Clustering the Graph Representation of Data

M. Tech. Project Stage 2 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 throughout 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 suggestions on direction of improvement and future work.

Rose Catherine K.
MTech. 2
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.

This report discusses a particular method of clustering, called Nibble Algorithm, which uses random walks over the graph representation of data. It was implemented and tested, to gain insight into its performance, advantages and shortcomings. This led us to propose a modified algorithm, which we call Modified-Nibble Algorithm that improves upon the former one. This algorithm was implemented and tested on the `dblp3` database and the `wikipedia` dataset, the results of which are also discussed.

Contents

1 Introduction

Keyword searching is an important paradigm of searching, which is receiving considerable attention 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, such that the equivalent graph representation does not fit in the main memory of the search system and hence, will have to 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.

Cluster

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. 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.

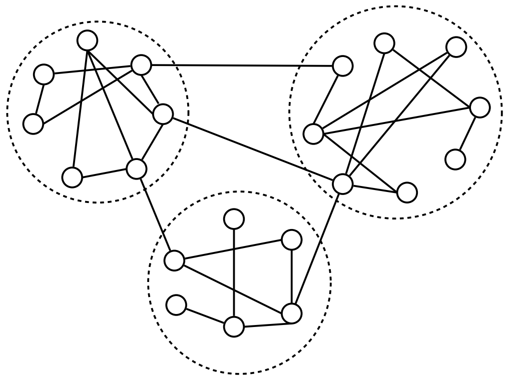
Figure 1: Example of clustering on a simple graph

Community

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 with each other more often than with those outside the group. A web community could be a set of web pages that link more to pages within the group. Determining communities has become a topic of great interest. As mentioned in [Dji06], it is a way to analyze and understand the information contained in the huge amount of data available on the world wide web. 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. Finding communities can be modeled as a graph clustering problem, where vertices of the graph represent entities and edges denote relationships between them. Hence, community-finding and clustering have become synonymous.

Objective function for clustering

All clustering techniques are based on optimizing an objective function. Some commonly used objective functions are distance-based measures, cut-size and community-related measures. We are particularly interested in the latter one, which is used for measuring the quality of the communitites found. Examples of these include Conductance (discussed in Section 2) and Modularity. A discussion on different methods that optimize the above objective functions can be found in [Cat08].

Graph representation of data

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. Another graph that is receiving considerable amount of attention is the wiki-graph. Here, nodes of the graph are the articles / web pages in the Wikipedia website [wik]. An edge is added between two nodes if there is a hyperlink between the corresponding articles. Similar technique can also be used to convert the web corpus into graph representation.

Overview

This report is organized as follows: Section 2 explains a method for quantifying the goodness of a community structure. Section 3 gives an overview of random walks and how it is employed in finding communities. It also discusses a particular technique for partitioning the graph using an algorithm called `Nibble`. In Section 4, we detail the `Modified Nibble` algorithm proposed by us. Section 5 explains some of the experiments conducted and its analysis, followed by conclusions and future work in Section 6.

2 Quantifying the goodness of community structure

Almost always, the underlying community structure of a given graph is not known ahead of time. In the absence of this information, we require a quantity that can measure the goodness of the clustering produced by an algorithm.

It is quite obvious that, usually, the cut surrounding a small number of nodes will be smaller than that of a large number of nodes. So, a minimum value of cut size doesn't reveal much about the structure, since it is biased towards clusters of smaller sizes. Similarly, there is no reason for the communitites to be of same size. Hence, partition techniques that group nodes of the graph into clusters of roughly the same size, cannot be applied for finding communitites. For measuring the strength of a community structure, we use the notion of conductance, which is explained below.

Graph Conductance

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

$$\Phi(S) = \frac{|\partial(S)|}{min(Vol(S), Vol(\bar{S}))} \tag{2.1}$$

3 Finding Communities using Random Walks on Graphs

Random walk can be considered as a graph traversal technique. The walk starts from a node designated as the $startNode$; at each step of the walk, the node explored next is one of the neighbors of the current node, chosen randomly with equal probability. Since this method of traversal doesn't distinguish between nodes already explored and those that are yet untouched, the walk may pass through some nodes, multiple number of times.

There are many variations to the above basic form of random walk. A popular variant is where the edges of the graph have weights associated with them, and the node explored next is chosen with a probability that is proportional to the weight of the edge connecting the current node to the neighbor [CS07]. Another variant of the walk allows self-transition: at each step, with certain amount of predetermined probability, the walk may remain at the current node; otherwise, the next node is chosen with equal probability, or with probability proportional to the edge-weights, as the case may be, from the set of neighbors of the current node [ST04].

Random walk analysis have been used in many fields to model the behaviour of many processes. Some of the popular examples include the set of web pages visited by a surfer, the path traced by a molecule in a medium, the price of stocks and the financial status of a gambler.

3.1 Probability distribution of a walk

In many applications, instead of performing discrete random walks, it is more interesting to find out the probability of a random walk of k steps which started at a particular $startNode$, touching a particular node [CS07]. In this scenario, the nodes of the graph have a quantity called *nodeProbability* associated with them, which gives the probability of the walk under consideration to be at that particular node, at the instant/step of inspection. So, before the walk starts, the $startNode$ has *nodeProbability* of 1 and the rest have their probabilities set to 0. If the $startNode$ has m neighbors, then, in the first step, all the neighbors have their *nodeProbability* set to $1/m$ and that of the $startNode$ is set to 0. In subsequent steps, each node which has a positive value for their *nodeProbability* will divide its current value, equally between its neighbors - this is called *spreading of probabilities.* Nodes with positive values for *nodeProbability* are said to be *active*, and hence, the above process can also be called *Spreading Activation*, though there are differences between the two concepts. If a node receives activation from multiple neighbors, they are accumulated. At any step of the walk, the node-probabilites of the nodes of the graph add up to 1.

Many variants exist for the above method of finding the probability distribution over the nodes of the graph, according as the variant of the underlying random walk that is used. A popular variant is the one which uses a threshold for activation. Here, a node is considered to be active only if its *nodeProbability* is greater than a predefined *threshold* value [ST04]. Yet another variant is based on truncated random walks. Here, if the *nodeProbability* of a node falls below a predefined *threshold* value, then its probability is set to 0 [ST04]. An important difference between this one and the previous methods is that, here the node-probabilities may not add up to 1; and in fact, monotonically decreases as the walk progresses.

3.2 Rationale

Clustering techniques using random-walks are based on the intuition that a walk started from a particular node will remain within the enclosing cluster for that node with high probability, since the nodes within the cluster are densely connected. Hence, if the probability distribution of nodes after a few steps of the walk is considered, they will be roughly in the order of their degree of belongingness to the cluster under consideration. As mentioned in [CS07], self-transitions in the walk allow it to stay in place, and reinforce the importance of the starting point by slowing diffusion to other nodes. But as the walk gets longer, the identity of nodes in the clusters blur together.

Consider the toy example given in Figure 2, where the *nodeProbability* of the nodes after a 3-step walk from the *startNode*, is shown. It can be noted that, the nodes within the cluster for the *startNode* have high probabilities associated with them and as soon as we cross the cluster, the probabilites drop suddenly, thus revealing the boundary. This notion is used in the algorithm for clustering using seed sets, proposed by Andersen and Lang in [AL06].

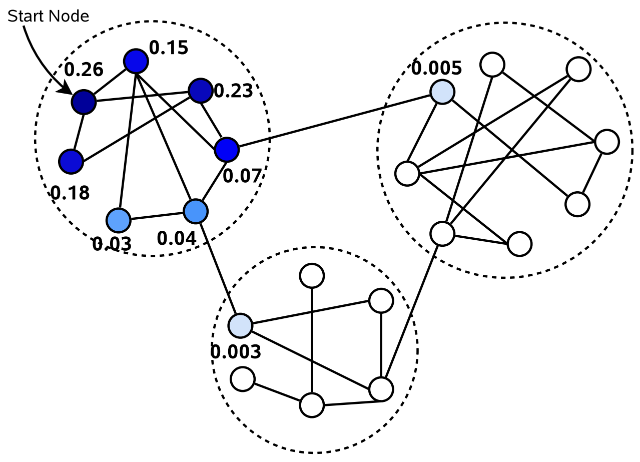

Figure 2: Example for sudden drop in probability outside the cluster

The above example shows that, the probability distribution of the random walk gives a rough ranking of the nodes of the graph. Hence, it is possible to find the nodes of the cluster by considering the first k of the top ranking nodes. But, this k cannot be fixed beforehand. Here, conductance comes to our rescue.

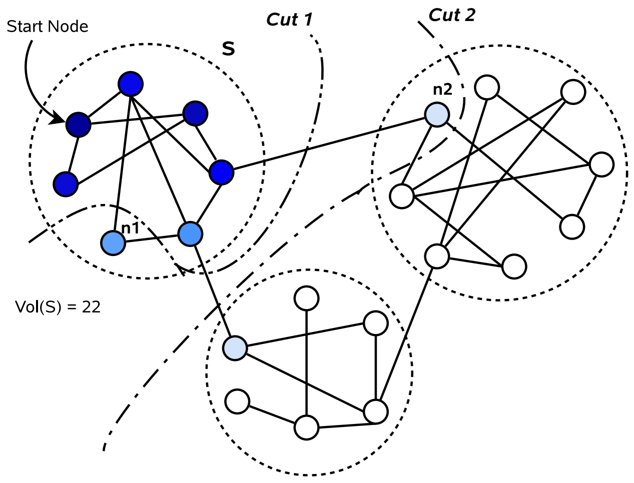

Figure 3: Example for choosing the best cluster based on conductance

Consider another toy example shown in Figure 3. The preferred cluster contains the first 7 top ranking nodes. It has 2 cut edges and its volume is 22. Conductance of this cut is 0.09. Suppose that the seventh node, $n1$, is not included. This corresponds to $Cut1$ in the figure. It decreases the volume by 2 and increases the cut size by 2, giving the conductance as 0.2. Similarly, suppose that we include the next highest ranking node, which is $n2$, also in the cluster ($Cut2$). It increases the volume by 3 and the cut size changes to 3, giving the conductance as 0.12.

The above example illustrates how conductance can be used to find the best cluster for a specified *startNode*. This notion is used in the algorithm for partitioning graphs using `Nibble`, proposed by Spielman and Teng in [ST04] (Section 3.4), and the `Modified-Nibble` algorithm proposed by us in this report (Section 4).

3.3 Advantages of using random walk based clustering

(i) With appropriate parameter settings, a random-walk based clustering technique can find good communities by touching only a small neighborhood of the *startNode*, without exploring the entire graph.

(ii) The time and space requirements to find a community for a particular node can be made independent of the graph size, by using truncated random walks ([AL06]) or by bounding the number of active nodes.

(iii) Rather than processing the entire graph at once, the clustering process can proceed by removing one cluster at a time, and then continuing on the remainder graph.

3.4 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.

3.4.1 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 2, 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:

$$\begin{aligned}
Vol_W(S) &= \sum_{v \in S} |w \in W : (v, w) \in E| \\
\partial_W(S) &= \sum_{v \in S} |w \in W - S : (v, w) \in E| \\
\Phi_W(S) &= \frac{|\partial_W(S)|}{min(Vol_W(S), Vol_W\bar{S})}
\end{aligned}$$

Random Walk - mathematical notations

Define two vectors χ and ψ as below:

$$\begin{aligned}
\chi_S(x) &= \begin{cases} 1 & for\ x \in S \\ 0 & otherwise \end{cases} \\
\psi_S(x) &= \begin{cases} d(x)/Vol_V(S) & for\ x \in S \\ 0 & otherwise \end{cases}
\end{aligned}$$

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.

3.4.2 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 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.

```
$C = $ RandomNibble$(G, \theta_0)$
(1) Choose a vertex $v$ according to $\psi_V$
(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 = $ Nibble$(G, v, \theta_0, b)$
```

Table 2: Pseudocode for `Random Nibble` algorithm

```
$D = $ 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 = $ 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} = $ 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 = $ 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

`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.

4 The Modified-Nibble Algorithm

Based on our implementation of the `Nibble` algorithm and the experiments conducted on the `IIT Bombay Electronic Submission of Theses and Dissertations Database (etd2)` graph (described in [Cat08]), we identified the following shortcomings of the algorithm.

(i) It is difficult to specify the conductance of the clusters, apriori. Hence, instead of taking it as a user-input, the algorithm must be capable of finding clusters with best value of conductance.

(ii) In the step 3(c) of `Nibble`, any value of j that satisfies the three conditions is accepted. Consider the case where the user-specified conductance value is greater than the actual conductance of a cluster. Then, the algorithm might terminate early, as soon as the larger value of conductance is reached, but before finding this better cluster.

(iii) Size of the cluster is an important property which the user may want to control to some extent. The maximum allowable size may be constrained by the size of external memory block or by the size of the main memory of machines in a distributed scenario. In `Nibble`, the user has no way of regulating the cluster size.

(iv) `etd2` contains tables for `department`, `faculty`, `program`, `students` and `thesis`. `Nibble` was not able to find the intuitive clustering which is the one based on `Department`. However, when we modified it to take as input, a value $maxClusterSize$ as an upper bound on the size of the clusters, it could find the required clustering, indicating that upper-bounding the cluster-size is key to finding better clusters.

(v) If unchecked, there is a high probability for the random walk to spread over the entire graph, especially when there are hub nodes. This situation is not desirable and the algorithm must be able to reduce the impact of misbehaving hub nodes. `Nibble` doesn't control the spread of the walk.

Keeping the above drawbacks in mind, we propose the `Modified-Nibble` algorithm, which is detailed below. It gives the user, adequate control over the properties of the clusters and at the same time, finds the cluster with best conductance without requiring the user to specify any value.

4.1 Terms and Definitions

`Modified-Nibble`, finds the probability distribution over the nodes of the graph. The random walk considered here has self-transition probability set to 0.5. The edge-weights are not taken into account while spreading probability to neighbors (these terms were described in Section 3.1).

If A represents the adjacency matrix of the graph, and D, the diagonal matrix with $(d(1), ..., d(n))$ on the diagonal, where $d(i)$ gives the degree of node i, then the transition probability matrix, P, can be expressed as $(AD^{-1} + I)/2$. The probability distribution of the random walk with start vertex v, obtained after t steps, is given by $p_t^v = P^t \chi_v$, where χ_v is the vector with all entries set to 0, except for v, which is set to 1.

Random walk step: By simulating a step of the walk, we mean that all nodes with non-zero node-probability will spread its current probability value to its neighbors. This is equivalent to computing $p_{t+1} = P\ p_t$. New nodes that become active during this step, can take part in spreading probabilities, only in the next step.

Batched random walk: A batch of i random walk steps just means that i steps of the walk are performed one after the other, without any intervention. This is equivalent to computing $p_{t+i} = P^i\ p_t$.

`MaxClusterSize`: This, taken as an input from the user, defines the upper bound on the number of nodes put into a single cluster. The values used in the experiments range from 100 to 1500. As reported in [DKS08], lower cluster sizes incur high IO cost, and larger sizes lead to considerable search overhead.

`MaxActiveNodeBound` and `f`: The former term upper bounds the maximum number of nodes that can be active at any time, during the clustering process. The latter term stands for $factor$ and is a user-input. Its values in the experiments range from 100 to 600. Given `MaxClusterSize` and `f`, `MaxActiveNodeBound` is set as the product of the two.

Arithmetic plus Geometric Progression (APGP): As the name suggests, the i^{th} term of an APGP series is the sum of i^{th} terms of an Arithmetic Progression and a Geometric Progression. $t_i^{apgp} = (a + id) + (a\ r^i),\ i = 0, 1, 2, ...$ The parameters can be used to get fine-grained control over the difference between successive terms of the series. r has to be kept small since otherwise, as i increases, the successive terms differ by a very large value. But then, for smaller values of i, the successive terms will be too close. To avoid this, we set d to a high value, and for larger values of i, the effect of the terms from AP will be overtaken by the terms of GP. For most of the experiments, the values used were: *a = 2, d = 7, r = 1.5.*

Degree Normalized Probability of a node is the ratio of its *nodeProbability* and degree. Nodes with large degree tend to acquire high values of probability even if they don't "belong" to the cluster. Normalizing the probability with the degree enables us to deal with such misbehaving nodes.

4.2 The Clustering Algorithm

The algorithm for clustering takes as input, the graph representation of data and outputs the clusters, which are disjoint sets of nodes of the graph. The core of the entire algorithm is the `Modified Nibble` procedure (detailed in Section 4.2.1), which in turn invokes the `Find Best Cluster` procedure (detailed in Section 4.2.2).

Algorithm

(i) Choose a start node, s. Here, if any prior knowledge of communitites is available, it can be used to select the start node. In our implementation, we chose the one with largest out-degree, similar to the `Random Nibble` algorithm (Table 2), except for the probability aspect. This decision was revised later (discussed in Section 4.3).

(ii) Invoke `Modified Nibble` procedure with s as the start node, on the current graph.

(iii) Let L be the list of nodes returned by `Modified Nibble`. This is the cluster for s. Remove L from the graph.

(iv) Repeat from step (i), until the entire graph is processed.

As can be seen from step (iii), instead of processing the entire graph at once, the algorithm proceeds by removing one cluster at a time, and then continuing on the remainder graph.

4.2.1 Modified Nibble procedure

`Modified Nibble` procedure accepts as input, a start node s and a graph G', and outputs a set of nodes belonging to the cluster for s. G' is the remainder graph, which consists of only unprocessed nodes.

Algorithm

(i) Set the *nodeProbability* of s to 1, and that of all other nodes of G' to 0. Let $totalWalkSteps$, initially set to 0, denote the number of steps of random walk simulated till now.

(ii) Get the next term, t_i (for i^{th} iteration), in the APGP series, and perform a batch of random walks for $(t_i - totalWalkSteps)$ number of steps. If t_i exceeds $maxClusterSize$, the batch of random walks is done for $(maxClusterSize - totalWalkSteps)$ steps.

(iii) If at any time during the batched random walk in step (ii), the number of active nodes exceed $maxActiveNodeBound$, then we don't simulate the remaining steps in the batch. But, directly proceed to step (iv). This decision was revised later and is discussed in Section 4.3.

(iv) Invoke `Find Best Cluster` to get the best cluster among the current active nodes.

(v) If the conductance of the best cluster returned in step (iv) is equal to or larger than the conductance of the best cluster obtained in the previous iteration of this loop (i.e., with the previous batch of random walks), then we make a greedy decision to stop, assuming that doing more walks may not give any improvement. So, return the cluster obtained in the previous iteration.

(vi) Else if the conductance has decreased, we assume that doing more walks might give a better cluster. But if t_i already exceeds $maxClusterSize$, or, if the number of active nodes have reached the $maxActiveNodeBound$, we stop and return the best out of the current and previous clusters. Otherwise, set $totalWalkSteps$ to t_i and repeat from step (ii).

In the algorithm, $totalWalkSteps$ is upper bounded by $maxClusterSize$ (step (vi)). As mentioned earlier, $maxClusterSize$ usually has its value in hundreds; and on performing that many walks, the probability distribution of nodes will tend towards the stationary distribution, which is not desirable. But, we still allow $maxClusterSize$ number of walks, to take care of the special situation where we have chains of nodes.

The decision to stop when the number of active nodes reach $maxActiveNodeBound$ was also changed later and is discussed in Section 4.3.

4.2.2 Find Best Cluster procedure

`Find Best Cluster` procedure accepts as input, a set of nodes, their current node-probabilities and in-degrees. It outputs a subset of these nodes, which can be considered as the best cluster among the nodes which are active at present.

Algorithm

(i) Find the degree-normalized probabilities of all nodes. Here, the probability of a node is normalized by its in-degree.

(ii) Sort the nodes in the decreasing order of their degree-normalized probabilities.

(iii) Find a j such that the first j entries of the sorted set have the smallest conductance. Here, j is constrained to lie between 1 and $min(numActiveNodes, maxClusterSize)$. If two values of j give the same conductance, then choose the larger one.

(iv) Return the first j entries of the sorted set as the best cluster, for the current active nodes.

Each invocation of the above procedure involves a sorting. Hence, from the aspect of running time of the entire clustering, adequate care must be taken to ensure that it is not invoked too often. This is dicussed in the section on heuristics (Section 4.4).

4.3 Revised version of the Modified-Nibble algorithm

Based on some of the experiments done on the `dblp3` database and the `wikipedia` dataset (detailed in Section 5.2), we have made a few changes to the proposed clustering algorithm.

Choosing the start node

Nodes with smaller degrees are very often, towards the periphery of the graph, and their clusters are easier to find. In particular, degree 1 nodes (pendant vertices) almost always belong to the cluster of its lone neighbor. Hence, they provide a good starting point for exploring the graph. Nodes with larger degrees are usually hub nodes and they are mostly towards the core of the graph. A random walk started from such a node can spread to a large proportion of the graph, in just a few steps, making it quite difficult to process. Proceeding from the peripheral clusters also decongests the core gradually, making it easier for exploring.

In step (i) of the clustering algorithm described in Section 4.2, the start node chosen was the one with largest degree. In the revised version, we choose the start node as the one with the lowest degree. If there are two or more of them, we choose one of them randomly.

Proceeding when $maxActiveNodeBound$ is reached

When the graph is tightly connected, even if the random walk is started from a node with low degree, the number of active nodes reach the $maxActiveNodeBound$ quite rapidly. This is also accelerated by the presence of a large number of hub nodes, as is the case in the wiki-graph. It is quite difficult to identify a good cluster with very few steps of the walk. Hence, terminating the walk as soon as the bound is reached and emiting the current best cluster, hurts the overall quality of the clustering, especially in the case of the wiki-graph.

In the revised version, we decided to continue the walk even when the bound is reached, but no more new nodes will be added to the set of active nodes. Hence the walk moves only to neighbors which are already explored. In step (iii) of the `Modified-Nibble` procedure (Section 4.2.1), we simulate all the steps of the batch, except that as soon as $maxActiveNodeBound$ is reached, the walk is confined to current active nodes. And in step (vi), we stop only when the total number of steps has crossed $maxClusterSize$, and not when the $maxActiveNodeBound$ is reached.

4.4 Heuristics

As explained in Section 4.2.2, every time the `Find Best Cluster` procedure is invoked, a sorting is done, which can hurt the total time for clustering. From the `Modified Nibble` algorithm (Section 4.2.1), it can be seen that, the procedure is invoked according to the terms of APGP series. Thus, the number of times sorting is done, is proportional to logarithm of $maxClusterSize$, which is acceptable.

Also, as mentioned before, there is no reason for communities to be of similar sizes. Hence `Modified Nibble` procedure returns clusters of sizes ranging from 1 to `MaxClusterSize`. When the size of the cluster becomes very small, in general, the cost of retrieving it increases. Keeping this in mind, when the entire graph is processed, we perform a compaction, which is explained below.

Compaction procedure

(i) Read the clusters one after another.

(ii) If the size of the current cluster is greater than `MaxClusterSize/2`, then emit it.

(iii) Else keep it on hold. If there is already a cluster on hold, then, merge them.

(iv) If on merging in step (iii), the size exceeds `MaxClusterSize/2`, emit the combined cluster as a new cluster.

(v) Repeat from step (i) until all clusters are processed.

Since compaction does a blind merging of small clusters, the final clustering could group together, nodes that may be unrelated.

5 Experiments and Analysis

The proposed `Modified-Nibble` algorithm was implemented in Java and experiments were conducted on the `Digital Bibliography Library Project (dblp)` database graph (2003 version), and the `Wikipedia` datagraph (2008 version). The revised version of the algorithm was also implemented and tested on the above two datasets. The details of experiments are explained in Sections 5.2 and 5.3.

5.1 Details of Datasets

dblp3 database

Tables: `author, cites, paper, writes` (Figure 4)
Number of nodes: 1,771,381
Number of undirected edges: 2,124,938
max degree = 784

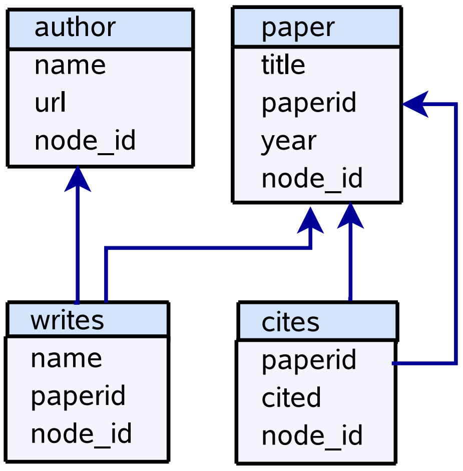

Figure 4: `dblp3` database schema

wiki database

Tables: `document, links` (Figure 5)
Number of nodes: 2,648,581
Number of undirected edges: 39,864,569
max degree = 267,884

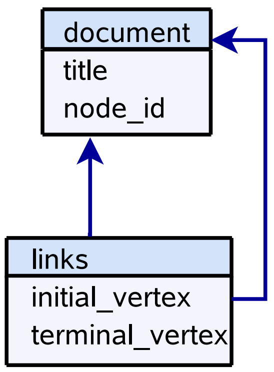

Figure 5: `wiki` database schema

5.2 Results for Modified-Nibble algorithm

5.2.1 Node and Edge compression

$$\begin{aligned}\text{Node Compression} &= \frac{\text{number of nodes in the original graph}}{\text{number of clusters}}\\ \text{Edge Compression} &= \frac{\text{number of edges in the original graph}}{\text{number of inter-cluster edges}}\end{aligned}$$

Node compression is easier to obtain; edge compression is the main indicator of quality of clustering. Higher the edge compression, better the clustering. Table 5 gives the compression ratios for different cluster sizes, on `dblp3` datagraph, and Table 6 gives the same for `wikipedia` datagraph.

maxClusterSize	**# clusters**	**# inter-cluster edges**	**node compression**	**edge compression**
100	24,113	206,040	73	10.31
200	12,698	166,219	139.5	12.78
400	6,709	136,784	264.0	15.53
800	3,505	114,536	505.39	18.55
1500	1,909	90,574	927.91	23.46

Table 5: Compression values for different cluster sizes on `dblp3`. Parameter settings: $a = 2$, $d = 7$, $r = 1.5$, $f = 500$ and with compaction

# clusters	**# inter-cluster edges**	**node compression**	**edge compression**
16,208	12,445,795	163.41	3.203

Table 6: Compression values for `maxClusterSize` = 200 on `wikipedia`. Parameter settings: $a = 2$, $d = 7$, $r = 1.5$, $f = 500$ and with compaction

5.2.2 Average Conductance

$$\text{Average Conductance} \quad = \quad \frac{\sum_{c:cluster} \Phi(c)}{\text{number of clusters}}$$

Conductance (Equation 2.1) was the objective used for the clustering process. Average conductance gives the conductance after the clustering process is done, averaged over all the clusters. Table 7 gives the same for different cluster sizes on `dblp3` datagraph.

maxClusterSize	100	200	400	800	1500
avg conductance	0.0838	0.0689	0.0579	0.0499	0.0401

Table 7: Average conductance values for different cluster sizes on `dblp3`. Parameter settings: *a = 2, d = 7, r = 1.5, f = 500* and with compaction

5.2.3 Effect of the factor `f` on compression and conductance

Factor `f` decides the bound on the number of active nodes. The latter is computed as the product of `f` and `maxClusterSize`. Table 8 summarizes the compression figures obtained for different values of `f` for the `dblp3` database, where `maxClusterSize` is set to 1500. Table 9 gives the average conductance figures for the same. The entry "no bounds" in these tables is for the case where there was no bound on the number of active nodes. The entry "cost" in Table 8 gives the time required for the clustering process on a 2.66GHz Intel Core2 Duo CPU machine with 3 GB RAM, running Fedora 8.

f	**# clusters**	**# inter-cluster edges**	**node compression**	**edge compression**	**cost (approximate)**
100	1,965	105,290	901.46	20.18	1.5 hrs
150	1,946	103,603	910.27	20.51	2 hrs
200	1,945	102,080	910.74	20.82	3 hrs
300	1,934	97,529	915.92	21.79	9.5 hrs
400	1,921	94,872	922.11	22.39	15 hrs
500	1,909	90,574	927.91	23.46	1 day
no bounds	1,862	78,973	951.33	26.91	2.5 days

Table 8: Compression for different values of `f` on `dblp3`, for `maxClusterSize` = 1500. Parameter settings: *a = 2, d = 7, r = 1.5* and with compaction

f	100	150	200	300	400	500	no bounds
avg conductance	0.0474	0.0467	0.0458	0.0436	0.0423	0.0401	0.0344

Table 9: Average conductance for different values of `f` on `dblp3`, for `maxClusterSize` = 1500. Parameter settings: *a = 2, d = 7, r = 1.5* and with compaction

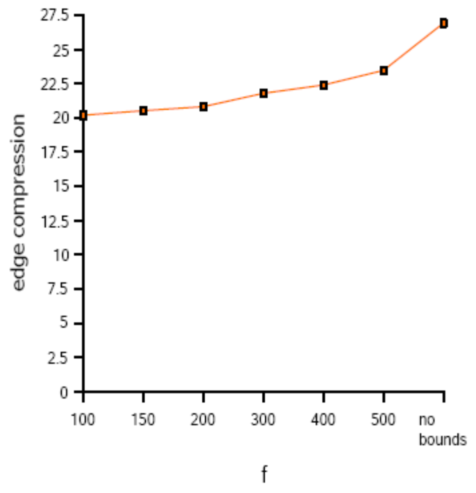

Figure 6: Effect of `f` on edge compression (Table 8)

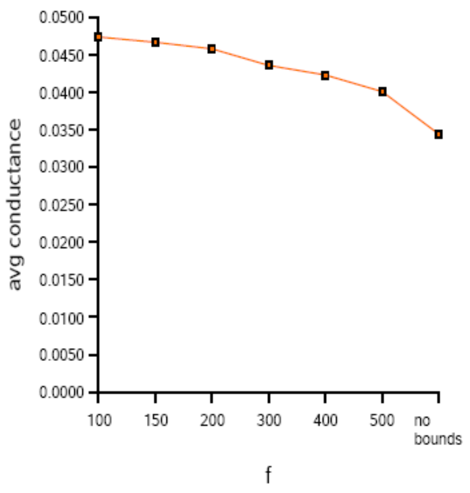

Figure 7: Effect of `f` on avg conductance (Table 9)

5.2.4 Comparison with EBFS Clusters

Edge-weight prioritized breadth first search (EBFS) is a clustering technique which takes weights of edges into account. It chooses an unassigned node as the start-node, and performs a BFS from it, where the neighboring nodes are explored in the order of the weight of the edges connecting them. The search is stopped when the number of explored nodes reach the predefined maximum supernode size. All the explored nodes form a cluster. The process is repeated till all nodes are processed. The BANKS system for external memory datagraphs uses the clusters produced by EBFS ([DKS08]). Figures 8 and 9 compare the edge compression and average conductance obtained by `Modified-Nibble` and EBFS, for different cluster sizes on `dblp3`.

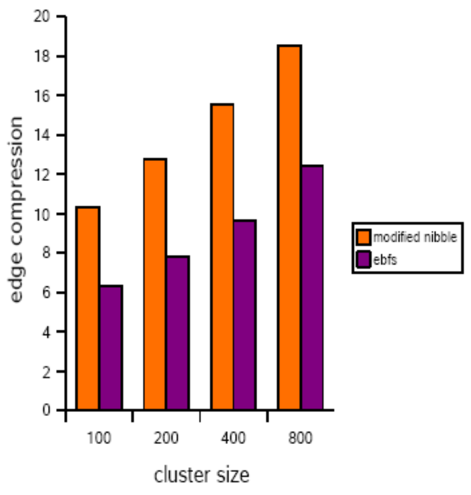

Figure 8: Comparison of edge compression between `Modified-Nibble` and EBFS, on `dblp3` datagraph

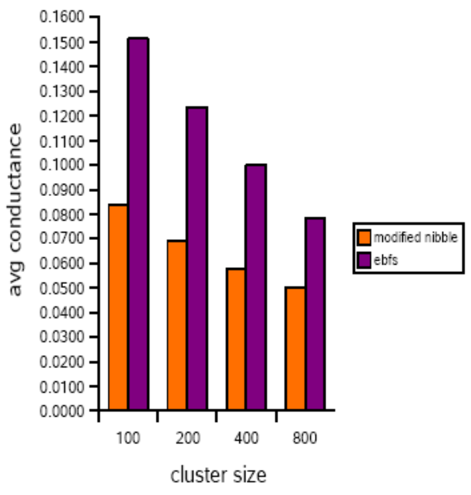

Figure 9: Comparison of avg conductance between `Modified-Nibble` and EBFS, on `dblp3` datagraph

5.2.5 Performance improvement in BANKS

The clusters obtained by `Modified-Nibble` algorithm on the `dblp3` datagraph were incorporated into the BANKS system and their effect on its performance was studied. Table 10 summarizes the improvement in performance when compared to the EBFS algorithm. The values give the percentage reduction in some of the performance metrics for cluster sizes of 100, 200, 400 and 800, averaged over 10 benchmark queries. Details of the performance analysis can be found in [Sav09, Agr09].

	% reduction			
Cluster Size	**100**	**200**	**400**	**800**
CPU + IO time taken	43.6	36.62	45.27	41.04
Number of nodes explored	-0.23	16.22	57.25	49.41
Number of nodes touched	22.85	26.49	59.09	60.47
Number of cache misses	48.71	36.41	39.74	17.53

Table 10: Summary of performance improvement in BANKS system for different cluster sizes - relative performance of Modified-Nibble and EBFS algorithms

5.3 Results for revised version of Modified-Nibble algorithm

The revised version of `Modified-Nibble` discussed in Section 4.3 was tested on the `dblp3` dataset to gauge its impact on clustering. Table 11 gives a comparison between the two on `dblp3` database for `maxClusterSize` of 400.

	Modified-Nibble	**Revised Modified-Nibble**
# clusters	6,709	6,709
# inter-cluster edges	136,784	125,651
edge compression	15.53	16.91

Table 11: Comparison between `Modified-Nibble` and `Revised Modified-Nibble` on `dblp3` datagraph for `maxClusterSize` = 400. Parameter settings: *a = 2, d = 7, r = 1.5, f = 500* and with compaction

max Cluster Size	**# clusters**	**# inter-cluster edges**	**node compression**	**edge compression**	**avg conductance**
200	17,713	11,485,314	149.5	3.471	0.350

Table 12: Compression and Conductance values for cluster size of 200 on `wiki`. Parameter settings: *a = 2, d = 7, r = 1.5, f = 100* and with compaction

5.4 Observations and Analysis

(i) As can be observed from Table 5, both node and edge compression figures improve with larger bound on the cluster size.

(ii) Edge compression on the wiki-graph, given in Table 6, is quite discouraging. The reason behind this could be any of the following:

  (a) Almost all the inherent clusters in wikipedia have their size greater than 200.

  (b) The decision to stop as soon as the `maxActiveNodeBound` is reached is hurting the clustering to a very large extent.

  (c) The community structure of `wikipedia` is different from that of `dblp`, and we are missing out on some important aspect of the former, which is absent in the latter.

(iii) From Tables 8 and 9, it is obvious that larger values of `f` give better clusters. Especially, for the case where there is no bound on the number of active nodes, the algorithm performs its best. But, from this, it cannot be concluded that `f` alone is affecting the quality. From Table 11, it can be seen that on continuing the processing even when the `maxActiveNodeBound` is reached, is giving improvement. Thus, this case requires more tests for confirmation.

(iv) From Figures 6 and 7, it can be noted that the compression and conductance values are comparable for values of `f` ranging from 100 to 500. Sharp improvement is observed only for the case of “no bounds”. In line with the earlier observation, this may be indicative of the fact that `f` is not directly affecting the quality of the clustering. So, lower values of `f` may suffice, also considering the marked reduction in the cost (Table 8).

(v) Figures 8 and 9 show that clusters produced by `Modified-Nibble` give much better compression and conductance values than those produced by EBFS. It can also be noted that, for both the algorithms, decrease in conductance is accompanied by better edge compression. This suggests that, the objective of our algorithm, which is to minimize the conductance of clusters, is in effect, giving better quality clusters. But, it requires experiments on different datasets for confirmation.

(vi) From Table 10, it is quite clear that the clusters found by `Modified-Nibble` is outperforming those found by EBFS, by a large margin. This indicates that clusters with better compression and conductance values, also perform well in the actual search system. However, as reported in [Sav09, Agr09], the number of nodes explored and number of nodes touched increases quite rapidly as the cluster size increases. Preliminary analysis points at the blind compaction step done after the clustering, since this groups together nodes that may be totally unrelated.

(vii) Even with the revised version of the algorithm, the compression and conductance values on wiki-graph have not improved significantly (Table 12). As suggested earlier, it might be due to a yet unrecognized aspect of the community structure of the graph. However, this requires more experiments for confirmation.

6 Conclusions and Future Work

Clustering is the technique of finding the underlying structure of a graph and is a very important area of research. An interesting development in the recent years is the application of random walks on graphs to find good quality clustering. One such method is the graph partitioning technique which uses an algorithm called `Nibble` that approximates the probability distribution of random walks on the nodes of the graph. This algorithm was implemented and its performance was studied. Based on the shortcomings identified, we proposed the `Modified-Nibble` algorithm, which was implemented and tested on the `dblp3` and `wiki` datasets, to understand its performance. Through this exercise, we identified a few shortcomings and it has given us insight into how we should proceed in the future. Following is the proposed direction of future work:

- As seen from the results of the experiments, the clustering algorithm could not perform well on the wiki-graph. The reason behind this could be that the structure of wiki-graph is quite different from dblp3. It is known that wikipedia has large number of hub pages, and disambiguation pages. The latter, usually connects many unrelated concepts together, thus adding to the noise of the structure. Another issue is related to the out-degree of nodes. While the largest out-degree in dblp3 was less than 800, the largest in wiki-graph is more than 200,000. The algorithm must be smart enough to deal with misbehaving nodes, to discover the underlying community structure.

- To deal with the vast disparity in the cluster sizes, the current approach is to compact the clusters found. But, since this is done blindly, it may group together nodes, that are totally unrelated. This can lead to an increase in the query answer time of the search system. A better approach would be to merge clusters that are comparatively related to each other. Similarity between clusters can be judged by the number of edges crossing them.

- Currently, the clustering process takes time in the order of days, for the wiki graph. Though this is not a serious issue since clustering is an offline process, this is still quite important when dealing with larger graphs. Parallelizing the clustering, and running the clustering algorithm in a distributed environment are promising areas of future work.

- Consider a mapping from the set of web pages in WWW to the wiki articles, which is based on some similarity between the two. A webpage may be connected to more than one wiki article. Once the clustering on wiki-graph is in place, then, given such a mapping, it may be possible to cluster the entire web corpus based on the clustering computed for their images in wiki-graph. In general, given a small graph and a mapping of the nodes of a larger graph to nodes in the former, it may be easier to find a good clustering for the latter, by finding a good clustering for the smaller graph.

- Wikipedia has an inbuilt category structure. Categories give a rough grouping of the articles, and are entered by the authors manually. They are organized in a hierarchical fashion. Even though an article can belong to multiple categories, they still can provide some insight into the underlying structure of wikipedia. The clustering algorithm may be able to perform better on the wiki-graph if, in addition to the link structure, it also takes the category structure into consideration. This is an important direction to explore.

References

[Agr09] Rakhi Agrawal. Keyword Search in Distributed Environment. *MTech. Project Stage 2 Report, Indian Institute of Technology, Bombay*, 2009.

[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.

[Cat08] K. Rose Catherine. Clustering. *MTech. Project Stage 1 Report, Indian Institute of Technology, Bombay*, 2008.

[CS07] Nick Craswell and Martin Szummer. Random Walks on the Click Graph. *SIGIR*, 2007.

[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.

[DKS08] Bhavana Bharat Dalvi, Meghana Kshirsagar, and S. Sudarshan. Keyword Search on External Memory Data Graphs. *VLDB*, 2008.

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

[Sav09] Amita Savagaonkar. Keyword Search in Distributed Environment. *MTech. Project Stage 2 Report, Indian Institute of Technology, Bombay*, 2009.

[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.

[wik] Wikipedia. `http://www.wikipedia.org/`.

A Documentation of the Java implementation of `Modified-Nibble` algorithm

Some of the important classes in the implementation are given below.

`ModifiedNibble`	:	implements the `Modified-Nibble` algorithm.
`Graph`	:	this class represents the datagraph used by the clustering process. It handles all graph related operations.
`Cluster`	:	this datastructure represents a cluster. It stores the nodes that belong to it, and has methods to get the properties like volume, cutsize and conductance of the cluster.
`Configuration`	:	provides methods for reading the configuration file provided by the user.
`Utils`	:	provides utility functions, such as quick sort.

Important methods of some of the classes are elaborated below.

A.1 `ModifiedNibble`

`FindClusters`	:	This implements the major portion of the `Modified-Nibble` algorithm. It invokes the APGP series generator with the user specified values for the parameters and then calls the `Graph.NextRandomWalkStep` method for that batch. This is followed by a call to `ModifiedNibble.GetCluster` to get the best cluster. On getting the best cluster, it makes the decision on whether to continue or not. It also keeps track of whether the `maxActiveNodeBound` has reached and whether the total number of walk steps is within `maxClusterSize`.
`ResumeFindClusters`	:	This method is used to resume the clustering process from a previously terminated execution. It reads the partially processed cluster output and creates the remainder graph by a call to `Graph.CreateNewGraph`. It then continues in a similar way as `ModifiedNibble.FindClusters`.
`GetCluster`	:	This implements the `FindBestCluster` procedure. It invokes `Graph.SortOnDegNormProb` method for getting the nodes in the decreasing order of their degree normalized probabilities. It then uses a sweeping method to find the first `j` nodes that give lowest conductance.
`WriteClusterToFile`	:	It outputs the cluster passed as argument, in the required format, to the user-specified outfile.

A.2 Graph

`CreateNewGraph`	:	It removes the nodes present in the cluster provided as the argument, from the current graph and adjusts the degrees of the remaining nodes.
`SetStartNode`	:	This is called just before starting the random walk for a cluster. It sets the `startNode` as the one with highest degree, or the one with lowest degree, as the case may be.
`NextRandomWalkStep`	:	This method performs one step of the random walk on the current graph, from the current set of active nodes.
`GetOutNeighbors`	:	It returns the out neighbors of the node provided as its argument, which are present in the current graph.
`GetActiveOutNeighbors`	:	This method is called only when the `maxActiveNodeBound` has reached. It is similar to the `Graph.GetOutNeighbors` method, except that only active neighbors are returned.
`SortOnDegNormProb`	:	It invokes `Utils.QuickSort` method for sorting the current active nodes on their degree normalized probabilities.

A.3 Data structures

`IntArrayList`	:	implements the `java.util.ArrayList<Integer>` in terms of an array of fixed length, where the maximum required length is known beforehand. It provides `size, get, set, clear, add` and `addAll` methods, similar to the `ArrayList`. This is used to store the current active nodes, since the maximum number of active nodes is limited by the total number of nodes in the graph. It improves the performance by avoiding runtime memory allocation.
`HashCache`	:	it is an instance of `java.util.HashMap<Integer, ArrayList<Integer>>` initialized with `maxActiveNodeBound` as its size. It is used for caching the current active out-neighbors of a node. When `Graph.GetActiveOutNeighbors` method is called for a node which is not already entered in the hashmap, it is retrieved from the current graph. This is then entered in the hashmap. Next time the active out-neighbors of this node is requested, it is retrieved directly from the hashmap. Every time `Graph.CreateNewGraph` is invoked, the hashmap is cleared. It was observed that the time taken by `Graph.GetActiveOutNeighbors` decreased from 78% to 29% of the overall processing time.