Multilevel Regularized Markov Clustering Algorithm

SOFTWARE PROJECT REPORT

Project Guide
Professor K.K Shukla

Group Members

Kumar Ankit
(09100EN051)

Amit Kumar Baranwal
(09100EN053)

B. Tech. Part III,
Computer Engineering,
Institute of Technology, BHU

Disclaimer: This report is submitted as a part requirement for the VI Semester Project in Department of Computer Engineering, Institute of Technology, Banaras Hindu University, Varanasi

INDEX

CERTIFICATE

This is to certify that Kumar Ankit, student of Department of Computer Engineering, IT BHU, has worked on the topic Multilevel Regularized MCL under my direct supervision and guidance, the findings of which have been incorporated in this project.
He has worked diligently, meticulously and methodically. He has prepared the report under my supervision and guidance and has been found satisfactory and is approved for submission.

Professor K. K Shukla
Department of Computer Engineering
IT BHU

CERTIFICATE

This is to certify that Amit Kumar Baranwal, student of Department of Computer Engineering, IT BHU, has worked on the topic Multilevel Regularized MCL under my direct supervision and guidance, the findings of which have been incorporated in this project.
He has worked diligently, meticulously and methodically. He has prepared the report under my supervision and guidance and has been found satisfactory and is approved for submission.

Professor K. K Shukla
Department of Computer Engineering
IT BHU

ACKNOWLEDGEMENTS

We would like to convey our deepest gratitude to Professor K.K Shukla, who guided us throughout this project. His keen interest, continuous motivation, suggestions and support helped us immensely in successfully completing this project.
We would also like to thank Dr. R. B. Mishra, Head of the Department, Department of Computer Engineering, for allowing us to avail all the facilities of the Department necessary for this project.

Kumar Ankit
Roll No. 09100EN051

Amit Kumar Baranwal
Roll No. 09100EN053

Computer Engineering,
Institute of Technology,
BHU

ABSTRACT

Algorithms based on simulating stochastic flows form natural solutions for the problem of graph clustering but due to their lack of scalability and fragmentation of output their use have been restricted. This project is an attempt to study implement and analyze the class of flow based graph clustering algorithms represented by **MCL.** MCL or Markov Clustering Algorithm are based on flows or transition probability between edges in graphs .Three versions of this algorithm namely MCL, R-MCL (Regularized MCL) and MLR-MCL (Multi-level Regularized MCL) have been successively analyzed and implemented. Each version has significant improvements over the previous one and seeks to do away with the drawbacks and limitations of previous one while retaining its strength.

First of all we discuss in brief the basic algorithm guiding the MCL. Its limitation and the reason for the limitations are highlighted. The first problem that is the fragmentation problem has suitably been redressed using a regularization step giving a variant of the algorithm we know as RMCL. Finally, a scalable multi-level variant of RMCL has been realized. In a multilevel algorithm the graph is successively coarsened to a manageable size, and a small number of iterations of flow simulation (RMCL) is performed on the coarse graph. The graph is then successively refined in incremental steps. The central intuition is that using flow values from simulation on coarser graph can lead to good initialization of flows in refined graphs.

To implement MLR-MCL a new graph representation technique has been adopted. Also to maintain coarsening information separate data structures have been used.. The Coarsening and the refinement step used in MLR-MCL implemented will be discussed in detail.

A comparison between the results of the three variants developed by us on two real graphs have been presented for analysis .

INTRODUCTION

Clustering is basically finding patterns in data or grouping similar groups of data points together in what we call *clusters.* In the context of Graphs, clustering a Graph means, grouping together vertices of a Graph based on edge structure so that we have many vertices within a cluster and very few edges between clusters. A graph cluster in its loosest sense is a connected component while in its strictest sense is a maximal clique of a graph.

Graph Clustering is an important problem with many applications in a number of disciplines. It ranges from discovering communities in social networks to image segmentation and from analyzing protein interaction networks to the circuit layout problem.

Given the importance of the problem numerous solutions have been proposed in the literature. Spectral methods that target weighted cuts form an important class of such algorithms and are shown to be very effective for problems such as image segmentation. Multi-level graph partitioning algorithms such as Metis are well known to scale well,and have been used in studies of some of the biggest graph datasets. Graclus is another multi-level partitioning algorithm that optimizes weighted cuts (including normalized cuts) by optimizing an equivalent weighted kernel K-means loss function. The avoidance of expensive
eigenvector computation gives Graclus a big boost in speed while retaining or improving upon the quality of spectral approaches. Divisive/agglomerative approaches have been popular in network analysis, but they are expensive and do not scale well.

Preliminaries: Notations and Definitions

Let *G* = (*V, E*) be our input graph with *V* and *E* denoting the node set and edge set respectively. Let *A* be the $|V| \times |V|$ adjacency matrix corresponding to the graph, with A[i][j] denoting the weight of the edge between the vertex *vi* and the vertex *vj* . The weight of edge can represent association between two nodes in the graph. In a transport network it can represent distance between two cities. If the graph is unweighted the weights of all the edges in the graph is taken as one. In our discussion we have considered the graphs to be **undirected**, so the matrix A[i][j] will be a symmetric matrix.

Stochastic flow or flow: The flow between two vertices vi and vj is defined as the transition probability form vi to vj.

Markov Chain: A **Markov chain** is a mathematical system that undergoes transitions from one state to another, between a finite or countable number of possible states. It is a random- process characterized as memory less - the next state depends only on the current state and not on the sequence of events that preceded it.

Column Stochastic Matrix: A column stochastic matrix is a matrix in which the summation of each column is 1. A column stochastic square matrix **M** of order NxN where N is the number of nodes in the graph can be interpreted as matrix of transition probabilities of a **Markov Chain** defined on graph. The ith column of M represents the transition probabilities out of the *vi* therefore M[j][i] represents the probability of a transition from vertex vi to vj.

Deriving a column Stochastic Matrix:

To derive a column stochastic Matrix **M**, each column in the adjacency matrix **A,** of the graph is normalized to sum to 1.

$$M(i,j) = \frac{A(i,j)}{\sum_{k=1}^{n} A(k,j)}$$

In matrix notation, $\mathbf{M = AD^{-1}}$, where D is the diagonal degree matrix of G with $D(i,i) = \sum_{j=1}^{n} A(j,i)$

Markov Clustering Algorithm

The Markov Clustering Algorithm (MCL), iteratively applies a series of operators – Expansion , Inflation and Pruning to an initial flow matrix or stochastic matrix of graph G. The steps are applied sequentially and successively until convergence. Each of the step is described below:

Expansion: Given an Input flow matrix **M**, the expansion space spreads the flow out of a vertex to potentially new vertices and also enhances the flow to the vertices that can be reached by multiple paths. Thus it enhances within cluster flow as the vertices in a cluster are reachable by multiple paths. Mathematically speaking Expansion is,

$$M_{exp} = Expand(M) \stackrel{def}{=} M * M$$

The *ith* column of *Mexp* can be interpreted as the final distribution of a random walk of length 2 starting from vertex *vi*, with the transition probabilities of the random walk given by *M*.

Inflation: Applying the expansion step repeatedly will result in all the columns of *M* becoming equal to the principal eigenvector of the canonical transition matrix M_G. The inflation step is meant to prevent this from happening by introducing a non-linearity into the process, while also having the effect of strengthening intra-cluster flow and weakening inter-cluster flow. Mathematically speaking Inflation is,

$$M_{inf}(i,j) \stackrel{def}{=} \frac{M(i,j)^r}{\sum_{k=1}^{n} M(k,j)^r}$$

Minf corresponds to raising each entry in the matrix *M* to the power *r* and then normalizing the columns to sum to 1.By default *r* = 2. Because the entries in the matrix are all guaranteed to be less than or equal to 1, this operator has the effect of exaggerating the inhomogeneity in each column.

Pruning: Pruning is done to remove the entries that are very small. Very small entries are defined with respect to a pre-defined threshold value and the retained entries are rescaled to have column sum 1.

The threshold value selected in our implementation is: *One fourth of the average of all the entries in a column.* The entries less than the threshold value are set to 0. And the column is normalized to sum to 1.

Convergence: After some number of iterations, most of the nodes will find one "attractor" node to which all of their flow is directed i.e. there will be only one non-zero entry per column in the flow matrix M. Convergence is declared at this stage and no more iterations are done.

Interpretation as Clusters: To interpret clusters, the vertices are split into two types*. Attractors,* which attract other vertices, and vertices that are *being* attracted by the attractors. Attractors have at least one positive flow value within their corresponding row (in the steady state matrix). Each attractor is attracting the vertices which have positive values within its row. Attractors and the elements they attract are swept together into the same cluster.

Analysis

Complexity: $O(N^3)$, where N is the number of vertices.

N^3 is the cost of one matrix multiplication on two matrices of dimension N.

Inflation can be done in $O(N^2)$ time

The number of steps to converge is not proven, but experimentally shown to be ~10 to 100 steps, and mostly consist of sparse matrices after the first few steps.

Limitations of MCL:

The Markov Clustering Algorithm suffers through the following two major limitations:

Lack of Scalability: MCL given its cubic time complexity due to the Matrix multiplication in inflation step is quite slow. In the first few iterations when the

matrix is not that sparse the expansion step comes out to be very costly. So scalability has always been an issue for MCL family of Algorithms as for larger graphs the complexity is unacceptable.

Fragmentation of Output: MCL tends to produce too many clusters. Also the number of clusters are not predefined as it varies on the Expansion step and inflation parameter, *r.*
High fragmentation is undesirable as it deviates the graph from its original topology.

Algorithm 1 MCL

$A := A + I$ // Add self-loops to the graph
$M := AD^{-1}$ // Initialize M as the canonical transition matrix

repeat
    $M := M_{exp} :=$ Expand(M)
    $M := M_{inf} :=$ Inflate(M, r)
    $M :=$ Prune(M)
until M converges

Interpret M as a clustering

Regularized MCL

The Regularized MCL aims at doing away with the fragmentation problem in MCL. Let us see why MCL produces so many clusters. This is because MCL allows the columns of many pairs of neighboring nodes in the flow matrix to diverge significantly. It is so because MCL uses the adjacency structure of the input graph only at the start, to initialize the flow to the canonical transition matrix M_g . After that the algorithm only works with the current flow matrix and there is nothing in the algorithm that prevent columns from diverging or differing widely without any penalty. Thus the diverging columns manifest themselves as separate clusters when they converge.

Regularized MCL implements smoothening or regularization of the flow distributions out of a node w.r.t its neighbors. It is the expansion step that updates the flow matrix , so to smoothen the flow we modify the Expansion step of MCL. The new expansion operation expressed in terms of Matrix notations can be written as:

$$Regularize(M) = M_{reg} \stackrel{def}{=} M * M_G$$

This does resolves the the fragmentation problem with the initial canonical transition matrix M_g
Has a binding effect that prevents the columns from diverging too much to form separate clusters on convergence.

Algorithm 2 Regularized MCL

$A := A + I$ // Add self loops and transform weights
$M := M_G := AD^{-1}$ //Initialize M as the canonical transition matrix

repeat
  $M := M_{reg} := M * M_G$
  $M := M_{inf} := Inflation(M, r)$
  $M := Prune(M)$
until M converges

The Regularized MCL still suffers from the lack of scalability problem as the input size dependent time consuming steps have not been modified in R-MCL. The next section is the main focus of our project – The MLR-MCL, a multilevel approach to MCL that is scalable as well as free from outputting too many clusters.

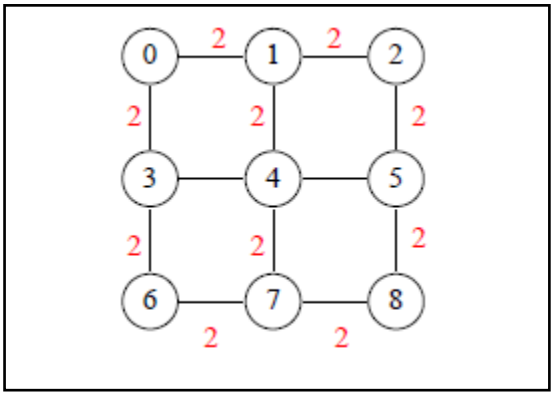
Multilevel Regularized MCL

The main intuition for using a multilevel approach to MCL is that the flow the flow values resulting from simulations on coarser graphs can effectively be used to initialize the flow for simulations on bigger graphs. The algorithm also runs faster because the initial few costly iterations are run on the coarsest or smallest graphs first.

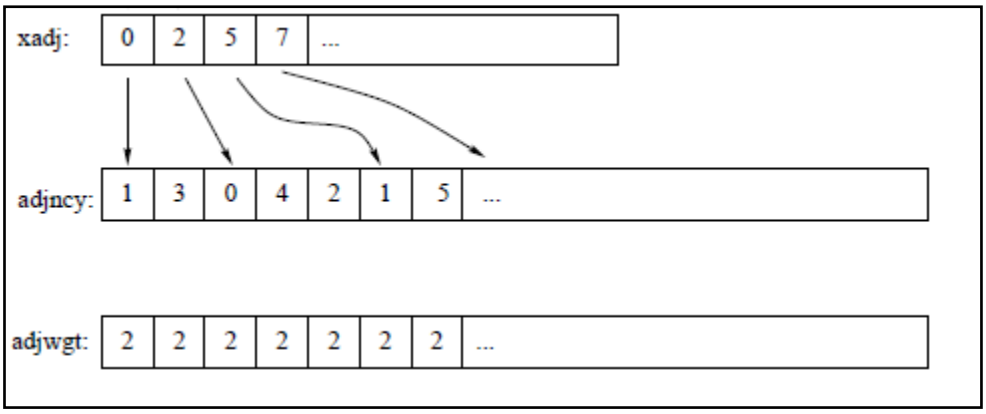
Graph Representation for Implementation of MLR-MCL

In MLR- MCL graphs are representing arrays. Given a graph of v vertices, vertices are implicitly labeled 0, 1, 2, . . . , v − 1. Information on edges is held in two arrays: xadj and adjncy. For all vertices i, the list of vertices adjacent to i are listed in the array adjncy from elements xadj[i] inclusive to xadj [i+1]) exclusive.

Thus, for xadj[i] <= j < xadj[i + 1], the value of adjncy[j] is connected to the vertex i. The array adjwgt holds the edge weights, thus adjwgt[j] holds the edge weight of adjncy[j]. The following example illustrates a graph and its array representation as per the scheme adopted for implementing MLR-MCL.



For the sample graph shown above the array representation of the graph a per our scheme is :



Description of MLR-MCL

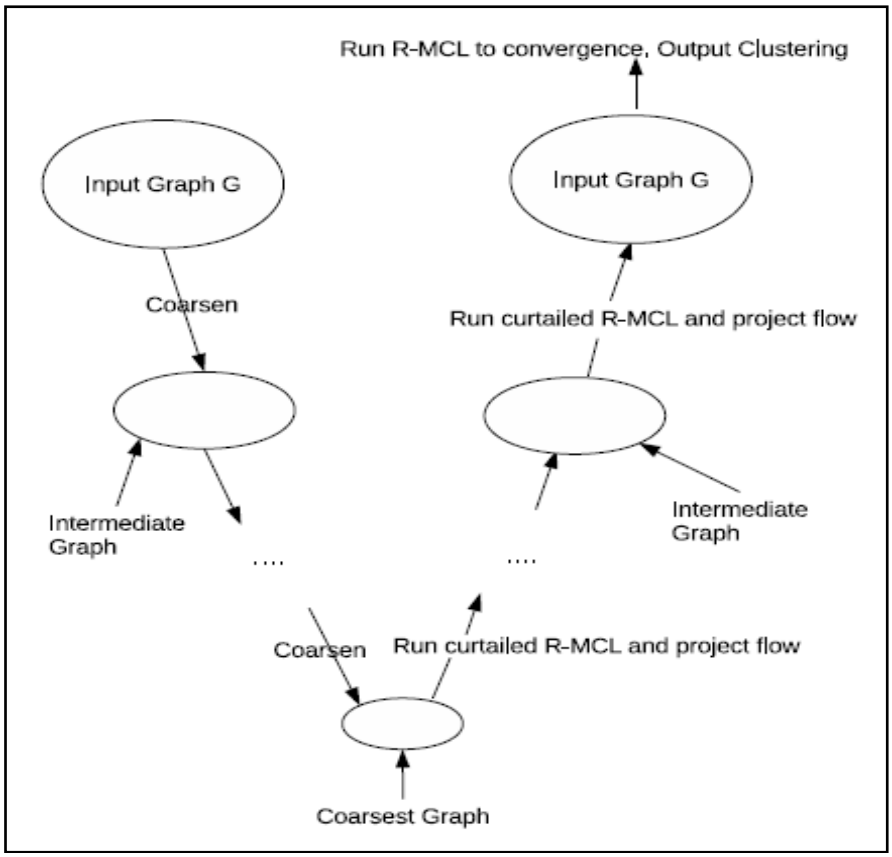
MLR-MCL operates in three phases. The three phases are namely:-

1) **Coarsening:** The input graph *G* is successively coarsened into a series of smaller graphs G_1, G_2, G_3.. and so on until we have a graph G_l of manageable size.

2) **Curtailed RMCL with refinement:** Beginning with the coarsest graph, R-MCL is run for a small number of iterations referred to as curtailed R-MCL.

3) **RMCL on original graph:** With flow values initialized from the previous phase, R-MCL is run on the final graph until convergence.

The three phases can be pictorially represented by the following diagram:-



Now the three phases will be discussed in details with their implementations.

Phase I:Coarsening:

Coarsening is successively reducing a graph G into smaller graphs by collapsing some pair of nodes. The input graph *G* is successively coarsened into a series of smaller graphs G_1,G_2,G_3.. and so on until we have a graph G_l of manageable size. The coarsening stage is very crucial as faster we approach the coarser graph more efficient the algorithm will be.

The process of coarsening can be subdivided into three stages:

1) Matching
2) Mapping
3) Creating a coarse graph

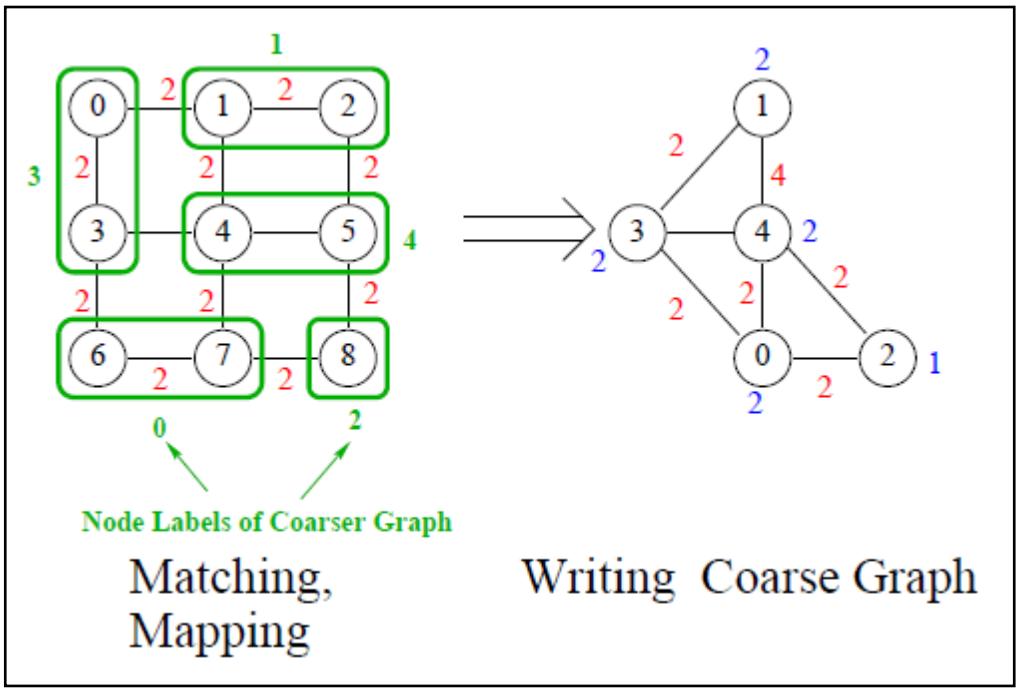
Matching:

In matching we greedily find as many pairs of connected vertices as possible, calling each pair a match. Selecting the pair (i, j), means i is matched with j and

vice versa. Pairs of vertices are matched until no vertex with an unmatched adjacent vertex remains. If a vertex i has no unmatched adjacent vertices, i is matched with itself.

For matching different algorithms can be adopted namely, HEM (Heavy edge matching), Random Matching. In our implementation we have used *Maximal Matching.*

A maximal matching is a matching *M* of a graph *G* with the property that if any edge not in *M* is added to *M*, it is no longer a matching, that is, *M* is maximal if it is not a proper subset of any other matching in graph *G*. In other words, a matching *M* of a graph *G* is maximal if every edge in *G* has a non-empty intersection with at least one edge in *M*.



The matching information is stored in an array named *match.* The array match is constructed in such a way that for all I such that 0< i <|V|, **match[match [i]] = i.** A consistency matching satisfies this requirement while an inconsistency matching doesnot. For the graph shown in the above example the match array representing the matching can be constructed as:

match:	3	2	1	0	5	4	7	6	8

It can be seen from the figure that the matched pair of vertices occupy the corresponding indices in the array match.

(0,3) is a matched pair, so *match[0]=3* and *match[3]=0*. Similarly , it can be verified for other matched pairs. A vertex left unmatched is considered to be matched with itself as it can be seen that *match [8] = 8.*

Mapping:

The second stage is mapping. Each matched pairs of vertices of G0 will represent one vertex of G1. Thus, we need implicit vertex labels for the vertices of G1. If G1 is to have v1 vertices, matched vertices i and j must map to a unique k such that 0 <=k < v1. Thus, mapping is actually mapping of vertices from a given graph to its successive coarse graph. In figure , the number outside the matched pairs are the mappings.

The mapping information is stored in another array called *cmap.* The array cmap is constructed in such a way that cmap [i] = cmap[match[i]]. This is so because the vertices in i and match[i] will be collapsed to a single vertex while forming the coarse graph. For the figure , the cmap of the matching will be:

cmap:	3	1	1	3	4	4	0	0	2

It can easily be verified that, the vertices (0 and 3) are forming a matched pair so cmap [0] = cmap [3] = 3. It means that the vertices 0 and 3 from the graph will be collapsed into a single vertex in the coarse graph and it will be labelled

Creating a coarse graph:

This is the third and perhaps the most crucial stage of coarsening. The mapping determines the number of vertices of G1.
An upper bound on the number of edges of G1 is the number of edges in G0. The edges connecting the matched pair of vertices are deleted and the the matched edges are collapsed to form a single edge as per the information stored in the arrays match[] and cmap[]. The weight information about the edges have to be carefully stored in the coarse graph as we can have overlapping edges when collapsing two or more pairs of matched edges. More specifically if there are edges connecting the two edges in a matched pair with the two vertices in another matched pair, then there will be overlapping of edges so the final edge weight in the coarse graph will be summation of edge

weights of the overlapping edge. After this stage we have two important data structures storing the information about the coarse graph:-

1) **Adjecency matrix of the coarse graph:** Adjecency matrix of the coarse graph with edge and weight information.

2) **NodeMap1 and Nodemap2**: NodeMap1[i] stores the first vertex of the matched pair in the graph that was collapsed to give vertex i in the coarse graph. NodeMap2[i] stores the second vetex of the matched pair in the graph that was collapsed to give vertex i in the coarse graph.

Phase II: Curtailed R-MCL

Beginning with the coarsest graph, R-MCL is run for a few iterations in our implementation we have run it for two times. This abbreviated version of R-MCL is referred to as Curtailed R-MCL. The flow values at the end of this Curtailed R-MCL run are then projected on to the refined graph of the current graph , and R-MCL is run again for a few iterations on the refined graph, and this is repeated until we reach the original graph.
The reason for running curtailed R-MCL instead of full fledged version of R-MCL is that on the coarse graphs the full RMCL or even a large number of iterations of R-MCL will lead to early convergence and thus at the successive uncoarsening levels we will get the same convergent flows .

Phase III: R-MCL on initial graph

With flow values initialized from the previous phase, R-MCL is run on the final graph until convergence. The flow matrix at the end is converted into a clustering in the usual way, with all the nodes that flow into the same "attractor" node being assigned into one cluster.

C++ Code for MLR-MCL (as developed by us)

Listing 1: Coarsening.cpp

Input: A graph file in which graph is represented n its adjacency matrix form.

Output: A file containing the adjacency matrix of the coarse graph and NodeMap1 and NodeMap2.

```
#include<stdio.h>
#include<cstdlib>
#include<iostream>
#include<cstring>
#include<list>
#define MAX 1000

using namespace std;float adjecency[MAX+1][MAX+1];

float coarse[MAX+1][MAX+1];

int cmap[MAX+1];

int matched[MAX+1];

int edj[MAX+1];

int xadj[MAX+1];

float adjwt[MAX+1];

int numedges[MAX+1];

int nodemap1[MAX+1];

int nodemap2[MAX+1];

void update1(int num,int n,int m)

{ int i,j,k;

     //Here's the weight handling

for(i=0;i<num;i++){

int count=0,a1,a2,b1,b2;

for(j=i+1;j<=num;j++)

{   for(int k=0;k<n;k++)
```

```
     {
if(cmap[k]==i){a1=k;a2=matched[k];nodemap1[i]=k;nodemap2[i]=ma
tched[k];}
if(cmap[k]==j){b1=k;b2=matched[k];}
          }
if(adjecency[a1][b1])
{ coarse[i][j]+=adjecency[a1][b1];
adjecency[a1][b1]=0;
}
if(adjecency[a1][b2])
{coarse[i][j]+=adjecency[a1][b2];
adjecency[a1][b2]=0;
}
if(adjecency[a2][b1])
{ coarse[i][j]+=adjecency[a2][b1];
adjecency[a2][b1]=0;
}
if(adjecency[a2][b2])
{ coarse[i][j]+=adjecency[a2][b2];
adjecency[a2][b2]=0; }
coarse[j][i]=coarse[i][j]; }}
//Weight handling ends here
int cc=0;
for(i=0;i<num;i++)
for(j=0;j<num;j++)
     if(coarse[i][j]) cc++;
```

```
cout<<num<<" "<<(cc>>1)<<endl;


for(i=0;i<num;i++)
      {
      for(j=0;j<num;j++)
      cout<<coarse[i][j]<<" ";
      cout<<endl;
      }
cout<<"Nodemap1:"<<endl;
      for(i=0;i<num;i++)cout<<nodemap1[i]<<" ";
            cout<<endl;
      cout<<"Nodemap2:"<<endl;
      for(i=0;i<num;i++)cout<<nodemap2[i]<<" ";
}
void update(int num,int n,int m)
{ int i,j;
for(i=0;i<n;i++){
for(j=xadj[i];j<xadj[i+1];j++)
{
    if(edj[j]==matched[i])
    { edj[j]=-1;
adjwt[j]=0;    }}
}
}
```

```
void match(int n,int m)

{

int num=0;

for(int i=0;i<n;i++)

{  if(matched[i] == -1)

    {

 for(int j=xadj[i];j<xadj[i+1];j++)

      {

       if(matched[edj[j]]==-1)

       {  matched[edj[j]]=i;

       matched[i]=edj[j];

       cmap[i]=cmap[edj[j]]=num;

       num++;

       numedges[num]=(xadj[i+1]-xadj[i])+(xadj[edj[j]+1]-
xadj[edj[j]]);

       break;

       }}

}}

for(int i=0;i<n;i++)

{  if(matched[i]==-1)

    {  matched[i]=i;

        cmap[i]=num++;

        numedges[num]=(xadj[i+1]-xadj[i]); } }

update(num,n,m);

update1(num,n,m);

}
```

```
int main(int argc,char** argv)
{ char str[100000]; freopen(argv[1],"r",stdin);
    int m,n,i,j,a,k,l;
    char *ptr;
     scanf("%d %d",&n,&m);
     i=0,k=0,l=0;
     int count=0;
for(i=0;i<n;i++)
     { xadj[k++]=count;
     for(j=0;j<n;j++)
     { cin>>adjecency[i][j];
     if(adjecency[i][j])
     { edj[count]=j;
          adjwt[count]=adjecency[i][j];
          count++; } } }
     xadj[k++]=count;
for(i=0;i<n;i++)  matched[i]=-1;
match(n,m);
    cout<<endl;
     return 0;
}
```

Listing 2: mlrmcl.cpp

Input: The adjacency matrix of the coarsest graph along with the Nodemap information in a text file.

Output: No of cluster form along with the vertex listing in each cluster.

```
#include <map>
#include <queue>
#include <stack>
#include <ctime>
#include <cmath>
#include <vector>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <climits>
#include <iostream>
#include <algorithm>


using namespace std;

long int temp,i,j,k,T;

#define CASE while(T--)
#define FOR(I,A,B) for(I=A;I<B;I++)
#define REP(i,n) FOR(i,0,n)
#define FORR(I,J,K) for(I=J;I>K;I--)
#define JAM(N) Case #N:
#define INPUT(A) freopen(A,"r",stdin);
#define OUTPUT(A) freopen(A,"w",stdout);
#define DEBUG cout<<"DEBUG"<<endl;
#define EXP 1e-10
#define INF 1e 10

#define F first
#define S second

typedef long long LL;
typedef pair<int,int> PII;
typedef pair<LL,LL> PLL;
typedef pair<int,PII> TRI;

#define s(n)                        scanf("%d",&n)
#define sl(n)                       scanf("%ld",&n)
#define sll(n)                           scanf("%lld",&n)
#define sf(n)                       scanf("%f",&n)
#define slf(n)                           scanf("%lf",&n)
```

```
#define ss(n)                           scanf("%s",n)

int next(){
    char c;int num=0;
    c=getchar_unlocked();
    while(!(c>='0' && c<='9')) c=getchar_unlocked();
    while(c>='0' && c<='9'){
        num=(num<<3)+(num<<1)+c-'0';
        c=getchar_unlocked();
    }
    return num;
}

//main code is here
#define MAX 100

char* itoa(int val, int base=10){

     static char buf[32] = {0};
     if(!val){buf[0]='0';return &buf[0];}
     int i = 30;

     for(; val && i ; --i, val /= base)

          buf[i] = "0123456789abcdef"[val % base];

     return &buf[i+1];

}

float M[MAX][MAX]={0},Mg[MAX][MAX]={0},conv[MAX][MAX]={0};
int v,e;

void Normalise(int v)
{
     int i,j,k;
     float csum;
     FOR(j,1,v)
     {
          csum=0;
          FOR(k,1,v)
          csum = csum + M[k][j];
          FOR(i,1,v)
          M[i][j] = M[i][j]/csum;
     }
}

void Regularize(int v)
{
     int i,j;
     float temp[MAX][MAX];
```

```
      FOR(i,1,v) FOR(j,1,v) {temp[i][j]=M[i][j];M[i][j]=0;}

      FOR(i,1,v)
      FOR(j,1,v)
      FOR(k,1,v)
      M[i][j] = M[i][j]+temp[i][k]*Mg[k][j];
}

void Inflate(int v,int r)
{
      int i,j;
      float temp[MAX][MAX],div;
      FOR(i,1,v) FOR(j,1,v) temp[i][j]=M[i][j];

      FOR(j,1,v)
      {
            div=0.0;
            FOR(k,1,v)
            div = div + pow(temp[k][j],r);
            FOR(i,1,v)
            M[i][j]=pow(temp[i][j],r)/div;
      }
}

void Prune(int v)
{
      int i,j;
      float temp[MAX][MAX],avg,mx;
      FOR(i,1,v) FOR(j,1,v) temp[i][j]=M[i][j];

      FOR(j,1,v)
      {
            avg=0;
            mx=-1.0;
            FOR(k,1,v)
            {
            avg = avg + temp[k][j];
            mx = max(mx,temp[k][j]);
            }
            avg = avg/(v-1);
            float threshold=avg/4;
            FOR(i,1,v)
            if(M[i][j] < threshold)
            M[i][j]=0;
      }
      Normalise(v);
}

void ProjectFlow(char *fname,int gi)
{     //cout<<"PF->"<<fname<<endl;
      FILE *fp=fopen(fname,"r");
```

```
	int i,j;

	fscanf(fp,"%d%d",&v,&e);
	e++;v++;
	float Gi[MAX][MAX];
	FOR(i,1,v) FOR(j,1,v) fscanf(fp,"%f",&Gi[i][j]);
	int Nodemap1[MAX]={0},Nodemap2[MAX]={0};
	char x[100];
	fscanf(fp,"%s",x);
	//cout<<x<<endl;
	FOR(i,1,v)
	{
		fscanf(fp,"%d",&Nodemap1[i]);
		//cout<<Nodemap1[i]<<" ";
	}
	fscanf(fp,"%s",x);
	//cout<<endl<<x<<endl;
	FOR(i,1,v)
	{
		fscanf(fp,"%d",&Nodemap2[i]);
		//cout<<Nodemap2[i]<<" ";
	}
	//cout<<endl;
	fclose(fp);

	float temp[MAX][MAX],avg,mx;
	FOR(i,1,v) FOR(j,1,v) temp[i][j]=M[i][j];

	float zero=((float)0.0)/4;
	FOR(i,1,v)
	FOR(j,1,v)
	{
		if(temp[i][j] != zero)
		{
		M[Nodemap1[i]+1][Nodemap1[j]+1]=temp[i][j];
		M[Nodemap1[i]+1][Nodemap2[j]+1]=temp[i][j];
		M[Nodemap2[i]+1][Nodemap1[j]+1]=0;
		M[Nodemap2[i]+1][Nodemap2[j]+1]=0;
		}
	}
	/*FOR(i,1,v)
	{FOR(j,1,v)
	cout<<Gi[i][j]<<" ";cout<<endl;}*/
	//cout<<v<<endl;
}
void mg(char *fname,int gi)
{	//cout<<fname<<endl;
	int i,j,k;
	FILE *fp=fopen(fname,"r");
	//int e,v;
	fscanf(fp,"%d%d",&v,&e);
```

```
	e++;v++;
	float A[MAX][MAX]={0};
	FOR(i,1,v)	FOR(j,1,v)	fscanf(fp,"%f",&A[i][j]);
				//simple adjacency matrix

	FOR(i,1,v) A[i][i]=1.0;
		//self loops

	float D[MAX][MAX]={0};
	FOR(i,1,v)
	FOR(j,1,v)
	D[i][i]+=A[i][j];
	float invD[MAX][MAX]={0};
	FOR(i,1,v) invD[i][i]=1.0/D[i][i];
	//Got D inverse

	FOR(i,1,v)
	FOR(j,1,v)
	FOR(k,1,v)
	M[i][j]=M[i][j] + A[i][k]*invD[k][j];
	//Got M

	FOR(i,1,v)
	FOR(j,1,v)
	Mg[i][j]=M[i][j];
	fclose(fp);
}

void CRmcl(char *fname,int r,int c)
{
	int snit=2;
	//cin>>snit;
	char a[100]=".hid/",temp[100];
	strcat(a,fname);
	strcpy(temp,a);
	strcat(temp,itoa(c));

	FILE *fp=fopen(temp,"r");
	//int e,v;
	fscanf(fp,"%d%d",&v,&e);
	e++;v++;
	float A[MAX][MAX]={0};
	FOR(i,1,v)	FOR(j,1,v)	fscanf(fp,"%f",&A[i][j]);
				//simple adjacency matrix

	FOR(i,1,v) A[i][i]=1.0;
		//self loops

	float D[MAX][MAX]={0};
	FOR(i,1,v)
	FOR(j,1,v)
```

```
	D[i][i]+=A[i][j];
	float invD[MAX][MAX]={0};
	FOR(i,1,v) invD[i][i]=1.0/D[i][i];
	//Got D inverse

	FOR(i,1,v)
	FOR(j,1,v)
	FOR(k,1,v)
	M[i][j]=M[i][j] + A[i][k]*invD[k][j];
	//Got M

	FOR(i,1,v)
	FOR(j,1,v)
	Mg[i][j]=M[i][j];
	fclose(fp);
	//cout<<temp<<endl;
	for(i=c;i>0;i--)
	{
		Normalise(v);
		REP(j,snit)
		{
			Regularize(v);
			Inflate(v,r);
			Prune(v);
		}
		//cout<<temp<<endl;
		ProjectFlow(temp,i);
		strcpy(temp,a);
		strcat(temp,itoa(i-1));
		mg(temp,i-1);
		//cout<<v<<" "<<temp<<" "<<i<<endl;;
	}
}

void Coarse(char *fname,int r,int c)
{
	char a[100]="./co ",temp[100];
	strcat(a,fname);

	strcpy(temp,a);
	strcat(temp," > .hid/");
	strcat(temp,fname);
	strcat(temp,"1");
	system(temp);

	strcpy(a,"./co .hid/");
	strcat(a,fname);
	for(i=1;i<c;i++)
	{

		char t1[10],t2[10],t[100];
```

```
            strcpy(t,a);
            if(i<10){t1[0]=i+'0';t1[1]=0;}
            else {t1[0]=(i/10+'0');t1[1]=(i%10+'0');t1[2]=0;}

            int j=i+1;
            if(j<10){t2[0]=j+'0';t2[1]=0;}
            else {t2[0]=(j/10+'0');t2[1]=(j%10+'0');t2[2]=0;}

            strcat(t,t1);
            strcat(t," > .hid/");
            strcat(t,fname);
            strcat(t,t2);
            //cout<<t<<endl;
            system(t);
      }
}

inline bool isconverge(int v)
{
      int flag=0;
      FOR(i,1,v)
      FOR(j,1,v)
      if(M[i][j] != conv[i][j])
      break;

      if(i==v-1 && j==v-1)
      return true;

      FOR(i,1,v)
      FOR(j,1,v)
      conv[i][j] = M[i][j];
      return false;
}

main(int argc,char** argv)
{
      if(argc < 4)
      {cout<<"USAGE : mlrmcl r clevel filename"<<endl;return
0;}

      system("rm -rf .hid");
      system("mkdir .hid -p");
      char tmp[100]={"cp "};
      strcat(tmp,argv[3]);
      strcat(tmp," .hid/");
      strcat(tmp,argv[3]);
      strcat(tmp,"0");
      system(tmp);

      int r,c;
      INPUT(argv[3]);
```

```
	r=atoi(argv[1]);
	c=atoi(argv[2]);

	Coarse(argv[3],r,c);
	CRmcl(argv[3],r,c);

	//RMCL ON Original Graph

	FILE *fp=fopen(argv[3],"r");
	int v;
	scanf("%d",&v);v++;
	fclose(fp);

	FOR(i,1,v)
	FOR(j,1,v)
	conv[i][j] = M[i][j];

	while(isconverge(v))
	{
	Regularize(v);
	Inflate(v,r);
	Prune(v);
	}

	int nc = 0,interpret[MAX]={0};
	FOR(i,1,v)
	{FOR(j,1,v)
	{
		//cout<<M[i][j]<<" ";
		if(M[i][j]>0 && interpret[i]==0)
		{
		nc++;
		cout<<i<<" ";
		interpret[i]=interpret[j]=1;
		FOR(k,j+1,v)
		if(M[i][k]>0  && !interpret[k])
		{
			interpret[k]=1;
			cout<<k<<" ";
		}cout<<endl;
		}
	}
	//cout<<endl;
	}
	cout<<endl<<"Number Of Clusters in the given Graph using
MLRMCL : "<<nc<<endl<<endl;
	return 0;
}
```

Experimental Results:

Result set 1: It is the cluster information obtained by running MCL, R-MCL, and MLR-MCL on the three graphs under consideration.

Graph1:
http://www.weizmann.ac.il/mcb/UriAlon/Papers/networkMotifs/prisonInter_st.txt

Graph2:
http://www.weizmann.ac.il/mcb/UriAlon/Papers/networkMotifs/leader2Inter_st.txt

Graph3:
A sample test graph(as mentioned in the example in the report)

Table 1: Results showing number of clusters formed on running the three algorithms as implemented by us.

No. of clusters	Graph1	Graph2	Graph3
MCL (r=2)	23	10	5
R-MCL (r=2)	23	10	5
MLR-MCL (r=2)	17 (c=10)	9 (c=5)	3 (c=2)

Table 2: Results showing number of clusters formed on running the three algorithms as available in standard package.

No. of clusters	Graph1	Graph2	Graph3
MCL (r=2)			
R-MCL (r=2)			
MLR-MCL (r=2)			

Future Scope and improvements of the work

The following can be taken up as further extension of the work.

1) The graph representation used as well as the way in which the matching and mapping information has been maintained are in well accordance with the Parmetis package – A package for K way partitioning of a graph. This can be taken as a reference if some parallel architecture is provided to parallelize the MLR-MCL.

2) The matrix multiplication, which constitute a costly part of the Expansion step, has been implemented using usual $O(N^3)$ algorithm. This becomes more costly when the graph gets sparser. So new schemes for matrix multiplication in case of sparse graph can be seen as a optimization step.

3) The algorithms have been implemented for weighted graphs. It can be taken up to enhance the algorithms in order to fit it well for directed graphs also.

4) The refinement step can be further enhanced to minimize the edge cuts by selecting that cluster for a given vertex which results in minimizing the edge cut.

References:

1) Scalable Graph Clustering Using Stochastic Flows: Applications to Community Discovery
By *Venu Satuluri and Srinivasan Parthasarathy*, Department of Computer Science, The Ohio State University.

2) Partial Parallelization of Graph Partitioning algorithm Metis
By *Zardosht Kasheff*

3) http://www.weizmann.ac.il/mcb/UriAlon/groupNetworksData.html
For collecting the graph datasets for validating the results of our implementations.