Graph Clustering for Keyword Search

M. Tech. Project Report

Submitted in partial fulfillment of the requirements
for the degree of

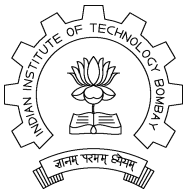
Master of Technology

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 imparting confidence in me, when I was facing difficulties in the project. I thank him for having patience to clarify my doubts and for bringing into perspective, the different aspects of the project topic. I also thank him for being a wonderful guide and mentor. Working with him was a great learning experience.

I would also like to thank Prof. Soumen Chakrabarti, for giving suggestions and directions for future work, during my first and second stage project assessments.

I am extremely grateful to my parents, Allesu Kanjirathinkal and Lisy Allesu, for giving me emotional support during the ups and downs that I went through. Without them, this work could not have been completed.

Last, but not the least, I also thank my friends over here, especially, Amita, Rakhi and Sayali, for making my life at IITB, enjoyable.

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 graph representation of data. Identifying graph nodes that are highly related to each other, and clustering them together, can localize the computing required to answer a particular keyword query, to a single or a few clusters. In the case that the graph is stored in external memory, it is possible to achieve good recall by exploring only a small portion of the graph which corresponds to these few relevant clusters. In the case that the search is distributed, splitting the data in accordance with the clustering will reduce the amount of inter-processor communication. Thus, creating good quality clustering of the data can bring down the keyword query answering time, considerably.

In this report, 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`, that improves upon the `Nibble` algorithm proposed earlier for clustering based on random walks. We outline several heuristics that can improve its performance. Then, we compare `Modified Nibble` with two graph clustering algorithms proposed earlier, EBFS and kMetis. Our performance metrics include edge compression, keyword search performance and the time & space overheads for clustering. Our results show that `Modified Nibble` outperforms EBFS uniformly, and outperforms kMetis in some settings.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The amount of data that is available today is enormous, and is growing at an extremely fast rate. It is stored in a variety of forms like web pages, XML files, relational data, etc. Searching for the right piece of data has become a very important task of everybody's life. Web search has already become a multi-billion dollar industry. Another domain of search, which is receiving considerable amount of attention is that on structured and semi-structured data.

Huge volumes of data, like enterprise data, company archives, etc. are stored in structured form, in relational databases, or in a variety of semi-structured formats such as XML. Query languages like SQL and XQuery, which are used to access relational/XML data, assume that the user knows the schema that is used to store data. A more intuitive way of querying is 'keyword searching', which is an unstructured method of querying and which has already been popularized by the web search engines. For using keyword queries, users are not required to know any query language, or do not need any knowledge of the underlying schema.

Representing data as a graph

To enable keyword searching on the data (both structured and semi-structured), it has to be represented in a suitable form. The most popular representation adopted is the graph model. For a relational database, the commonly used construction 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. Another graph that is receiving considerable amount of attention is the wiki graph. Here, nodes of the graph are the articles in the Wikipedia website. An edge is added between two nodes, if there is a hyperlink between the corresponding articles. Similar technique can be used to convert the web corpus into its graph representation. According to the granularity required, nodes may represent an entire article/page, or a smaller portion like the introduction part (text that comes before any subheading).

Representing data as a graph reveals its high level structure, and the relationship between its components. It also allows the search algorithms to generate answers in the form of graphs or trees, making the relationship between the keywords, more clear and intuitive.

External memory keyword search

Initial algorithms for keyword search such as BANKS [BHN+02], 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 the external memory search algorithm, Incremental Expanding Search [DKS08], which searches on a smaller memory resident supernode graph, to minimize IO. In this case, good clustering of nodes into supernodes, when constructing the supernode graph, is a key to efficient search.

Clustering

A cluster can be defined as a collection of objects that are similar to each other, in some way. Given a set of objects and a similarity measure, clustering is the process of finding out clusters in the given set, with respect to the measure. Graph clustering processes a graph, where presence of an edge between nodes is considered as the indicator of some similarity between them. A graph cluster can be considered as a set of nodes such that, edges connecting nodes within the cluster are more, when compared to those linking to nodes outside the cluster. A simple example is shown in Figure 1.1.

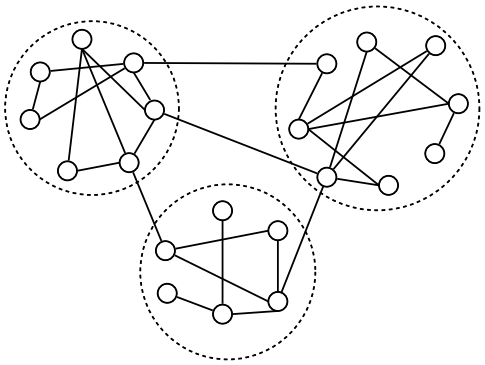


Figure 1.1: Example of clustering on a simple graph

Clustering is an already well researched topic. Geometric, hierarchical and partitioning methods are some popular classes of clustering algorithms. But some of them, like geometric clustering algorithms cannot be used for graph clustering. For graph clustering, some of the popular algorithms are kMetis and EBFS (used in Incremental Expansion Search algorithm [DKS08]). In this report, we look into the community related aspect of clustering. Below, we describe briefly, the concept of communities (detailed discussion in Section 2.3) and explain our intuition for community based clustering.

A community is a set of real-world entities that form a closely knit group. It gives a natural division of the graph nodes into densely connected subgroups ([NG04]). All real datasets have many communities within them. Take the example of bibliographic citations in the area of computer science. Here, we see the presence of groupings based on different topics in computer science, like databases, machine learning, etc., where citations within a group are much more frequent, than to outside.

Since nodes within a community are closely knit together, a search started from one of its nodes, will remain within its boundary to a very large extent, thus localizing the search. In addition to that, 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 very small fraction of the entire graph. Also, there is no reason why communities must be of similar sizes. Hence, 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 memory or distributed keyword search to produce answers in significantly lesser time.

Contributions

We propose an algorithm called `Modified Nibble`, for clustering the graph representation of data, using the technique of random walks. It improves upon an algorithm proposed earlier, called

the `Nibble` algorithm. We outline several heuristics that can improve the performance of our proposed algorithm. Then, we compare `Modified Nibble` with two graph clustering algorithms proposed earlier, EBFS and kMetis. Our performance metrics include edge compression, keyword search performance and the time & space overheads for clustering. Our experimental results show that `Modified Nibble` is able to outperform EBFS consistently, and outperform kMetis in some settings.

The rest of the report is organized as follows: Section 2 gives an overview of existing clustering algorithms. Using random walks for finding graph communities is motivated in Section 3. It also discusses two existing algorithms which use random walks for clustering. Section 4 describes in detail, our proposed algorithm, 'Clustering using `Modified Nibble` algorithm'. Heuristics which improve its performance are also discussed in the same section. Section 5 analyzes the effect of parameters and heuristics on edge compression achieved on two sample datagraphs. This is followed by the comparison of edge compression and search performance of our algorithm, and two other clustering algorithms, in Section 6. Conclusions and direction for future work are presented in Section 7.

Chapter 2

Related Work

A good amount of work has already been done in the area of graph clustering. This can be broadly classified into methods for clustering by graph-partitioning and those for finding communities. In this section, we will discuss briefly, the algorithms that belong to each of these categories. Other than these, there are numerous techniques that address the problem of graph summarization, with or without loss of information, and with different semantics for the supernode graph. A few such methods are discussed in Section 2.4. But first, we describe an external memory search algorithm from Dalvi et al. [DKS08], in Section 2.1.

2.1 External memory keyword search

BANKS (Browsing ANd Keyword Searching) [BHN+02] is a keyword search system for querying data that is represented as a graph. Dalvi et al. [DKS08] propose an algorithm, called Incremental Expansion Backward Search, for keyword search on external memory graphs. It maintains a summarized view of the data, called a supernode graph, in memory. The supernode graph is a condensed view of the much larger datagraph, obtained by clustering the nodes. Each node in the supernode graph corresponds to a set of nodes in the original graph. If there is an edge between any two inner nodes, then a superedge is created between their respective supernodes.

2.1.1 Keyword queries and in-memory search algorithms

Keyword querying is an unstructured method of querying. Connection queries and Near queries are examples for the same.

Connection queries: In this form of querying, the input is a set of words which the user thinks is important to the piece of data, (s)he is searching for. An example is 'sudarshan soumen'. The output has to show how the input words are connected to each other.

Near queries: Near queries form an important class of querying, which is not yet supported by web search engines. Its query model is slightly different from connection querying, but is very much intuitive. An example for near query is '`author (near data mining)`'. Here, `author` defines the *type* of answer required by the user. `data` and `mining` are keywords, to which the user wants the author to be close to, in the graph.

In-memory keyword search

Here, we will give a brief outline of an in-memory keyword search algorithm, called Backward Expanding search proposed in [BHN+02]. Given a set of keywords, the algorithm initially identifies nodes that are relevant to the keywords. It then proceeds to find a node, which has a path in the graph, to these nodes. The algorithm explores the graph by moving backwards along edges incident on the current node. And hence the name. The search starts from the keyword nodes, and is successful, when it finds a node to which search from all the keywords could reach. This results in generating an answer, which is represented as a tree, rooted at the common node found.

2.1.2 Answering keyword queries

In this section, we outline algorithms for answering keyword queries in external memory graphs.

Connection queries

Incremental Expansion Backward search proposed in [DKS08], is similar to the backward expanding search, except that the search is done on the supernode graph. The search starts from those supernodes, which contain the keyword nodes as their inner nodes, and proceeds to connect these supernodes. The answer thus generated, may contain supernodes. In such cases, the algorithm expands only the closest supernode per keyword on the path from the keyword to the root of the result. Then, the search proceeds on the partially expanded graph (also called, a multi granular graph). An answer is said to be pure if it doesn't contain any supernodes. The search has to continue until the required number of top ranking pure answers are generated.

Near queries

Dalvi et al. [DKS08] doesn't handle near queries on external memory graphs. Amita and Rakhi have implemented the following algorithm (described in [Sav09, Agr09]), for answering near queries on external memory graphs.

(i) Identify all (inner) nodes which contain one or more of the near keywords.

(ii) For all these innernodes, calculate their initial activation (which might require their corresponding supernodes to be read from disk). These nodes are then entered into a queue, which is prioritized on the amount of unspread activation of the node.

(iii) While the queue is not empty, remove the top node n from it, and do the following:

  (a) If the supernode of n is not already expanded, read it from disk.

  (b) Spread n's activation to its neighbors, and add them as well, into the priority queue.

  (c) If n is of the type requested by the user, then add it to the answer heap, which is max heapified on node-activation.

(iv) Output the top k answers from the answer heap.

Figure 2.1: Answering near queries in external memory graph search

An important observation in the above two algorithms is that, if the search can be localized to a few supernodes, number of disk accesses can be reduced and hence, the overall query answer time. This requires that, most of the edges incident on the inner nodes of a supernode, be contained within the supernode, and links to nodes in other supernodes, be minimized. Thus, by using a good clustering of the graph, we can improve the efficiency of search.

2.1.3 EBFS clustering algorithm

Edge-weight prioritized breadth first search (EBFS) is a clustering technique which uses BFS to create clusters. It starts with an unassigned 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.

Advantages: The time and space required by EBFS clustering is of the order of the size of the graph.

Disadvantages: The supernode graph generated by EBFS is quite dense. Hence, when keywords matched a large number of nodes, it was observed that, the search spread to a very large fraction of the data graph, and either it ran out of memory, or the answering time was above the acceptable limits.

2.2 Graph partitioning

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. To define formally, (as given in [KK98]): Given a graph $G = (V, E)$ and an integer k, partition the set of nodes of the graph into k subsets V_1, V_2, ..., V_k, such that:

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

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

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

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

To motivate this, consider the example given in [NG04], that arises in parallel computing. Suppose there are n intercommunicating computer processes, which should be distributed over g computer processors. The pattern of communications between processes can be represented by a graph or network in which the vertices represent processes and edges join process pairs that need to communicate. The problem is to allocate the processes to processors in such a way as roughly to balance the load on each processor, while at the same, time minimizing the number of edges that run between processors, so that the amount of inter-processor communication is minimized.

In general, finding an exact solution to graph partitioning problem is NP-complete. Also, when the graphs are very large, even an $O(E^2)$ algorithm is too expensive. Below, we discuss an algorithm which finds a k-way partitioning of a graph in $O(E)$ time.

2.2.1 Metis

The Metis algorithm proposed by Karypis and Kumar in [KK98], works as follows: the graph is first coarsened down to a small number of vertices by collapsing vertices and edges. Now, a k-way partitioning is found on this smaller graph. This partitioning is projected back onto the original graph, by refining it at the intermediate levels, to get a k-way partitioning of the original graph. For the graph $G = (V, E)$, the algorithm runs in $O(|E|)$ time.

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

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

Methods to find matching:

- Random Matching (RM): Visit the vertices in random order. If a vertex has not been matched yet, then select one of its unmatched adjacent vertices, randomly. The edge connecting them, is then included in the matching.
- Heavy Edge Matching (HEM): Visit the vertices in random order, as in the case of RM. But, if a vertex is found to be unmatched yet, match it with that adjacent vertex for which, the weight of the edge connecting them is the maximum over all valid incident edges.
- Modified Heavy Edge Matching (HEM*): Suppose if v is an unmatched vertex. Let H denote the set of its adjacent vertices which are unmatched and are connected to it by an edge of maximum weight. For each vertex u in H, find the sum of the weights of edges that connect u to vertices adjacent to v. Then, v is matched with that u for which is sum is the maximum. This is similar to HEM, but this scheme gives a smaller average degree for the coarser graph.

Initial Partitioning Phase: A balanced k-way partitioning is computed on this smaller graph using a multilevel bisection algorithm. A multilevel recursive bisection (MLRB) algorithm is used to compute a k-way partition, by first obtaining a 2-way partitioning of graph, and then recursively obtaining a 2-way partitioning of each partition for $log\ k$ times.

Uncoarsening Phase: In this phase, the partitioning of the coarsest graph G_m is projected back to the original graph, by going through the sequence of graphs $G_{m-1}, G_{m-2}, \ldots, G_1, G$. Since G_{i-1}

is finer than G_i, the partitioning projected onto it is refined to further decrease the edge-cut. The k-way refinement is done in the following way: for each vertex v in its nodeset, define $N(v)$ as the neighborhood of v, which is the union of the partitions to which its adjacent vertices belong. During the refinement process, v can move to any of its neighboring partitions. For each partition b in $N(v)$, define $ED[v]_b$, which is the external degree of v to partition b, as the sum of the weights of the edges connecting v to nodes in b. Also, define $ID[v]$, the internal degree of v as the sum of the weights of the edges that connect v to nodes in its own partition. Now the gain of moving vertex v to partition b is defined as $ED[v]_b - ID[v]$. The partitioning refinement algorithm will move a vertex only if it satisfies the following:

> Balancing Condition: Suppose that a and b are two partitions of a graph $G = (V, E)$. Let $W(i)$ denote the weight of the partition i, let $w(v)$ denote the weight of the node v. Define $W^{min} = 0.9|V|/k$ and $W^{max} = C|V|/k$, where C is a positive real number, which can be used to vary the degree of imbalance among partitions. Now, a vertex v, can be moved from a to b only if:
>
> - $W(b) + w(v) \leq W^{max}$
> - $W(a) - w(v) \geq W^{min}$

Metis uses a greedy strategy for refinement where v is moved to the partition which gives largest reduction in the edge-cut, or if no reduction in the edge-cut is possible, then v is moved to a partition which improves balance, but doesn't increase the edge-cut.

Advantages:

(i) It uses a computationally inexpensive refinement algorithm, which can be used to project the initial partitioning to increasingly uncoarsened version of the graph. This enables the entire partitioning procedure to run in a time, which is faster by a factor of $O(log\ k)$ than previously proposed multilevel recursive bisection algorithms.

(ii) Though it uses heuristics and doesn't guarantee an optimal partitioning, in practice, it finds good partitions, even on large graphs, in comparatively lesser processing times.

Disadvantages:

(i) Since it emphasizes on finding similar sized clusters, it cannot be used to find communities of varying sizes.

(ii) Since it creates multiple (coarse) versions of the input graph, it requires a good amount of main memory, which may not be available when the graph size is huge. This behaviour was observed when metis was used to cluster the wikigraph.

2.3 Clustering for finding communities

A community is a set of real-world entities that form a closely knit group. The community structure gives natural divisions of the nodes into densely connected subgroups ([NG04]). 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. The ability to find and analyze such groups can provide invaluable help in understanding and visualizing the structure of networks.

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. 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. This section begins with a study of measures used for quantifying the goodness of communities (Section 2.3.1), followed by a discussion on four approaches for finding communities. Later, in Section 3, we discuss a different class of approach that is based on random walks.

2.3.1 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 low 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 communities 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 communities. The goodness of a community structure is measured using conductance and modularity, 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:

$$\Phi(S) = \frac{|\partial(S)|}{min(Vol(S), Vol(\bar{S}))} \tag{2.3.1}$$

Modularity

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

Specifying formally, let V_1, ... V_k be the node subsets induced by the clustering, on an n-vertex m-edge graph G. Then, modularity can be expressed as:

$$Q = \frac{1}{m}\sum_{i=1}^{k}(|E(V_i)| - Ex(V_i, \mathcal{G})) \tag{2.3.2}$$

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

The expected number of edges for different random graph models can be computed as follows:

- Erdös-Rényi Random Graph Model

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

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

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

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

- Chung-Lu Random Graph Model

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

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

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

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

$$Q = \sum_i (e_{ii} - a_i^2) = Tr\ \mathbf{e} - \parallel \mathbf{e}^2 \parallel \tag{2.3.3}$$

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

2.3.2 Divisive method using edge betweenness

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

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

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. Without recalculating the measure, any divisive algorithm will fail to recover the cluster structure, except probably in the simplest of the cases.

Different Betweenness measures:

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

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

Limitations:

(i) The input graph is assumed to have undirected and unweighted edges.

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

2.3.3 Extremal optimization

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

In Section 2.3.1, Modularity, which is a measure of the goodness of clustering, was explained. Consider the expression of Modularity given by Newman and Girvan ([NG04]):

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

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

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

where $\kappa_{r(i)}$ is the number of links that i has to nodes which are in its community, and k_i is the degree of node i. The quantity $a_{r(i)}$ is the number of links whose at least one incident vertex is in the community of i.
From the above definition, it can be seen that,

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

where L is the total number of links in the network.
The $fitness$ of a node i, given by λ_i, is defined as:

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

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

Algorithm

Extremal Optimization (EO) algorithm optimizes the global variable modularity (Q), by improving extremal local variables. Searching for the optimal modularity value is made difficult by the fact that the space of possible partitions, blows up very fast. Hence, heuristics are used to restrict the search space.

The proposed heuristic search algorithm proceeds in the following steps:

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

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

(iii) Self Organization step: Move a node with low value of $fitness$ (extremal), from one partition to the other. Specifically, the τ - EO probabilistic selection is used, where initially, the nodes are ranked according to their $fitness$ values, and then, a node of rank q is selected with probability proportional to $q^{-\tau}$ where, $\tau \sim 1 + 1/ln(N)$ (N is the number of nodes).

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

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

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

The proposed algorithm has a complexity of $O(N^2 ln(N))$.

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

Limitations: As remarked in [Upa08], the biggest drawback is that shifting any vertex changes the contribution of all the nodes, and hence, all of them needs to be examined in the next round, for selecting the next node. Computation of the contribution thus can not be reduced further.

2.3.4 Modularity-weight prioritized BFS

In the paper [Upa08], Upadhyaya suggests an algorithm called Modularity-Weight Prioritized BFS, which is based on the Modified Extremal Optimization algorithm suggested by Duch and Arenas in [DA05] and which was discussed in Section 2.3.3.

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

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

Algorithm: The proposed algorithm proceeds in the following steps:

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

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

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

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

(v) If highest increase in modularity obtained in step (iv) is less than zero, then, this iteration is called as an *iteration-without-improvement*. If the number of such iterations exceed a particular number, then, stop iterating. Else, repeat steps from (ii) onwards.

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

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

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

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

Limitations:

(i) Since the next node to be shifted is chosen from the set of boundary nodes, it increases the probability of getting stuck at a local optimum.

(ii) To bring down the running time, not all nodes are tested for gain in modularity; only a fixed number of them are chosen randomly from the set of boundary nodes and tested. Hence chances of stopping before the global optimum is achieved, is high.

2.4 Miscellaneous clustering methods

In this section, we discuss a few clustering techniques that are markedly different from the approaches that we have discussed so far. Though they don't suit our task, a brief discussion is included here to give insight into the larger class of clustering algorithms.

2.4.1 K-means clustering

K-means method comes under the class of geometric clustering methods, which optimize a distance based measure, such as a monotone function of the diameters or the radii of the clusters, and finds clustering based on the geometry of points in some D-dimensional space ([CRW90]). The input to K-means consists of N points in some D-dimensional space, and a number K, which is the number of clusters required. The goal is to find K points (in the D-dimensional space), called *means*, that represent the K clusters, and an assignment of the N input points to one of the clusters, such that, the sum of the squares of the distances of each data point to the *mean* of its cluster, is a minimum.

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

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

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

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

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

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

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

Limitations:

(i) The direct implementation of K-means algorithm can be slow, since in the first half-step of each iteration, it is necessary to compute the distance between every data point and every *mean*. Improvements have been suggested that take advantage of the triangle inequality for distances.

(ii) The dissimilarity between a data point and a *mean* point, is taken as square of the Euclidean distance between them. Hence, it is necessary that, the data being clustered must be points in some D-dimensional space - this is not true always. K-medoids algorithm is an improvement over this, where the dissimilarity between every pair of points is specified beforehand, and the *mean* points, called *medoid*s in this case, are constrained to be chosen from the data points.

2.4.2 Graph summarization

Rastogi et al. [NRS08] suggest a graph compression method which exploits the similarity of the link structure present in the graph to realize space savings. The compressed representation of the input graph G, consists of a graph summary S and a set of edge corrections C, which are used to recreate the original graph from S. The graph summary S is similar to a supernode graph, but has a slightly different semantics for superedges. Each supernode v in S, corresponds to a set A_v of nodes in G. Each superedge (u, v) in S represents edges between **all** pair of nodes in A_u and A_v. However, all of these may not be present in G. These are added as edge corrections in C, and annotated as negative $(-)$. Edges of G, which are not implied by the supernode graph, are added as positive edge corrections $(+)$.

In the best case, if there is a complete bi-partite subgraph, then the two bi-partite cores can be collapsed into two supernodes and all edges can be simply replaced with a superedge between the supernodes. Similarly, a complete clique can be collapsed into a single supernode with a self-edge. In other cases, given two supernodes u and v, a superedge is added between them only if the negative edge corrections are lesser than the positive corrections. This is based on MDL (Minimum Description Length) principles. The supernode graph is created by merging nodes iteratively, starting from the original graph. Supernodes chosen to be merged are the ones, which give maximum reduction in cost, where cost of a supernode is the sum of the costs of representing the superedges incident on them.

Limitations:

Though this algorithm is able to achieve impressive compression, the semantics of the supernode graph is not suitable for the graph search algorithms that we consider.

Chapter 3

Finding Communities using Random Walks on Graphs

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.

Many a times, the walk is adapted to the graph at hand. For example, when the edges of the graph have weights associated with them, the node explored next is chosen with probability 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 behavior 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. 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*, 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, all nodes have non-negative probabilities and they 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 reset 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

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 [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.

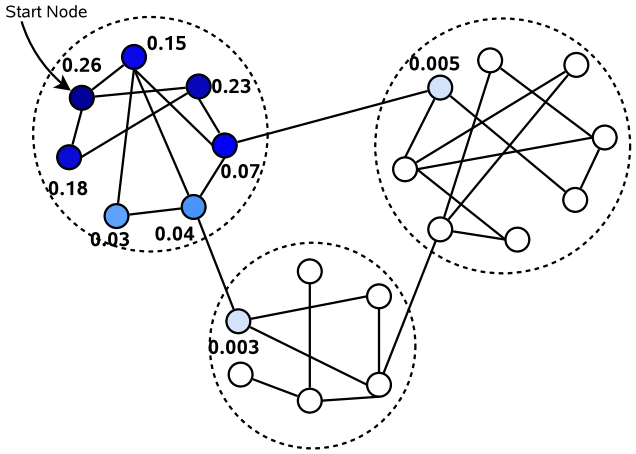


Figure 3.1: Example for sudden drop in probability outside the cluster

Consider the toy example given in Figure 3.2, 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 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 [AL06].

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.

Consider another toy example shown in Figure 3.2. 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

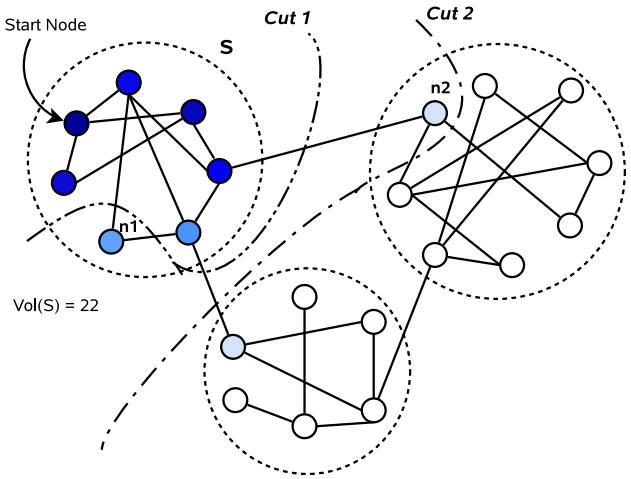


Figure 3.2: Example for choosing the best cluster based on conductance

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 3.3), and the `Modified-Nibble` algorithm proposed by us in this report (Section 4).

3.3 Clustering using Nibble algorithm

In the paper [ST04], Spielman and Teng describe a nearly-linear time algorithm, `Partition`, for computing crude partitions of a graph, by approximating the distribution of random walks on the graph. In this section, we outline all the procedures. Detailed pseudocode is given in Appendix A.

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

(i) Compute the bound on maxIterations, t_0, and threshold, ϵ.

(ii) Start spreading probabilities from v.

(iii) When the $nodeProbability$ falls below ϵ, truncate the walk by setting it to 0.

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

(v) 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.

(vi) If a j was found, then return the first j nodes of the sorted set, as the enclosing cluster of v.

(vii) Otherwise, do the next step of spreading probabilities and repeat from Step (iii).

Random Nibble Algorithm: input: Graph G, Max Conductance θ_0

(i) Choose v randomly with probability proportional to its degree.

(ii) Call `Nibble`(G, v, θ_0).

`Random Nibble` is an intermediate algorithm which calls `Nibble` on carefully chosen start nodes.

Partition Algorithm: input: Graph G, Max Conductance θ_0

(i) Compute the bound on maxIterations, t

(ii) Call `RandomNibble` with current graph and θ_0.

(iii) Keep the cluster returned by `RandomNibble` on hold. If there is already a cluster on hold, then merge them.

(iv) If on merging in step (iii), the volume exceeds a predetermined fraction of G, then stop and return the merged cluster.

(v) Else, remove these nodes from the graph and repeat for at most t iterations.

`Partition` calls `Nibble` through the `Random Nibble` method, for at most, a fixed number of times. It then collects the clusters found by `Nibble`. As can be seen from Step (iv) of the algorithm, as soon as the volume of this collection exceeds a predetermined fraction of that of the entire graph, it returns the collection.

The final clusters of the graph are obtained by the `Multiway Partition` procedure. It uses `Partition` to get an initial partitioning of the graph and then invokes `Partition` again, on the two partitions thus obtained. This is repeated for a fixed number of times.

3.3.1 Shortcomings

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, a priori. 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 (v) 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`.

(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.

(vi) Testing for good community, which involves sorting the nodes, is done after each step of spreading of probabilities, and could lead to considerable slowdown of clustering on large graphs.

The overall algorithm processes the entire graph in a top-down manner. Hence, it becomes difficult to handle graphs of very large size.

3.4 Clustering using Seed Sets

In the paper [AL06], Andersen and Lang present an algorithm to discover the enclosing community of a given cohesive set of nodes, called the "seed set". They modify the algorithm proposed by Spielman and Teng ([ST04]) (discussed in Section 3.3) to expand the seed set for discovering the enclosing community, that has small conductance, while examining only a small portion of the entire graph.

Seed set expansion is commonly done in the link-based analysis of the web. It first came into prominence with the HITS algorithm proposed by Jon Kleinberg ([Kle98]) where, a search engine was used to retrieve a set of pages related to a particular input, which served as the seed set. A fixed-depth neighborhood expansion was performed on this set, to get a larger set of pages upon which the HITS algorithm was run.

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

Algorithm:

(i) Assign equal probabilities to all nodes in the seed set, and start spreading probabilities.

(ii) Sort the vertices in the descending order of their degree-normalized probabilities.

(iii) Truncate the walk for nodes with probabilities lesser than a predefined threshold, ϵ.

(iv) Find a j such that the set of first j nodes, C, satisfy the test for a good community:
the probability outside C is lesser than a predetermined fraction of $\Phi(C) \times T$, where T is the number of steps of the random walk done.

(v) If a j is found, stop and return that set as the community.

(vi) Else, continue the random walk and repeat from step (ii) onwards.

Good seed sets:

Consider the amount of probability that has escaped from a community C after T steps of the random walk, that started from the the seed set S. The seed set is good if the amount of probability that has escaped is not much larger than $\Phi(C)\ T$, provided C has a small conductance.

Following are also good seed sets:

(i) Any set that is fairly large and nearly contained in the target community.

(ii) Sets chosen randomly from within a target community.

3.4.1 Advantages and shortcomings

Advantages:

(i) The major advantage of the algorithm is that it explores only the local locality. The algorithm is able find a small community enclosing the seed set, by touching only a few number of nodes. This is accomplished by using truncated walk distributions in place of exact walk distributions.

(ii) The algorithm is capable of finding nested clusters that enclose the seed set. This is achieved by simulating the walk for a larger number of steps.

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

Chapter 4

Clustering using Modified Nibble Algorithm

Keeping in mind the ideas suggested by Spielman and Teng in [ST04] (discussed in Section 3.3) and Andersen and Lang in [AL06] (discussed in Section 3.4), and based on their shortcomings that we have identified, we propose the `Modified Nibble` algorithm, which is discussed in this section.

4.1 Outline of Modified Nibble algorithm

The proposed method of clustering takes as input, the graph G, and a user-specified upper bound on the size of clusters, `maxClusterSize`.

Overall clustering algorithm

(1) Choose a start node.

(2) Nibble out a cluster for the start node using the `Modified Nibble` algorithm and remove it from G.

(3) Repeat from step (1), until the entire graph is processed.

Figure 4.1: The overall clustering algorithm

The core of the proposed clustering method is the `Modified Nibble` procedure (Figure 4.2). It explores the locality of the specified start node, by performing random walks on the remainder graph. As can be noted, maximum conductance is not a user-input. Instead, the algorithm finds the best available cluster.

`Find Best Cluster` algorithm (Figure 4.3), is internally invoked by the `Modified Nibble` algorithm, to find the best available cluster out of the current active nodes. The algorithm described here is same as that discussed in Section 3.3 and Section 3.4.

An important point to note is that, the proposed graph clustering algorithm proceeds by removing one cluster at a time rather than processing the entire graph at one go. This can be beneficial for clustering massive graphs.

Modified Nibble algorithm

(1) Initialize `nodeProbability` of the `startNode` to 1 and add it to the `activeNodes` set.

(2) `Batch i`: for each node in `activeNodes`, do the following for a specified number of times:

  (a) spread `spreadProbability` fraction of its current `nodeProbability`, to all its out-neighbors, equally.

  (b) update `nodeProbability` of all nodes with the probabilities accumulated from their neighbors.

  (c) update `activeNodes` set to contain all nodes with positive values for their `nodeProbabilities`.

(3) Invoke `Find Best Cluster` algorithm and obtain the best cluster for the `startNode`, out of the current set of `activeNodes`.

(4) If the cluster obtained in step (3) has same or higher conductance than the best cluster obtained in `Batch i-1`, stop and output the latter, as the cluster for the `startNode`.
Assumption: spreading probabilities further could blur the cluster boundary and hence, may not give better results. This decision is greedy.

(5) Else, if in Step 4, the conductance has reduced, repeat from Step 2 onwards (`Batch i+1`).
Assumption: all nodes of the best cluster may not yet be explored. Thus, performing more walks could improve the results.

Figure 4.2: `Modified Nibble` algorithm

Find Best Cluster

(1) Sort the nodes in the `activeNodes` set, in the decreasing order of their `nodeProbabilities`.

(2) Define the candidate clusters C^i to be the set of nodes from 1 to i, in the sorted order.

(3) Compute the conductance of all candidate clusters.

(4) Return the one with smallest conductance as the best cluster.

Figure 4.3: `Find Best Cluster` algorithm

4.2 Sample execution of Modified Nibble algorithm

In this section, we show a sample execution of the `Modified Nibble` algorithm on a toy graph.

Consider the graph in Figure 4.4, 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.

Batch 1

$\Phi(Best\ Cluster) = 4/12 = 0.33$
The intuitive cluster S, not found yet.

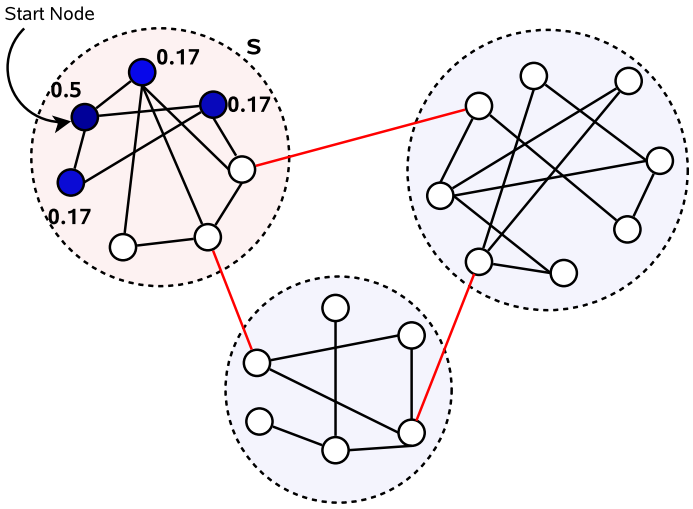
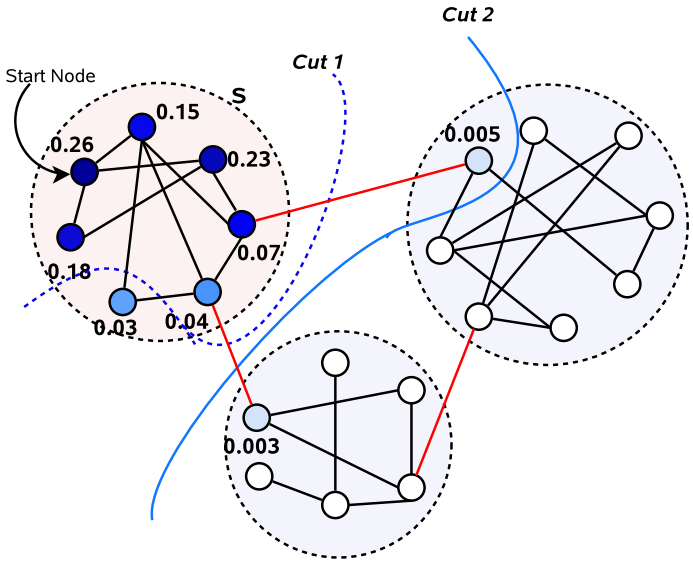


Figure 4.4: Probability distribution after 1 step

Figure 4.5 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 *nodeProbabilities* belong to S. Hence, by considering nodes in the decreasing order of their *nodeProbabilities*, it is possible to determine S.



Batch 2

$\Phi(S) = 2/22 = 0.09$
$\Phi(Cut1) = 4/(22-2) = 0.2$
$\Phi(Cut2) = 3/(22+3) = 0.12$
Best Cluster $= S$

Figure 4.5: Probability distribution after 3 steps

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 such sets formed. Figure 4.5 shows two cuts in the graph, in addition to S. $Cut1$ corresponds to the case where we choose the first 6 nodes in the sorted order. Its conductance is 0.2. $Cut2$ corresponds to choosing the first 8 nodes, and its conductance is 0.12. Choosing the first 7 nodes creates 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 the previous batch of random walks. 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 walks.

Batch 3

$\Phi(Cut3) = 4/28 = 0.14$
$\Phi(Cut4) = 6/32 = 0.18$
Best Cluster = S

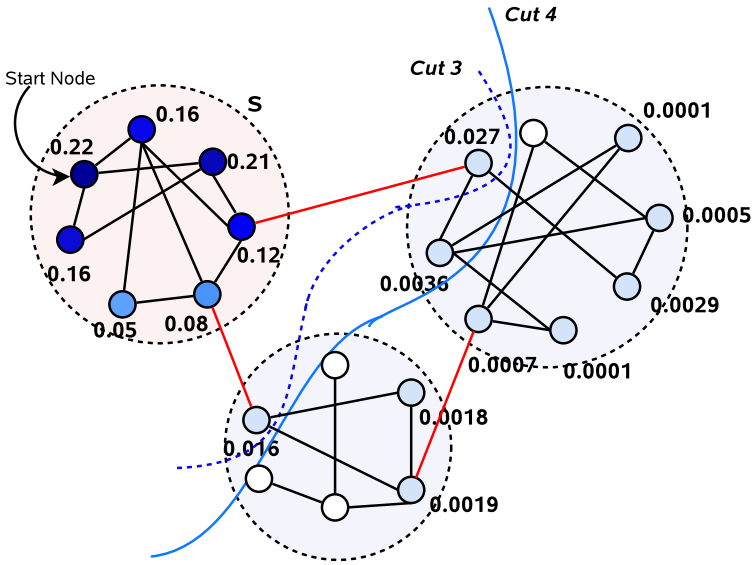


Figure 4.6: Probability distribution after 5 steps

Figure 4.6 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 4.5). $Cut3$ has the first 9 nodes in the sorted order, which includes all the nodes in S and its conductance is 0.14. $Cut4$ has the 10^{th} node added to $Cut3$, 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.

An important observation in Figure 4.6 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.

4.3 Parameters and Heuristics

The outline of `Modified Nibble` clustering algorithm given in Figures 4.1, 4.2 and 4.3 is under-specified. In this section, we will discuss a few heuristics and parameters to the same, before presenting the detailed algorithm, in Section 4.5. Section 4.3.1 below, describes the base set of parameters and heuristics, which have been considered in our first cut implementation of the clustering algorithm. Some additional heuristics are discussed in Sections 4.3.2 and 4.4.6.

4.3.1 Base set

H1. **Start node:** In the ideal setting where communities in the graph are known beforehand, the node which is most '*central*' to the cluster can be chosen as the start node. However, since we do not have this information in advance, we have to heuristically make a call on where to start. Following are two options:

(a) **Max degree:** As a first cut, node with highest out-degree in the remainder graph was used as the start node in most of the experiments.

(b) **Min degree:** Nodes with large out degree are mostly hub nodes and are usually towards the center of the graph. Links from hub nodes have to be treated cautiously, because it is not uncommon for them to connect nodes which are barely related, creating short-cut paths, all over the place. Hence, a random walk started from such nodes can spread to a large proportion of the graph, in a few steps. Nodes with lower out-degree are usually towards the periphery of the graph, and may provide good starting points for exploring the graph. In addition to this, removing clusters from the periphery will gradually decongest the core, thus making its processing easier.

H2. **Nodes spreading in each step**: In step 2 of the `Modified Nibble` procedure (Figure 4.2), all active nodes spread their probabilities. But, another alternative is that only a single node is chosen to spread. Thus, following are the two cases:

(a) Spread from all active nodes.

(b) Only a single node spreads in each step. But to achieve meaningful results in this case, following changes need to be done:

- Let δ denote the amount of probability received by a node, which is yet to be spread to its neighbors. Then, when a node is chosen to spread next, it spreads `spreadProbability` fraction of only this δ, remaining of which gets added to its `nodeProbability`, which is not transferable.
- In each step, the node to spread next, is the one with largest value for δ.
- Since in each step of a batch, only a single node spreads its activation, the number of iterations in a batch was set to `m` $\times$ `maxClusterSize`. Here, `m` stands for *multiple*. It can control the amount of spreading in the graph, prior to testing for best cluster.

H3. **Self-transition probability of a random walk:** This is determined by the parameter, `spreadProbability` (Figure 4.2, step 2(a)). Lower values of `spreadProbability` 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, `spreadProbability` was set to 0.5.

H4. **Number of iterations in a Batch:** In the ideal case, after every step of spreading of probabilities, all candidate clusters must be checked to find the best cluster. But, this can slow down the clustering process considerably, since each invocation of `Find Best Cluster` (Figure 4.3) involves sorting. To avoid this, the concept of a `Batch` of random walks is used, and `Find Best Cluster` is invoked only after a batch. The number of steps in a batch of random walks is heuristically chosen from the APGP series, described below:

Arithmetic Plus Geometric Progression (APGP): 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 a, d and r, can be used to get fine-grained control over the difference between successive terms of the series. It is advisable to set r to a comparatively small value, so that the difference between successive terms of the series is not very large when i increases. But then, for smaller values of i, the successive terms will be too close. To avoid

this, set d to a higher value. For larger values of i, terms of GP will surpass those of AP, and hence, the number of times sorting is done, is $O(log$ `maxClusterSize`$)$, which is acceptable.

H5. **Upper bound on number of random walk steps:** In step 5 of the `Modified Nibble` procedure (Figure 4.2), if the conductance of the best cluster found in each iteration decreases when compared to that found in the previous iteration, the spreading of probabilities is continued. An upper bound on the number of random walk steps could be `maxClusterSize`, which in turn decides the maximum number of iterations. This bound ensures that, all nodes of a cluster whose diameter is `maxClusterSize`, are touched before spreading of probabilities is discontinued.

H6. **Upper bound on number of active nodes:** Spreading probabilities from all active nodes can propagate to the entire graph, if left checked. According to the intuition for random walk based clustering (Section 3.2), it is possible to extract a cluster by exploring only a local neighborhood of the start node. Hence, the size of this neighborhood was restricted to be within `maxActiveNodeBound`, calculated as `f` $\times$ `maxClusterSize`. `f` is referred to as *factor*.

H7. **Behavior on `maxActiveNodeBound`:** If the number of active nodes is restricted using H6, 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. The intuition behind this approach is as follows: when the graph is strongly connected, the number of active nodes can reach the `maxActiveNodeBound` quite rapidly. This is also accelerated by the presence of a large number of hub nodes, as is the case in wiki graph. In such a scenario, identifying a good cluster in a very few steps of the walk, becomes difficult. Hence, terminating the walk as soon as the bound is reached and emitting the current best cluster, can hurt the overall quality of the clustering.

H8. **Compaction procedure:** `Modified Nibble` procedure may return clusters of sizes much smaller than `MaxClusterSize`. Retrieving many tiny clusters incur heavy IO cost, thus hurting the search performance. This can be avoided to some extent by bundling together, multiple such clusters. Following 3 methods of compacting the clustering were tested:

CP1. **Blind and greedy compaction of all clusters:** This was a first cut approach. It read clusters in the order of their generation and collected as many as possible such that the total number of nodes doesn't exceed `maxClusterSize/2`. As soon as the combined size exceeded this value, the collected set of clusters are emitted as a new cluster. All clusters of size at least `maxClusterSize/2` are directly emitted.

CP2. **Edge aware compaction of all clusters:** After the clusters were created by the `Modified Nibble` algorithm, the Metis procedure (Section 2.2.1) was invoked with a suitable k, on the supernode graph, where the node weight was set to the number of nodes assigned to that particular supernode, and edge weight, to the number of edges between the two supernodes. Metis returns the supernodes that must be bundled together, of which only those are combined, where the total number of nodes is at most `maxClusterSize`.

CP3. **Naïve compaction of tiny clusters:** It was observed that CP1 and CP2 usually created dense supernode graphs, which is detrimental to the search algorithm. Hence, in this compaction method, only "tiny" clusters which do not have any cut edges are combined. In the implementation, clusters of size lesser than 10 were chosen and CP1 was executed, where a combined cluster is emitted, if adding the next tiny cluster to it would make its size greater than `maxClusterSize`. Note that, performing this compaction will not affect the number of cut edges.

4.3.2 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. In many real world datasets, like wikipedia, there are some nodes which are linked to, by a very large number of nodes. If all these co-cited nodes were in a single cluster, all edges to them will be condensed to a very few superedges, thus giving higher edge compression. This is the notion behind the following heuristic:

H9. **Remove hub nodes:** Select nodes of indegree at least `maxClusterSize`, and choose the top `maxClusterSize` number of nodes (in the order of decreasing degree). Group these nodes together into a single cluster, and remove from the original graph. Alternatively, choose the top `t` $\times$ `maxClusterSize` and create `t` clusters of size, `maxClusterSize`. Execute the clustering procedure on the remainder graph.

4.4 Graph formations

In the `Find Best Cluster` procedure discussed in Section 4.1, the candidate clusters were generated by considering the graph nodes in the order of their increasing probabilities. Same was the approach used in the clustering method using nibble algorithm (Section 3.3) and clustering using seed sets (Section 3.4). In experiments conducted on `dblp3` and `wiki` datagraphs, it was observed that, a downright implementation of the above leads to some interesting formations in the supernode graph all of which can hurt the searching algorithm.

In this section, we illustrate 3 formations that have been identified, and also discuss possible causes and solutions to avoid or minimize their occurrence. The first two formations, namely, Bridge and V, are applicable only to nodes which have exactly two neighbors, whereas, the third, named Umbrella formation is generic. Bridge and V are relevant for the `dblp3` dataset, since 2 out of its 4 relations, namely `cites` and `writes`, correspond to graph nodes with exactly 2 neighbors (structure of these datasets are discussed in Section 5).

4.4.1 Bridge formation

Figure 4.7 shows an example of Bridge formation in a clustering for the `dblp3` datagraph. A Bridge is formed when a node n_c which has only 2 neighbors, is separated from both its neighbors in a clustering, and both the neighbors are themselves in different clusters. Thus, both edges connecting n_c to its neighbors are cut edges and n_c can be visualized as being in the center of a bridge that connects its two neighbors.

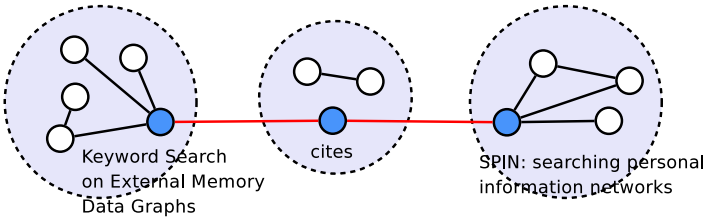


Figure 4.7: Bridge formation

4.4.2 V formation

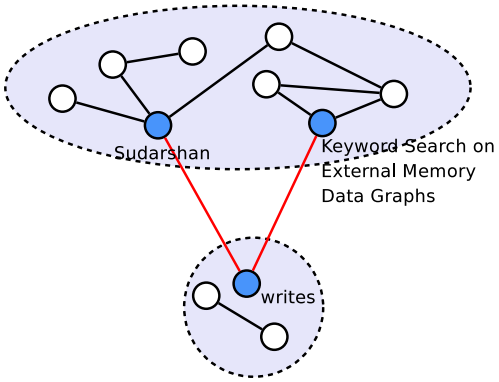


Figure 4.8: V formation

Figure 4.8 illustrates an example of V formation in a clustering for the `dblp3` datagraph. V formation is quite identical to the Bridge, except that here, both neighbors of n_c are in the same cluster. Both edges contribute to the cut set, but form only a single super edge. Here, n_c can be visualized as being at the vertex of a V which connects its two neighbors through it.

4.4.3 Umbrella formation

An Umbrella formation is a generic term for all cases where a node is separated from all its neighbors, in the clustering under consideration. Here, n_c has 1 or more neighbors, and in the clustering, n_c's cluster is different from those of its neighbors. The neighbors may or may not be in the same cluster.

All edges incident on n_c are in the cut-set, and n_c can be visualized as being at the top notch of an open umbrella, with its incident edges forming the radial ribs (Figure 4.9). Henceforth, nodes that are separated from all its neighbors will be referred to as *abandoned* nodes.

4.4.4 Possible reasons and solutions to graph formations

Reasons:

(i) Each neighbor of an abandoned node n_c, is an authoritative node in its respective domain, and n_c is a hub node that merely connects them. This might cause each neighbor to be absorbed into the cluster for its domain, leaving out n_c.

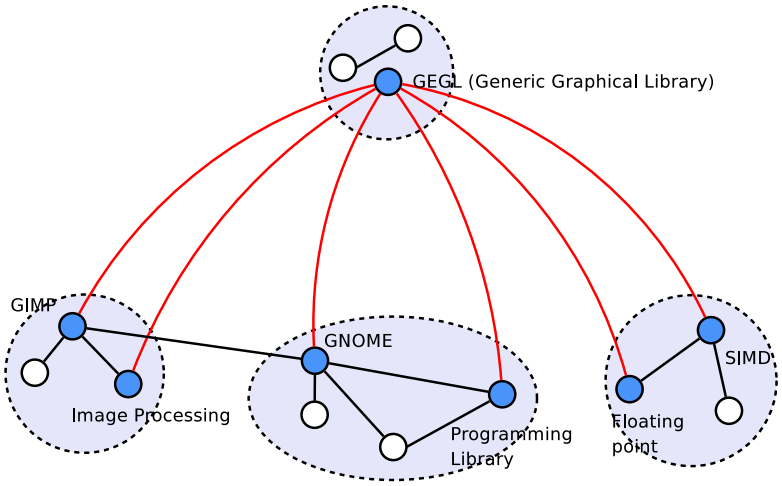


Figure 4.9: Umbrella formation

(ii) All the neighbors of n_c, have relatively large out degree, because of which, the amount of probability that is transferred from them to n_c is much low. This could cause n_c to be listed much later than its neighbors, in the sorted order. Due to this reason, even if the neighbors of n_c belong to the same cluster, n_c may not get added to it (V formation).

Solutions:

(i) Add all abandoned nodes to a cluster after it is formed: in step 4 of the `Modified Nibble` procedure (Figure 4.2), if a cluster is found, then, identify all nodes which will now be abandoned, and add them to the newly found cluster. This approach has the drawback that the size of the resulting cluster could be larger than `maxClusterSize` parameter.

(ii) Add abandoned nodes as and when they are found: in step 2 of the `Find Best Cluster` procedure (Figure 4.3), each candidate cluster generated is altered to contain all abandoned nodes that would be created, if that candidate were to be finalized as a cluster. A candidate whose resulting size goes beyond `maxClusterSize` is discarded. This approach ensures that clusters are within the `maxClusterSize` parameter.

(iii) Degree-normalize node probabilities prior to sorting: this addresses the second reason mentioned above. Nodes with lower degrees will become comparable to larger degree nodes, when the accumulated probabilities are normalized by their in-degrees.

4.4.5 Implications on search performance

Presence of formations can hamper graph search, in the following way: Let a and b be two keyword nodes, which are in clusters A and B (both may be same), respectively. Node n_c which links a and b is abandoned, and is in another cluster C which is different from both A and B, thus giving rise to a Bridge or V formation. Now, to find a path connecting a and b, the search algorithm will fault for C, even though all the other nodes of this cluster are barely related to the search at hand. Same is the case for any instance of search, which requires a path to, say b, and that passes through a, thus escalating the number of cache misses and consequently, the query answering time.

4.4.6 Graph formation heuristics

For eliminating graph formations, following two methods were tried out:

H10. **Graph formation heuristic**

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

(b) **Abandoned node awareness:** Prevent the occurrence of formations right from the creation of candidate clusters (Step 2 of `FindBestCluster` procedure in Figure 4.3), by adding all abandoned nodes to the candidate clusters. Candidates whose size goes beyond `maxClusterSize` are discarded.

4.5 Detailed pseudocode of Modified Nibble clustering algorithm

This section describes the pseudocode of `Modified Nibble` clustering algorithm in detail. It is obtained by attaching the different parameters and heuristics described in Section 4.3 to the outline given in Section 4.1.

Overall clustering algorithm *input: Graph G*

(1) Set $G' = G$. But, if co-citation heuristic H9 is used, set G' to the remainder graph, after removing hub nodes.

(2) Choose start node n_s according to H1.

(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 H8 procedure.

Figure 4.10: Detailed pseudocode for the overall clustering algorithm

Figure 4.10 describes the overall algorithm which clusters the input graph. It calls `Modified Nibble` algorithm, which is detailed in Figure 4.11, to nibble out a cluster for the selected start node. `Modified Nibble` internally invokes `ModifiedFindBestCluster` algorithm, which is described in Figure 4.12, to find the best cluster out the current active nodes.

In the experiments done in Section 5, the effect of varying the parameters and heuristics used in these three algorithms are studied. Section 6 compares the performance of our algorithm with two other clustering algorithms.

`ModifiedNibble` *input: start node* n_s*, Graph* G'

(1) initialization:

- set `nodeProbability` of n_s to 1 and add it to the `activeNodes` set.
- set `maxSteps` according to H5.
- if number of active nodes are bounded, calculate `maxActiveNodeBound` using H6.
- set `totalSteps` to 0.

(2) `Batch i`:
initialization:

- get term t_i from the series chosen using H4.
- set `batchSteps` to (t_i - `totalSteps`).
- but, if t_i exceeds `maxSteps`, set `batchSteps` to (`maxSteps` - `totalSteps`).

do the following for `batchSteps` number of times:

(a) spread from all nodes in `activeNodes` or a single node, according to H2.

(b) the amount of spreading is determined by `spreadProbability` as chosen in H3.

(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 `nodeProbabilities`.

(e) if number of active nodes are bounded, check if `maxActiveNodeBound` has been reached. If yes, then, according to the choice of H7, do as below:

- H7(a) : stop this batch, and proceed directly to step 3.
- H7(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 graph heuristic H10 is being used, and is set to H10(a),
set C_{best} to $C_{best} \cup \{n_c \mid n_c \textit{ is abandoned by } C_{best}\}$

(7) return C_{best} as the best cluster of n_s.

Figure 4.11: Detailed pseudocode for `Modified Nibble` algorithm

`ModifiedFindBestCluster` *input: set* `activeNodes`*, graph* G'

(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 `nodeProbabilities`.

(3) define candidate clusters C^j to be the set of nodes from 1 to j, in the sorted order, where $j = min(\texttt{maxClusterSize}, |\texttt{activeNodes}|)$.

(4) if the graph heuristic H10 is used, and is set to H10(b), then do the following:

- set each C^j to $C^j \cup \{n_c \mid n_c \textit{ is abandoned by } C^j\}$
- 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 4.12: Detailed pseudocode for `ModifiedFindBestCluster` algorithm

Chapter 5

Experiments and Analysis of Parameters and Heuristics

In this section, our goal is to study the effect of different settings of parameters and heuristics, on graph compression. Comparison with other algorithms are done in Section 6. The experiments start off, with a base implementation of `Modified Nibble` clustering algorithm, whose settings are specified in Section 5.2. During this exercise, we refine the values of parameters and the choices for heuristics, to arrive at a fine-tuned implementation of our proposed clustering algorithm.

`Modified Nibble` clustering algorithm (detailed pseudocode in Section 4.5)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). 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 `wiki` were conducted on a blade of eight 2.50 GHz Intel Xeon CPUs, with a combined RAM of 8 GB, running Debian 4.0. Observations are analyzed, as and when the results are presented.

5.1 Details of datasets

dblp3 database

Tables: `author, cites, paper, writes` (Figure 5.1)
Number of nodes: 1,771,381
Number of undirected edges: 2,124,938
max degree = 784

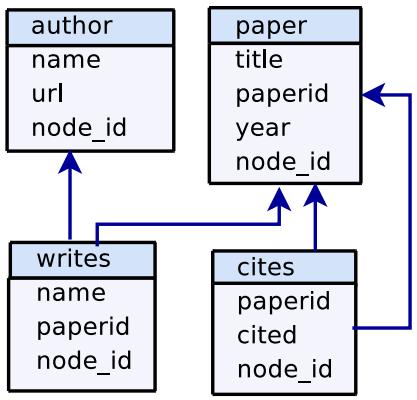


Figure 5.1: `dblp3` database schema

wiki database

Tables: `document, links` (Figure 5.2)
Number of nodes: 2,648,581
Number of undirected edges: 39,864,569
max degree = 267,884

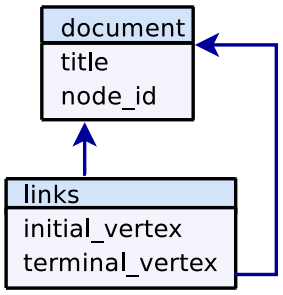


Figure 5.2: `wiki` database schema

5.2 Base implementation of Modified Nibble clustering

Base implementation (BI) is our first cut implementation of `Modified Nibble` clustering algorithm, and doesn't take care of the graph formations. The settings of parameters and heuristics, H1 to H9, for BI are specified in Table 5.1. Experiments in this section use BI with these settings as the default. Deviations if any, from the base implementation are specified, when the results are presented.

Heuristic / Parameter		**Choice / Value**
H1 - start node	:	(a) - max degree
H2 - nodes spreading in each step	:	(a) - all active nodes
H3 - self-transition probability	:	0.5
H4 - number of iterations in a batch	:	APGP series with `a` = 2, `d` = 7, `r` = 1.5
H5 - upper bound for number of steps	:	`maxClusterSize`
H6 - `maxActiveNodeBound`	:	`f` = 500
H7 - behavior on H6	:	(a) - stop on `maxActiveNodeBound`
H8 - compaction	:	CP1 - blind and greedy compaction
H9 - co-citation	:	no

Table 5.1: Settings for the base implementation

5.3 Node and edge compression

This section uses the base implementation for obtaining node and edge compression values, on `dblp3` and `wiki` datasets.

$$\text{Node Compression} = \frac{\text{number of nodes in the original graph}}{\text{number of clusters}}$$

$$\text{Edge Compression} = \frac{\text{number of edges in the original graph}}{\text{number of inter-cluster edges}}$$

Node compression is easier to obtain. The main indicator of quality of clustering is edge compression. Higher the edge compression, better the clustering.

5.3.1 Compression on dblp3

mcs	# 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 5.2: Compression values for different cluster sizes on `dblp3`

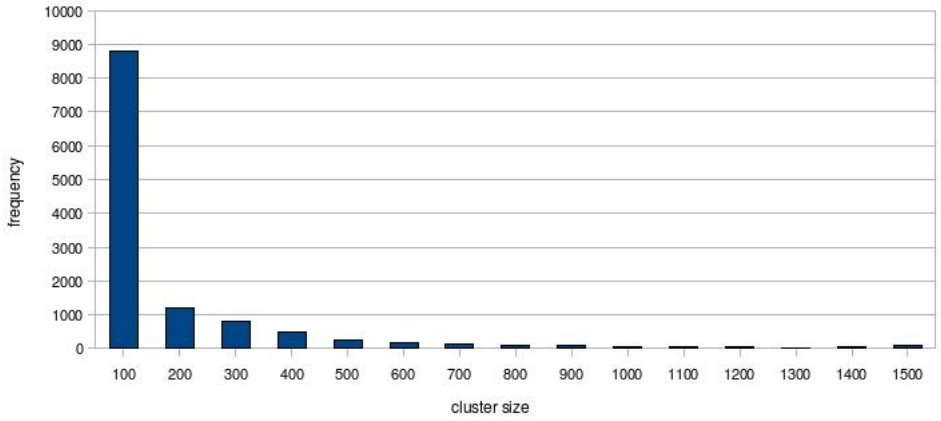


Figure 5.3: Chart of cluster size vs. frequency of `dblp3` (without compaction)

Observations:

- It can be observed that in Table 5.2, edge compression improves with increasing `maxClusterSize`. But, the improvement is only about 2 times when the parameter changes from 100 to 1500, which is not very substantial. Figure 5.3 can explain this observation, better.
- From Figure 5.3, it can be observed that, most of the clusters of `dblp3` have sizes up to 400. Hence, increasing the `maxClusterSize` parameter beyond 400, may not give enormous gain, in terms of edge compression. So, all experiments on `dblp3` use 400 as the setting for `maxClusterSize`, unless mentioned otherwise.
- Figure 5.3 also shows that, there are relatively large number of 'tiny' clusters in `dblp3`. This might escalate disk access time, in case of external memory search systems, unless a proper compaction scheme is in place.

5.3.2 Compression on wiki

mcs	# clusters	# inter-cluster edges	edge compression
200	16,208	12,445,795	3.203
400	8,052	7,924,106	5.031
1500	2,205	1,871,661	21.299

Table 5.3: Compression values for different cluster sizes on `wikipedia`

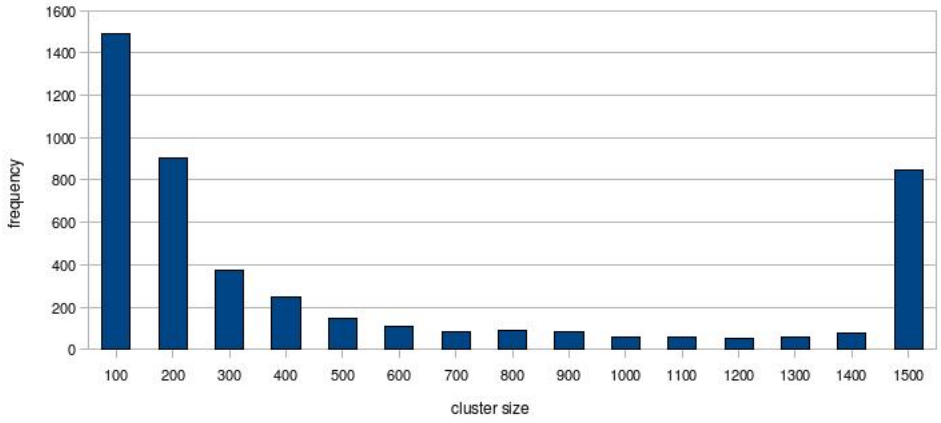


Figure 5.4: Chart of cluster size vs. frequency of `wiki` (without compaction)

Observations:

- From Table 5.3, it can be seen that, edge compression improves by more than 6 times, when `maxClusterSize` increases from 200 to 1500. As in the case of `dblp3`, the distribution of cluster sizes (given in Figure 5.3) can explain this better.
- Figure 5.4 shows the distribution of cluster sizes in the wikipedia dataset. It can be observed that, unlike `dblp3`, wikipedia has many communities of large size.
- The frequency for cluster size of 1500 in Figure 5.4 indicates that there are communities of much larger size than 1500, and in the present clustering, they might have been broken into smaller ones, to enforce the bound on maximum size of clusters. But for ease of handling, we set the `maxClusterSize` to 1500 for experiments on `wiki`, unless specified otherwise.

5.4 Effect of parameters and heuristics on edge compression

In this section, we study the effect of parameters and heuristic choices (discussed in Section 4.3) on edge compression. Here, we don't bother with node compression, since, as explained in Section 5.3, it is not difficult to obtain a large value for the same.

For testing the effect of a particular choice, we change the value of that variable alone, keeping the others constant. This section uses the base implementation with `maxClusterSize` set to 400 for `dblp3` (as explained in Section 5.3.1) and 1500 for `wiki` (as explained in Section 5.3.2). Deviations from BI are specified when the results are presented.

5.4.1 H1 - start node

In the base implementation, start node was heuristically chosen to be the one with highest degree in the remainder graph. In this section, we test the effect on edge compression, when the start node is the one with lowest degree. Table 5.4 compares edge compression on `dblp3` for the two choices.

	edge compression		
	maxClusterSize		
start node	200	400	800
min degree	11.81	14.39	16.95
max degree	12.78	15.53	18.55

Table 5.4: Edge compression for different choices of start node, on `dblp3`.

Observations:

- From Table 5.4, it can be observed that the compression obtained with the base implementation is always higher than that were the start node is the min degree node. Hence, we stick with max degree start node in all of the experiments, unless specified explicitly.

5.4.2 H2 - nodes spreading in each step

In this section, we compare the effect of the following two options for H2, on edge compression:

(a) all active nodes spread in each step of the walk

(b) only a single node spreads in each step

But, before we proceed, we will first test the effect of `m` on H2(b).

Effect of `m`

`m` decides the number of iterations in a single batch, which is calculated as `m` × `maxClusterSize`. Table 5.5 shows the effect of `m` on edge compression. It is plotted in Figure 5.5.

m	# clusters	# inter-cluster edges	edge compression
1	74,356	121,460	17.494
3	73,739	119,357	17.803
5	73,839	118,406	17.946
10	73,693	119,147	17.834

Table 5.5: Edge compression for different values of `m` on `dblp3`. (*settings: spreadProbability = 75, maxClusterSize = 1500, no compaction)*

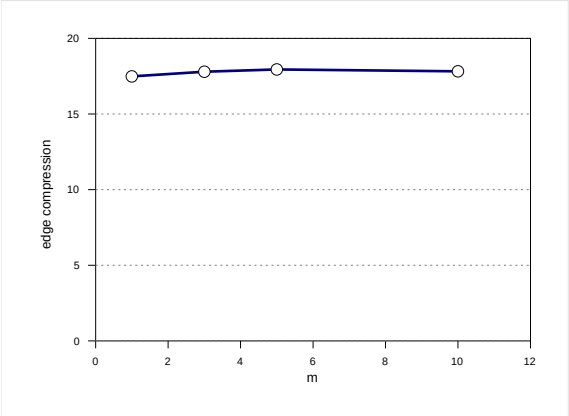


Figure 5.5: Effect of `m` on edge compression in `dblp3` (values from Table 5.5)

Observations:

- From Figure 5.5, it can be seen that, the effect of `m` on edge compression is negligible, as well as, inconsistent (the compression decreases when `m` increases from 5 to 10, in Table 5.5).

Comparison between choices a and b of H2

For comparison of choices (a) all active nodes spreading vs. (b) single node spreading in each step, we will use the best clustering obtained for H2(b), which is the one with `m` set to 5 (refer Table 5.5). For H2(a), we use the base implementation. Table 5.6 compares the two.

H2	# clusters	# inter-cluster edges	edge compression
(a)	61,633	96,101	22.115
(b)	73,839	118,406	17.946

Table 5.6: Edge compression for different choices of H2 on `dblp3`. (*settings: maxClusterSize = 1500, no compaction)*

Observations:

- As can be seen from Table 5.6, BI is able to achieve much higher compression than the clustering with H2(b). Hence, in all experiments, we use H2(a), i.e. spreading from all active nodes, unless stated explicitly.

5.4.3 H3 - spread probability

The fraction of probability that a node shares with its neighbors, is determined by the parameter `spreadProbability`. Table 5.7 summarizes the effect of varying this parameter, on edge compression in `dblp3`.

spreadProbability	# clusters	# inter-cluster edges	edge compression
25	79,065	132,374	16.052
50	78,435	131,466	16.163
75	74,356	121,460	17.495
85	71,364	115,668	18.371
95	65,616	109,715	19.367

Table 5.7: Edge compression for different values of `spreadProbability` on `dblp3`. (*settings: H2(b) (single node spreads in each step, with m = 1), maxClusterSize = 1500, no compaction.)*

Observations:

- From Table 5.7, as the value of `spreadProbability` increases, edge compression also increases. Also note that, the number of clusters reduce by about 13,500 which suggests that, on the average, clusters found are of larger size. Since with higher `spreadProbability`, larger fraction of total probability can escape the cluster boundary, many of the large clusters found, could be merging together multiple smaller ones. To avoid such effects, we stick to the setting of 0.5 for all the experiments, unless mentioned otherwise.

5.4.4 H6 - upper bound on active nodes

Factor `f` decides the bound on the number of nodes that are active at any instant of spreading. The bound is computed as `f` × `maxClusterSize`. Table 5.8 summarizes the compression figures obtained for different values of `f`. The entry 'no bounds' in the table is for the case where there was no bound on the number of active nodes. `f` vs. edge compression is plotted in Figure 5.6.

f	# clusters	# inter-cluster edges	node compression	edge compression	time (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 5.8: Compression for different values of `f` on `dblp3`. (*settings: maxClusterSize = 1500*)

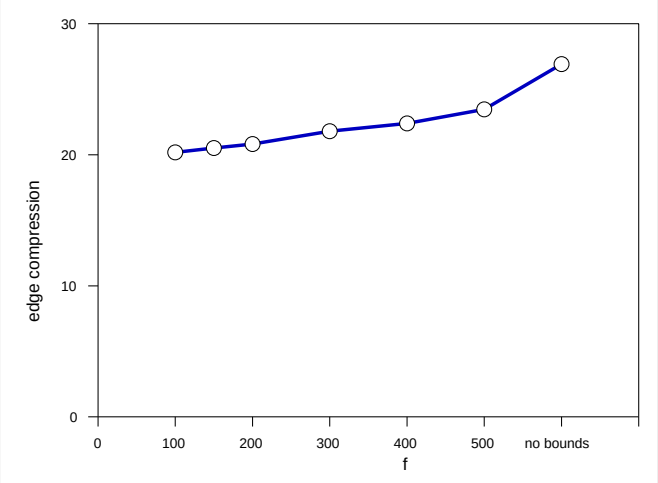


Figure 5.6: Effect of `f` on edge compression in `dblp3` (refer Table 5.8)

Observations:

- It can be noticed that, in Table 5.8, edge compression improves with increase in `f`.
- For the case where the number of active nodes are not bound, compression improves to about 27. But on comparing it with the compression of 23.4 obtained when `f` is 500, we observe that, this is not very substantial. Also, for an improvement in compression by a factor of 1.14, we incur 2.5 times the processing cost. Hence, in all experiments, we restrict the number of active nodes to `f` $\times$ `maxClusterSize`, where `f` is set to 500, unless explicitly specified.

5.4.5 H7 - behavior on maxActiveNodeBound

When the number of active nodes are constrained to be within the `maxActiveNodeBound`, the algorithm has two options when this bound is reached - abort the search (H7(a)) or continue spreading in the explored neighborhood (H7(b)). For the latter choice, the algorithm will stop only if it finds that the conductance is not improving with more steps of the walk. Table 5.9 has the numbers, showing the implication of this decision.

	# clusters	**# inter-cluster edges**	**edge compression**	**time**
H7(a)	77,462	147,663	14.39	1.5 hrs
H7(b)	65,883	128,466	16.54	4 days

Table 5.9: Effect of H7 on compression of `dblp3` (*settings: startnode - minDegree, no compaction*)

Observations:

- In the Table 5.9, it can be seen that, edge compression does improve when the search for clusters is continued on reaching the bound.
- But, also observe that the processing time shoots up, to 4 days, which is just not acceptable. Thus, we use option H7(a), i.e., stop on `maxActiveNodeBound`, henceforth, unless specified otherwise.

5.4.6 H8 - compaction techniques

In Section 5.3.1 and 5.3.2, we discussed about the size distribution of clusters in `dblp3` and `wiki` datasets. It was found that both datasets have a large number of tiny clusters. Presence of such clusters in large numbers affected keyword search adversely, as was observed in our routine experiments. This served as a motivation to use a compaction method, which will process the clustering that is output by the `Modified Nibble` algorithm.

In the Heuristics section (Section 4.3), we proposed 3 compaction techniques - CP1, CP2 and CP3. CP1 and CP2 improves edge compression, since they combine clusters which may have edges across them. But it was observed that, applying CP1 and CP2, made the supernode graph, denser. In a dense supernode graph, search quickly spreads to a very large fraction of it, which in turn, incurs more cache misses. This increases the query answer time.

CP3 neither affects edge compression, nor does it make the supernode graph denser, since it groups only tiny clusters, that don't have cut edges. We choose CP3, since we want to strike a balance between the following:

- number of 'tiny' supernodes
- denseness of the supernode graph

Table 5.10 shows the number of supernodes before and after applying CP3 compaction on a `dblp3` and a `wiki` clustering.

dataset	before	after
`dblp3`	70,189	31,341
`wiki`	11,808	11,304

Table 5.10: Effect of CP3 compaction on the number of clusters, in `dblp3` and `wiki`.

5.4.7 H9 - co-citation heuristic for wikipedia

In this section, we try to achieve better edge compression by leveraging on the co-citation in wikipedia, using the H9 heuristic (remove hub nodes, prior to clustering). The number of hub nodes removed is calculated as `t` $\times$ `maxClusterSize`.

t	# clusters	# inter-cluster edges	edge compression
-	2,350	1,777,217	22.431
1	2,294	1,334,752	29.867
2	2,290	1,304,721	30.554

Table 5.11: Effect of H9 on edge compression of `wiki`. (*settings: start node - minDegree, H7(b) - continue on maxActiveNodeBound*)

Observations:

- From Table 5.11, we observe that, when top indegree nodes are removed, edge compression increases from 22.4 to 29.8. This suggests that, degree of co-citation of these nodes are high.

- It can also be observed in Table 5.11 that, by removing twice the number of top indegree nodes, improvement in compression is negligible. This could be because, co-citation drops with decreasing degree.

5.4.8 H10 - heuristics for graph formations

The base implementation of `Modified Nibble` clustering algorithm that we considered until now, did not prevent the creation of graph formations (Section 4.4). Table 5.12 gives the numbers for BI, as well as, for different combinations of heuristics, on `dblp3` and Table 5.13 gives the same for the `wiki` datagraph. It is evident from these numbers that, graph formations are much more prevalent than thought before.

Heuristic	**maxClusterSize**	**Bridge**	**V**	**Umbrella**
BI	200	480	148	3,466
BI	400	412	126	3,014
BI + H1(b)	400	584	95	4,588
BI + H1(b) + H7(b)	400	327	22	1,058

Table 5.12: Graph formations on `dblp3` (*settings: no compaction*)

Heuristic	**maxClusterSize**	**Umbrella**
BI	1500	180,725
BI + H1(b) + H7(b)	1500	291,068
BI + H1(b) + H7(b) + H9	1500	246,864

Table 5.13: Graph formations on `wiki` (*settings: no compaction*)

Following are the two options that we considered in Section 4.4.6, to remove graph formations (H10):

(a) Post-process: add the abandoned nodes, after finding the best cluster.

(b) Abandoned node awareness: prevent the occurrence of formations right from the creation of candidate clusters and discard those for which, the final size goes beyond `maxClusterSize`.

Dataset	**maxClusterSize**	**Final maxClusterSize**
dblp3	200	323
wiki	1500	5627

Table 5.14: Increase in the final cluster size using H10(a)

Observations:

- Heuristic H10(a) removes all formations from the clustering. But, it is obvious that, rearranged clusters can have sizes greater than `maxClusterSize`. Table 5.14 shows the final maximum size of clusters for `dblp3` and `wiki` datasets, using H10(a). Though the increase in size for the `dblp3` dataset is within acceptable limits, the increase for `wiki` is not. H10(b) will produce formation-free clusters of size within the `maxClusterSize` parameter. So, we will use H10(b) henceforth.

5.5 Final settings for Modified Nibble clustering

Table 5.15 specifies the parameter values and heuristic choices for the final implementation (FI) of the `Modified Nibble` clustering algorithm.

Heuristic / Parameter		**Choice / Value**
H1 - start node	:	(a) - max degree
H2 - nodes spreading in each step	:	(a) - all active nodes
H3 - self-transition probability	:	0.5
H4 - number of iterations in a batch	:	APGP series with `a` = 2, `d` = 7, `r` = 1.5
H5 - upper bound for number of steps	:	`maxClusterSize`
H6 - `maxActiveNodeBound`	:	`f` = 500
H7 - behavior on H6	:	(a) - stop on `maxActiveNodeBound`
H8 - compaction	:	CP3 - naïve compaction of tiny clusters
H9 - co-citation	:	no
H10 - graph formation	:	(b) - abandoned node awareness

Table 5.15: Heuristic choices for the final implementation (FI)

5.5.1 Compression using FI

Table 5.16 shows the compression obtained on `dblp3` using the final implementation, and Table 5.17 gives the same for `wikipedia`.

mcs	**# clusters**	**# inter-cluster edges**	**node compression**	**edge compression**
100	50,102	202,436	35.36	10.497
200	36,227	161,973	48.89	13.119
400	31,215	136,347	56.75	15.585
800	28,390	116,360	62.39	18.262

Table 5.16: Compression values for different cluster sizes on `dblp3` using FI

mcs	**# clusters**	**# inter-cluster edges**	**edge compression**
1500	11,260	2,440,205	16.336
1600	11,305	2,304,976	17.295

Table 5.17: Compression on `wikipedia` using FI

mcs	# clusters	# inter-cluster edges	edge compression
1500	15,314	3,064,809	13.007

Table 5.18: Compression on `wikipedia` using BI + CP3

Observations:

- Comparing the compression obtained by FI and BI on `dblp3` (Tables 5.16 and 5.2), we observe that, in spite of the fact that BI uses CP1 compaction, which reduces cut-edges, FI is able to beat BI for sizes 100, 200 and 400. For 800, the compression achieved by both are very close.
- Comparing the compression obtained by FI with CP3 and BI with CP1 on `wiki` (Tables 5.17 and 5.3), we observe that, the compression obtained by FI + CP3 is much below the latter. However, CP1 can cause bad search performance. A more meaningful comparison is to look at the compression obtained by BI, with CP3 compaction, given in Table 5.18. From Tables 5.17 and 5.18, we observe that, FI achieves a compression of 17 on `wiki`, when compared to 13 obtained by BI, for `maxClusterSize` set to 1500.

Chapter 6

Comparison with Other Clustering Algorithms

In this section, we compare the final implementation of `Modified Nibble` clustering algorithm (FI), against EBFS (discussed in Section 2.1.3) and Metis (Section 2.2.1), with regard to the following metrics:

- edge compression on `dblp3` and `wiki` datasets (described in Section 5.1).
- connection query performance for a set of select queries, using the Incremental Expansion Backward search algorithm (outlined in Section 2.1.2), on the `dblp3` dataset.
- near query performance for a set of select queries, using the external memory near query algorithm described in Section 2.1.2, again on the `dblp3` dataset.
- time and space requirements for clustering.

Performance 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. All results presented were taken on a cold cache. Experiments on `wiki` were conducted on a blade of eight 2.50 GHz Intel Xeon CPUs, with a combined RAM of 8 GB, running Debian 4.0. Observations are analyzed, as and when the results are presented. Please note that, in the charts, we refer FI as `modNib` (short for 'modified nibble'). Also, we refer the implementation of Incremental Expansion Backward Search algorithm on BANKS, as 'external memory BANKS'.

6.1 EBFS

Since the processing done by EBFS is minimal, we do not consider the time and space taken by it. In this section, we compare only its edge compression and search performance, with FI.

6.1.1 Edge compression

Table 6.1 gives the compression values of EBFS on `dblp3`. Figure 6.1 compares the edge compression values of FI with EBFS.

Observations:

cluster size	# clusters	# inter-cluster edges	edge compression
100	17,714	335,819	6.327
200	8,857	272,161	7.807
400	4,429	219,591	9.676
800	2,215	170,731	12.446

Table 6.1: EBFS - Edge compression on `dblp3` for different cluster sizes.

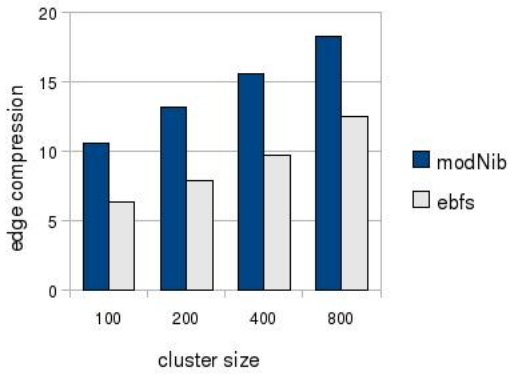


Figure 6.1: Comparison of edge compression on `dblp3` between FI and EBFS

- From Figure 6.1, it is quite obvious that FI is able to achieve better edge compression than EBFS, on the `dblp3` dataset.

6.1.2 External memory connection queries

The queries used for connection query search on external memory BANKS are specified in Table 6.2. Comparison of search performance between FI and EBFS can be found in Figures 6.2, 6.3 and 6.4. The FI and EBFS clusterings used for performance measurements are the ones whose `maxClusterSize` is 400.

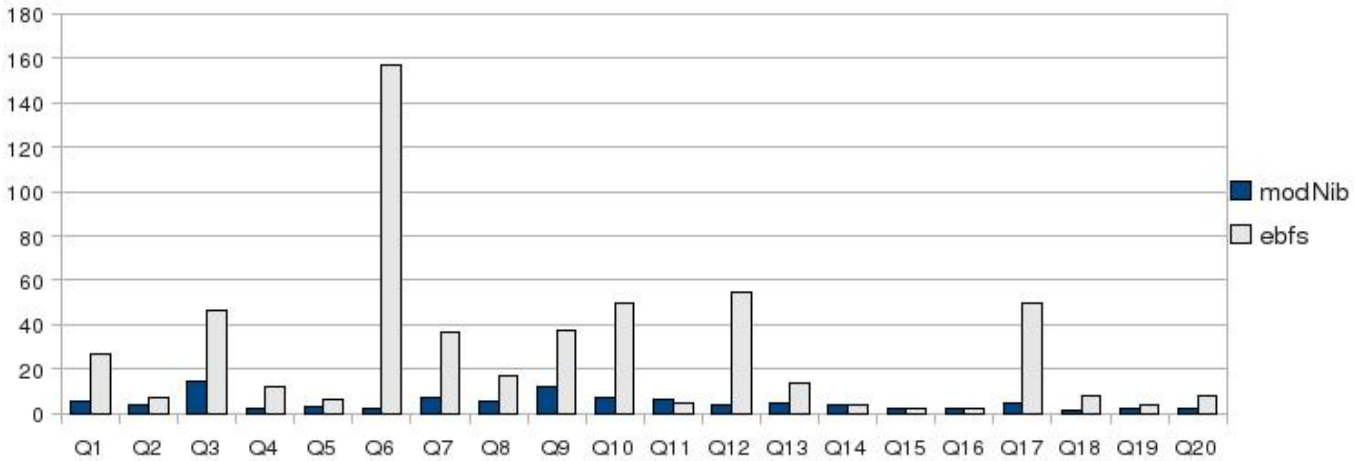


Figure 6.2: CPU + IO time (sec) : connection query on `dblp3`

Q1	sudarshan soumen
Q2	vapnik support vector
Q3	divesh jignesh jagadish timber querying XML
Q4	sudarshan widom
Q5	giora fernandez
Q6	david fernandez parametric
Q7	chaudhuri agrawal
Q8	widom database
Q9	raghu deductive databases
Q10	“prabhakar raghavan” “raghu ramakrishnan”
Q11	rozenberg “petri nets”
Q12	rozenberg janssens “graph grammars”
Q13	silberschatz “disk arrays”
Q14	ramamritham “real time”
Q15	“howard siegel” SIMD
Q16	frieze “random graphs”
Q17	romanski ada
Q18	banerjee “distributed memory” multicomputers
Q19	didier “possibilistic logic”
Q20	tamassia “graph drawing”

Table 6.2: connection queries for `dblp3` dataset

Observations:

- From Figures 6.2, 6.3 and 6.4, it is obvious that the final implementation of modified nibble is out-performing ebfs by a very large margin.

6.1.3 External memory near queries

The queries used for near keyword query on external memory BANKS are specified in Table 6.3. Comparison of near query performance between FI and EBFS can be found in Figures 6.5, 6.6 and 6.7.

Observations:

- Figure 6.5 shows the number of supernodes which contain the near set nodes, for the FI and EBFS clusterings. For the near queries considered, these numbers are identical for both. However, in almost all cases, FI has lower values than EBFS.
- From Figure 6.7, it can be seen that FI has lesser number of cache misses, which in turn, explains the lower CPU + IO time obtained, given in Figure 6.6.

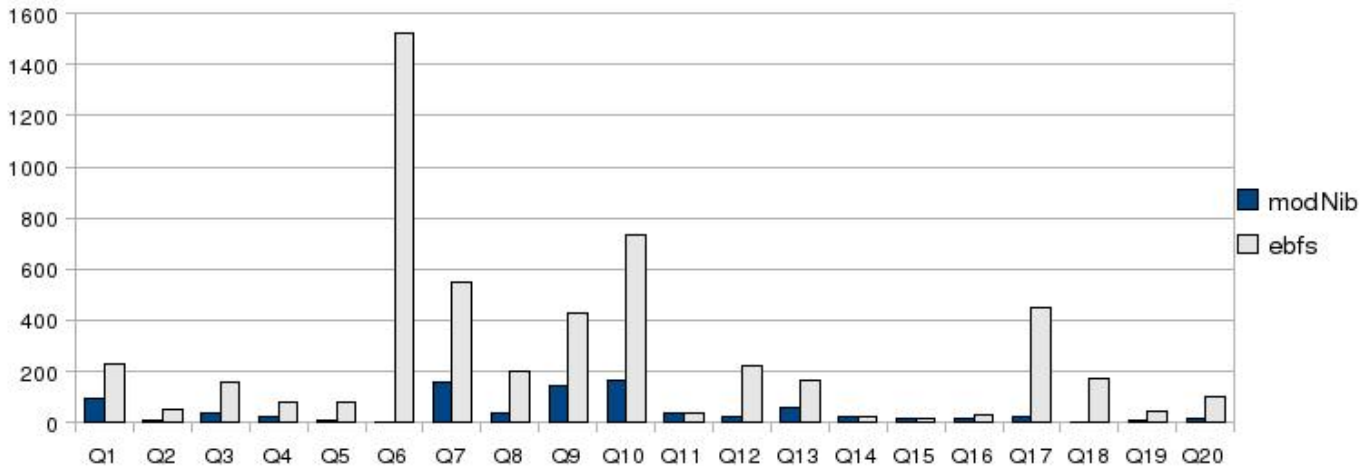


Figure 6.3: cache misses : connection query on `dblp3`

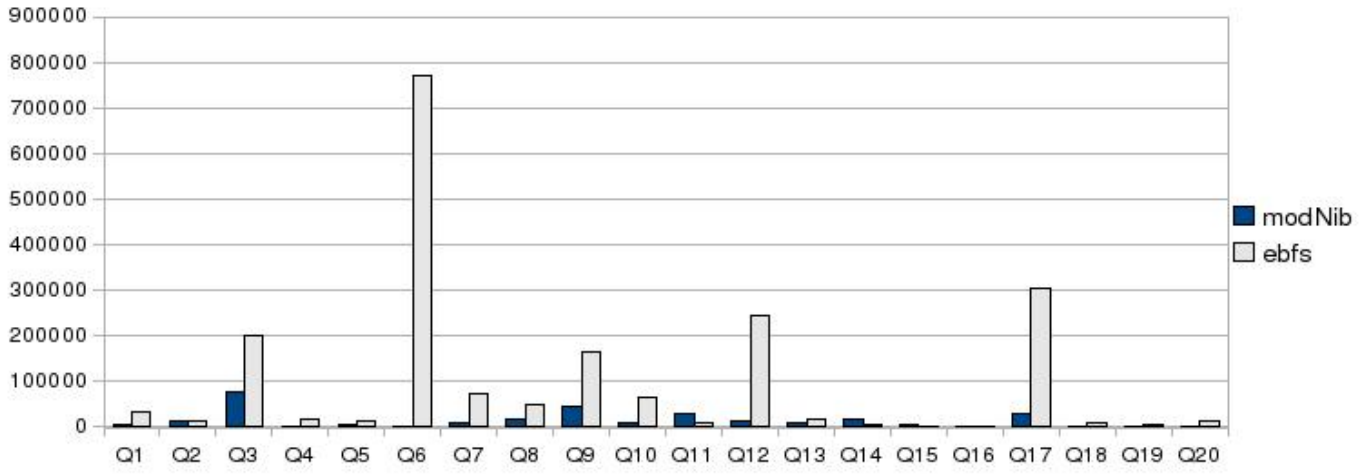


Figure 6.4: number of nodes explored : connection query on `dblp3`

N1	author (near “data mining”)
N2	paper (near christos faloutsos nick roussopoulos)
N3	author (near “query processing”)
N4	author (near “possibilistic logic”)
N5	paper (near chaudhuri agrawal)
N6	paper (near “deductive databases”)
N7	paper (near “random graphs”)
N8	author (near “handwriting recognition” “subgraph isomorphism”)
N9	paper (near “branching programs”)
N10	paper (near “petri nets” “context free grammars”)
N11	author (near “graph grammars”)
N12	author (near “load balancing”)
N13	author (near “scan circuits”)
N14	author (near “kolmogorov complexity” “match making”)
N15	author (near “distributed memory” multicomputers)
N16	author (near “image retrieval”)
N17	author (near “reliability performance”)
N18	paper (near smith siegel McMillen)
N19	author (near “maximum matchings” “game trees”)
N20	author (near “NP complete”)

Table 6.3: near queries for `dblp3` dataset

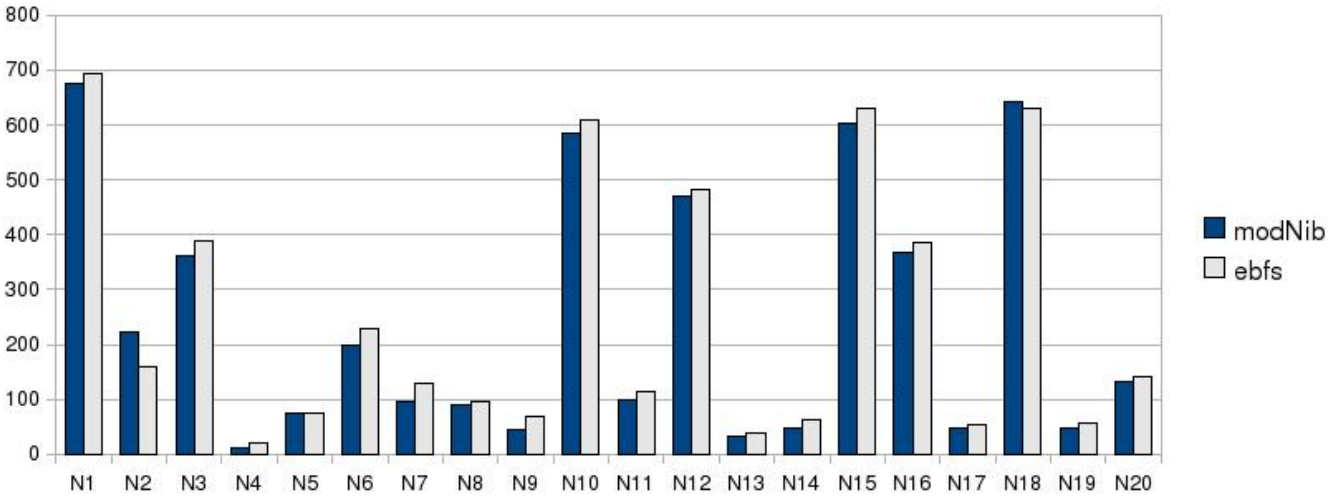


Figure 6.5: number of supernodes with near keywords match : near queries on `dblp3`

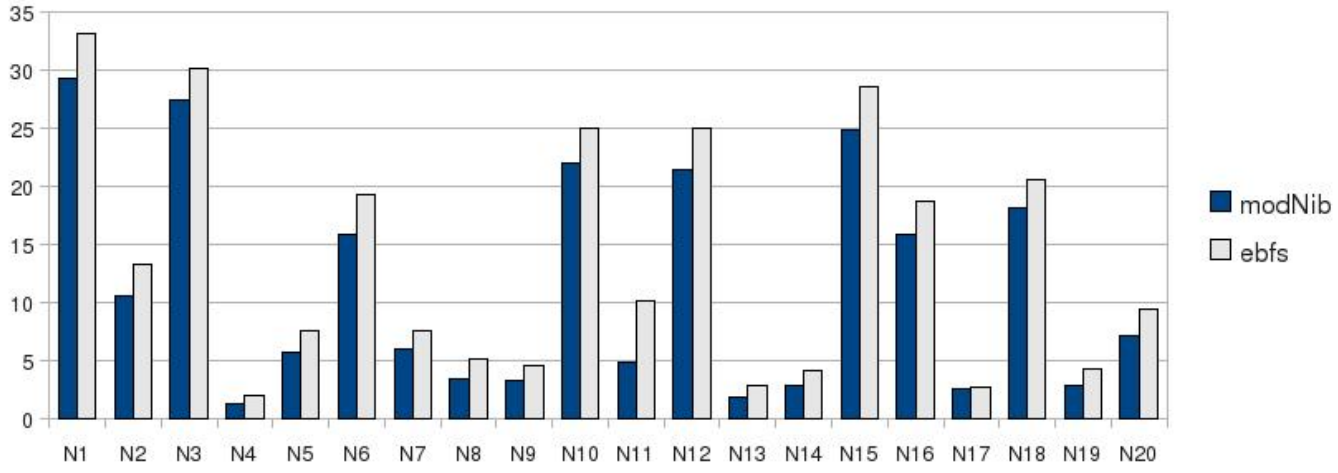


Figure 6.6: CPU + IO time (sec) : near queries on `dblp3`

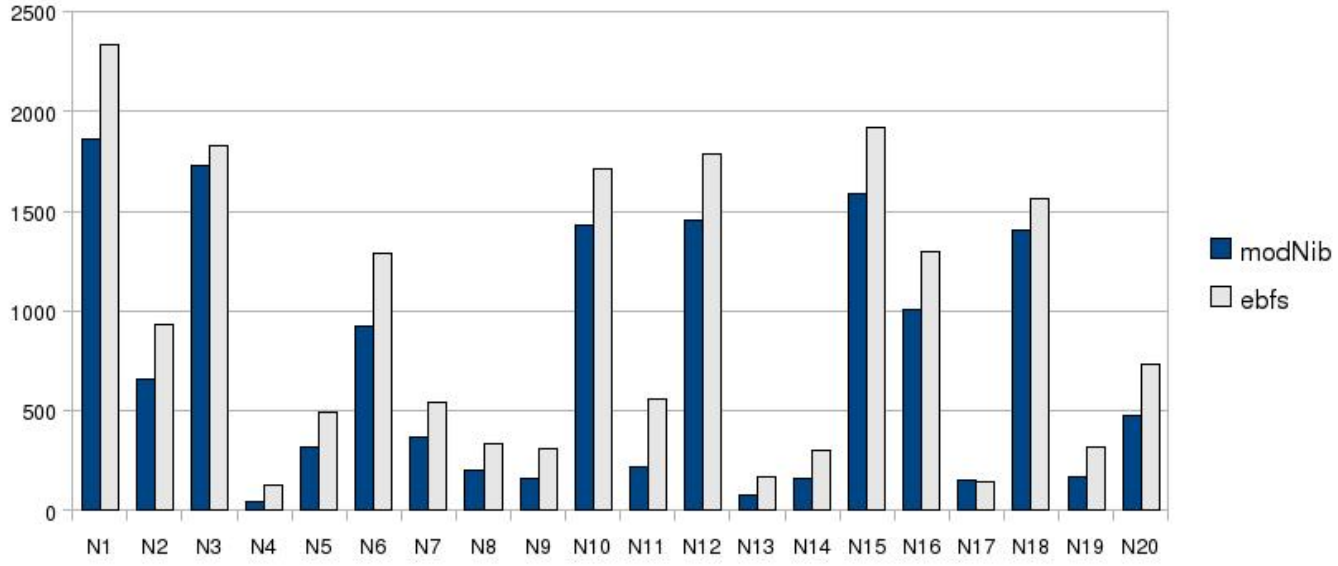


Figure 6.7: cache misses : near queries on `dblp3`

6.2 Metis

Metis is a graph partitioning algorithm (discussed in Section 2.2.1), and its parameters and objectives are much different from FI. This poses a difficulty in comparing the two. But, for comparison purposes, we use clusterings of each, where the `maxClusterSize` and average cluster sizes are comparable. Here, we use kMetis for clustering. It belongs to the Metis family, and uses k-way partitioning method.

6.2.1 Edge compression

Table 6.4 gives the edge compression obtained by Metis for different settings of `k`, on `dblp3` and Table 6.5 gives the same for `wiki` dataset.

k	maxClusterSize	# inter-cluster edges	edge compression
4,000	476	105,666	20.110
5,000	401	114,189	18.609
10,000	332	164,233	12.938
30,000	335	220,961	9.616

Table 6.4: Metis - Edge compression on `dblp3` for different `k`.

k	maxClusterSize	# inter-cluster edges	edge compression
2,000	1,663	1,441,411	27.657
3,000	1,096	2,535,532	15.722
4,000	16,353	4,364,488	9.134

Table 6.5: Metis - Edge compression on `wiki` for different `k`.

Observations:

- From Table 6.4, note that, edge compression drops considerably, when the number of supernodes (`k`) increase.
- Maintaining the number of supernodes to around 30,000 and thus, the average super node size, we observe that FI is able to obtain a compression of 15.6 (Table 5.16), when compared to the compression of 9.6 obtained by Metis.
- Compression obtained on wikipedia by Metis, given in Table 6.5, also shows a similar trend as that for `dblp3`. When `k` changes from 2,000 to 4,000, the compression falls from 27.6 to 9.1.
- Also, it is interesting to note that, the clustering with `k` set to 4,000 (Table 6.5), has a cluster of size 16,353 while the average cluster size is about 660, despite the claim that Metis creates clusters of roughly the same size.
- Comparing with FI, we see that, it is able to get a compression of 17.3, with 11,305 clusters where maximum cluster size is 1600 (Table 5.17).

6.2.2 External memory connection queries

The queries used for connection query on external memory BANKS are specified in Table 6.2. Comparison of search performance between FI and Metis can be found in Figures 6.8, 6.9 and 6.10.

FI clustering used is the one with `maxClusterSize` = 400 (and it has 31,215 supernodes), and the Metis clustering used is the one with `k` set to 30,000 (and its maximum cluster size is 335), to make them comparable.

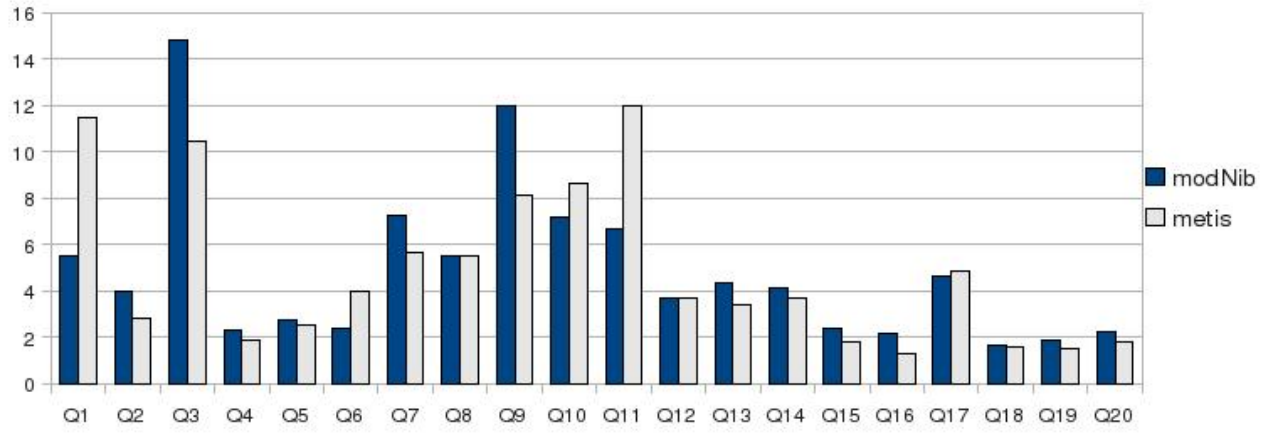


Figure 6.8: CPU + IO time (sec) : connection query on `dblp3`

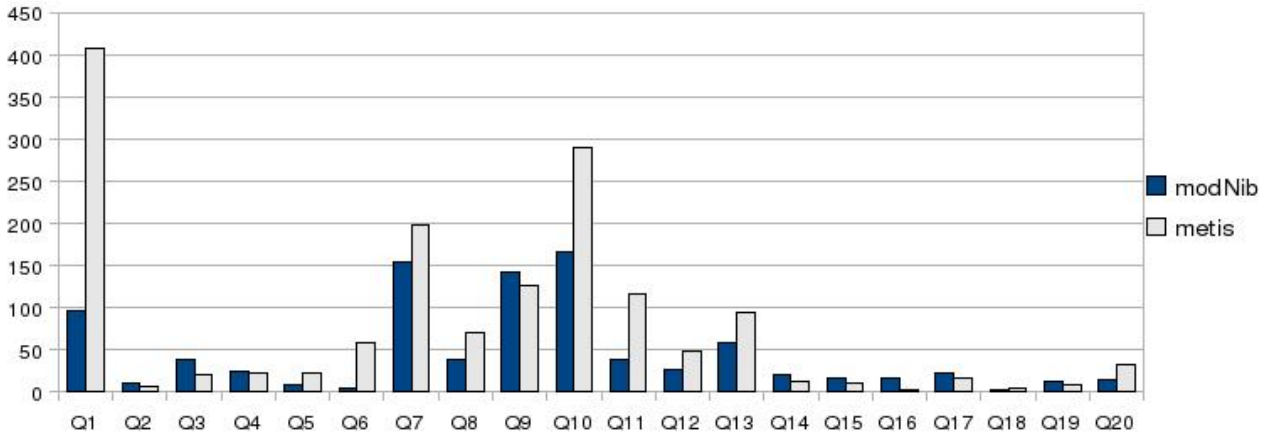


Figure 6.9: cache misses : connection query on `dblp3`

Observations:

- From Figures 6.8, 6.9 and 6.10, we observe that, Metis performs really well on some keyword queries, while FI outperforms Metis on some others. At the same time, none of the clustering algorithm 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.

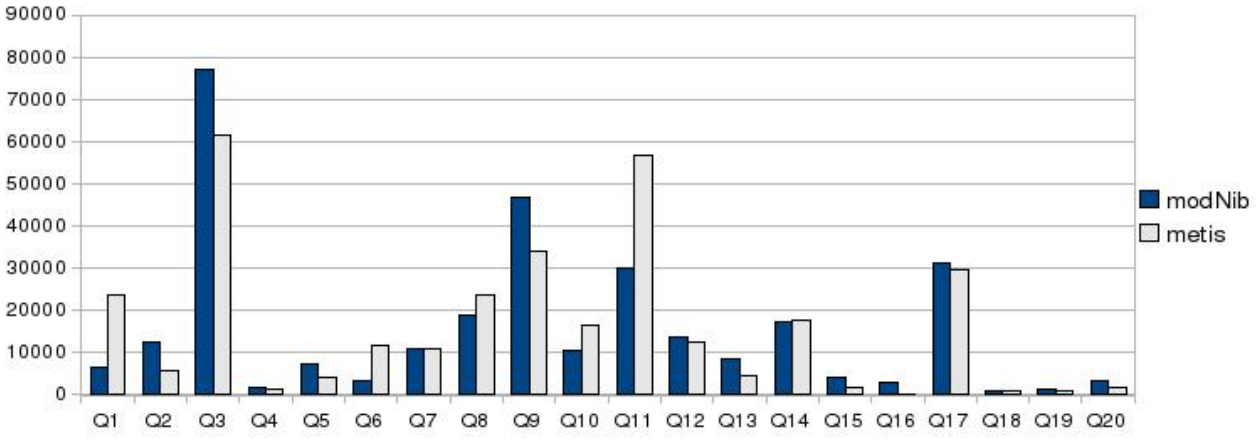


Figure 6.10: number of nodes explored : connection query on `dblp3`

6.2.3 External memory near queries

The queries used for near query search on external memory BANKS are specified in Table 6.3. Comparison of near query performance between FI and Metis can be found in Figures 6.11, 6.12 and 6.13.

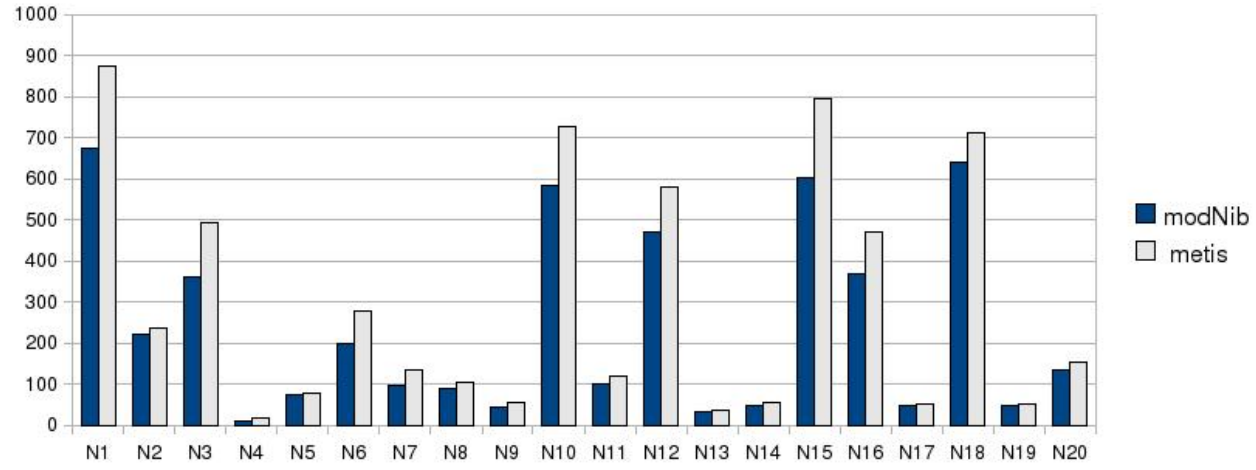


Figure 6.11: number of supernodes with near keywords match : near queries on `dblp3`

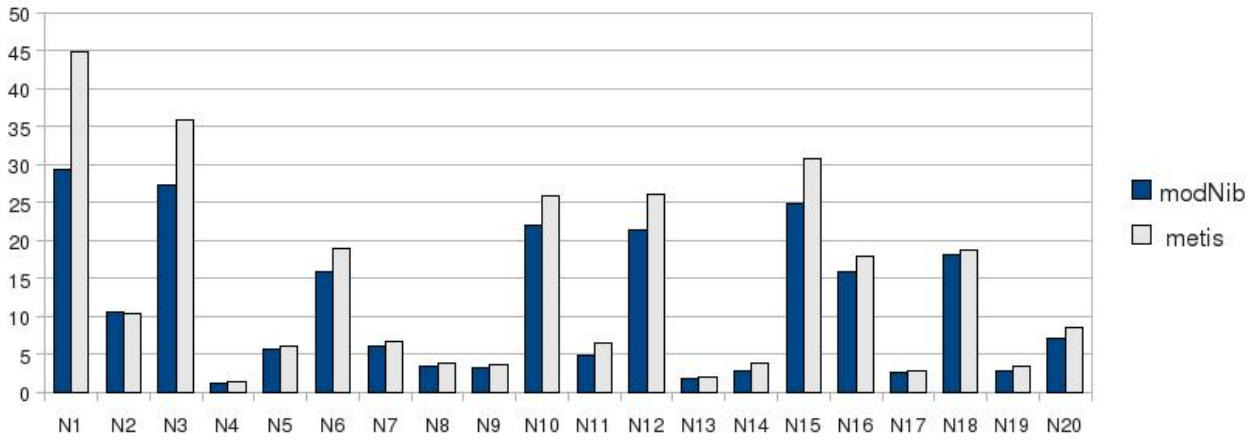


Figure 6.12: CPU + IO time (sec) : near queries on `dblp3`

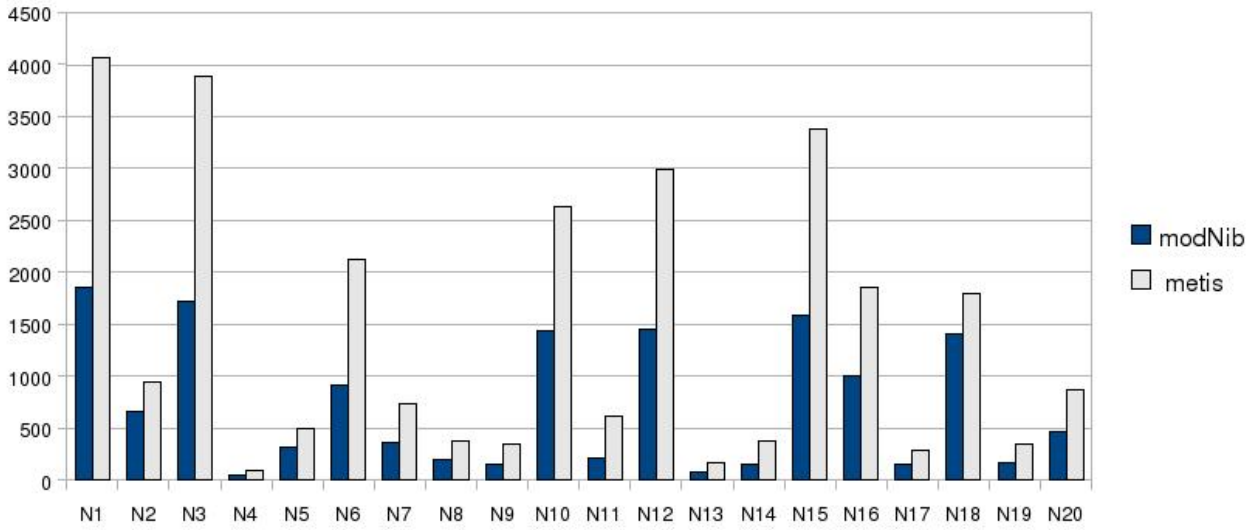


Figure 6.13: cache misses : near queries on `dblp3`

Observations:

- Figure 6.11 compares the number of supernodes which contain the near set nodes, for FI and Metis clusterings. Note that, in all cases, the number of supernodes for FI are lesser than that of Metis. This indicates that, clusters produced on the link graph by FI, also clusters the paper titles in `dblp3`.
- From Figure 6.13, it can be seen that FI has significantly lesser number of cache misses, and hence it is able to beat Metis on all queries (Figure 6.12).

6.2.4 Time and space requirements for clustering

In this section, we compare the time and memory required by FI and Metis for clustering `dblp3` and `wiki` datasets. Table 6.6 shows the sizes of these two datasets. Time and space required by FI is shown in Table 6.7, and that for Metis is in Figures 6.14 and 6.16. Space requirement of Metis is plotted in Figures 6.15 and 6.17.

dataset	**size**
dblp3	132 MB
wiki	1.9 GB

Table 6.6: Size of datasets

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

Table 6.7: FI: time and space requirements, for all values of `maxClusterSize`

Observations:

- From Table 6.7, it is clear that the space requirements of FI is very close to the size of the graph. Thus, the graph has to just fit in memory; additional memory requirements are very little.
- For all values of `maxClusterSize`, the difference in the time and space requirements of FI is negligible. Though `maxClusterSize` will affect the requirements, since its value is mostly much lesser when compared to the size of the input graph, this effect can be ignored.
- From Figures 6.14 and 6.16, we observe that the space required by Metis is many times higher than the size of the graph. For example, when `k` is 40,000 (Figure 6.14), the memory required is about 96 times the size of `dblp3`.
- From the plot of `k` vs. space, for Metis, given in Figures 6.15 and 6.17, we observe that, space requirement grows almost linearly with `k`, but the constants are quite huge. Also, for `dblp3` (Figure 6.15), the relationship between `k` and space required, is initially super-linear, and then becomes linear when `k` increases beyond 20,000.
- Comparing the time required by FI and Metis, we observe that FI takes much more time than Metis. But, we also note that, almost always, clustering is done offline. Thus, time may not always be an issue; but space might be.

k	space	time
4000	928 MB	
5000	968 MB	
10000	1.53 GB	$\sim$ 5 mins
20000	3.88 GB	
30000	7.89 GB	
40000	12.8 GB	

Figure 6.14: Metis: time and space requirements on `dblp3`, for different values of `k`

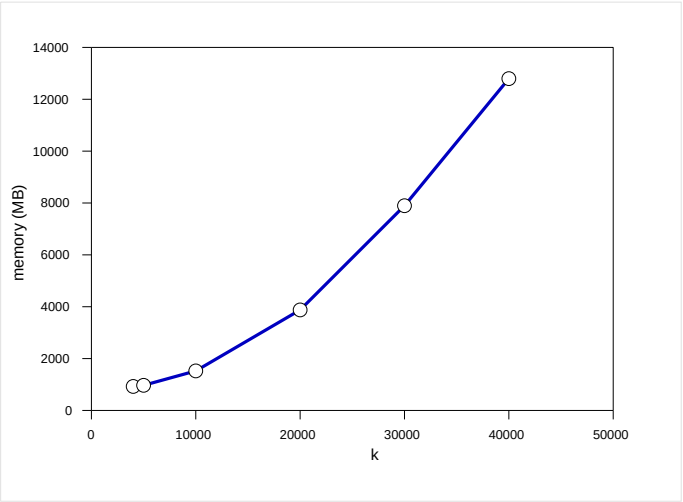


Figure 6.15: space required for Metis for different `k` on `dblp3`

k	space	time
2000	4.8 GB	
3000	5.16 GB	$\sim$ 1.5 hrs
4000	5.53 GB	

Figure 6.16: Metis: time and space requirements on `wiki`, for different values of `k`

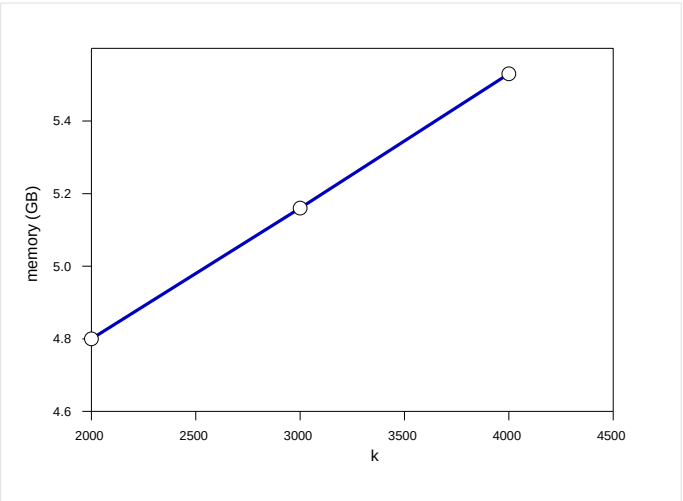


Figure 6.17: space required for Metis for different `k` on `wiki`

Chapter 7

Conclusions and Future Work

Clustering is the process of finding out a grouping of the nodes of a graph such that, nodes belonging to the same cluster link to each other more often, than to nodes in other clusters. Clustering is important for external memory as well as distributed keyword search ([Sav09, Agr09]), since it helps to substantially reduce the query answering time, by localizing the search to a few clusters.

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` clustering algorithm. Maintaining its core structure and attaching various heuristics, we obtained a base implementation for the same. This was tested on the `dblp3` and `wiki` datasets, to understand its performance and the effect of heuristics. During this exercise, we refined the algorithm and the heuristics, to obtain its final implementation.

`Modified Nibble` clustering algorithm was tested on the external memory keyword search algorithm used in BANKS for a set of keyword queries and near queries. Its performance was compared with that of clusterings produced by EBFS and Metis. It was observed that, `Modified Nibble` is able to consistently outperform both EBFS and Metis, on near-queries. For keyword queries, though `Modified Nibble` beats EBFS substantially, it is not able to beat Metis consistently.

Following is the proposed direction of future work:

- The objective of minimizing conductance while clustering the graph, has enabled us to get good near-query performance on that graph. However, the same objective gave an average performance on keyword queries, when compared to Metis. Thus, the search for a clustering objective that can improve keyword query performance on external memory search systems, has to continue.

- In the evaluation of heuristics, we studied the effect of each one in isolation. But, as a matter of fact, interaction amongst heuristics is possible, and the combined result could be much better than their individual results. We presented 10 heuristics, and they together have more than 2^{10} combinations, which makes it infeasible to study them all. But, it would be a good idea to test the effect of a few of the more intuitive combinations, thereby achieving a better compression and search performance.

- 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 available ([BV04]), graphs that are much larger than wikipedia, can be read into main memory. Since the memory requirements of our algorithm are linear in graph size, it should be possible to cluster any graph that fits in memory. It will be interesting to test 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. By modifying the algorithm to run in a distributed environment, such cases can be handled. In addition to that, nibbling out multiple clusters in parallel, to speed up the entire process, is also a promising area of future work.

Bibliography

[Agr09] Rakhi Agrawal. Keyword Search on External Memory and Distributed Graph. *MTech. Project 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.

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

[BV04] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. *Proc. of 13th International World Wide Web Conference*, pages 595–601, 2004.

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

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

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

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

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

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

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

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

[Kle98] Jon M. Kleinberg. Authoritative Sources in a Hyperlinked Environment. *Proc. 9th ACM-SIAM Symposium on Discrete Algorithms*, 1998.

[Luc] Lucene. `http://lucene.apache.org/`.

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

[NRS08] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. Graph Summarization with Bounded Error. *SIGMOD*, 2008.

[Sav09] Amita Savagaonkar. Distributed Keyword Search. *MTech. Project 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.

Appendix A

Clustering using Nibble Algorithm: Detailed Pseudocode

This section gives the pseudocode for different procedures described in the paper [ST04] by Spielman and Teng.

A.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 2.3.1, except that here, conductance is referred to as *sparsity*.

The above terms can also be defined for a subgraph of G induced by a subset of the vertices $W \subseteq V$, where $S \subseteq W$, as below:

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

Let $d(v)$ denote the degree of vertex v, A, the adjacency matrix of the unweighted graph, and D, the diagonal matrix with $(d(1), ..., d(n))$ on the diagonal. Then, the matrix P that represents the random walk with self-transition can be defined as $P = (AD^{-1} + I)/2$, where I is the identity matrix.

Also, define:

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

The 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:

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

A.2 Nibble

$C = \texttt{Nibble}(G, v, \theta_0, b)$

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 $\emptyset$.

Figure A.1: Pseudocode for `Nibble` algorithm

A.3 Random Nibble

$C = \texttt{RandomNibble}(G, \theta_0)$

(1) Choose a vertex v according to ψ_V

(2) Choose a b in $1, ..., \lceil log(m) \rceil$ according to

$$Pr[b = i] = 2^{-i}/(1 - 2^{-\lceil log(m) \rceil})$$

(3) $C = \texttt{Nibble}(G, v, \theta_0, b)$

Figure A.2: Pseudocode for `Random Nibble` algorithm

A.4 Partition

$D =$ `Partition`(G, θ_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 $D = V - W_{j+1}$

Figure A.3: Pseudocode for `Partition` algorithm

A.5 Multiway Partition

$\mathcal{C} =$ `MultiwayPartition`(G, θ, p)

(0) Set $\mathcal{C}_1 = V$ and $S = \emptyset$

(1) For $t = 1$ to $\lceil log_{17/16} m \rceil \cdot \lceil lg(m) \rceil \cdot \lceil lg(2/\epsilon) \rceil$

(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 $\mathcal{C} = C_{t+1}$

Figure A.4: Pseudocode for `Multiway Partition` algorithm

Appendix B

Documentation of the Java implementation of `Modified Nibble` Algorithm

Some of the important classes in the implementation are given below, two of which are elaborated in Sections B.1 and B.2.

`ModifiedNibble`	:	implements the `Modified Nibble` clustering algorithm.
`Graph`	:	this class represents the datagraph used by the clustering process. It handles all graph related operations.
`Cluster`	:	this data structure 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`	:	specifies the configuration for a particular run of the clustering, like `maxClusterSize`, `maxActiveNodeBound`, `spreadProbability` and the files for input and output.
`Utils`	:	provides utility functions, such as quick sort, and a cluster writer, which handles writing or appending to the output file, according to whether the clustering started from scratch, or resumed from an earlier run which was terminated forcefully by the user.

B.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. While computing the conductance of each set of nodes, it also makes sure that its abandoned nodes are added.

B.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.

B.3 Data structures

`IntArrayList` : It 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.

For speeding up the processing, 3 integer arrays and a boolean array, all of which had size set to the number of nodes in the graph, were used in place of `ArrayList`s, to minimize run time memory allocation, and these were re-used to the extent possible.

Appendix C

BANKS on Wikipedia

BANKS system [BHN+02], had earlier been tested on the graph representation of structured data such as Digital Bibliography Library Project (dblp) database, Internet Movie Database (imdb) and the US Patent database, and it was found to do well. A graph for semi-structured data, which is gaining popularity is the Wikipedia graph (wiki graph for short), which is formed from the articles in the Wikipedia site. A simple construction of wiki graph is as follows: each article is represented as a graph node, and a link between two articles is represented as an edge between their corresponding graph nodes. Such a graph was prepared and modifications required to input the graph to BANKS were done by Amita and Rakhi ([Sav09, Agr09]). Following were observed when keyword-querying wikipedia, using BANKS:

- Since the entire text of an article is used while finding the keyword nodes, many sets of keyword queries gave articles which contained all the keywords, but most of these occurrences were some lesser known senses of the keyword. An example is the query 'amte kiran bedi', which gave 'Padma Shri' as the top answer. It has the following: 'Baba Amte', 'Kiran Chandra Banerjee' and 'P.S. Bedi'.

- The entire article is treated as a single entity, and all the text in it is given equal importance. Words that occur in the reference section and those that occur in the title are treated in a similar way. Because of this, for many queries, answers returned were totally unintuitive. An example is the query 'sourav filmfare' which listed 'Nagma' as the top answer. The article contained a link to 'Filmfare Best Actress Award' and in the references, an external link to HindustanTimes article which contained 'Sourav' in the title, but it is not mentioned anywhere in the main text.

- Wikipedia has many pages that are lists of events that occurred in a particular year, topics related to a particular country, etc. Occurrence of keywords in these, may not always signify a strong relationship. But, in many queries, such pages were returned as the top ranked answer. For example, the query 'shahrukh booker' returned '1965' as the first ranked answer, which listed 'Shahrukh Khan' born on November 2, 1965, and 'Booker T.', American wrestler, born on March 1, 1965.

The above observations served as our motivation to do the following:

- Propose a graph representation for wikipedia, which can be used to find intuitive answers for keyword queries on the same. Our approach is described in Section C.1.

- Improve the preprocessing used in BANKS, to handle semi-structured data (wikipedia articles, in the present context). This is elaborated in Section C.2.

C.1 Fragmented graph for wikipedia

We do not treat the entire wikipedia article as a single entity. Instead, we view it as a set of fragments which are connected to each other, through their main article. Following are the salient points of this representation:

- Each article is fragmented into multiple parts, each consisting of the text that comes under a second level heading (sub-heading). The introduction (text before table of contents) and the infobox, constitute a fragment on their own, which we refer to as, the main fragment.
- The main fragment retains the title of the article, where as, the other fragment titles are set to `<main title>#<sub-heading title>`. All fragments are assigned new nodeids.
- Each fragment is now a node in the graph.
- Links are created from the main fragment to the other fragments of the same article.
- Links from a particular fragment now belongs only to that fragment, and not to the entire article, as was the case before. And links to a sub-heading of an article will now link to the fragment corresponding to that sub-heading, instead of the entire article.

The nodes in the answer tree returned by BANKS, using this graph, will now correspond to the fragments, instead of the article.

C.2 Modifications to preprocessing

Following are the modifications to the preprocessing stage, that we implemented:

- Earlier, node prestige of articles were computed after the addition of backedges. We observed that, because of this, lists accumulate fair amount of prestige and thus are ranked higher in the answer list, very often. We calculate the node prestige only with forward edges. Backedges are added to the graph, only after this computation is done.
- As mentioned before, all the text in an article were treated equally, regardless of where they occur. But, we treat titles differently from the remaining text. This is done through a Lucene index ([Luc]), by creating different fields to represent title and text, in a `Document` object.
- Answer ranking in BANKS takes into consideration, the node prestige of the articles. Earlier, since the entire text was treated equally, occurrence of a keyword in an article with a very high prestige, however unrelated it may be to the article, resulted in that article being listed in the top, in a search for that query. Thus, it is not able to judge the relevance of a word to the article in which it occurs. Our approach was to incorporate the relevance returned by Lucene, into the ranking. To do this, while indexing the fragments, we set the `boost` of the `Document` object, as the precomputed node prestige of the fragment. Now, for ranking, instead of node prestige, we use the score returned by Lucene (obtained from the `Hit` object), which is now a function of the node prestige of the fragment and the relevance of the word to the fragment.

C.3 Observations and future work

The above graph representation was created on a subset of articles from wikipedia. We started with about 52,000 articles, which on fragmentation resulted in 331,000 fragments. Thus, on an average, each article split into a main fragment and 5 new fragments.

BANKS was run on this graph, after preprocessing it with the modifications presented in Section C.2, and following are the observations:

- The top answers were more intuitive. For example, the query 'apollo purdue annan' returned 'Massachusetts Institute of Technology#Noted alumni' as the top answer. Apollo 11 Lunar Module Pilot - Buzz Aldrin, Former UN Secretary General - Kofi Annan, and the 10^{th} president of Purdue University - Martin Jischke, were alumni of MIT.
- List pages now occur with lesser frequency, in the top results.

The proposed direction of future work is as below:

- Presently, our implementation has been tested only on a subset of articles in wikipedia. Scaling it to deal with the entire set, which has over 2 million articles, is definitely worth pursuing.
- Near queries form an important class of keyword querying, which is not yet supported by web search engines. BANKS supports near querying. But, presently, there is no type for the answer (everything is just a page or an article). Using categories as types is one way of solving this, and will be an interesting area of future work.
- Currently, category pages are ignored. But, they can definitely provide additional information which can improve the results. Adding category nodes to the graph, and connecting them to articles which belong to them, can create shorter paths that link fragment nodes.