Clustering

M. Tech. Project Stage I Report

Submitted in partial fulfillment of the requirements
for the degree of

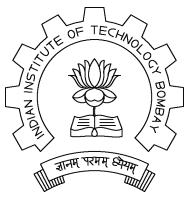
Master of Technology

by

Rose Catherine K.
Roll No: 07305010

under the guidance of

Prof. S. Sudarshan



Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

Acknowledgements

I would like to express my sincere thanks and gratitude to my guide, Prof. S. Sudarshan, for the constant motivation and guidance he gave me through out the project work, and for having patience to clarify my doubts and for bringing into perspective, the different aspects of the project topic. Working with him was a great learning experience.

I would also like to thank Prof. Soumen Chakrabarti, for the clarifications regarding random walks, thereby giving me a good insight into how they are employed in finding communities.

Rose Catherine K.
MTech. 1
CSE, IIT Bombay.

Abstract

Clustering is the process of finding out a grouping of the given set of objects, such that those in the same collection are similar to each other. This is important because, it reveals the high level organization of the data.

It is also important from the point of view of keyword searching in databases. Identifying nodes that are highly related to each other, and grouping them together, can localize the computing required to answer a particular keyword query, to a single or a few clusters. Thus, creating good quality clustering of the graph representation of the database can bring down the keyword query answer time, considerably.

Clustering has different aspects, like the objective function to be optimized, the method used to achieve this optimum, and measuring the goodness of a particular clustering on a graph. A large number of algorithms have been proposed for optimizing different objective functions. This report discusses a few clustering techniques, their advantages and shortcomings, and also studies a particular algorithm called `Nibble`, in detail. The latter was implemented and tested, to gain insight into its performance and how it can be adapted for the particular domain of keyword searching.

Contents

1 Introduction

Keyword searching is an important paradigm of searching, which is receiving considerable interest during the past few years. Keyword search in databases is becoming increasingly important.

In general, a keyword search system, for example, the BANKS System ([BHN+02]), begins by identifying nodes that are relevant to the keywords. It then proceeds to connect these nodes according to the search algorithm. This involves moving along the edges of the graph, to the neighbors of the nodes. Now, consider databases that have millions of nodes and edges. For example, DBLP has 1,771,381 nodes and 2,124,938 edges, in the equivalent graph representation. This entire data will not fit in the main memory of the search system and hence, will be stored in external memory. This increases the time required to access the data graph, and hence affects the query answer time.

As mentioned in [Upa08], the external memory BANKS could perform better if the nodes that are connected to each other are retrieved together - either by storing them in the same block of memory, or, by storing them in the same machine, if the data is distributed across machines. This can be done by first clustering the nodes, and then assigning each cluster to a machine. Hence, by creating good quality clustering of the graph representation of the database, the keyword query answer time can be brought down considerably.

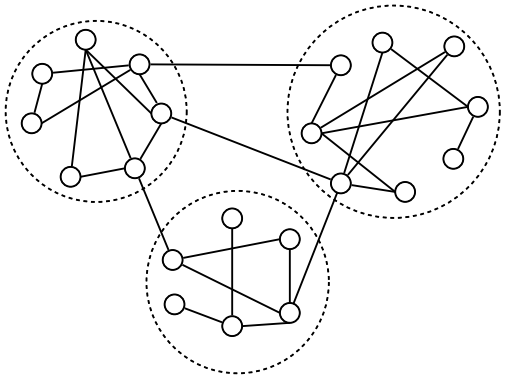


Figure 1: Example of clustering on a simple graph

A cluster can be defined as a collection of objects that are similar to each other. Clustering is the process of finding out the clusters in the given set of objects. The similarity measure has to be given; i.e., there should be a method to find the degree of similarity or distance between any two objects.

Consider a graph consisting of vertices/nodes and edges connecting them. Let the edges represent some similarity between the incident nodes. Then a cluster can be considered as a set of nodes such that, edges connecting nodes within the cluster are more than those linking to nodes outside the cluster. i.e., connections within it is dense and inter-cluster edges are comparatively low. A simple example is shown in Figure 1.

Clustering can be done on any data, by representing it as a graph. For a relational database, the commonly used construction of data graph, is as follows: the tuples of the database form the

nodes and the cross references between them, like foreign key references, inclusion dependencies, etc., form the edges of the graph. The nodes may also be a cell in the table, according as the granularity required. Similar techniques can be used to convert XML data and the web corpus into graph representations.

Overview

This report is organized as follows: Section 2 gives an overview of different aspects of clustering, followed by a discussion on quantifying the strength of community structure in Section 3. Section 4 discusses an approach to clustering, using graph partitioning technique. In Section 5, a few methods for finding communities are discussed, followed by a discussion of using the idea of random walks in determining the same, in Section 6. Section 7 explains some of the experiments conducted and its analysis, followed by conclusions and future work in Section 8.

2 Overview of Clustering Techniques

In this Section, we discuss some of the aspects of clustering, such as objective functions and methods of clustering.

All clustering techniques are based on optimizing an objective function. Following are some of the commonly used objective functions:

- Distance based measures: The objective function is based on the geometry of points in some D-dimensional space. Here, the dissimilarity of any two points is usually taken as the Euclidean distance between them. Hence, it is necessary that the data must have a D-dimensional representation. K-means (discussed in this Section) is an example for a clustering technique that uses a distance based function as its objective.

- Cut Size: Cut size is given by the number of edges that connect nodes which are in different clusters. The goal is to minimize cut size. The METIS Algorithm (discussed in Section 4), is an example for an algorithm whose objective is to minimize cut size.

- Community related measures: These are used for measuring the quality of the communitites found. Examples of these include Conductance and Modularity (discussed in Section 3.1 and Section 3.2 respectively). Examples for algorithms that optimize these measures include Modularity Weight Prioritized BFS (Section 5.4) and Clustering using Seed Sets (Section 6.2).

There are many methods of clustering, that optimize different objective functions. Some of these are listed below:

Hierarchical Clustering Methods:

Hierarchical methods are of two types - agglomerative and divisive ([HK06]). Agglomerative methods proceed by adding edges to the set of nodes, to identify the clusters. They begin with

a graph that has all the nodes of the original graph, and no edges. Then starting with the vertex pairs with highest similarity, edges are added. Divisive methods work on the original graph. Here, in every step of the clustering process, least similar vertices are found out and the edge between them is removed. In both the techniques, the clustering procedure can be stopped at any step, and the resulting components in the network give the clusters. Example for hierarchical clustering method, is the divisive method using Edge Betweenness ([NG04]), discussed in Section 5.1.

Partition Refinement Method:

Given n objects and a number k, a partitioning method constructs k partitions of the data, where each partition represents a cluster. In general, these methods proceed by first creating an initial partitioning of the data into k clusters. It then attempts to improve the partitioning by moving objects from one group to another (more details can be found in [HK06]). Examples include K-means (discussed in this Section) and Extremal Optimization (discussed in Section 5.3).

Random Walk based Techniques:

Random walk based methods are used to discover communitites in a network graph. The goal of these methods is usually to find the cluster to which a particular node belongs, or the enclosing cluster of a set of nodes, called the seed set. The objective function is usually one of the community goodness measures. Random walk based techniques are discussed in detail in Section 6.

Geometric Clustering Methods:

Geometric clustering methods try to optimize a distance based measure, such as a monotone function of the diameters or the radii of the clusters, and finds clustering based on the geometry of points in some D-dimensional space ([CRW90]). Below, we discuss the K-means clustering algorithm which tries to minimize a *distortion* function.

K-means Clustering

K-means clustering takes as input, N points in some D-dimensional space, and a number K, which is the number of clusters required. The goal is to find K points (in the D-dimensional space), called *mean*, that represent the K clusters, and an assignment of the N input points to one of the clusters, such that, the sum of the squares of the distances of each data point to the *mean* of its cluster, is a minimum.

The objective can be formally stated as given in [Bis06]: Let the data points be $\mathbf{x}_1, ..., \mathbf{x}_N$. Consider the $1 - of - K$ coding scheme for representing the assignment of a data point $\mathbf{x}_i$, where a set of binary variables $r_{ik} \in \{0, 1\},\ k = 1, ..., K$ are associated with it, such that, if $\mathbf{x}_i$ is assigned to cluster k, then, $r_{ik} = 1$ and $r_{ij} = 0$ for all $j \neq k$. Let the *mean* of cluster k be

μ_k. Then, the objective function, (also called as *distortion measure*) is given by:

$$J = \sum_{i=1}^{N} \sum_{k=1}^{K} r_{ik} \ \| \mathbf{x}_i - \mu_k \|^2$$

Algorithm:

The goal is to minimize the objective function stated above. The algorithm proceeds in the following steps:

(i) Choose some initial values for μ_k for all $k = 1, ..., K$.
Then, the algorithm proceeds in two half steps:

(ii) In the first half-step, assign each of the data points to its closest *mean*. i.e., assign $\mathbf{x}_i$ to that μ_k that minimizes $\| \mathbf{x}_i - \mu_k \|^2$.

(iii) In the second half-step, set μ_k to the mean of all the data points assigned to cluster k.

The algorithm is assured to converge, since each step reduces the value of the objective function.

Limitations:

- The direct implementation of $K - means$ algorithm can be slow, since in the first half-step of each iteration, it is necessary to compute the distance between every data point and every *mean*. Improvements have been suggested that take advantage of the triangle inequality for distances.
- The dissimilarity between a data point and a *mean* point, is taken as square of the Euclidean distance between them. Hence, it is necessary that, the data being clustered must be points in some D-dimensional space - this is not true always. $K - medoids$ algorithm is an improvement over this, where the dissimilarity between every pair of points is specified beforehand, and the *mean* points, called *medoid*s in this case, are constrained to be chosen from the data points.

3 Quantifying the strength of community structure

A community is a set of real-world entities that form a closely knit group. The community structure in networks gives natural divisions of network nodes into densely connected subgroups. (This is discussed in detail in Section 5 and Section 6)

All clustering algorithms produce a grouping of the nodes of the graph. And usually, the community structure may not be known ahead of time. Measuring the goodness of the clusters found is hence important. But, a minimum value of cut size doesn't reveal much about the structure, since mostly, a small number of nodes will be surrounded by a smaller cut, than a large number of nodes. Also, there is no reason for the communitites to be of same size. Therefore, these cannot be used to quantify the strength of the clustering found. Goodness of a community structure is measured using the following quantities:

3.1 Graph Conductance

Graph conductance (as given in [AL06]), also known as the normalized cut metric, is defined as given below:

Let $G = (V, E)$ be a graph. Now, define the following:

- $d(v)$ is the degree of vertex v.
- For $S \subseteq V$, $Vol(S) = \sum_{v \in S} d(v)$
- Let $\bar{S} = V - S$. Then, S defines a cut and $(S, \bar{S})$ defines a partition of G.
- The cutset is given by $\partial(S) = \{\{u, v\} \mid \{u, v\} \in E, u \in S, v \notin S\}$, which is the set of edges that connect nodes in S with those in $\bar{S}$. The cutsize is denoted by $|\partial(S)|$.

Then, the *conductance* of the set S is defined as:

$$\Phi(S) = \frac{|\partial(S)|}{min(Vol(S), Vol(\bar{S}))} \tag{3.1}$$

3.2 Modularity

The definition of modularity given in [Dji06], measures the difference between the number of in-cluster edges and the expected value of that number in a random graph on the same vertex set.

Specifying formally, let V_1, ... V_k be the node subsets induced by the clustering. Then, modularity can be expressed as:

$$Q = \frac{1}{m} \sum_{i=1}^{k} (|E(V_i)| - Ex(V_i, \mathcal{G})) \tag{3.2}$$

where, $E(V_i)$ is the set of all edges of G with both endpoints in V_i and $Ex(V_i, \mathcal{G})$ is the expected number of such edges in a random graph from a given random graph distribution $\mathcal{G}$, with a vertex set V_i.

Expected number of edges in Erdös-Rényi Random Graph Model

In the random graph model $G(n,p)$ of Erdös and Rényi, each edge out of the $\binom{n}{2}$ edges (between every pair of nodes) is materialized with a fixed probability p. Hence, the expected number of edges is $\binom{n}{p}\ p$ (as mentioned in [Upa08]).

If the expected number of edges in the graph is m, then

$$p = \frac{m}{\binom{n}{2}}$$

The expected number of edges in a partition V_i is given by $\binom{|V_i|}{2}\ p$.

Expected number of edges in Chung-Lu Random Graph Model

In the paper [CL02], the authors suggest a model for random graphs with a given expected degree sequence. Here, the probability that a particular edge exists, is proportional to the product of the expected degree of its end points. i.e., for a given expected degree sequence $\mathbf{w} = (w_1, w_2, ..., w_n)$, the probability p_{ij} that there is an edge between v_i and v_j is given by:

$$p_{ij} = \frac{w_i w_j}{\sum_k w_k}$$

assuming that $max_i\ w_i^2 < \sum_k w_k$.

The expected number of edges in a partition can be found by the following method, as given in [Upa08]: Let V_d be a partition of the graph, and E_d^T represent all the possible edges in the subgraph generated by V_d. Also, let X_d be a random variable that represents the number of edges in V_d. For the edges $e \in E_d^T$, if I_e is the indicator random variable that e exists, then, $X = \sum_{e \in E_d^T} I_e$. Hence, expected number of edges in the partition can be expressed as:

$$E(X) = \sum_{e \in E_d^T} E(I_e)$$

$$= \frac{1}{\sum_k w_k} \sum_{(i,j) \in E_d^T} w_i w_j$$

Modularity can also be measured in the following manner, as given in [NG04]: Consider a particular division of a network into k clusters. Now, define a $k \times k$ symmetric matrix $\mathbf{e}$ such that, the element e_{ij} gives the fraction of all edges in the network that link vertices in cluster i to vertices in cluster j. Hence, the quantity, $Tr\ \mathbf{e} = \sum_i e_{ii}$, which is the trace of this matrix, gives the fraction of edges in the network that connect vertices in the same cluster. A good division into clusters should have a high value of this trace. Now, define the row sums $a_i = \sum_j e_{ij}$, which represents the fraction of edges that connect to vertices in cluster i. In a random graph, where edges connect vertices without regard for the communities they belong to, $e_{ij} = a_i a_j$. Then, modularity measure can be defined as:

$$Q = \sum_i (e_{ii} - a_i^2) = Tr\ \mathbf{e} - \parallel \mathbf{e}^2 \parallel \tag{3.3}$$

Values of Q approaching 0, indicate that the clustering is no better than random and values closer to 1, indicate strong community structure.

4 Clustering to Minimize Cut Size using Graph Partitioning

The graph partitioning problem is defined as follows([KK98]): Given a graph $G = (V, E)$ and an integer k, partition the set of nodes of the graph into k subsets V_1, V_2, ..., V_k, such that:

(i) The subsets are pairwise disjoint. i.e., $V_i \cap V_j = \emptyset$ for $i \neq j$

(ii) The union of all the subsets give the entire node-set. i.e., $\bigcup V_i = V$

(iii) The number of nodes in each of the subsets is the same, and is equal to $|V|/k$

(iv) The number of edges of E whose incident vertices belong to different subsets is minimized. This quantity is called the cut-size.

To motivate this, consider the example given in [NG04], that arises in parallel computing. Suppose there are n intercommunicating computer processes, which should be distributed over g computer processors. The pattern of communications between processes can be represented by a graph or network in which the vertices represent processes and edges join process pairs that need to communicate. The problem is to allocate the processes to processors in such a way as roughly to balance the load on each processor, while at the same, time minimizing the number of edges that run between processors, so that the amount of inter-processor communication is minimized.
In general, finding an exact solution to graph partitioning problem is NP-complete. Also, the graphs tend to be very large and even an $O(E^2)$ algorithm is too expensive. Below, we discuss an algorithm which can find k-way partitioning of a graph in $O(E)$ time. It is also capable of processing very large graphs.

METIS:

The paper by Karypis and Kumar, [KK98], presents an algorithm for k-way partitioning of graphs, minimizing the cut size. The overall idea of the algorithm is as follows: The graph is first coarsened down to a small number of vertices by collapsing vertices and edges. Now, a k-way partitioning is found on this smaller graph. This partitioning is projected back onto the original graph, by refining it at the intermediate levels, to get a k-way partitioning of the original graph. For the graph $G = (V, E)$, the algorithm runs in $O(|E|)$ time. The paper proposes a computationally inexpensive refinement algorithm, which can be used to project the initial partitioning to increasingly uncoarsened version of the graph. This enables the entire partitioning procedure to run in a time, which is faster by a factor of $O(log(k))$ than previously proposed multilevel recursive bisection algorithms.

Coarsening Phase:

In this phase, a sequence of smaller graphs $G_i = (V_i, E_i)$, is constructed from the original graph $G = (V, E)$, by collapsing vertices and edges. When a set of vertices in G_i is combined to form a single vertex in G_{i+1}, weight of the new vertex is set to the sum of the weights of the vertices combined. And, all the edges incident on the combined vertices are made incident on the new vertex, in order to preserve the connectivity information in the coarser graph. When

more than one of the combined vertices have edges to the same vertex u, the weight of the edge connecting the new vertex to u is set to the sum of the weights of these edges.

Coarsening the graph by collapsing edges can be defined in terms of matchings. A matching of a graph is defined as a set of edges such that, the edges don't share any vertices. A matching of $G = (V, E)$ can be found in time $O(|E|)$. If a matching is found on a graph, then the next coarsened graph can be found by combining the incident vertices of the edges in the matching, into a single vertex. Coarsening is stopped when the number of vertices in the coarsest graph becomes less than ck for some constant c, or if the reduction in the size of successively coarser graphs becomes less than a certain constant factor.

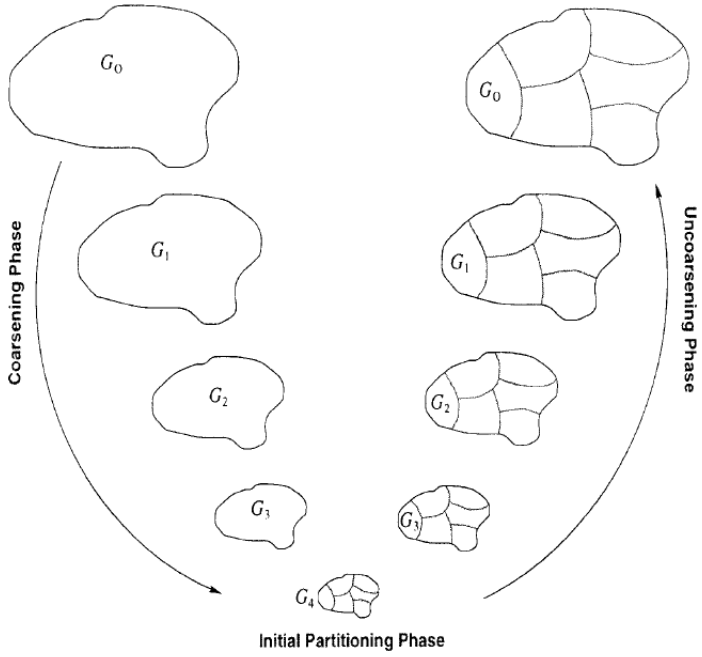


Figure 2: Various phases of the METIS algorithm

Methods used to find matching:

Following methods can be used to find a matching of a graph:

- Random Matching (RM): Visit the vertices in random order. If a vertex has not been matched yet, then select one of its unmatched adjacent vertices, randomly. The edge connecting them, is then included in the matching.
- Heavy Edge Matching (HEM): Visit the vertices in random order, as in the case of RM. But, if a vertex is found to be unmatched yet, match it with that adjacent vertex for which, the weight of the edge connecting them is the maximum over all valid incident edges.

- Modified Heavy Edge Matching (HEM*): Suppose if v is an unmatched vertex. Let H denote the set of its adjacent vertices which are unmatched and are connected to it by an edge of maximum weight. For each vertex u in H, find the sum of the weights of edges that connect u to vertices adjacent to v. Then, v is matched with that u for which is sum is the maximum. This is similar to HEM, but this scheme gives a smaller average degree for the coarser graph.

Initial Partitioning Phase:

A k-way partitioning is computed on this smaller graph using a multilevel bisection algorithm, such that, it is balanced and satisfies the minimum edge-cut requirements. A multilevel recursive bisection (MLRB) algorithm works by first coarsening the graph to get a smaller graph. A bisection of this graph is computed, and the partitioning is projected back to the original graph by refining the partitioning. A recursive bisection method can be used for computing k-way partition, by first obtaining a 2-way partitioning of graph, and then recursively obtaining a 2-way partitioning of each partition. Repeating this for $logk$ times, the graph will be partitioned into k partitions.

Uncoarsening Phase:

In this phase, the partitioning of the coarsest graph G_m is projected back to the original graph, by going through the sequence of graphs G_{m-1}, G_{m-2}, . . . , G_1, G. The main idea employed during the uncoarsening phase is refinement. Since G_{i-1} is finer than G_i and has more degrees of freedom, the partitioning projected onto it can be refined to further decrease the edge-cut. The paper describes a new k-way refinement algorithm, whose complexity is independent of the number of partitions being refined. Consider the partitioning of an intermediate graph. For each vertex v in its nodeset, define $N(v)$ as the neighborhood of v, which is the union of the partitions to which its adjacent vertices belong. During the refinement process, v can move to any of its neighboring partitions. For each partition in $N(v)$, compute the gain of moving v into it. For each partition b in $N(v)$, define $ED[v]_b$, which is the external degree of v to partition b, as the sum of the weights of the edges connecting v to nodes in b. Also, define $ID[v]$, the internal degree of v as the sum of the weights of the edges that connect v to nodes in its own partition. Now the gain of moving vertex v to partition b is defined as $ED[v]_b - ID[v]$. The partitioning refinement algorithm will move a vertex only if it satisfies the following
Balancing Condition: Suppose that a and b are two partitions of a graph $G = (V, E)$. Let $W(i)$ denote the weight of the partition i, let $w(v)$ denote the weight of the node v. Define $W^{min} = 0.9|V|/k$ and $W^{max} = C|V|/k$, where C is a positive real number, which can be used to vary the degree of imbalance among partitions. Now, a vertex v, can be moved from a to b only if the following conditions are satisfied:

- $W(b) + w(v) \leq W^{max}$
- $W(a) - w(v) \geq W^{min}$

Greedy Refinement (GR):

Consider a graph and its partitioning. The vertices are visited in random order. Let v be a vertex and $N(v)$, its neighborhood. Let $N'(v)$ be the subset of $N(v)$ such that, the movement of vertex v to any partition in $N'(v)$ satisfies the Balancing Condition. Now vertex v is moved to one of the partitions b in $N'(v)$, if either one of the following conditions is satisfied:

- $ED[v]_b > ID[v]$ and $ED[v]_b$ is maximum among all b in $N'(v)$.
- $ED[v]_b = ID(v)$ and $W(a) - W(b) > w(v)$.

That is, move v to a partition that gives the largest reduction in the edge-cut, or if no reduction in the edge-cut is possible, then move v to the partition that doesn't increase the edge-cut but improves the balance.

5 Clustering for Finding Communities

A community is a set of real-world entities that form a closely knit group. As mentioned in [NG04], community structure in networks gives natural divisions of network nodes into densely connected subgroups. Example for a community in social network analysis could be a set of people such that, they interact amongst themselves more when compared to others outside the group. A web community could be a set of web pages that link more to pages within the group. The ability to find and analyze such groups can provide invaluable help in understanding and visualizing the structure of networks.

Determining communities is a topic of great interest. A few examples are mentioned in [Dji06]. It is a way to analyze and understand the information contained in the huge amount of data available on the WWW and the relationships between the individual items. Communities also correspond to entities such as collaboration networks, online social networks, scientific publications or news stories on a given topic, related commercial items, etc. They also arise in other types of networks such as biological networks (protein interaction networks, genetic networks).

Finding communities can be modeled as a graph clustering problem, where vertices of the graph represent entities and edges describe relationships between them. Finding communitites and clustering have become synonymous.

In this section, we discuss four approaches for finding communitites, and in Section 6, we discuss another class of approaches based on random walks.

5.1 Divisive Method using Edge Betweenness

In the paper [NG04], Newman and Girvan describe an algorithm which is divisive in nature, but is different from the conventional hierarchical clustering techniques by two features - firstly, the edges to be removed are identified using a "betweenness" measure, and secondly, this measure is recalculated after each removal.

Approach:

Divisive clustering methods, as described in Section 2 proceed by removing iteratively, the edge connecting the least similar vertices. Here, the algorithm proceeds by finding edges in the network that are most "between" other vertices.

Betweenness is some measure that favors edges that lie between communities and disfavors those that lie inside communities. Hence, they are responsible for connecting many pairs of vertices. The main intuition is that, if the number of times, a set of paths traverse an edge can be counted, then, this number would be higher for edges that connect communitites, and hence, can be used to identify them.

The idea behind recalculation of the measure after every removal is that, once an edge is removed, the betweenness values for the remaining edges will no longer reflect the situation in the new graph. Thus without recalculating the measure, any divisive algorithm will fail to recover the cluster structure, in the simplest of the cases.

Different Betweenness measures:

- Shortest-Path Betweenness of an edge is the number of shortest paths between all pairs of nodes, that traverse that edge. It can be calculated for all edges in time $O(mn)$, where m is the number of edges in the graph and n is the number of vertices.
- Random-Walk Betweenness of an edge is the sum of the expected number of times, a random walk between a particular pair of nodes traverses that edge, over all node pairs.
- Current-Flow Betweenness: Consider a circuit that is obtained by placing a unit resistance on all the edges of the graph, and a unit current source and sink at a pair of nodes. The current-flow betweenness for an edge can then be defined as the absolute value of the current flowing through that edge, summed over all source-sink pairs.

Algorithm:

The algorithm for finding community structure is as follows:

(i) Calculate the betweenness scores for all edges in the graph.

(ii) Find the edge with the highest betweenness score and remove it from the graph.

(iii) Recalculate betweenness for all remaining edges.

(iv) Repeat from step (ii).

Shortest-path betweenness for all edges can be calculated in time $O(mn)$. But, since this has to be repeated in every iteration of the algorithm, which can go upto m, the overall time complexity (worst-case) of the algorithm is $O(m^2n)$, or $O(n^3)$ on a sparse graph.

Limitations:

- The input graph is assumed to have undirected and unweighted edges.
- The fastest of the implementations, which is based on shortest-path betweenness, takes $O(n^3)$ time on a sparse graph, which is intractable for large graphs with millions of nodes and edges.

5.2 Clustering using Minimum Weighted Cut

In the paper [Dji06], Djidjev shows that the problem of finding a graph clustering which maximizes the modularity, can be converted into the problem of finding a minimum weighted cut (MWC) on a complete graph with the same vertices as the original graph. The intuition behind this mapping is that, for any good clustering of a graph, the cut set, which consists of edges that connect nodes present in different partitions, should be small. A major difficulty in using the MWC for clustering is that, it does not consider the sizes of the subgraphs induced by the cut. Also, the MWC problem is known to be NP-hard, in case of real valued weights. Hence, the resulting minimum cut problem is solved by a modified algorithm for graph partitioning.

Reducing clustering to a minimum cut problem:

The minimum weighted cut (MWC) problem can be defined formally as follows: Given a graph $G = (V, E)$ with real valued weights on its edges, find a partition of V such that the cut set is of minimum weight.

In Section 3.2, modularity was expressed as:

$$Q = \frac{1}{m}\sum_{i=1}^{k}(|E(V_i)| - Ex(V_i, \mathcal{G}))$$

The clustering task is to find out a partition $\mathcal{P} = \{V_1 \cap ... \cap V_k\}$ such that modularity of the resulting cluster is maximized. i.e.,

$$\begin{aligned} & max_{\mathcal{P}}\left\{\sum_{i=1}^{k}(|E(V_i)| - Ex(V_i, \mathcal{G}))\right\} \\ & = -min_{\mathcal{P}}\left\{-\sum_{i=1}^{k}(|E(V_i)| - Ex(V_i, \mathcal{G}))\right\} \\ & = -min_{\mathcal{P}}\left\{(|E(G)| - \sum_{i=1}^{k}|E(V_i)|) - (|E(G)| - \sum_{i=1}^{k}Ex(V_i, \mathcal{G}))\right\} \\ & = -min_{\mathcal{P}}\left\{|Cut(\mathcal{P})| - ExCut(\mathcal{P}, \mathcal{G})\right\} \end{aligned}$$

where $Cut(\mathcal{P})$ is the cut of $\mathcal{P}$ and $ExCut(\mathcal{P}, \mathcal{G})$ is the expected value of $Cut(\mathcal{P})$ for a random graph from $\mathcal{G}$.

Now, consider a complete graph G', with the same vertex set as G. The edges of G', given by (i, j) are assigned weights as below:

$$weight(i, j) = \begin{cases} 1 - p_{ij} & if (i, j) \in G \\ -p_{ij} & if (i, j) \notin G \end{cases}$$

where p_{ij} is the probability that there is an edge between vertices i and j in a random graph from the class $\mathcal{G}$.
From the above, it can be shown that the problem of maximizing modularity can be equivalently posed as the problem of finding a partition $\mathcal{P}'$ of G' such that the size of the cut induced by $\mathcal{P}'$ on G' is minimized.

Solving the Minimum Cut Problem:

Minimum cut problem can be solved by a modified graph partitioning algorithm. In the implementation, the graph partitioning algorithm used was based on METIS [KK98]. Modifications like dropping the requirement for balance of the partition and determining the cardinality of the partition that minimizes the cut size, must be done. But the main challenge was to bring the complexity of the algorithm close to linear on the size of the original graph, rather than on the size of the transformed one, since the latter graph is always dense. But, as mentioned in the paper, it is possible to simulate the execution of the multilevel algorithm on the transformed

graph, by explicitly maintaining information only about the edges from the original graph and implicitly taking into account the remaining edges by modifying the formulae for computing weights and gains.

Complexity:

If d is the depth of the dendrogram describing the clustering hierarchy, then, finding an optimal clustering which has k parts takes $O((nlog(n) + m)d)$ time. Typically d is $O(log(k))$

5.3 Extremal Optimization

In the paper [DA05], Duch and Arenas propose a divisive algorithm to find the community structure in complex networks using a heuristic search, which is based on extremal optimization of the value of modularity.

Definitions:

In Section 3.2, Modularity, which is a measure of the goodness of clustering, was explained. Consider the expression of Modularity given by Newman and Girvan ([NG04]):

$$Q = \sum_r (e_{rr} - a_r^2)$$

- Contribution of an individual node i to the value of modularity, for a certain partitioning of the graph, is defined as:

  $$q_i = \kappa_{r(i)} - k_i \, a_{r(i)}$$

  where $\kappa_{r(i)}$ is the number of links that i has to nodes which are in its community, and k_i is the degree of node i. The quantity $a_{r(i)}$ is the number of links whose atleast one incident vertex is in the community of i.

  From the above definition, it can be seen that,

  $$Q = \frac{1}{2L} \sum_i q_i$$

  where L is the total number of links in the network.

- The $fitness$ of a node i, given by λ_i, is defined as:

  $$\lambda_i = \frac{q_i}{k_i} = \frac{\kappa_{r(i)}}{k_i} - a_{r(i)}$$

  The above quantity is the degree normalized contribution of node i. λ_i can be used to compare the relative contribution of individual nodes to the community structure.

Algorithm

Extremal Optimization (EO) algorithm optimizes a global variable by improving extremal local variables. Here, modularity value, Q, is the global variable to be optimized.

As remarked in the paper, since the space of possible partitions grows faster than any power of the system size, searching for the optimal modularity value seems to be a NP-hard problem. Hence, heuristics must be used to restrict the search space while searching for a partitioning that optimizes modularity.

The proposed heuristic search algorithm proceeds in the following steps:

(i) Split the nodes of the whole graph into two random partitions, having the same number of nodes in each, so that an initial clustering is obtained.

(ii) Calculate the $fitness$ of each of the nodes.

(iii) Self Organization step: Move a node with low value of $fitness$ (extremal), from one partition to the other. Specifically, the $\tau - EO$ probabilistic selection is used, where initially, the nodes are ranked according to their $fitness$ values, and then, a node of rank q is selected according to the probability distribution:

$$P(q) \propto q^{-\tau}$$

where, $\tau \sim 1 + 1/ln(N)$ (N is the number of nodes).

(iv) Recalculate the $fitness$ values of nodes.

(v) If the "optimal state" where Q has a maximum value, is not yet reached, then repeat the process from step (iii) onwards.

(vi) Otherwise, this partitioning can be accepted. Therefore, delete all links between both the partitions and recurse on each of the resulting components.

(vii) The algorithm stops when the value of modularity cannot be further improved.

Complexity:

In the implementation, at each self-organization step, the previously obtained maximum value of Q is stored, and if this maximum is not improved in N steps, the search is stopped. Also, ranking process has a cost of $O(Nln(N))$ associated with it. After every self-organization step, $fitness$ of nodes have to be calculated, thus giving an overall computational cost of $O(N^2ln^2(N))$. However, using heap data structures can reduce the time for the ranking selection process, to $O(N)$. Then, total cost of the algorithm will be $O(N^2ln(N))$.

Advantages:

- During the self-organization step, since the node to be moved across the partition is chosen by probabilistic selection method, the final result will be independent of initialization and can escape from local optima.

Limitations:

- As remarked in [Upa08], the biggest drawback is that shifting any vertex changes the contribution of all the nodes, and hence, all of them needs to be examined in the next round, for selecting the next node.Computation of the contribution thus can not be reduced further.
- Also, a large number of communities of very small sizes get formed.

5.4 Modularity-Weight Prioritised BFS

In the paper [Upa08], Upadhyaya suggests an algorithm called Modularity-Weight Prioritised BFS, which is based on the Modified Extremal Optimization algorithm suggested by Duch and Arenas in [DA05], which was discussed in Section 5.3.

The proposed algorithm differs from the latter algorithm in the following:

- All nodes are put in `partition 1` when the procedure begins, instead of randomly assigning to one of the partitions.
- Only those nodes in the boundary are considered for shifting to the other partition (`partition 0`), unlike the Modified Extremal Optimization algorithm, where all nodes are considered.
- All nodes whose every neighbor is in the other partition, is moved to that partition, with out regard to whether they give the maximum increase in modularity.

Algorithm:

The proposed algorithm proceeds in the following steps:

(i) Initially, assign all nodes to `partition 1` and initialize the `fringeNodeVector` with the vertex with lowest degree.

(ii) Choose the `top` element of `fringeNodeVector` and move it to `partition 0`. Update the `fringeNodeVector` by adding the neighboring nodes of the `top` element, which are in `partition 1`.

(iii) Now, choose a fixed number of elements, randomly, from `fringeNodeVector`. From these, move all nodes whose every neighbor is in the other partition, to `partition 0`.

(iv) For other nodes in the chosen set, calculate the increase in modularity on moving it to `partition 0`. Select that node with the highest increase and make it the `top` element of the `fringeNodeVector`.

(v) Repeat steps from (ii) onwards.

(vi) If highest increase in modularity obtained in step (iv) is less than zero, then, this iteration is called as an $iteration-without-improvement$. If the number of such iterations exceed a particular number, then, we stop iterating.

(vii) Once the iteration is stopped, then, undo all node exchanges done after the one which gave the maximum value of modularity over all the iterations. This gives two partitions. Now recursively call `Modularity-Weight Prioritised BFS` on these two partitions.

(viii) If the size of a partition goes below a minimum value, or if the minimum expected modularity change is lesser than a user-provided value, then it is not partitioned further.

Time Complexity:

Time complexity is $O(n\ ln(k))$, where n is the number of nodes and k is the number of clusters approximately desired. In general, $k \sim \frac{2n}{size}$, where $size$ is the user-specified lower bound on the cluster size.

Advantages:

- The time complexity of $O(n\ ln(k))$ is one of the best achieved upper bounds.

Limitations:

- Since the next node to be shifted is chosen from the set of boundary nodes, it increases the probability of getting stuck at a local optimum.
- To bring down the running time, not all nodes are tested for gain in modularity; only a fixed number of them are chosen randomly from the set of boundary nodes and tested. As remarked in the paper, the solution given by this could be an approximate one.

6 Finding Communities using Random Walk based Methods

The objective of these methods is usually to find the cluster to which a particular node belongs, or the enclosing cluster of a set of nodes, called the seed set.

In general, it functions by starting a random walk from the specified node. The intuition is that, since the nodes within a cluster will be densely connected, with a large probability, the walk will remain within the cluster.

Another version of random walk is the truncated walk. In this, probability is spread as usual to neighboring vertices, but is then removed from any vertex whose probability is below a certain threshold. Truncated walk allows the computation to be made more local. As mentioned in [AL06], the main advantage of clustering techniques based on truncated random walks is that its time and space requirements can be made independent of the graph size.

6.1 Clustering using Nibble Algorithm

In the paper [ST04], Spielman and Teng describes a nearly-linear time algorithm, `Partition`, for computing crude partitions of a graph. The algorithm works by approximating the distribution of random walks on the graph. It employes truncated random walks to speed up the procedure.

Definitions:

The definition of $Vol(S)$, $\partial(S)$ and conductance, $\Phi(S)$, for a subset $S \subseteq V$ of the graph $G = (V, E)$ is given in Section 3.1, except that here, conductance is referred to as *sparsity*.

In addition to this, the *balance* of a cut S is defined as

$$bal(S) = \frac{Vol_V(S)}{Vol_V(V)}$$

These terms are defined for the subgraph of G induced by a subset of the vertices $W \subseteq V$, where $S \subseteq W$, as below:

$$Vol_W(S) = \sum_{v \in S} |w \in W : (v, w) \in E|$$

$$\partial_W(S) = \sum_{v \in S} |w \in W - S : (v, w) \in E|$$

$$\Phi_W(S) = \frac{|\partial_W(S)|}{min(Vol_W(S), Vol_W \bar{S})}$$

Nibble Algorithm:

`Nibble` (Table 1) is an intermediate algorithm that is called implicitly by `Partition`. `Nibble` takes a vertex as input, which is called the seed vertex, and returns the enclosing cluster of that node. The algorithm executes a few steps of a random walk starting at the seed vertex and approximately computes the probability distribution. If this random walk does not mix rapidly, then, from this probability distribution, a small cut can be found. The time and space required

to compute this approximation can be minimized by executing a truncated random walk, where, after each step of the walk, those probabilities that are lower than a particular threshold are set to 0.

$C = \texttt{Nibble}(G, v, \theta_0, b)$
G a graph, v a vertex, $\theta_0 \in (0,1)$, b a positive integer.
(1) Set $\tilde{p_0}(x) = \chi_v$
(2) Set $t_0 = 49\ ln(me^4)/\theta_0^2$, $\gamma = \frac{5\theta_0}{7.7.8\ ln(me^4)}$, and $\epsilon_b = \frac{\theta_0}{7.8\ ln(me^4)t_0 2^b}$
(3) For $t = 1$ to t_0
  (a) Set $\tilde{p_t} = [P\ \tilde{p_{t-1}}]_{\epsilon_b}$
  (b) Compute a permutation $\tilde{\pi_t}$ such that $\tilde{p_t}(\tilde{\pi_t}(i)) \geq \tilde{p_t}(\tilde{\pi_t}(i+1))$ for all i.
  (c) If there exists a $\tilde{j}$ such that
    (i) $\Phi(\tilde{\pi_t}(\{1, ..., \tilde{j}\}) \leq \theta_0$,
    (ii) $\tilde{p_t}(\tilde{\pi_t}(\tilde{j})) \geq \gamma / Vol_V(\tilde{\pi_t}(\{1, ..., \tilde{j}\})$, and
    (iii) $5\ Vol_V(V)/6 \geq Vol(\tilde{\pi_t}(\{1, ..., \tilde{j}\}) \geq (5/7)\ 2^{b-1}$
    then output $C = \tilde{\pi_t}(\{1, ..., \tilde{j}\}$ and quit.
(4) Return $\emptyset$.

Table 1: Pseudocode for `Nibble` algorithm

Random Walk - mathematical notations

Define two vectors χ and ψ as below:

$$\chi_S(x) = \begin{cases} 1 & for\ x \in S \\ 0 & otherwise \end{cases}$$

$$\psi_S(x) = \begin{cases} d(x)/Vol_V(S) & for\ x \in S \\ 0 & otherwise \end{cases}$$

The walk that is considered here, is such that, at each step, it remains in the same vertex with half the probability and otherwise, moves along one of the randomly chosen edges incident on this vertex, to its neighbor. This can be represented in matrix form as $P = (AD^{-1} + I)/2$, where, A is the unweighted graph, and D is the diagonal matrix with $(d(1), ..., d(n))$ on the diagonal. The probability distribution of the random walk with start vertex v, obtained after t steps, is given by $p_t^v = P^t \chi_v$.

The truncation operation can be represented as:

$$[p]_\epsilon(v) = \begin{cases} p(v) & if\ p(v) \geq 2\epsilon d(i) \\ 0 & otherwise \end{cases}$$

Truncation operation is done after every step of the random walk, and for the nodes whose probability, $p_t(i)$ is lesser than $2\epsilon d(i)$, is rounded off to 0.

$C = \texttt{RandomNibble}(G, \theta_0)$
(1) Choose a vertex v according to ψ_V
(2) Choose a b in $1, ..., \lceil log(m) \rceil$ according to
$$Pr[b = i] = 2^{-i}/(1 - 2^{-\lceil log(m) \rceil})$$
(3) $C = \texttt{Nibble}(G, v, \theta_0, b)$

Table 2: Pseudocode for `Random Nibble` algorithm

$D = \texttt{Partition}(G, \theta_0, p)$
where G is a graph, $\theta_0, p \in (0, 1)$.
(0) Set $W_1 = V$
(1) For $j = 1$ to $56m \lceil lg(1/p) \rceil$
(a) Set $D_j = \texttt{RandomNibble}(G(W_j), \theta_0)$
(b) Set $W_{j+1} = W_j - D_j$
(c) If $Vol_{W_{j+1}}(W_{j+1}) \leq (5/6)\ Vol_V(V)$, then go to step (2)
(2) Set $D = V - W_{j+1}$

Table 3: Pseudocode for `Partition` algorithm

$\mathcal{C} = \texttt{MultiwayPartition}(G, \theta, p)$
(0) Set $\mathcal{C}_1 = V$ and $S = \emptyset$
(1) For $t = 1$ to $\lceil log_{17/16} m \rceil$. $\lceil lg(m) \rceil$. $\lceil lg(2/\epsilon) \rceil$
(a) For each component $C \in \mathcal{C}_t$,
$D = \texttt{Partition}(G(C), \theta_0, p/m)$
Add D and $C - D$ to $\mathcal{C}_{t+1}$
(2) Return $\mathcal{C} = C_{t+1}$

Table 4: Pseudocode for `Multiway Partition` algorithm

`Random Nibble` (Table 2) is an intermediate algorithm which calls `Nibble` on carefully chosen random inputs.

`Partition` (Table 3) calls `Nibble` through the `Random Nibble` method, for atmost, a fixed number of times. It then collects the clusters found by `Nibble`. As can be seen from the *Step* (1)(*c*) of the algorithm, as soon as the volume of this collection exceeds $\frac{1}{6}^{th}$ of the volume of the entire graph, it returns the collection.

`Multiway Partition` (Table 4) uses `Partition` to get a partitioning of the graph. It then invokes the latter again, on the two partitions thus obtained. This is repeated for a fixed number of times.

`Nibble` can run in time $O(2^b ln^4(m)/\theta_0^5$. If `Nibble` returns a non-empty set C, then,

(i) $\Phi_V(C) \leq \theta_0$

(ii) $Vol_V(C) \leq (5/6) Vol_V(V)$

`RandomNibble` can run in time $O(ln^4(m)/\theta_0^5$, and the set output by it satisfies the same properties as that output by `Nibble`.
`Partition` can run in $O(mlg(1/p) ln^4(m)/\theta_0^5$. Let D be the output of $\texttt{Partition}(G, \theta_0, p)$. Then,

(i) $Vol_D(D) \leq (31/36) Vol_V(V)$

(ii) $\Phi_V(D) \leq \theta$, where, $\theta_0 = (5/36)\theta$.

The expected running time of `Multiway Partition` is $m\ (lg(1/p)\ lg^{O(1)}(m))/\theta^5$.

6.2 Clustering using Seed Sets

In the paper [AL06], Andersen and Lang present an algorithm to discover the enclosing community of a given cohesive set of nodes, called the "seed set". They modify the algorithm proposed by Spielman and Teng ([ST04]), which can find the enclosing cluster with a single seed vertex. This algorithm is desirable since, it can expand the seed set to find a community, that has small conductance, while examining only a small portion of the entire graph. They also study the effects of the size and quality of the seed set on the results.
Seed set expansion is commonly done in the link-based analysis of the web. It first came into prominence with the HITS algorithm, proposed by Jon Kleinberg where, a search engine was used to retrieve a set of papes related to a particular input, which served as the seed set. A fixed-depth neighborhood expansion was performed on this set, to get a larger set of pages upon which the HITS algorithm was run.

The main intuition behind the algorithm can be explained as follows: Consider the random walk which begins from the nodes in the seed set. Since, within a cluster, the nodes are expected to link to each other more often, much of this walk will be contained in the cluster. As soon as we move outside the cluster, the probability will fall, revealing the cluser boundary.

Algorithm:

The definition of $Vol(S)$, $\partial(S)$ and conductance, $\phi(S)$, for a subset S of the graph is given in Section 3.1.

The algorithm proceeds in the following steps:

(i) The initial probability distribution p_0 is set to ψ_S, where

$$\psi_S = \begin{cases} d(x)/Vol(S) & if\ x \in S \\ 0 & otherwise \end{cases}$$

(ii) Define the random walk transition matrix, M as $1/2(I+AD^{-1})$. Here, A is the unweighted graph, and D is the diagonal matrix with $(d(1), ..., d(n))$ on the diagonal ($d(i)$ is the degree of node i).

(iii) Simulate the next step of the random walk to obtain the probability distribution, $p_{t+1} = Mp_t$.

(iv) Sort the vertices in descending order of their degree-normalized probabilities: $r_t(v) = p_t(v)/d(v)$.

(v) For the truncated walk, set the probability on any vertex for which $r_t(v) \leq \epsilon$ to 0, where ϵ is a constant, called the threshold.

(vi) Let v_i^t be the i^{th} vertex after sorting, such that $r(v_i^t) \geq r(v_{i+1}^t)$. Then, this ordering defines a collection of sets $S_0^t, ..., S_J^t$, where $S_j^t = \{v_i^t | 1 \leq i \leq j\}$, and J is the number of vertices with nonzero values of $p(u)/d(u)$.

(vii) Each of the sets S_j^t, are tested for a good community.

(viii) If none of the sets qualify as a good community, then the random walk is continued, from step (iii) onwards.

Good seed sets:

Consider the amount of probability that has escaped from a community C after T steps of the random walk. This depends on both C and the seed set S. If C has a small conductance, then a seed set is good, if the amount of probability that has escaped is not much larger than $\phi(C)\ T$.

Following are also good seed sets:

- Any set that is fairly large and nearly contained in the target community.
- Sets chosen randomly from within a target community.

Advantages:

- The major advantage of the algorithm is that it explores only local locality. This is achieved by using truncated random walk instead of an exact walk.
- The algorithm is able find a small community enclosing the seed set, by touching only a few number of nodes. This is accomplished by using truncated walk distributions in place of exact walk distributions.
- The algorithm is capable of finding nested clusters that enclose the seed set. This is achieved by simulating the walk for a larger number of steps.

Disadvantages:

The major disadvantage of the proposed algorithm is the method of selection of the seed set, which should be cohesive. In the experiments done by the authors, the target cluster set was initially identified and nodes were chosen from this set, randomly, to form the seed set. But, this cannot be done, when the underlying clustering of the graph is not known beforehand.

7 Experiments and Analysis

The `Nibble` algorithm proposed in Section 6.1 was implemented in Java, with some minor modifications that are given below:

- In the step 3(b) of the algorithm, Nibble computes a permutation of the nodes such that the edges are ordered according to their probabilities. The method of computing the permutation is not specified. Hence, in the implementation, a randomized *QuickSort* procedure is used to sort the nodes in the decreasing order of their probabilities.
- In the step 3(c) of the algorithm, Nibble doesn't specify how $\tilde{j}$ is chosen. In the implementation, the condition 3(c) is checked for all j starting from 0 till J , where J is the index of the last node with non-zero probability. $\tilde{j}$ is then set to the first j for which the condition is satisfied.
- For reducing the running-time, the *sweep* method of computing *sparsity* (or *conductance*) as given in the paper [AL06], was implemented. This is not suggested in `Nibble`.

Details of the Experiments:

`Nibble` was executed on `IIT Bombay Electronic Submission of Theses and Dissertations Database` (`etd2`). `etd2` contains tables for `department, faculty, program, students` and `thesis`. It is comparitively small, with just 4329 nodes and 21432 edges, in the equivalent graph representation. But, having a small database will enable us to manually check if the results obtained are intuitive, and make changes accordingly, before proceeding on to larger ones, with millions of nodes.

Result I:

Execution of `Nibble` for $sparsity = 0.15$, produced the result given below:

Cluster #1:

```
start node = 537
number of nodes in the cluster = 2189
sparsity = 0.14952751528627015
cut size = 269
subset vol = 8917
number of iterations = 32
```

Cluster #2:

```
start node = 538
number of nodes in the cluster = 226
sparsity = 0.1495798319327731
cut size = 89
subset vol = 595
number of iterations = 2
```

Cluster #3:

```
start node = 540
number of nodes in the cluster = 62
sparsity = 0.12547528517110265
cut size = 33
subset vol = 263
number of iterations = 2
```

Cluster #4:

```
start node = 542
number of nodes in the cluster = 39
sparsity = 0.14705882352941177
cut size = 20
subset vol = 136
number of iterations = 2
```

Result II:

`Nibble` was modified slightly to run with a manually chosen node. The cluster obtained for $startnode = 0$ and $sparsity = 0.15$, is given below:

start node = 0

```
number of nodes in the cluster = 39
sparsity = 0.14979757085020243
cut size = 74
subset vol = 494
number of iterations = 31
```

Result III:

`Nibble` was executed for different values of sparsity. The details of the first cluster found, are given below:

max sparsity = 0.70 :

```
start node = 537
number of Nodes = 330
sparsity = 0.6994061215166743
cut size = 1531
subset vol = 2189
number of iterations = 1
```

max sparsity = 0.50 :

```
start node = 537
number of Nodes = 767
sparsity = 0.49983676134508653
cut size = 1531
subset vol = 3063
number of iterations = 1
```

max sparsity = 0.30 :

```
start node = 537
number of Nodes = 1566
sparsity = 0.2995990070651136
cut size = 1569
subset vol = 5237
number of iterations = 2
```

max sparsity = 0.15 :

```
start node = 537
number of Nodes = 2189
sparsity = 0.14952751528627015
cut size = 269
subset vol = 8917
number of iterations = 32
```

max sparsity = 0.10:

```
start node = 537
no cluster was found even after 1000 iterations.
```

Result IV:

`Nibble` was modified to take as input, a value for `maxClusterSize` as well, and find clusters such that, the number of nodes in them doesn't exceed this value. Hence, when checking for condition $3(c)$ of the algorithm, $\tilde{j}$ was constrained to be between 0 and `maxClusterSize`. The results of the execution for `maxClusterSize = 500` and `max sparsity = 0.15`, are given below:

Cluster #1:

```
start node = 537
number of Nodes = 10
sparsity = 0.13043478260869565
cut size = 3
sub set vol = 23
number of iterations = 664
```

Cluster #2:

```
start node = 537
number of Nodes = 46
sparsity = 0.1320754716981132
cut size = 21
sub set vol = 159
number of iterations = 726
```

The next cluster could not be found even after 3000 iterations.
This modified `Nibble` was also executed for `maxClusterSize` set to 100, 200, 750, 1000 and 1500, and it gave the same two clusters every time.

node id	database table entry
537	MTech. program
538	PhD. program
540	MDes. program
542	MPhil. program
0	Computer Center

Table 5: Nodeid to Database Entry mapping for a few nodes

Observations:

Following are the observations:

- For the clusters given in `Result I`, the nodes grouped together were related to the start node. For example, the nodes added to `Cluster #1` were MTech. students, MTech. theses, professors who have guided many MTech. students etc., and nodes added to `Cluster #2` were PhD. students, PhD. dissertations etc. This is the same in the case of manually chosen node (`Result II`) - 0 represents the `Computer Center` and nodes added to the cluster were faculty closely related to it and the departments of `CSE` and `IT`.
- Sizes of the clusters differ drastically - 2189, 226, 62 and 39 in the order of discovery.
- Effect of `Start Node`: When there is no constraint on the maximum size of the cluster, the clustering is on `program` basis - `MTech., PhD., MDes., MPhil.` etc. which were the starting nodes.
- Effect of `sparsity`: From `Result III`, it seems that sparsity is affecting the cluster size inversely - for higher sparsity, smaller is the cluster size and for lower sparsity, higher is the cluster size. But, this could not be verified, since it could be because of the particular database under consideration.
- Effect of `maxClusterSize`:

- The clusters given in `Result IV` are on the basis of `department` - the first cluster is for `School of Management` and the second for `Humanities and Social Science Department`.
- The effect of `start node` is no more visible and it itself was not added to both of the clusters.
- Varying the value of `maxClusterSize` did not affect the clustering, indicating that the clusters found are of good quality.

Conclusions:

- The first implementation of `Nibble` is not very successful in finding the clustering which is more intuitive, ie. the one based on `Department`.
- The dependance on the starting node is not a desirable property.
- The modified version of `Nibble` with the bound `maxClusterSize` added, is also not very successful; eventhough it could find clusters on the basis of `Department`, it was unable to find all the clusters.
- When the start node or the seed vertex is itself not in the cluster, it seems to defeat the entire logic applied in the algorithm, since it was supposed to find the enclosing cluster for the start node.

8 Conclusions and Future Work

Clustering is the technique of finding the underlying structure of a graph and is a very important area of research. It is employed in a wide variety of fields and each field uses a particular objective function and a method for clustering, that is suited to its goal. A few of such objectives and clustering techniques were studied. One of the proposed community finding method, based on random walks, called the `Nibble` algorithm was implemented and executed on the `etd2` database, for understanding its functioning and performance. Through this exercise, a few shortcomings of the algorithm were identified and it gave insight into how we must proceed in the future.

Following is the proposed direction of future work:

- As seen from the analysis of the results of the experiments conducted on `etd2` database, the current implementation of the `Nibble` algorithm doesn't perform well, on the clustering task. Implementation of a new method for choosing the seed vertex, finding all the clusters and handling the case of the seed not being included in the cluster, are some of the future work in this direction.

- In many of the cases, the basis of clustering is intuitive and known beforehand. For example, in the case of the `Bibliography` database, the clustering is expected to be on the basis of the research area, which in turn is determined by the `conference`, and in the case of `etd` database, the basis is expected to be `department`. When some information is known in addition to just the graph structure, the clustering algorithm must be able to take advantage of the situation, to produce good quality clusters.

- Wikipedia can be converted into a graph representation with vertices as the articles and edges as the inter-Wiki links. Then, this graph can be clustered to get a higher level organization of topics. Now, consider a mapping from the set of web pages in WWW to the Wiki-pages, which is based on some similarity between the two. Once this mapping is in place, then the entire web corpus can be clustered based on the clustering that has been computed for their image in Wikipedia. In general, given a small graph and a mapping of the nodes of a larger graph to nodes in the former, it will be possible to find a good clustering for the latter, by finding a good clustering for the smaller graph.

- The long term goal is to reduce the query answer time for database keyword query systems, by finding a good quality clustering on the database graph. The graph obtained after clustering nodes, called the supernode graph, must be sufficiently small - with high node and edge compression, so that, a large number of the queries can be processed locally. This objective has to be formalized and a method of clustering for achieving the objective has to be devised.

- Testing some of the methods and heuristics proposed in the papers discussed in this report, to understand their functioning and performance.

References

[AL06] Reid Andersen and Kevin J. Lang. Communities from Seed Sets. *Proceedings of the 15th international conference on World Wide Web*, pages 223–232, 2006.

[BHN+02] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. *ICDE*, 2002.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[CL02] Fan Chung and Linyuan Lu. Connected Components in Random Graphs with Given Expected Degree Sequences. *Annals of Combinatorics*, 6:125–145, 2002.

[CRW90] Vasilis Capoyleas, Gunter Rote, and Gerhard Woeginger. Geometric Clusterings. *Journal of Algorithms*, 1990.

[DA05] J. Duch and A. Arenas. Community detection in complex networks using extremal optimization. *Physical Review E, 72:027104*, 2005.

[Dji06] Hristo N. Djidjev. A scalable multilevel algorithm for graph clustering and community structure detection. *In Workshop on Algorithms and Models for the Web Graph*, 2006.

[HK06] Jiawei Han and Micheline Kamber. *Data Mining - Concepts and Techniques*. Elsevier, 2006.

[KK98] George Karypis and Vipin Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distributed Computing 48*, pages 96–129, 1998.

[NG04] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Phys. Rev. E, 69(2):026113*, 2004.

[ST04] Daniel A. Spielman and Shang-Hua Teng. Nearly-Linear Time Algorithms for Graph Partitioning, Graph Sparsification, and Solving Linear Systems. *ACM STOC-04*, pages 81–90, 2004.

[Upa08] Prasang Upadhyaya. Clustering Techniques for Graph Representations of Data. *Technical report, Indian Institute of Technology, Bombay*, 2008.