Matt Howa

Modularity Research

cluster_walktrap - This function tries to find densely connected subgraphs, also called communities in a graph via random walks. The idea is that short random walks tend to stay in the same community. This function is the implementation of the Walktrap community finding algorithm, see Pascal Pons, Matthieu Latapy: Computing communities in large networks using random walks, ***References:*** http://arxiv.org/abs/physics/0512106

cluster_edge_betweenness - Many networks consist of modules which are densely connected themselves but sparsely connected to other modules. The edge betweenness score of an edge measures the number of shortest paths through it, see edge_betweenness for details. The idea of the edge betweenness based community structure detection is that it is likely that edges connecting separate modules have high edge betweenness as all the shortest paths from one module to another must traverse through them. So if we gradually remove the edge with the highest edge betweenness score we will get a hierarchical map, a rooted tree, called a dendrogram of the graph. The leafs of the tree are the individual vertices and the root of the tree represents the whole graph. cluster_edge_betweenness performs this algorithm by calculating the edge betweenness of the graph, removing the edge with the highest edge betweenness score, then recalculating edge betweenness of the edges and again removing the one with the highest score, etc. edge.betweeness.community returns various information collected throught the run of the algorithm. See the return value down here. ***References*** - M Newman and M Girvan: Finding and evaluating community structure in networks, Physical Review E 69, 026113 (2004)

cluster_fast_greedy - This function tries to find dense subgraph, also called communities in graphs via directly optimizing a modularity score. This function implements the fast greedy modularity optimization algorithm for finding community structure, ***Reference***: A Clauset, MEJ Newman, C Moore: Finding community structure in very large networks, http://www.arxiv.org/abs/cond-mat/0408187 for the details.

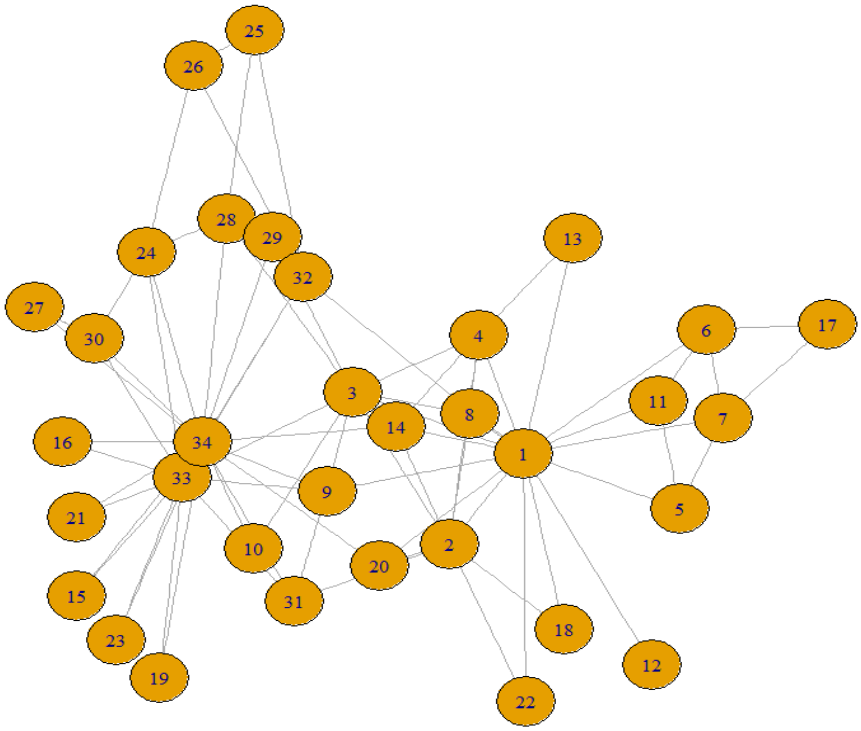
cluster_spinglass - This function tries to find communities in graphs via a spin-glass model and simulated annealing. This function tries to find communities in a graph. A community is a set of nodes with many edges inside the community and few edges between outside it (i.e. between the community itself and the rest of the graph.) This idea is reversed for edges having a negative weight, ie. few negative edges inside a community and many negative edges between communities. Note that only the 'neg' implementation supports negative edge weights. The spinglass.cummunity function can solve two problems related to community detection. If the vertex argument is not given (or it is NULL), then the regular community detection problem is solved (approximately), i.e. partitioning the vertices into communities, by optimizing an energy function. If the vertex argument is given and it is not NULL, then it must be a vertex id, and the same energy function is used to find the community of the the given vertex. See also the examples below. ***References*** - J. Reichardt and S. Bornholdt: Statistical Mechanics of Community Detection, Phys. Rev. E, 74, 016110 (2006), http://arxiv.org/abs/cond-mat/0603718 ; M. E. J. Newman and M. Girvan: Finding and evaluating community structure in networks, Phys. Rev. E 69, 026113 (2004) ; V.A. Traag and Jeroen Bruggeman: Community detection in networks with positive and negative links, http://arxiv.org/abs/0811.2329 (2008).

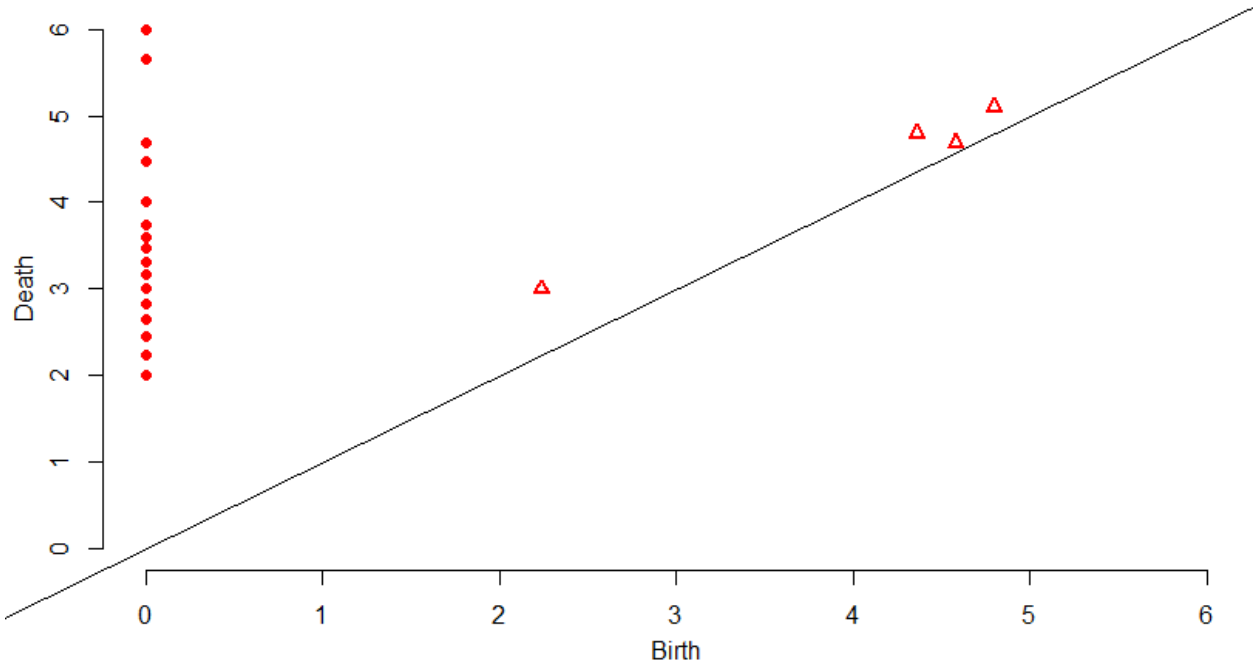
cluster_leading_eigen - This function tries to find densely connected subgraphs in a graph by calculating the leading non-negative eigenvector of the modularity matrix of the graph. The function documented in these section implements the 'leading eigenvector' method developed

by Mark Newman, see the reference below. The heart of the method is the definition of the modularity matrix, B, which is B=A-P, A being the adjacency matrix of the (undirected) network, and P contains the probability that certain edges are present according to the ‘configuration model’. In other words, a P[i,j] element of P is the probability that there is an edge between vertices i and j in a random network in which the degrees of all vertices are the same as in the input graph. The leading eigenvector method works by calculating the eigenvector of the modularity matrix for the largest positive eigenvalue and then separating vertices into two community based on the sign of the corresponding element in the eigenvector. If all elements in the eigenvector are of the same sign that means that the network has no underlying comuunity structure. Check Newman's paper to understand why this is a good method for detecting community structure. ***References***: MEJ Newman: Finding community structure using the eigenvectors of matrices, Physical Review E 74 036104, 2006.

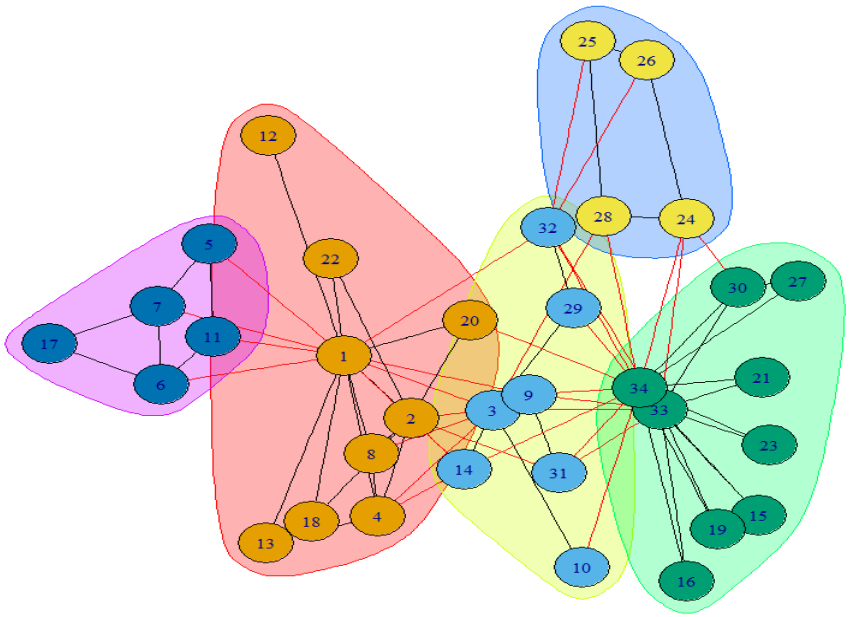
```
## Load package
library(igraph)
library(TDA)

#Zachs karate club
modularityTest1 <- function(){

  xlist<-read.graph("C:\\Users\\jdbba\\OneDrive\\Documents\\School\\Research\\DataSets\\Zach's karate
club\\karate\\karate.gml", format="gml")
  plot(xlist)

  ##Persistant Homology of 1st graph
  Diag<-ripsDiag(distances(xlist),1,6, dist = "euclidean", library = "Dionysus",location = TRUE, printProgress =
FALSE)
  plot(Diag[["diagram"]], col='red')

  ##Cluster WalkTrap
  #Modularity
  wtc<-cluster_leading_eigen(xlist)
  plot(wtc, xlist )


  ##Persistant Homology of modular graph
  mDiag <- ripsDiag(merges(wtc),1,25, dist = "euclidean", library = "Dionysus",location = TRUE, printProgress =
FALSE)
  plot(mDiag[["diagram"]], col='red')

  bottleneck(Diag[["diagram"]],mDiag[["diagram"]])
}


modularityTest1()
```

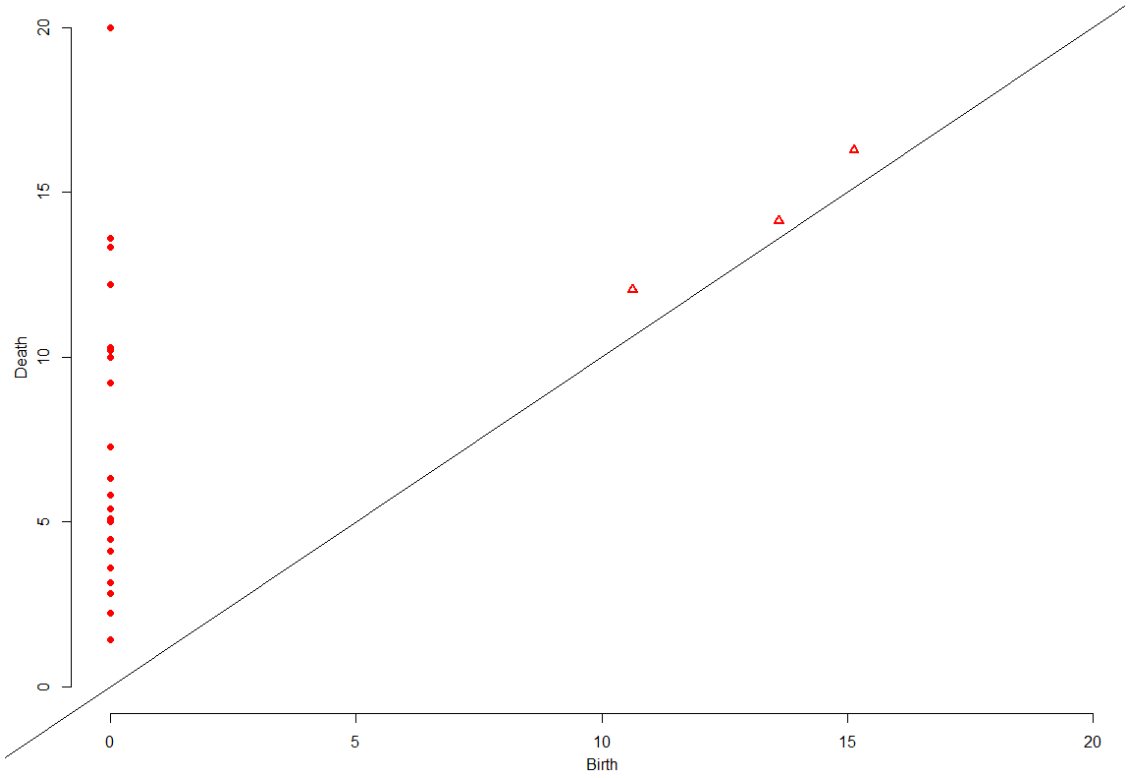
Original Graph





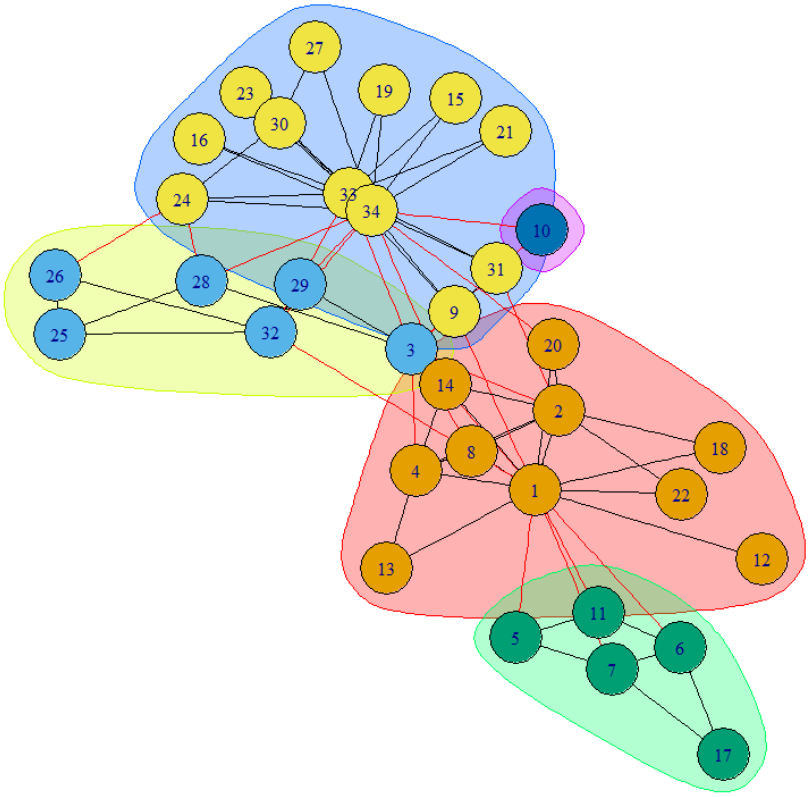
Cluster Walk Trap

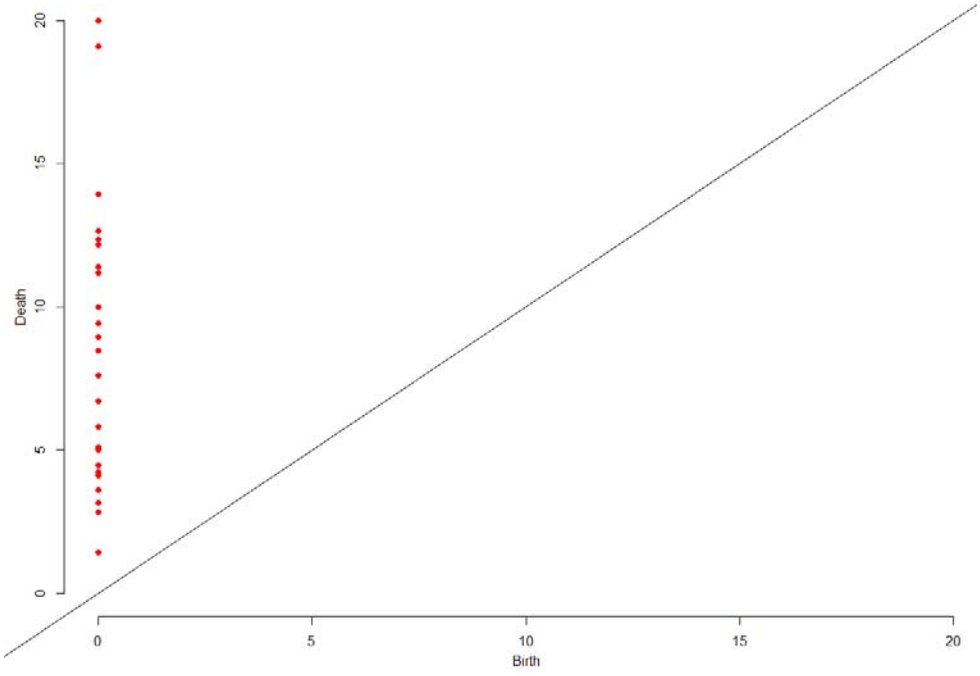




Bottleneck: 0.7057244

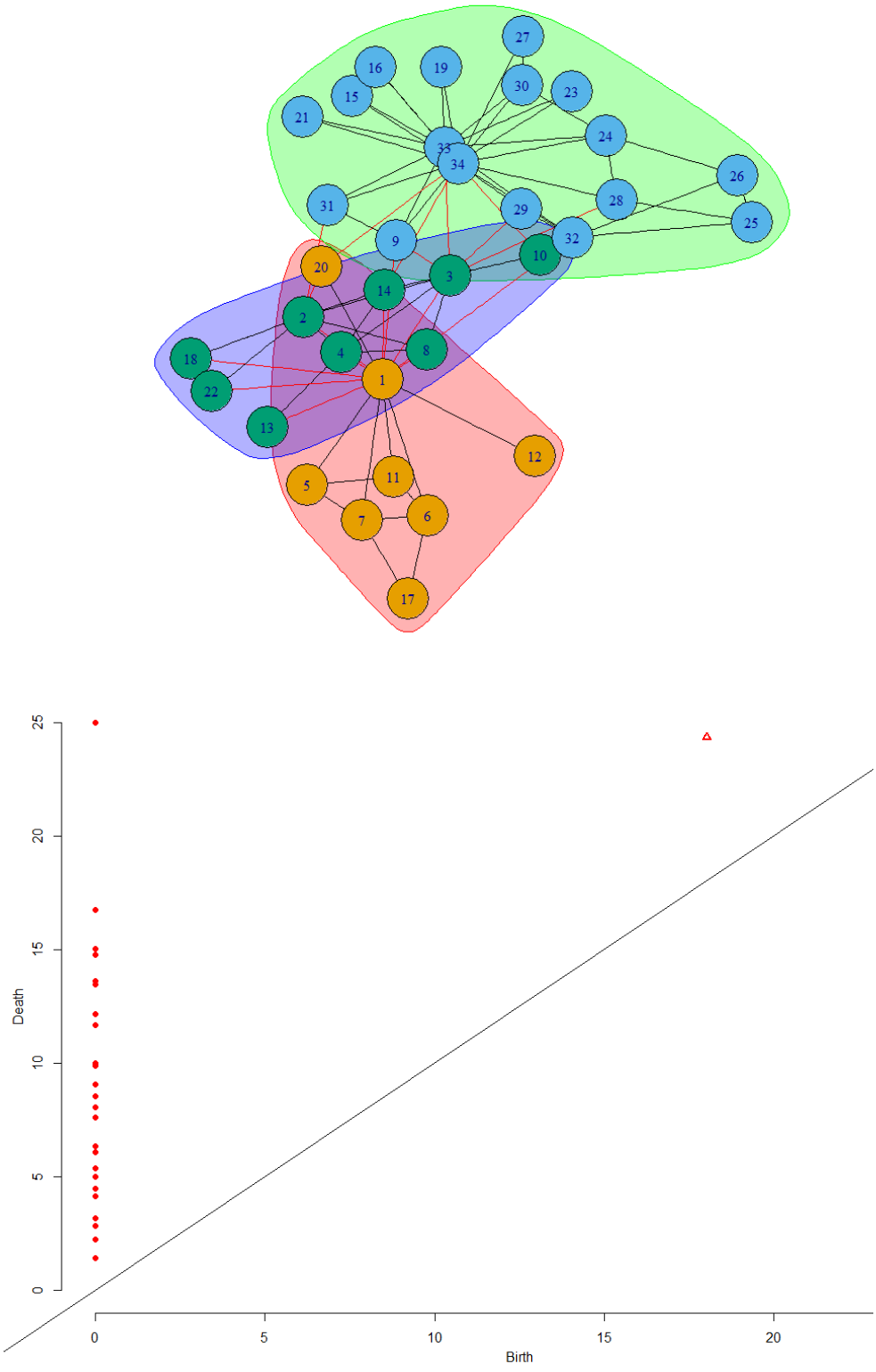
Cluster_edge_betweeness





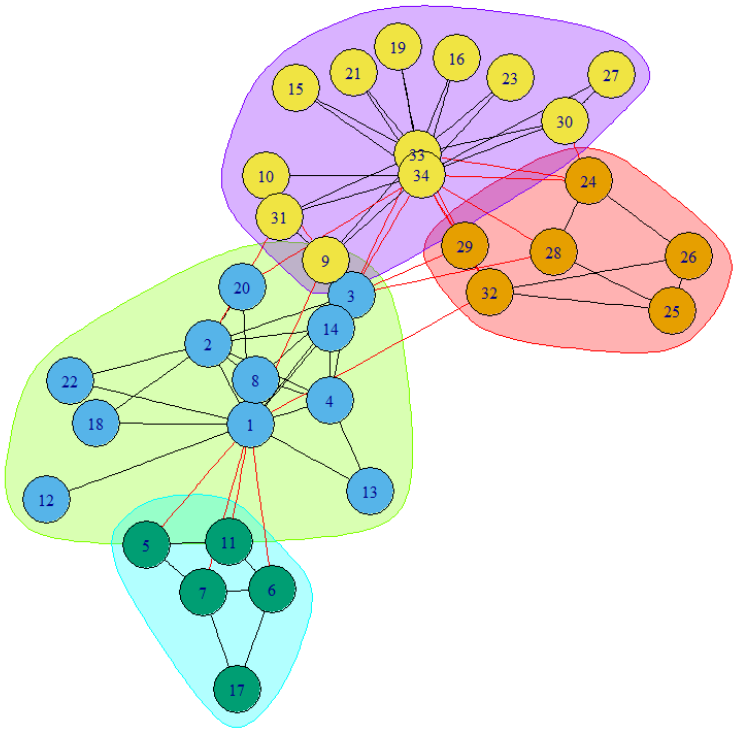
Bottleneck distance: 0.5

Cluster_fast_greedy

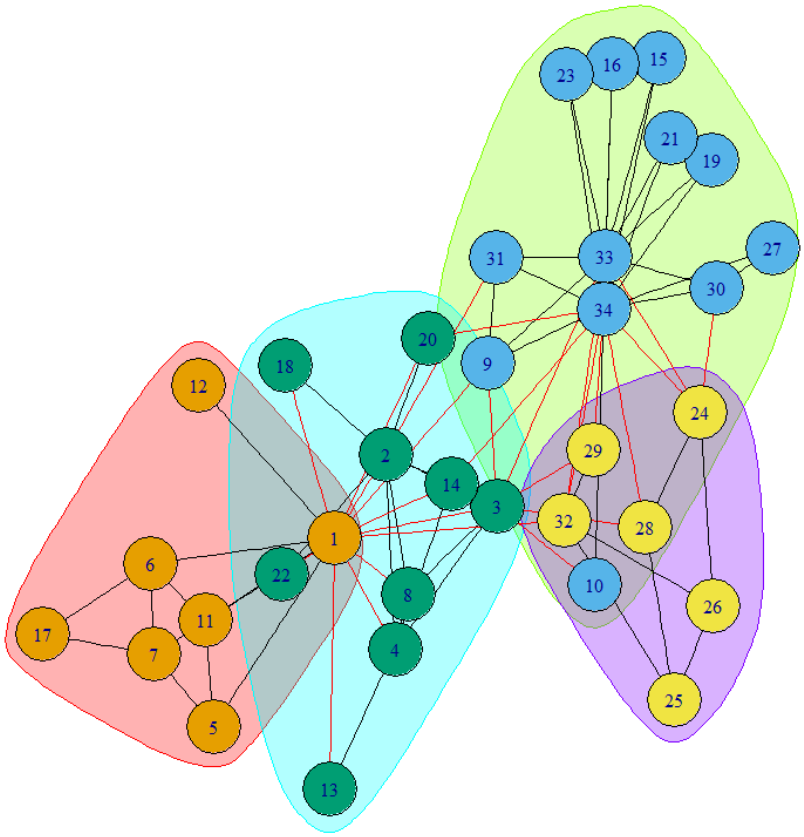


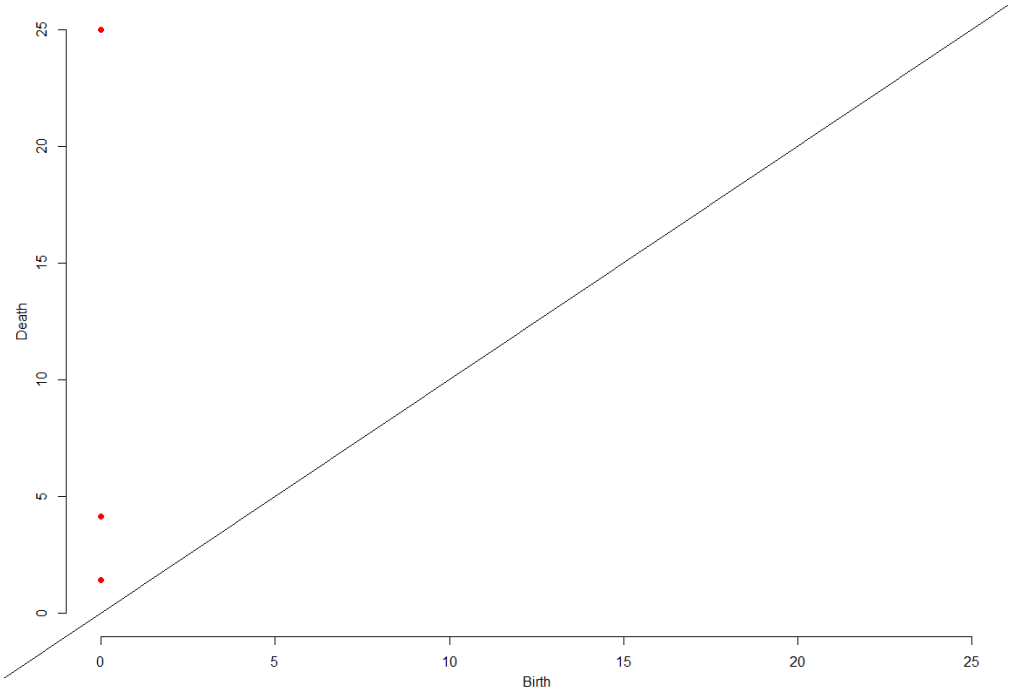
Bottleneck distance: 3.161917

Cluster_spinglass



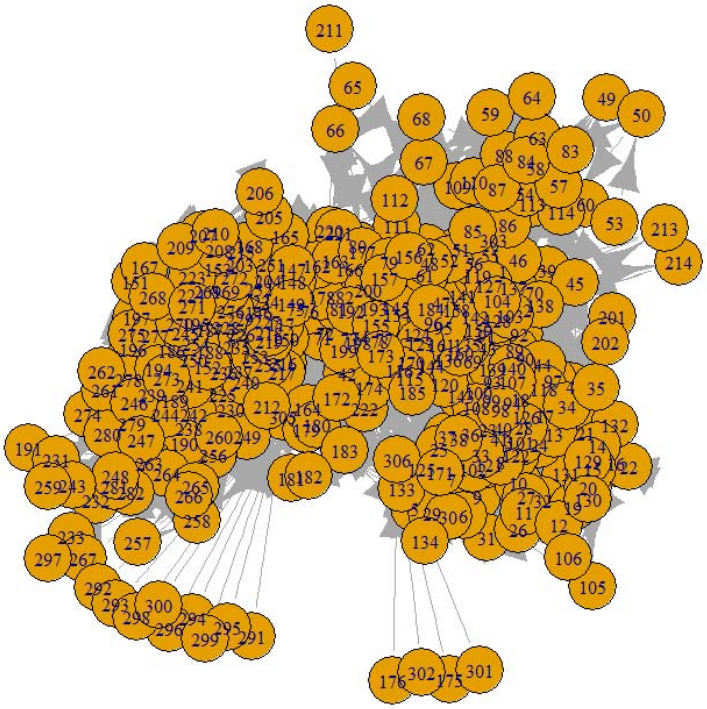
Cannot figure out how to return a matrix to run Persistant Homology

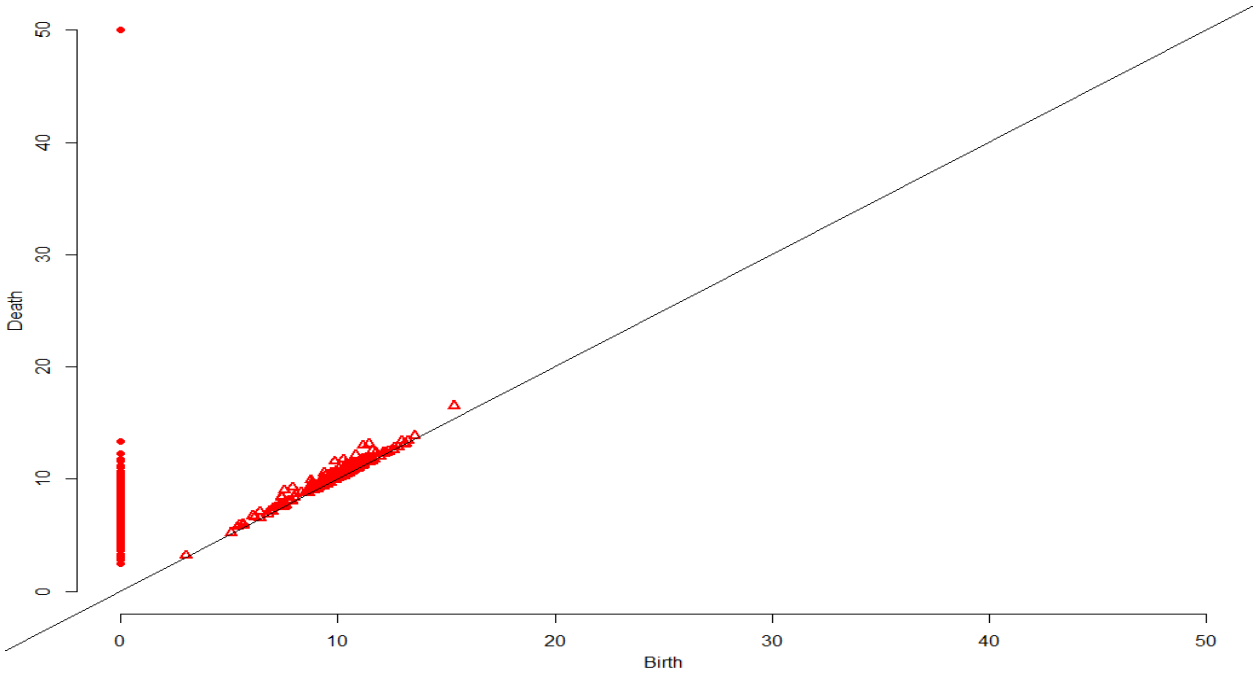


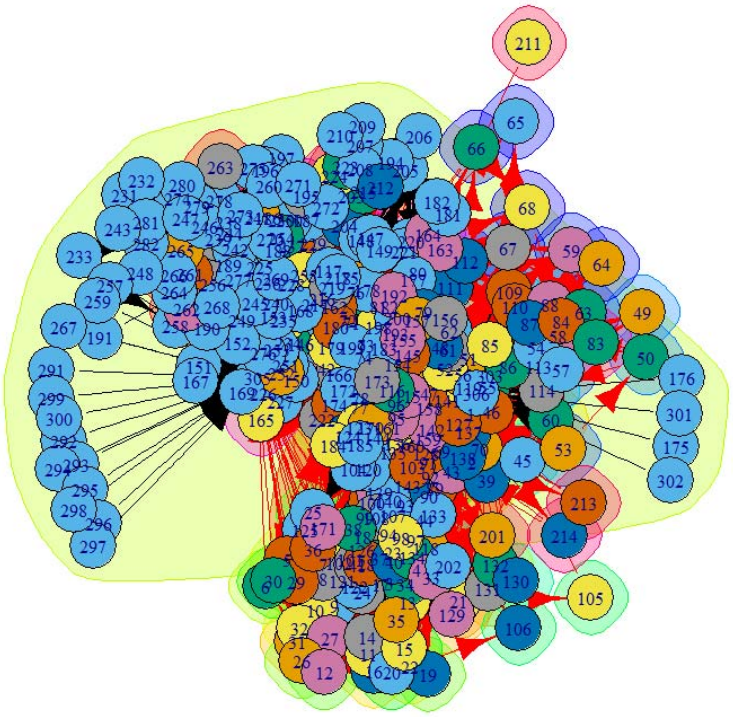


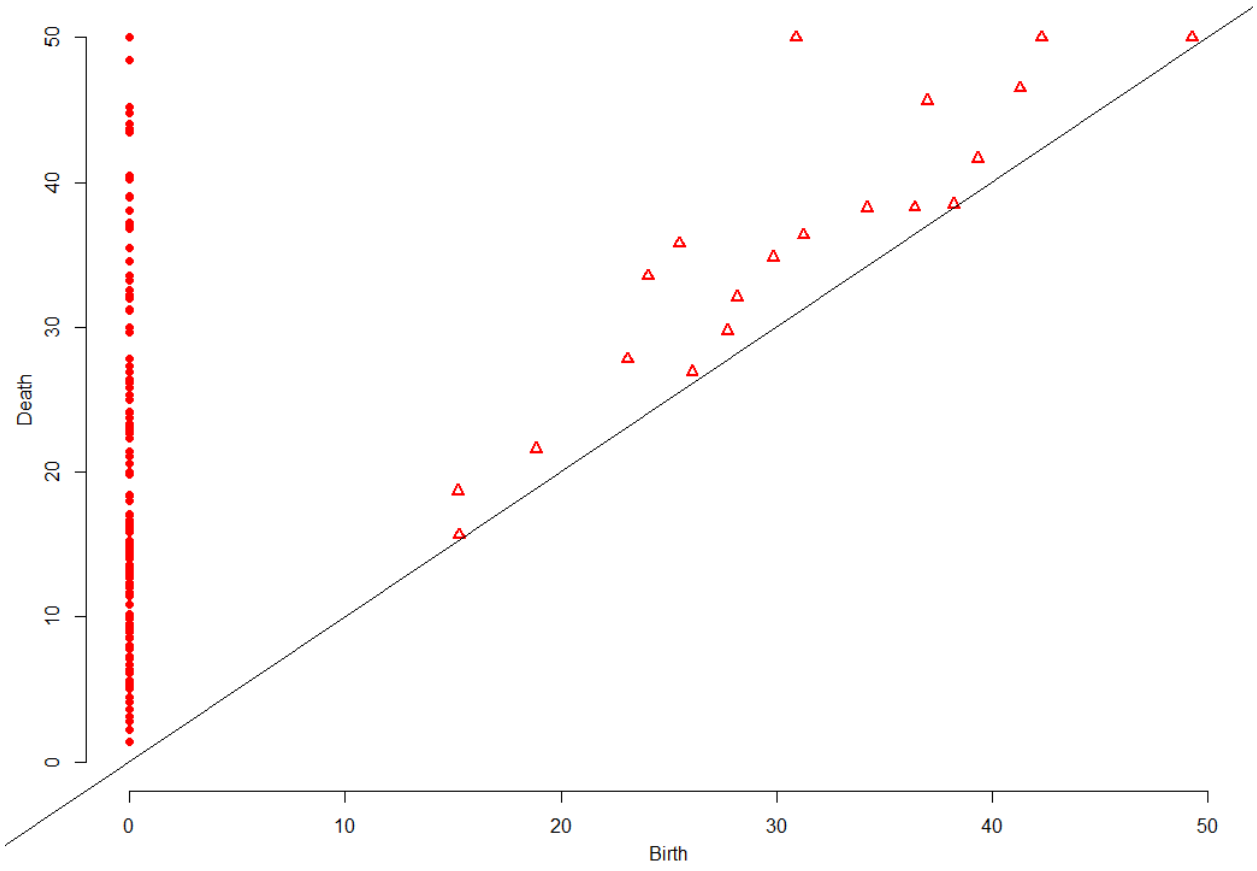
Bottleneck: 0.5

Neural Network








Death
Birth
0
10
20
30
40
50

Bottleneck: 9.556555

At this points ripsDiag is taking way too long. its time to look into scalable solutions

Started research for DIPHA

Got help from SOURABH PALANDE who gave me some python scripts for converting objects to the correct format for DIPHA and back again.

```
import numpy as np
from struct import *
Function to write distance matrix to binary dipha input file
Inputs :
  dist  : input pairwise distance matrix
          (it must be a square symmetric matrix
           with 0 on diagonal. The function doesn't perform this
           input verification)
  fname : a string with
           [full path to location]/filename.bin
          to write file (extension .bin is required by dipha)

def writeDistBin(dist, fname):
                magicN = 8067171840;
                ftype = 7;
                m,n = np.shape(dist);
                print('input matrix size : ' +str(m) + ', ' + str(n) + '\n');

                ## flatten the matrix
                dflat = [ii for sub in dist for ii in sub];

                ## first three values are integers
                ## magicN : indicates that this is a dipha file
                ## ftype  : file type (7 => distance matrix)
                ## m      : number of rows / columns (dist must be m x m matrix)
                packBin = pack('q'*3+'d'*(m*m), *([magicN, ftype, m] + dflat));
                with open(outf, 'wb') as f:
                        f.write(packBin);

                return(True);

Function to read binary dipha persistent diagram file and
write in to a plain text file

Inputs :
  infile  : string containing
             [full-path-to-location]/fname.bin
            of the dipha persistent diagram binary file
            (this is the output file generated by dipha)

  outfile : string containing
            [full path to location]/fname.txt
           to write the plain text persistent diagram file
'''

def writeDiagTxt((infile, outfile)):
                with open(inf, 'rb') as f:
                        binData = f.read();

                flag = False;
                if(len(binData) >= 24):
                        magicN, fType, nPoints = unpack('3q', binData[:24]);

                        if (magicN != 8067171840):
```

```
                print('Magic number does not match. Not a dipha file');

        elif(ftype != 2):
                print('File type does not match');

        else:
                diag = unpack('qdd'*nPoints, binData[24:]);
                diag = np.asarray(diag).reshape(nPoints,3);

                with open(outf, 'w') as fp:
                        for row in diag:
                                fp.write(" ".join([str(i) for i in row]) + '\n');

                flag = True;

else:
        print('file is too small. Re-check the input');

return(flag);
```