compute across levels

First major requirement of the system would be able to compute the shortest path between nodes at different depth, that is compute a path between the theory group of IIIT-Delhi to the algorithms group of GaTech(which would be at a different depth than the source, as it is a specialised group within the theory group of GaTech). It is obvious to see that if such a path exists across depths, then there would be exist a path in the original graph connecting at least one researcher from theory group of IIIT-Delhi to one researcher of the theory group of GaTech. The other way round is also true, and that is also trivial to see.

6.1.2 Efficiency

Second major requirement of the system is to be able to compute these paths very fast and very efficiently. This is because we can not pre-compute the shortest path to all pair of nodes in the community forest. Also, it would be unfair to ask the user to wait for 10 minutes to show her the shortest path while the backend computes the shortest path till then.

6.2 Methodology

We developed a method to compute the shortest path for clustering tress of height 2. This works for the following three scenarios:

1. Source : Cluster at depth n, Target : Cluster at depth n
2. Source : Cluster at the lowest depth, Target : A leaf node in the community tree
3. Source : A leaf node in the community tree, Target : Cluster at the lowest depth

We need to modify the model in a way such that it also works for the following four scenarios:

1. Source : Cluster at depth n, Target : Cluster at depth n-i, where i as a non negative integer such that depth n-i still exists
2. Source : Cluster at depth n-i, Target : Cluster at depth n, where i as a non negative integer such that depth n-i still exists
3. Source : Cluster at level n, Target : A leaf node in the community tree, which will obviously be a leaf node in the community tree.

4. Source : A leaf node in the community tree, Target : Cluster at level n

Though, we hope and suspect that the method we have developed scales for larger depths, this is one of the things that we plan to look in upcoming versions of SuperBatCat. For now let us assume that the model is for trees of depth 2.

6.3 The naive way

For the sake of discussion, let us take the source to be community A and the target as community B. The most naive way of computing the shortest path between A and B would be to compute the shortest path between all nodes of community A and all nodes of community B in the original graph and then abstract the level of the shortest path to the level of the communities. Example, if community A has nodes 1,5,7,9,11,12 and community B has nodes 2,3,4, the naive way would constitute computing all pair shortest path of nodes of community A and community B, which in this case is around 12 pairs. After finding all pair shortest paths, we would choose the path which is shortest amongst these set of paths. Now say, if the path comes something like the following $1 \rightarrow 15 \rightarrow 17 \rightarrow 2$ and node 15 and 17 lie in community D and E respectively, then we can simply say that the shortest path from community A to community B is as follows $A \rightarrow D \rightarrow E \rightarrow B$.

The proof of correctness of the above algorithm falls trivially from the following facts:

1. We are computing the all pair shortest path between nodes of the communities
2. If a path exists between a pair of communities, then a path exists between the underlying nodes as well th
3. If a path exists between a pair of nodes, then a path exists between the communities to which the nodes belong to

The only thing left to prove is that that the path will be the shortest path. Let us suppose not, $\exists$ a path $opt(P)$ such that the $length(opt(P)) < length(P)$ where P is the path returned by our naive algorithm. By fact 2, $\exists$ a path $opt(P)$ in the original graph. Since the graph is assumed to be unweighted, therefore the length of the path in the original graph will be the same as the length in the community graph (because edges are neither removed nor added, we define edges between communities as the union of edges between the nodes of the two communities, and the weight of the edges is one. So, which means $\exists$ an edge between community A and community B iff $\exists$ an edge between any pair of nodes which belong to either community). Since we are taking

the shortest of all the shortest paths available, therefore our algorithm will choose $opt(P)$ as the path. Therefore, a path $opt(P)$ such that the $length(opt(P)) < length(P)$ can not exist. Hence, our algorithm is correct.

Now, having proved that the algorithm works, we had to study the feasibility of the algorithm, because finding all pair shortest paths is not cheap, the Floyd-Warshall algorithm takes $O(n^3)$ to compute the shortest paths, which is way too much for real world data. As the sizes of the clusters increase, the running time of the algorithm also goes up. We tested this method on Enron using Gremlin and Neo4j and it took 12 hours to execute. Thus we had to rule out the algorithm because of the running time and the efficiency constraints.

6.4 A shortest path heuristic

In the previous algorithm, we modelled the inter-community edges as unweighted and undirected. We define a new heuristic which models the inter-community edges as directed and weighted. The weight of an edge going from A to B is defined as:

$$w(A, B) = \frac{size(\text{community A})}{\text{number Of Links Between A and B}}$$

Intuitively, the denominator is basically a measure of connectedness of the two communities, if the communities are closely related, there will be more links joining them, and thus we can exploit this property to define edge weights. Also, it is important to note that we are replacing the undirected (A, B) edge with two edges $A \to B$ and $B \to A$, so the graph is not getting disconnected. Since we have only one level of clustering, there will be clusters and leaf nodes. Computing cluster to cluster shortest path is straight forward with the new definition of the edge-weights, as we are just converting the undirected graph to a directed graph. Now we have edge-weights at the top level, and thus we can simply run with the traditional shortest path algorithms like Djikstra. The interesting scenario is the one where we expand one of the clusters in the path. Example, if the shortest path between communities A and B is $A \to C \to D \to B$ and the user wants to expand community C for clustering. We need to model the shortest path now such that that it incorporates the new nodes that have been introduced due to the expansion of the community. We tackle this problem by treating the new introduced nodes as clusters on the same level. The interesting scenario is the one where we expand one of the clusters in the path. Example, if the shortest path between communities A and B is $A \to C \to D \to B$ and the user wants to expand community C for clustering. We need to model the shortest path now such that that it incorporates the new nodes that have been introduced due to the expansion of the

community. We tackle this problem by treating the new introduced nodes as new communities with only one node, that is they are treated as communities on the same level, and the edge-weights are recomputed so that the changes are incorporated. Another interesting scenario is when a leaf node to a community shortest path is to be computed, example, if we need to find out the shortest path from a person P to the IIIT-Delhi community in the graph. This problem can be solved by abstracting the level of person P, that is find out which community she belongs to. Say she belongs to the GaTech community, thus if we are able to find out the shortest path from the GaTech community to the IIIT-Delhi community, we have found the shortest path from P to the IIIT-Delhi community (the proof of correctness follows from the proof of correctness of 4.3).

6.5 Problems and Limitations

The proofs and algorithms given in this section assume that in one community, every person is reachable from one another. That is inside a community, there would not be more than one connected component. This, in a real world is a very reasonable assumption to make. But most of the community detection algorithms we used did quite the opposite, they put disconnected components in one community, even if we ran them on the largest connected component, we observed several cases when parts of the graph were disconnected and then put in one community. This makes our claim that if a path exists between two communities, then a past must exist between at least two nodes of the two communities false. Consider the following scenario, let there $\exists$ a path $A \rightarrow B \rightarrow C$ and let community B have two connected components $B1$ and $B2$. Now, $\exists$ a path from A to $B1$ and $\exists$ a path from $B2$ to C. This implies, that $\exists$ a path from A to C, which may not be the case in the underlying graph. All nodes of A and all nodes of C may belong to completely different connected components of the graph. This motivated us for start working on a community detection algorithm of our own. We came up with two algorithms which have been described in the previous section.

Another limitation of the model is that that this has been tested and verified on only one level of clustering, that is there is no hierarchical clustering. We need to develop the model for arbitrary source and target, as explained earlier.

Chapter 7

Architecture

The architecture of SuperBatCat is very simple. We have assumed that the graph has been stored on Neo4j (or could be any other graph database for that matter) and we take out the graph using Gremlin scripts to store them in edgelist format. After the edgelist has been extracted, or has been fed to the system, we extract the largest connected component from the whole graph and then run our community detection algorithm on the largest connected component. After the data has been put into different communities, we convert the data into a json which is fed to the visualisation. The following shows the block diagram of the architecture of the system.

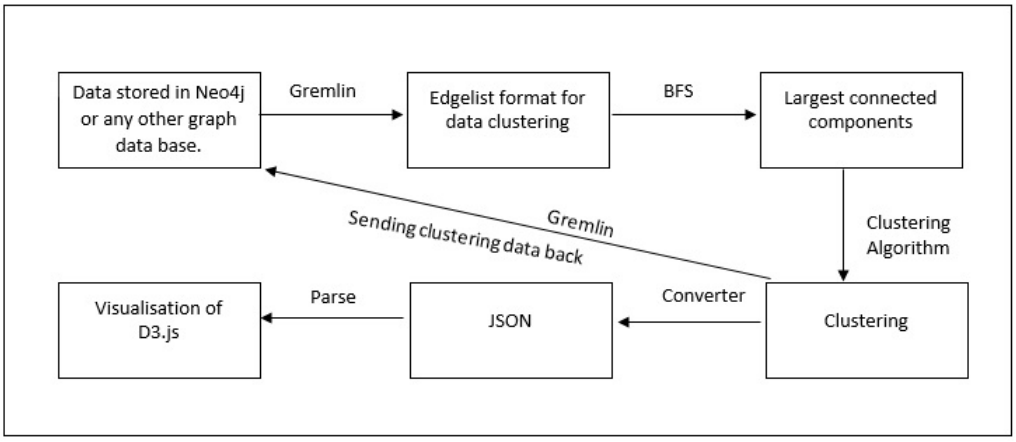

Figure 7.1: Block diagram of the system

7.1 Backend

We stored our graphs in Neo4j [6], which is a NOSQL, ACID supported, highly scalable graph database. It is one of the most popular graph databases and is very efficient. Neo4j allows support for many graph traversal languages like Gremlin, which is open source . We chose Gremlin to query the database as its use is not restricted to just Neo4j, and we could have the flexibility of switching to another database in the future without major changes to our code. We used Neo4j as our major database, we read our graph from neo4j and then converted it to an edgelist using a Gremlin script. This edgelist was used in community detection.

7.2 Implementation of the algorithm

We implemented the whole algorithm in Java using hashmaps for adjacency lists and for storing the edges removed. Though HashMap gives us $O(1)$ access we can do a better job implementing it using union find data structure which we plan to do before proceeding with our future plans.

Chapter 8

Future Work and Extension

We have done some work regarding the existing community detection algorithms, their short comings and why they were not useful for us. We have developed and implemented a new community detection algorithm, the implementation is a bit slow right now and there is much scope for improvement, we could use the union find datastructure to run the algorithm in one parse over the edges.

Even for the visualisation, though it works good for now, it could be made much smoother and other layouts could be experimented with (right now we use the force-directed layout).

As stated earlier, the shortest path method needs to be extended for arbitrary number of levels and for arbitrary level source, target nodes.

We also plan to extend SuperBatCat for attributed and temporal graphs which will help us explore how graph structures change overtime.

Bibliography

[1] Barabási, A.-L., Albert, R., and Jeong, H. Scale-free characteristics of random networks: the topology of the world-wide web. *Physica A: Statistical Mechanics and its Applications 281*, 1 (2000), 69–77.

[2] Barnes, J., and Hut, P. A hierarchical o (n log n) force-calculation algorithm.

[3] Bostock, M. D3. js-data-driven documents, 2012.

[4] Clauset, A., Newman, M. E., and Moore, C. Finding community structure in very large networks. *Physical review E 70*, 6 (2004), 066111.

[5] Csardi, G., and Nepusz, T. The igraph software package for complex network research. *InterJournal, Complex Systems 1695*, 5 (2006).

[6] Developers, N. Neo4j. *Graph NoSQL Database [online]* (2012).

[7] Gephi, N. Gephi-the open graph viz platform, 2010.

[8] Girvan, M., and Newman, M. E. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences 99*, 12 (2002), 7821–7826.

[9] Karypis, G., and Kumar, V. Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0.

[10] Kleinberg, J. The small-world phenomenon: An algorithmic perspective. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing* (2000), ACM, pp. 163–170.

[11] Newman, M. E. Finding community structure in networks using the eigenvectors of matrices. *Physical review E 74*, 3 (2006), 036104.

[12] Smith, M., Milic-Frayling, N., Shneiderman, B., Mendes Rodrigues, E., Leskovec, J., and Dunne, C. Nodexl: a free and open network overview, discovery and exploration add-in for excel 2007/2010. *Social Media Research Foundation* (2010).

[13] Yang, J., and Leskovec, J. Community-affiliation graph model for overlapping network community detection. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on* (2012), IEEE, pp. 1170–1175.