Graph Clustering for Keyword Search

Rose Catherine K.* S. Sudarshan

Indian Institute of Technology Bombay
rosecatherinek@gmail.com, sudarsha@cse.iitb.ac.in

Abstract

Keyword search on data represented as graphs, is receiving lot of attention in recent years. Initial versions of keyword search systems assumed that the graph is memory resident. However, there are applications where the graph can be much larger than the available memory. This led to the development of search algorithms which search on a smaller memory resident summary graph (supernode graph), and fetch parts of the original graph from the disk, only when required. In this scenario, good clustering of nodes into supernodes, when constructing the summary graph, is a key to efficient search.

In this paper, we address the issue of graph clustering for keyword search, using a technique based on random walks. We propose an algorithm, which we call Modified Nibble clustering algorithm, that improves upon the Nibble algorithm proposed earlier. We outline several policies that can improve its performance. Then, we compare our algorithm with two graph clustering algorithms proposed earlier, EBFS and kMetis. Our performance metrics include edge compression, keyword search performance, and the time and space overheads for clustering. Our results show that Modified Nibble outperforms EBFS uniformly, and outperforms kMetis in some settings. Further, the memory requirements of our algorithm are much lower than that of kMetis, making it practical even with a very large number of nodes, unlike kMetis.

1 Introduction

Keyword search on data represented as graphs, has become a topic of great interest in recent years. It can be attributed to the following two factors, to some extent: graphs can represent all forms of data - structured, semi-structured, and unstructured, along with relationships between its various entities. And, keyword searching allows users to query the data, without knowing any particular query language, or the underlying schema used to format the data.

Initial algorithms for keyword search, such as BANKS [3], assume that data is memory resident. But there are applications where the data can be much larger than the available memory. This led to the development of external memory search algorithms, such as Incremental Expanding Backward Search [7], which searches on a smaller memory resident supernode graph, to minimize IO. It fetches parts of the original graph from memory, when required. The efficiency of search in this case, can be improved by using a good clustering of the graph nodes.

Graph clustering is the process of grouping graph nodes, such that most edges are inside individual clusters, and inter-cluster edges are comparatively few. Clustering is an already well researched topic. Some popular classes of clustering algorithms are geometric, hierarchical and partitioning methods. However, not all of them can be used for graph clustering. For graph clustering, some of the popular algorithms are kMetis [12], and EBFS (used in Incremental Expansion Search algorithm [7]).

In this paper, we focus on clustering based on communities. Below, we describe briefly the concept of communities and the intuition for community based clustering.

A community is a set of real-world entities that form a closely knit group. Communities provide a natural division of graph nodes into densely connected subgroups [14]. Since nodes within a community are closely knit together, a keyword search started from one of its nodes, will remain within its boundary to a large extent, thus localizing the search. In addition, since inter-community connections are weak, the supernode graph produced will be sparse, which in turn, will restrict the spread of the search to a small fraction of the entire graph. Also, since there is no reason why communities must be of similar sizes, while clustering, we don't have to force equal partitioning of the nodes.

By dividing the data in accordance with the underlying community structure, and storing them in the same or adjacent disk blocks, or in the same machine if the data is distributed across machines, related data can be retrieved together. This can enable external

* Current affiliation: IBM IRL, Bangalore

*15th International Conference on Management of Data
COMAD 2009, Mysore, India, December 9–12, 2009
©Computer Society of India, 2009*

memory or distributed keyword search to produce answers in less time.

The contributions of this paper are as follows:

1. We propose an algorithm called Modified Nibble, for graph clustering, using the technique of random walks. Our algorithm improves upon a community based clustering algorithm proposed earlier, called the Nibble algorithm [16], avoiding several drawbacks of the Nibble algorithm which we detail in Section 4.4.

2. We outline, in Section 5.3, several policies that can improve the performance of our proposed algorithm.

3. We compare Modified Nibble with two graph clustering algorithms proposed earlier, EBFS and kMetis. Our experimental results in Section 6 show that Modified Nibble is able to outperform EBFS consistently, and outperform kMetis in some metrics. In particular, the memory requirements of our algorithm are much lower than that of kMetis, making it practical even with a very large number of clusters, unlike kMetis. This is particularly important for our target application of keyword search on graphs, where the number of clusters can be of the order of tens or hundreds of thousands.

2 Related work

There is a large amount of prior work on graph clustering, which can be broadly classified into methods based on graph-partitioning and methods based on finding communities.

The objective of graph partitioning methods is to minimize the number of cut edges, while distributing the nodes into partitions of roughly the same size. An example is kMetis proposed in [12]. Partitioning algorithms process the graph in a top-down fashion. The graph is initially partitioned into two. Then, the procedure is repeated on these partitions for a fixed number of times, to obtain the required number of clusters.

In the paper [14], Newman and Girvan describe a divisive hierarchical clustering algorithm for finding communities, which uses a 'betweenness' measure to identify the edges to be removed. Betweenness is a measure which favors edges that lie between communities and disfavors those that lie inside communities. After every removal, the betweenness measure has to be recalculated since the betweenness values for the remaining edges will no longer reflect the situation in the new graph.

Duch and Arenas propose a divisive algorithm in [9], to find the community structure in graphs by extremal optimization of modularity. Modularity is a community goodness measure, computed as 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 [8]. An improved version of this algorithm is proposed in [17].

In [18], van Dongen describes Markov Cluster algorithm (MCL) which finds clusters by simulating a flow within the graph. It alternately strengthens the flow where it is already strong, and weakens it where it is weak. This is repeated until convergence, to get a number of regions with strong internal flow (clusters), separated by dry boundaries with no flow.

Bader et al. propose a clustering algorithm called MCODE in [2], for molecular complex prediction. It uses a vertex-weighing scheme based on the clustering coefficient which measures the density of the neighborhood of a vertex.

Spielman et al. [16] propose a partitioning algorithm which uses random walks on graphs to find good clusters. A modified version of the same is described in [1]. A detailed discussion of the key ideas of both these algorithms is in Section 4.3.

There are many other algorithms that are used for clustering, but which cannot be applied to the problem at hand, namely graph clustering for keyword search. K-means is a method that comes under the class of geometric clustering methods, which optimizes 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 ([5]). Rastogi et al. [13] suggest a graph compression method which exploits the similarity of the link structure present in the graph to realize space savings. The graph summary is similar to a supernode graph, but has a slightly different semantics for superedges. Each superedge represents edges between **all** pair of nodes belonging to each of the supernodes.

3 Clustering for finding communities

A community is a set of real-world entities that form a closely knit group. As mentioned in [8], it is a way to analyze and understand the information contained in the huge amount of data available today.

Finding communities can be modeled as a graph clustering problem, where vertices of the graph represent entities and edges denote relationships between them. However, when clustering is done to discover the community structure, no emphasis is given to creating clusters of similar sizes, though sometimes it is appropriate to upper bound and/or lower bound the cluster size. In addition to that, since communities provide a natural partitioning of the graph, the users are not required to input the number of clusters in the graph, beforehand.

3.1 Quantifying the goodness of community structure using conductance

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. We use the conductance of clusters (defined below), for this purpose.

Graph conductance (as given in [1]), also known as the normalized cut metric, is defined as follows:

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 $\partial(S) = \{\{u, v\} \mid \{u, v\} \in E, u \in S, v \notin S\}$. The cutsize is denoted by $|\partial(S)|$.

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

$$\Phi(S) = \frac{|\partial(S)|}{min(Vol(S), Vol(\bar{S}))} \tag{1}$$

4 Finding communities using random walks on graphs

A random walk is a graph traversal technique, which starts from the designated *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.

4.1 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 ([6]). 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.

In the initialization step prior to the walk, *nodeProbability* of the *startNode* is set to 1 and probabilities of the rest are set to 0. During the walk, at each step, each node which has a non-zero value for its *nodeProbability* will divide its current value, equally between its neighbors - this is called *spreading of probabilities*. Nodes with non-zero values for *nodeProbability* are said to be *active*. If a node receives activation from multiple neighbors, they are accumulated. At any step of the walk, all nodes have non-negative probabilities and they add up to 1. All these conditions may not always be enforced, to suit the problem at hand.

4.2 Rationale

The core idea of random-walk based clustering techniques is that a walk started from a particular node will remain within the cluster enclosing 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 [6], 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.

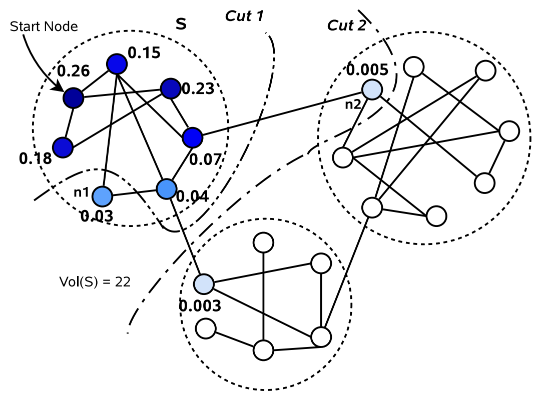

Figure 1: Example for clustering based on random walks

Consider the toy example given in Figure 1, 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 probabilities drop suddenly, thus revealing the boundary. This notion is used in the algorithm for clustering using seed sets, proposed by Andersen and Lang in [1].

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, the conductance measure comes to our rescue.

In the example of Figure 1, the preferred cluster (S) 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. Thus, the preferred cluster has the lowest conductance.

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 [16], and the Modified Nibble algorithm proposed by us in this paper.

4.3 Clustering using Nibble algorithm

Spielman and Teng [16] describe a nearly-linear time algorithm, Partition, for computing crude partitions of a graph, by approximating the distribution of random walks on the graph. The main procedure of the proposed clustering method is the Nibble algorithm, which, given a start node, finds the cluster that encloses that node. The walk allows self-transition with 50 percent probability and otherwise, moves along one of the randomly chosen edges incident on the vertex, to its neighbor. To speed up the procedure, it employs truncated random walks.

Nibble algorithm (outline):

input: Start node v, Graph G, Max Conductance θ_0

1. Compute the bound on maxIterations, t_0, and threshold, ϵ.
2. Start spreading probabilities from v.
3. When the *nodeProbability* falls below ϵ, truncate the walk by setting it to 0.
4. Sort the nodes in the decreasing order of their degree normalized probabilities.
5. Check if a j exists such that:
   - Conductance of the first j nodes in the sorted order, is lesser than or equal to θ_0
   - The above group of nodes satisfy a set of predefined requirements on its volume.
6. If a j was found, then return the first j nodes of the sorted set, as the enclosing cluster of v.
7. Otherwise, do the next step of spreading probabilities and repeat from Step 3.

Partition uses the Nibble procedure to nibble out clusters from the graph, which it merges, till the volume of the merged set becomes a predetermined fraction of the entire graph. At this point, the graph is partitioned into two, with the merged set forming one partition, and the remaining nodes forming the other. To cluster the graph, this procedure is repeated for a fixed number of times, on the partitions obtained. Clustering using Nibble algorithm processes the graph in a top-down fashion, which could make it difficult for large graphs.

The clustering algorithm proposed by Andersen and Lang in [1], uses the Nibble algorithm with minor modifications, to find the enclosing community for a 'seed set' of nodes. But, the seed set was chosen manually (after identifying the target cluster), which limits its application, to cases where the communities present in the graph are known beforehand.

4.4 Shortcomings of Nibble

Based on our implementation of the Nibble algorithm and the experiments conducted on sample datasets, we identified the following shortcomings of the algorithm.

1. Nibble was not able to find all clusters, for reasonable values of conductance, even on moderate sized graphs: If the conductance was increased, the cluster-quality was badly affected. And if the conductance was reduced, it pulled a large number of nodes into the cluster. We tried rectifying the latter behavior, by bounding the maximum size of the clusters. With this bound in place, Nibble was not able to find all the clusters.
2. Specifying the conductance of the clusters, a priori, is difficult.
3. In step 5 of Nibble procedure, any value of j that satisfies the conditions, is accepted. Due to this, the algorithm might terminate early, as soon as a cluster of user-specified conductance is found, and may miss out better clusters existing in the graph.
4. 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, user has no way of regulating the cluster size.
5. 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.
6. Testing for good community, which involves sorting the nodes, is done after each step of spreading of probabilities, and could lead to considerable overheads when clustering large graphs.

5 Modified Nibble clustering algorithm

Keeping in mind the ideas suggested by Spielman and Teng in [16], and Andersen and Lang in [1], and based on the shortcomings identified, we propose the Modified Nibble clustering algorithm, which is discussed in this section.

```
Overall clustering algorithm
input: Graph G, maxClusterSize

(1) Set $G' = G$. But, if co-citation heuristic P9 is used, set $G'$ to the remainder graph, after removing hub nodes.
(2) Choose start node $n_s$ according to P1.
(3) Obtain cluster $C_s$ = ModifiedNibble($n_s, G'$)
(4) Set $G' = G' - C_s$, and save $C_s$.
(5) Repeat from step (2), until $G'$ is null.
(6) Compact the clusters obtained, using P8 procedure.
```

Figure 2: The overall clustering algorithm

The proposed method of clustering is composed of 3 procedures - the overall clustering algorithm (Figure 2), the Modified Nibble algorithm (Figure 4) and the Modified FindBestCluster algorithm (Figure 3). Input to the clustering algorithm consists of the graph G, and a user-specified upper bound on the size of clusters, `maxClusterSize`. The algorithm works on undirected graphs; it can be applied to directed graphs by simply ignoring edge directions.

The algorithm is parametrized by a number of parameter setting and policies, which are referred to as P1, P2, and so on. These parameter settings and policies are described later, in Section 5.3.

5.1 Algorithm

The overall clustering (Figure 2) proceeds by identifying and removing one cluster at a time, rather than processing the entire graph at once. This could be beneficial for clustering massive graphs.

The core of our clustering method is the Modified Nibble procedure (Figure 4). It explores the locality of the specified start node, by performing random walks on the remainder graph (that is, the part of the graph not yet assigned to any cluster). The spread of this random walk is limited to the locality of the start node, by constraining the maximum number of steps of the walk and the maximum number of active nodes at any time of the walk.

The maximum conductance of the clusters is not a user-input, unlike for the original Nibble algorithm. The Modified Nibble algorithm instead finds the best cluster for the given start node. This is done by observing the fall in conductance of the current best cluster, as the walk progresses. As long as the conductance diminishes, it continues the walk. When the conductance value starts to increase, it stops and returns the one with lowest conductance.

The Modified FindBestCluster procedure (Figure 3), is internally invoked by Modified Nibble, to find the best available cluster out of the current active nodes. This involves sorting of the nodes. Unlike in the case of the original Nibble algorithm, sorting is invoked only when a batch of random walk steps has been done; thus the impact of sorting on the time required for the clustering process is minimized. The size of the batch is a parameter (P4).

```
Modified FindBestCluster
input: set activeNodes, graph $G'$, maxClusterSize

(1) normalize the nodeProbability of all nodes in activeNodes, with their degree in $G'$
(2) sort the nodes in activeNodes set, in the decreasing order of their degree-normalized nodeProbability.
(3) define candidate clusters $C^j$ to be the set of nodes from 1 to $j$, in the sorted order, where $j = min$(maxClusterSize, |activeNodes|).
(4) if abandoned node heuristic is set to P10(b), then do the following:
    • set each $C^j$ to $C^j \cup \{n_c \mid n_c \text{ is abandoned by } C^j\}$
    • if for any $j$, $|C^j|$ exceeds maxClusterSize, discard $C^j$.
(5) for all remaining candidate clusters, compute the conductance w.r.t $G'$.
(6) return that candidate, which has the smallest conductance, out of all the remaining candidate clusters, as the best cluster.
```

Figure 3: Modified FindBestCluster algorithm

Before delving into the details of parameters and policies, we explain the working of the two main procedures, namely Modified Nibble and Modified FindBestCluster, using a simple example in the next section.

5.2 Sample execution of Modified Nibble algorithm

Consider the (very small) example graph in Figure 5, with the start node as indicated. The cluster marked as S is the cluster for this particular start node. In this figure, we have performed one step of random walk, and this forms `Batch 1`. The best cluster amongst the current set of active nodes, is the one with all the 4 active nodes and its conductance is 0.33. Since, the intuitive cluster S has not been found yet, we continue with the spreading of probabilities.

Figure 6 shows the probability distribution after 3 steps (`Batch 2`). Here, the probability has spread beyond S. But, it can be observed that, amongst the active nodes, those with largest *nodeProbability* belong to S.

To decide on the number of nodes in the sorted order that should be taken as a cluster, we check the conductance of each of the sets formed. Figure 6 shows two cuts in the graph, in addition to S. $Cut1$ corre-

```
ModifiedNibble
input: start node $n_s$, Graph $G'$, maxClusterSize

(1) initialization:
      • set nodeProbability of $n_s$ to 1 and add it to the activeNodes set.
      • set maxSteps according to P5.
      • calculate maxActiveNodeBound using P6.
      • set totalSteps to 0.
(2) Batch i:
    initialization:
      • get term $t_i$ from the series chosen by P4.
      • set batchSteps to ($t_i$ - totalSteps).
      • but if $t_i$ exceeds maxSteps, set batchSteps to (maxSteps - totalSteps).
    do for batchSteps number of times:
      (a) spread from all nodes in activeNodes or a single node, according to P2.
      (b) the amount of spreading is determined by spreadProbability as chosen in P3.
      (c) update nodeProbability of all nodes, with the probabilities accumulated from
          their neighbors.
      (d) update activeNodes set to contain all nodes with positive values for their
          nodeProbability.
      (e) if number of active nodes are bounded, check if maxActiveNodeBound has been
          reached. If yes, then, according to the choice of P7, do as below:
            • P7(a) : stop this batch, and proceed directly to step 3.
            • P7(b) : continue this batch, but in step 2(a) above, spreading is done to
              only those nodes, which are already in activeNodes.
(3) obtain cluster $C_i$ = ModifiedFindBestCluster(activeNodes, $G'$).
(4) find conductance of $C_i$ w.r.t the current graph $G'$, $\Phi_{G'}(C_i)$.
      • if $\Phi_{G'}(C_i) \geq \Phi_{G'}(C_{i-1})$, set $C_{best}$ to $C_{i-1}$, and go to step 6.
      • else, set $C_{best}$ to $C_i$
(5) do the following and repeat from step 2 onwards (Batch i+1).
      • if $t_i$ exceeds maxSteps, go to step 6.
      • else, set totalSteps to $t_i$.
(6) if abandoned node heuristic is set to P10(a), set $C_{best}$ to $C_{best} \cup$
          $\{n_c \mid n_c \text{ is abandoned by } C_{best}\}$
(7) return $C_{best}$ as the best cluster of $n_s$.
```

Figure 4: Modified Nibble algorithm

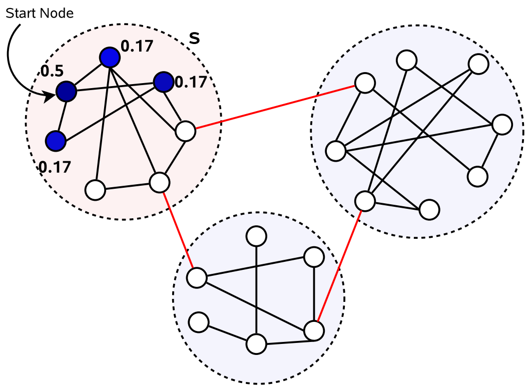

Figure 5: Probability distribution after 1 step

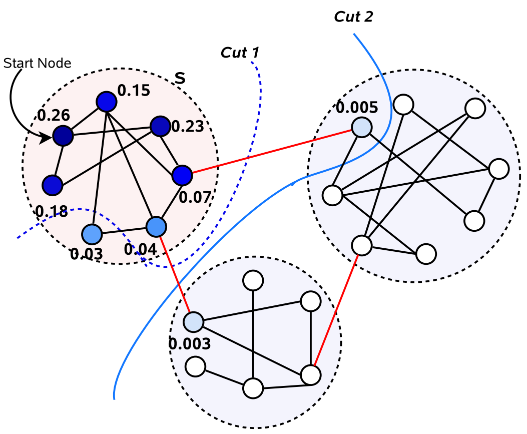

Figure 6: Probability distribution after 3 steps

sponds to the case where we choose the first 6 nodes in the sorted order. Its conductance is 0.2. *Cut*2 corresponds to choosing the first 8 nodes, and its conductance is 0.12. Choosing the first 7 nodes forms the cluster S, whose conductance is 0.09. Here, S has the lowest conductance and hence, is the best cluster for `Batch 2`. Note that, conductance of the best cluster has lowered when compared to `Batch 1`. But, at this point, it is not possible to determine if S is the best, over all clusters for the start node. Hence we continue with the random walk.

Figure 7 shows the probability distribution after 5 steps of random walk (`Batch 3`). Here, the probability has spread to a larger fraction of nodes in the graph. But, note that, much of the probability is still within S. In this figure, we consider two more cuts (in addition to the cuts that we inspected in Figure 6). *Cut*3 has the first 9 nodes in the sorted order, which includes all the nodes in S and its conductance is 0.14. *Cut*4 has the 10^{th} node added to *Cut*3, making its conductance 0.18. (In total, there are 17 cuts in this graph, but for the ease of illustration, we are considering only a few). Once again, S has the lowest conductance out of all cuts. At this point, we stop and return S as the cluster for the specified start node.

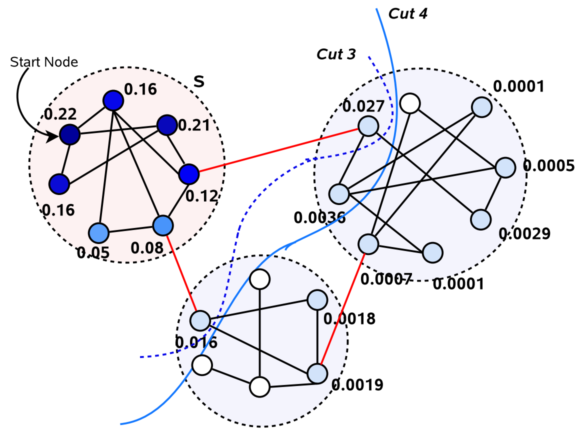

Figure 7: Probability distribution after 5 steps

An important observation in Figure 7 is that, at the point when we finalize on the cluster for the start node, there are nodes in the graph, that have not been touched yet. This illustrates our intuition that, it is possible to find clusters by inspecting only a local neighborhood of the start node and without exploring the entire graph.

5.3 Parameters and Policies

The key ideas of the proposed algorithm were demonstrated in Section 5.2. But the detailed algorithm presented in Section 5.1 uses several parameters and policies which we discuss in this section.

P1. **Start node:** a node from the remainder graph, from where exploring the graph will start. Two choices are the node with the maximum degree, the or node with the minimum degree, in the remainder graph.

P2. **Nodes spreading in each step**: spreading of probabilities in each step, can be done in the following ways:

  (a) Spread from all active nodes.

  (b) Only a single node spreads in each step, in which case, it spreads only that probability that it received and which has not been spread yet.

P3. **Self-transition probability of a random walk** is determined by the parameter, `spreadProbability`. Lower values of the parameter tend to over-emphasize proximity to the start node, while higher values can blur the cluster boundary rapidly, by allowing a larger fraction of probability to escape the boundary. For most of the experiments, it was set to 0.5.

P4. **Number of iterations in a Batch** is chosen from the APGP series, described below:

Arithmetic Plus Geometric Progression (APGP): i^{th} term of an APGP series, $t_i^{apgp} = (a + id) + (a\, r^i)$. The parameters a, d and r, can be used to get fine-grained control over the difference between successive terms of the series.

P5. **Upper bound on number of random walk steps** was chosen to be `maxClusterSize`, which ensures that, all nodes of a cluster whose diameter is `maxClusterSize`, are touched before spreading of probabilities is discontinued.

P6. **Upper bound on number of active nodes:** The walk, if left unchecked, can spread to the entire graph. According to the intuition for random walk based clustering (Section 4.2), it is possible to extract a cluster by exploring only a local neighborhood of the start node. We restrict the size of this neighborhood to be within `maxActiveNodeBound`, calculated as `f` $\times$ `maxClusterSize`.

P7. **Behavior on `maxActiveNodeBound`:** If the number of active nodes is restricted using P6, then, when the number of active nodes reach the `maxActiveNodeBound`, there are two options:

  (a) Stop processing and output the best cluster obtained so far.

  (b) Continue with spreading, but propagate to only those nodes that are already active, so that no more new nodes get added to the `activeNodes` set.

P8. **Compaction procedure:** Modified Nibble procedure may return clusters of sizes much smaller than `MaxClusterSize`, thus increasing the number of supernodes in the summary graph. To avoid this, multiple clusters can be bundled together. While experimenting with many compaction methods, which combined clusters that had edges between them, it was observed that, they made the supernode graph denser, which is detrimental to the search algorithm. To strike a balance between the number of supernodes and the denseness of the supernode graph, we use the following procedure:

Naïve compaction of tiny clusters: In this compaction method, only tiny clusters which do not have any cut edges are combined. Note that, performing this compaction will not affect the number of cut edges.

P9. **Co-citation heuristic:** Co-citation of articles A_1 and A_2 is said to occur, when another article C links to both A_1 and A_2. If all nodes co-cited by a large number of articles are assigned to a single cluster, all edges to them will be condensed to a very few superedges, thus giving higher edge compression. However, it was observed that, using this heuristic created a number of short-cut paths in the supernode graph.

P10. **Abandoned Node heuristics:** In Nibble algorithm (Section 4.3), the candidate clusters were generated by considering the graph nodes only in the order of their increasing probabilities. In experiments conducted on sample datasets, it was observed that, due to this, there were a large number of nodes, which were separated from all neighbors, in the clustering. We will refer to such nodes as *abandoned* nodes. Presence of abandoned nodes hurt the search by faulting for many more supernodes than necessary.

To eliminate abandoned nodes, following two methods were tried out:

(a) **Post-process:** After each cluster is found, check for abandoned nodes and add them to the cluster.

(b) **Abandoned node awareness:** Prevent abandoning of nodes right from the creation of candidate clusters, by adding all abandoned nodes to the candidate clusters. Candidates whose size goes beyond `maxClusterSize` are discarded.

It is obvious that using P10(a) can increase the size of the final cluster beyond `maxClusterSize`. But, P10(b) will produce abandoned-node-free clusters of size within the `maxClusterSize` parameter.

5.4 Final settings for Modified Nibble clustering

Table 1 specifies the parameter values and policy decisions for the implementation of the Modified Nibble clustering algorithm, that we used for comparison purposes (Section 6). These choices were made after a thorough evaluation of their effect on edge compression, time taken for clustering and keyword search performance. A detailed analysis can be found in [11].

The time complexity of the Modified Nibble Algorithm is linear in the number of nodes of the graph, and polynomial in `maxClusterSize`, `maxActiveNodes` and the highest degree of any node in the graph.

6 Comparison with other clustering algorithms

In this section, we compare the implementation of Modified Nibble clustering algorithm (ModNib for short), with the parameter values and policy decisions specified in Table 1, against alternative clustering algorithms.

P1	:	max degree
P2	:	(a) - all active nodes
P3	:	0.5
P4	:	APGP series with $\mathtt{a} = 2$, $\mathtt{d} = 7$, $\mathtt{r} = 1.5$
P5	:	`maxClusterSize`
P6	:	$\mathtt{f} = 500$
P7	:	(a) - stop on `maxActiveNodeBound`
P8	:	naïve compaction of tiny clusters
P9	:	not used
P10	:	(b) - abandoned node awareness

Table 1: Parameter values and policy decisions for the implementation of Modified Nibble clustering algorithm

6.1 Clustering algorithms compared

- Edge-weight prioritized breadth-first-search (EBFS): 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. External memory keyword search in BANKS currently uses EBFS clusters [7].

- kMetis: It is a k-way graph partitioning algorithm, proposed by Karypis and Kumar in [12]. Firstly, it coarsens the graph, by collapsing edges and grouping nodes, thus creating multiple versions of the input graph. Then, on the smallest graph, it finds a good partition. This partition is then projected back onto the original graph, by refining at the intermediate levels.

6.2 Experimental setup

Following are the datasets that we used:

- Digital Bibliography Library Project database (2003 version): dblp3 has 4 tables, viz. `author, cites, paper` and `writes`. Each tuple is represented by a node, and foreign-key constraints, by edges. This resulted in 1,771,381 nodes and 2,124,938 edges in the graph.

- English Wikipedia (2008 version): Nodes in the Wikipedia graph represent articles, and edges represent hyperlinks between corresponding articles (multiple links were ignored). Category pages and redirects were removed while constructing the graph. The Wikipedia graph has 2,648,581 nodes and 39,864,569 edges.

Experiments on dblp3 were conducted on a machine with two 3.00GHz Intel Pentium CPUs with a combined RAM of 1.5 GB, running Ubuntu 9.04. Experiments on Wikipedia were conducted on a blade of eight

2.50 GHz Intel Xeon CPUs, with a combined RAM of 8 GB, running Debian 4.0. The results presented for performance experiments were taken on a cold cache.

6.3 Comparison metrics

The following metrics are used for comparison:

1. Edge compression
2. Connection query performance
3. Near query performance
4. Time and space requirements for clustering

6.4 Edge compression

Edge compression is the ratio of number of edges in the original graph to that in the supernode graph. Node compression can be defined in a similar way, but since it is easier to obtain, we do not use it as a metric. In this section we compare Modified Nibble clustering algorithm with EBFS and kMetis, with respect to the edge compression obtained on sample datasets.

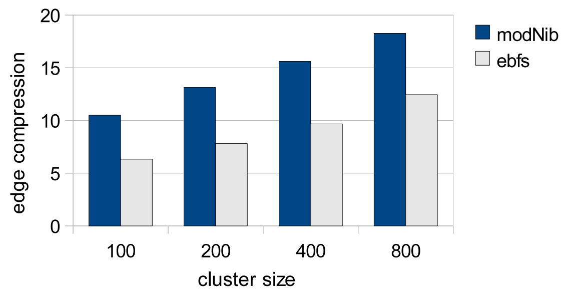

Figure 8: Comparison of edge compression on dblp3 between ModNib and EBFS

Figure 8 compares the edge compression values obtained by ModNib with EBFS, on the dblp3 dataset. It is quite obvious from the figure that ModNib is able to achieve better edge compression than EBFS.

Unlike EBFS, the input parameter of kMetis is much different from that of ModNib. For kMetis, the size of the clusters have to be controlled indirectly through k, which is the number of required clusters. This makes their comparison difficult. However, for comparison purposes, we use clusterings whose maximum cluster size and number of clusters (and hence, average cluster size) are comparable.

	#clusters	**max Cluster-Size**	**edge compres-sion**
ModNib	31,215	400	15.6
kMetis	30,000	335	9.616

Table 2: Comparison of edge compression on dblp3 between ModNib and kMetis

	#clusters	**max Cluster-Size**	**edge compres-sion**
ModNib	11,305	1,600	17.3
kMetis	3,000	1,096	15.7
kMetis	4,000	16,353	9.13

Table 3: Comparison of edge compression on wiki between ModNib and kMetis

From Table 2, it can be seen that the edge compression obtained on dblp3 by ModNib is much higher than that obtained by kMetis.

Table 3 gives the compression values on wiki, for the two algorithms. For kMetis, observe that, when the number of clusters (k) increases to 4000 from 3000, the compression falls sharply. Also, it creates a cluster of size 16,353 while the average cluster size is about 660, despite the claim that it partitions the graph into clusters of roughly the same size. Comparing with ModNib, we see that, it is able to get a compression of 17.3, with 11,305 clusters.

For both dblp3 and wiki, we were unable to run Metis for larger values of k than those reported in the Tables 2 and 3. This is discussed in Section 6.7.

6.5 Connection query performance

A connection query is a keyword query, except that the result returned is a sub-graph or a tree, which shows how the keywords are connected in the original graph. e.g. `krishnamurthy parametric query optimization`. The algorithm used in our performance measurements is the Incremental Expansion Backward search algorithm described in [7]. The answers obtained for different clusterings are the same, regardless of the clustering algorithm. Also, a cache manager was used to cache the expanded supernodes read from the disk. The size of the cache was set to the number of supernodes in the clustering, so that the measurements are not affected by the cache replacement policy.

ModNib and EBFS clusterings used for performance measurements have `maxClusterSize` set to 400. For the kMetis clustering, `k` was set to 30,000[1], which gave a maximum cluster size of 335. The ModNib clustering has 31,215 clusters, which makes the two clusterings comparable.

Figures 9 and 10 show the number of cache misses and the combined CPU and IO time respectively, for answering a set of 20 sample connection queries, using

[1]For connection and near queries, the algorithm initially searches on the supernode graph. It was observed that, if the number of supernodes is small, the supernode graph is very dense; as a result the search spreads to a very large fraction of the graph, and produces a large number of answers in the wrong order, both of which greatly increase the time to answer a query. For the dblp3 graph a supernode graph with about 30,000 clusters gave the best results.

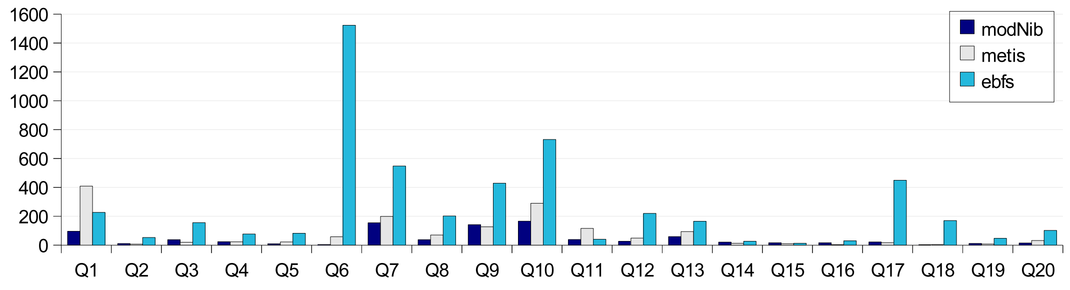

Figure 9: cache misses : connection query on dblp3

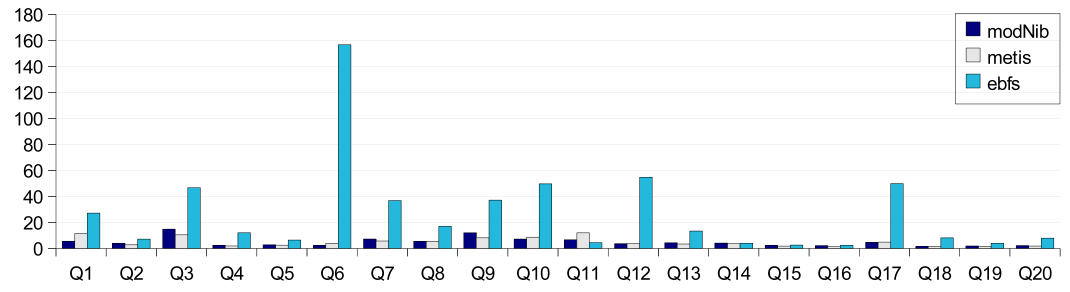

Figure 10: CPU + IO time (sec) : connection query on dblp3

ModNib, kMetis and EBFS clusters. From the figures, it can be seen that, ModNib is out-performing EBFS by a very large margin for most of the queries.

Comparing the performance of ModNib and kMetis, we see that kMetis is performing well on some connection queries, while ModNib is outperforming kMetis on others. At the same time, none of the clustering algorithms, is a clear winner over the other. When the clusterings are on par with each other, the difference in performance could be attributed to the particular queries under consideration, since, eventually, the performance depends on the clusters in which the keyword nodes appear. The sample queries used and the detailed analysis of the results can be found in [11].

6.6 Near query performance

An example of a near query is '`author (near data mining)`'. Here, `author` defines the *type* of answer required by the user. `data` and `mining` are keywords (which form the near set), to which the user wants the author to be close to, in the graph. Intuitively, if an author is close to multiple nodes containing the specified keywords, that author would rank higher in the result of the near query.

An algorithm for near query answering in memory-resident graphs is given in [10]. A simple extension of the same for external memory graphs is described in [11]. We use the latter algorithm for comparing the performance of ModNib, kMetis and EBFS clusters, on a set of 20 sample near queries. Clusters used are same as those used for connection query performance measurements (described in Section 6.5). Similar to connection queries, here too, the answers remain the same, regardless of the clustering algorithm. A cache manager was used in these experiments also, to cache the expanded supernodes, with the size of the cache set to the number of supernodes.

Comparison of near query performance between the three algorithms, can be found in Figures 11 and 12. From Figure 11, it can be seen that ModNib has the lowest number of cache misses out of the three, with a very large margin when compared to kMetis.

Figure 12 shows the combined CPU and IO time taken for answering the sample queries, using the three clustering algorithms. It is quite obvious that out of the three algorithms, ModNib performs the best, which in turn can be partially explained by the lower number of cache misses (Figure 11). The sample queries used and the detailed analysis of the results can be found in [11].

6.7 Time and space required for clustering

The graphs of dblp3 and wiki are of sizes 132 MB and 1.9 GB, respectively (in our representation). Table 4 gives the time and space requirements of ModNib.

dataset	**time**	**space**
dblp3	$\sim$ 1.5 hrs	190 MB
wiki	$\sim$ 1.5 days	2 GB

Table 4: Time and space requirements for ModNib

From Table 4, it is clear that the space requirements of ModNib is very close to the size of the graph. Also, it was found that the difference in time and space required for different `maxClusterSize`, is negligible.

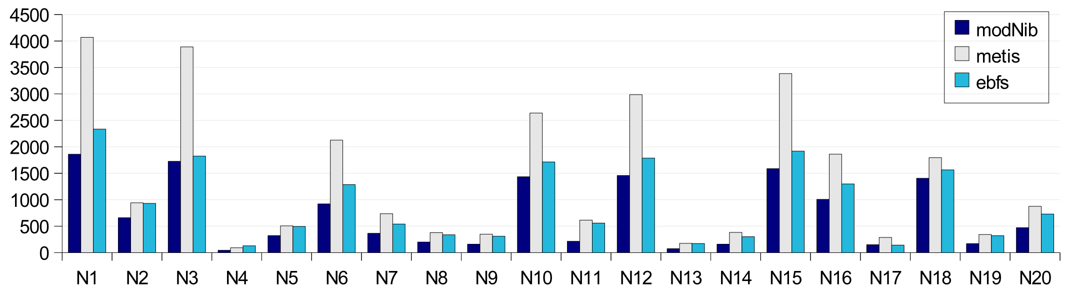

Figure 11: cache misses : near queries on dblp3

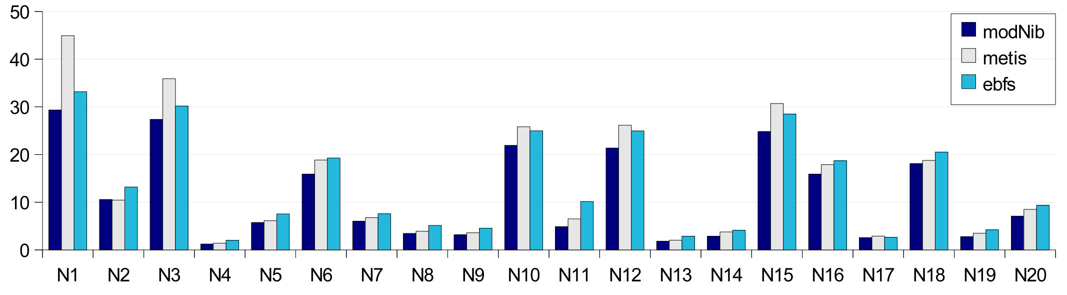

Figure 12: CPU + IO time (sec) : near queries on dblp3

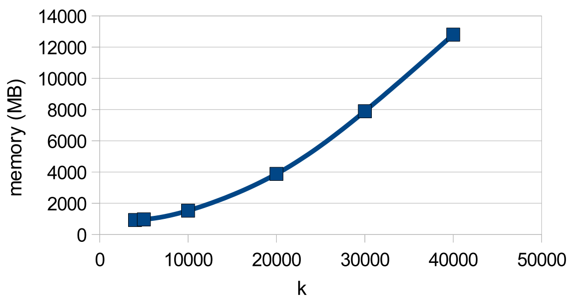

Figure 13: space required for Metis for different `k` on dblp3

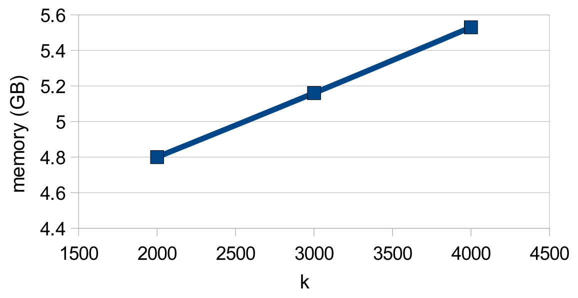

Figure 14: space required for Metis for different `k` on wiki

The space requirement of kMetis for dblp3 and wiki are plotted in Figures 13 and 14. It can be seen from these figures that, the space required by kMetis, grows almost linearly with k. But, the constants are quite huge. For example, when k is 40,000 on dblp3, memory required is 12.8 GB, which is about 96 times the size of the graph. For dblp3, with k greater than 30,000, kMetis couldn't run on an 8 GB RAM machine.[2] Memory requirements are even higher for kMetis on the wiki dataset.

The time taken by kMetis, to cluster dblp3 is approximately 5 mins, and for wiki is 1.5 hrs. This remained the trend, for varying values of k. Comparing the time required by ModNib and kMetis, we observe that our algorithm takes much more time than kMetis. But, we also note that, almost always, clustering is done offline. Thus, time may not always be an issue; but space might be.

EBFS is a small modification of BFS, and EBFS clustering takes only 7 seconds for clustering dblp3 (ignoring file write time); its space requirement is only marginally more than the size of the graph.

Satuluri et al. [15] have conducted experiments on various graphs, with the MCL algorithm. In the results given, a 75,877 node graph was clustered by MCL in 1.2 hours, on a dual core machine with 2.4 GHz CPU and 8 GB of RAM, while Modified Nibble can cluster the dblp3 graph which is 20 times larger, in 1.5 hours, with similar resources.

7 Conclusions and future work

In this paper, we considered the issue of graph clustering, using a technique for finding communities based on random walks. We proposed an algorithm, called Modified Nibble Clustering algorithm, which improves

[2]The memory requirement values in the plot were obtained from the error messages.

upon the Nibble algorithm proposed earlier. Its performance was enhanced using several policies, which were also outlined. We then compared our algorithm with two other graph clustering algorithms, viz. EBFS and kMetis on several metrics. Our experimental evaluation showed that Modified Nibble outperforms EBFS uniformly, and outperforms kMetis in edge compression, in near query performance, and most importantly on space required for clustering, where it is able to handle large numbers of clusters in situations where kMetis runs out of memory even on a moderate number of clusters.

We believe that it should be possible to bring down the time taken by Modified Nibble clustering algorithm, while maintaining the quality of the clustering, by further tuning the heuristics; this is an important area of future work. Also, currently, the algorithm works on undirected graphs, and for directed graphs, the directions are ignored. It will be interesting to study the quality of clustering when the directions are taken into account for a directed graph, and also, when the edges have weights associated with them, which may be asymmetric.

We have tested Modified Nibble on graphs with around 2.6M nodes and 40M edges (Wikipedia link graph). And we believe that it will be able to handle larger graphs. Using the compression techniques such as those described by Boldi and Vigna [4], graphs that are much larger than Wikipedia, can be stored in main memory. Since the additional memory requirements of our algorithm is minimal, it should be possible to cluster any graph that fits in memory. We are working towards testing the performance of Modified Nibble on such large graphs.

Massive graphs like the link graph of the world wide web may have to be stored in a distributed fashion, on multiple machines. Extending the Modified Nibble algorithm to work in a distributed environment, nibbling multiple clusters in parallel, is a promising area of future work.

References

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

[2] G. D. Bader and C. W. Hogue. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinformatics*, 2003.

[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. *ICDE*, 2002.

[4] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. *Proc. of 13th International World Wide Web Conference*, pages 595–601, 2004.

[5] V. Capoyleas, G. Rote, and G. Woeginger. Geometric Clusterings. *Journal of Algorithms*, 1990.

[6] N. Craswell and M. Szummer. Random Walks on the Click Graph. *SIGIR*, 2007.

[7] B. B. Dalvi, M. Kshirsagar, and S. Sudarshan. Keyword Search on External Memory Data Graphs. *VLDB*, 2008.

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

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

[10] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional Expansion For Keyword Search on Graph Databases. *VLDB*, 2005.

[11] Rose Catherine K. Graph Clustering for Keyword Search. *MTech. Thesis, Indian Institute of Technology Bombay*, 2009. www.cse.iitb.ac.in/dbms/Pubs/MTech/rose.pdf

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

[13] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph Summarization with Bounded Error. *SIGMOD*, 2008.

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

[15] V. Satuluri and S. Parthasarathy. Scalable Graph Clustering Using Stochastic Flows: Applications to Community Discovery. *KDD*, 2009.

[16] D. A. Spielman and S.-H. Teng. Nearly-Linear Time Algorithms for Graph Partitioning, Graph Sparsification, and Solving Linear Systems. *ACM STOC-04*, pages 81–90, 2004.

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

[18] S. van Dongen. MCL - Graph Clustering by Flow Simulation. *Ph.D. Thesis, University of Utrecht*, 2000.