Homework 1

Professor - Vwani Roychowdhury

Class - EE232E Graphs and Network Flows

April 16, 2015

**1. Create random networks**

**(a) Create three undirected random networks with 1000 nodes, and the probability p for drawing an edge between two arbitrary vertices 0.01, 0.05 and 0.1 respectively. Plot the degree distributions.**

|  |
| --- |
| **[Code]** |
| p <- 0.01 # Change p value to 0.05, 0.1  g <- erdos.renyi.game(n, p, "gnp", directed = FALSE, loops = FALSE)  plot(degree.distribution(g)) |

|  |  |
| --- | --- |
| **p** | **Degree Distribution Plot** |
| 0.01 |  |
| 0.05 |  |
| 0.1 |  |

**(b) Are these networks connected or disconnected? What are the diameters of these networks?**

|  |
| --- |
| **[Code]** |
| connectedindicator = is.connected(g) #check connected or not  diam\_unconnected\_false= diameter(g, unconnected=FALSE)  diam\_unconnected\_true = diameter(g, unconnected=TRUE) |

|  |  |  |  |
| --- | --- | --- | --- |
| **p** | **Connection** | **Diameter**  **(unconnected = FALSE criteria)** | **Diameter**  **(unconnected = TRUE criteria)** |
| 0.01 | Connected | 5 | 5 |
| 0.05 | Connected | 3 | 3 |
| 0.1 | Connected | 3 | 3 |

**(c) Try to numerically find a value pc (to three significant figures), so that when p < pc the generated random networks are disconnected, and when p > pc the generated random networks are connected.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **1st simulation** | | **2nd simulation** | | **3rd simulation** | |
| **p** | **connected** | **p** | **connected** | **p** | **connected** |
| 0.005 | FALSE | 0.005 | FALSE | 0.005 | FALSE |
| 0.008 | TRUE | 0.008 | TRUE | 0.0075 | TRUE |
| 0.007 | TRUE | 0.007 | TRUE | 0.006 | FALSE |
| 0.006 | FALSE | 0.006 | FALSE | 0.007 | TRUE |
| 0.0065 | FALSE | 0.0065 | FALSE | 0.0065 | FALSE |
| 0.00675 | FALSE | 0.00675 | FALSE | 0.00658 | FALSE |
| 0.0069 | FALSE | 0.00677 | TRUE | 0.00662 | FALSE |
| 0.00695 | FALSE | 0.00676 | FALSE | 0.00668 | FALSE |
| 0.00698 | FALSE |  |  | 0.00673 | TRUE |
| 0.00699 | TRUE |  |  | 0.00671 | FALSE |
| 0.006985 | FALSE |  |  | 0.00672 | FALSE |
| T | **0.00699** | T | **0.00676** | T | **0.00672** |

Average T : 0.00682 (3 significant figures)

**(d) Can you analytically derive the value of pc?**

Referring to the paper on this link: <http://www.cs.cornell.edu/courses/cs4850/2010sp/Course%20Notes/Random-graphs-from-jeh-Feb-06-2010.pdf>

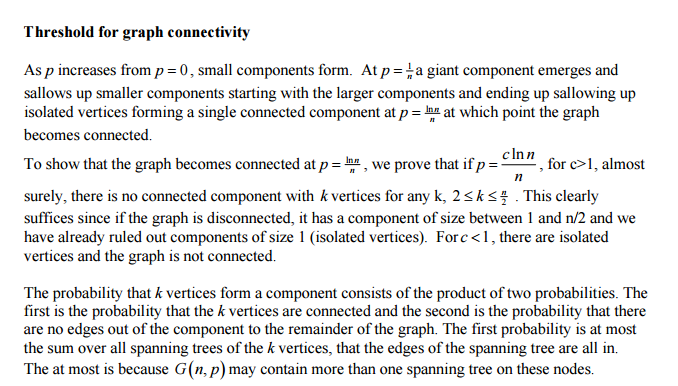


Figure 1 Paper excerpt

P = ln(n) / n = ln(1000) / 1000 = 0.006907 = 0.00691 (3 significant figures)

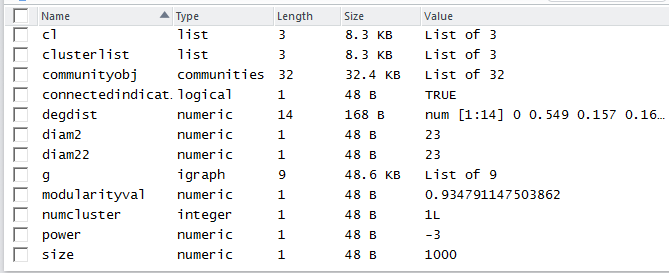
This result is close to the simulation above.

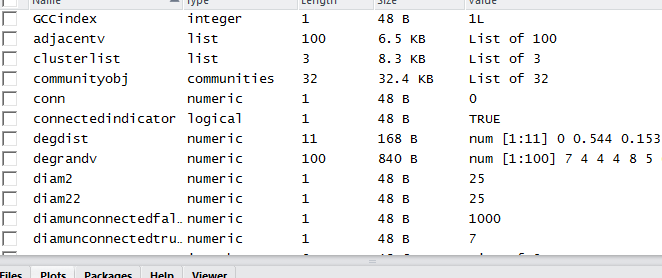
**2. Create a network with a fat-tailed degree distribution**

**(a) Create an undirected network with 1000 nodes, whose degree distribution is proportional to x-3. Plot the degree distribution. What is the diameter?**

|  |
| --- |
| **[Code]** |
| # Generate Random Network for Power law degree distribution  power = -3  numnodes = 1000  g<-barabasi.game(numnodes, power, directed = FALSE)  plot(degree.distribution(g))  # Calculate Diameter  diam\_unconnected\_true = diameter(g, unconnected=TRUE)  diam\_unconnected\_false = diameter(g, unconnected=FALSE) |

|  |  |
| --- | --- |
| **[Degree distribution n=1000]** | |
| Linear Scale |  |
| Log Scale |  |

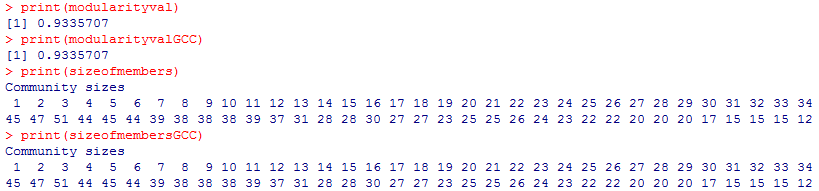




Diameter varies between 21 and 27, most frequently appears between 23 and 25. Hence, on average diameter is 24.

**(b) Is the network connected? Find the giant connected component (GCC) and use fast greedy method to find the community structure. Measure the modularity. Why is the modularity so large?**

|  |
| --- |
| **[Code]** |
| # Clustering Graph  clusterlist <- clusters(g)  size <- clusterlist$csize  numcluster <- clusterlist$no  # Find Community structure of Overall Graph  communityobj <- fastgreedy.community(g, merges=TRUE, modularity=TRUE, membership=TRUE, weights=NULL)  # Modularity & Sizes of Communities  modularityval <- modularity(communityobj)  sizeofmembers <- sizes(communityobj)  # Find the GCC  GCCindex <- which.max(size)  nonGCCnodes<-(1:vcount(g))[clusterlist$membership!=GCCindex] #compares the membership vector with the index , then multiply TRUE with node list  GCCfinal <- delete.vertices(g,nonGCCnodes)  # Find Community structure of GCC  communityobjGCC = fastgreedy.community(GCCfinal, merges=TRUE, modularity=TRUE, membership=TRUE, weights=NULL)  # Modularity & Sizes of Communities  modularityvalGCC = modularity(communityobjGCC)  sizeofmembersGCC= sizes(communityobjGCC) |



The network is connected. This is checked by using is.connected(graph) command in the code.

The question can be interpreted in two ways, after finding the GCC, do we apply the fast greedy method to the GCC or to all the original graph.

So, firstly, apply the fast greedy method to the original graph, before vertices are deleted to find the GCC. The results for modularity is 0.9335707

The modularity is large because nodes which have existed longer enjoys higher preferential for newer nodes to attach. Especially, since the power is -3, it means that older nodes have exponentially higher chance to attract newer nodes. Hence, as time progresses, more edges are incident on the oldest nodes, which means that there are more groups are more likely to form, because newer nodes tend to cluster around older nodes.

|  |
| --- |
| **[Graph of GCC]** |
|  |

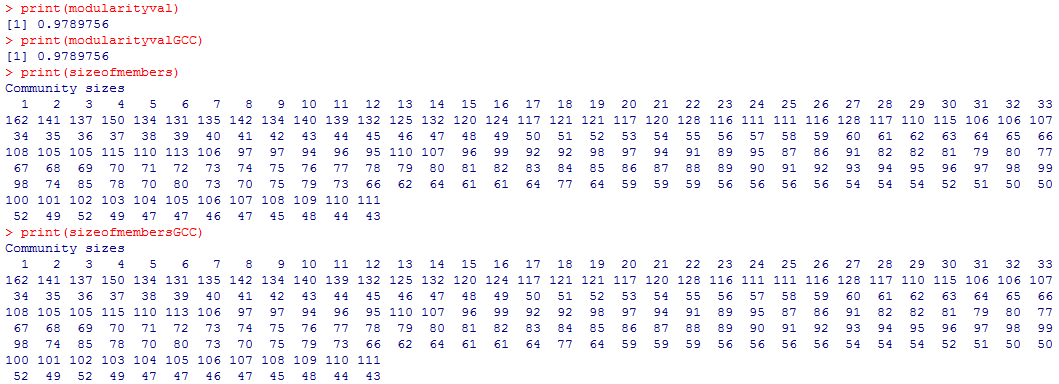
Measuring modularity value within the GCC:

Modularity value of GCC = 0.9335707

The explanation of why modularity is large is similar to before. The modularity is large because nodes which have existed longer enjoys higher preferential for newer nodes to attach. Especially, since the power is -3, it means that older nodes have exponentially higher chance to attract newer nodes. Hence, as time progresses, more edges are incident on the oldest nodes, which means that there are more groups are more likely to form, because newer nodes tend to cluster around older nodes.

**(c) Try to generate a larger network with 10000 nodes whose degree distribution is proportional to x-3. Compute the modularity. Is the same as the smaller network’s?**

|  |  |
| --- | --- |
| **[Degree distribution n=10000]** | |
| Linear Scale |  |
| Log Scale |  |



Modularity value of both original graph (including non GCC) and GCC only is 0.9789756, higher than value for 1000 nodes.

One possible reason why the modularity value is higher is because newer nodes have more options of older nodes to attach to. Hence, forming higher number of segments within the GCC.

**(d) You can randomly pick a node I, and then randomly pick a neighbor j of that node. Measure and plot the degree distribution of nodes j that are picked with this process.**

|  |
| --- |
| **[Code]** |
| iter = 100  # Pick up random vertices iter times  randv<-igraph.sample(1, numnodes, iter)  # For each randv,pick up a random adjacent vertex  # So first calculate the degree of the random vertex , based on vertex iD  degrandv<-degree(g, randv)  # Find adjacent vertex IDs  adjacentv = g[[randv,]] # or use adjcentv = get.adjlist(g) #adjacentv is a vector  # Find degree values of random neighbor nodes  nei\_deg <- c(0)  for (i in 1:length(adjacentv)) {  nei\_deg[i] <- degree(g, adjacentv[[i]][sample(1:length(adjacentv[[i]][]),1)])  }  # Make Plot the Degree Distribution  h <- hist(nei\_deg, breaks = 10, plot=FALSE)  h$counts = h$counts/sum(h$counts)  plot(h, ylab="degree.distribution") |

|  |
| --- |
| **[Degree Distribution, n = 1000]** |
|  |

|  |
| --- |
| **[Degree Distribution, n = 10000]** |
|  |

**3. Creates a random graph by simulating its evolution**

**(a) Each time a new vertex is added it creates a number of links to old vertices and the probability that an old vertex is cited depends on its in-degree (preferential attachment) and age. Produce such an undirected network with 1000 nodes. Plot the degree distribution.**

|  |
| --- |
| **[Code]** |
| g1 <- aging.prefatt.game(1000, pa.exp=1, aging.exp=0, aging.bin=1000,directed=F)  g2 <- aging.prefatt.game(1000, pa.exp=1, aging.exp=-1, aging.bin=1000,directed=F)  g3 <- aging.prefatt.game(1000, pa.exp=1, aging.exp=-3, aging.bin=1000,directed=F) |

|  |  |
| --- | --- |
| **‘aging.exp’ Value** | **Degree Distribution** |
| 0 |  |
| -1 |  |
| -3 |  |

**(b) Use fast greedy method to find the community structure. What is the modularity?**

|  |
| --- |
| **[Code]** |
| # Clustering Graph  clusterlist <- clusters(g1)  size <- clusterlist$csize  numcluster <- clusterlist$no  # Find Community structure of Overall Graph  communityobj <- fastgreedy.community(g1, merges=TRUE, modularity=TRUE, membership=TRUE, weights=NULL)  # Modularity & Sizes of Communities  modularityval <- modularity(communityobj)  sizeofmembers <- sizes(communityobj)  # Find the GCC  GCCindex <- which.max(size)  nonGCCnodes<-(1:vcount(g1))[clusterlist$membership!=GCCindex] #compares the membership vector with the index , then multiply TRUE with node list  GCCfinal <- delete.vertices(g1,nonGCCnodes)  # Find Community structure of GCC  communityobjGCC = fastgreedy.community(GCCfinal, merges=TRUE, modularity=TRUE, membership=TRUE, weights=NULL)  # Modularity & Sizes of Communities  modularityvalGCC = modularity(communityobjGCC)  sizeofmembersGCC= sizes(communityobjGCC)  print(modularityval)  print(modularityvalGCC)  print(sizeofmembers)  print(sizeofmembersGCC) |

|  |  |
| --- | --- |
| **‘aging.exp’ Value** |  |
| 0 |  |
| -1 |  |
| -3 |  |

|  |  |  |
| --- | --- | --- |
| **‘aging.exp’ Value** |  | **Modularity** |
| 0 | Overall | 0.9194936 |
| GCC | 0.9194936 |
| -1 | Overall | 0.9355602 |
| GCC | 0.9355602 |
| -3 | Overall | 0.9361098 |
| GCC | 0.9361098 |

**4. Use the forest fire model to create a directed network**

**(a) This is a growing network model, which resembles how the forest fire spreads by igniting trees close by. Plot the in and out degree distributions.**

|  |
| --- |
| **[Code]** |
| g\_fire <- forest.fire.game(10000, fw.prob=0.37, bw.factor=0.32/0.37, directed=TRUE)  plot(degree.distribution(g\_fire,mode="in"))  plot(degree.distribution(g\_fire,mode="out")) |

|  |  |
| --- | --- |
| **Degree Direction** | **Degree Distribution Graph** |
| in |  |
| out |  |

**(b) Measure the diameter.**

|  |
| --- |
| **[Code]** |
| # Calculate Diameter  diam\_unconnected\_true = diameter(g, unconnected=TRUE)  diam\_unconnected\_false = diameter(g, unconnected=FALSE)  print(diam\_unconnected\_true)  print(diam\_unconnected\_false) |

Diameter Unconnected True : 471

Diameter Unconnected False : 471

**(c) Measure the community structure and modularity.**

|  |
| --- |
| **[Code]** |
| # Problem 4 (c)  # Clustering Graph  clusterlist <- clusters(g\_fire)  size <- clusterlist$csize  numcluster <- clusterlist$no  # Find Community structure of Overall Graph  communityobj <- walktrap.community(g\_fire, merges=TRUE, modularity=TRUE, membership=TRUE, weights=NULL)  # Modularity & Sizes of Communities  modularityval <- modularity(communityobj)  sizeofmembers <- sizes(communityobj)  # Find the GCC  GCCindex <- which.max(size)  nonGCCnodes<-(1:vcount(g\_fire))[clusterlist$membership!=GCCindex] #compares the membership vector with the index , then multiply TRUE with node list  GCCfinal <- delete.vertices(g\_fire,nonGCCnodes)  # Find Community structure of GCC  communityobjGCC = walktrap.community(GCCfinal, merges=TRUE, modularity=TRUE, membership=TRUE, weights=NULL)  # Modularity & Sizes of Communities  modularityvalGCC = modularity(communityobjGCC)  sizeofmembersGCC= sizes(communityobjGCC)  print(modularityval)  print(modularityvalGCC)  print(sizeofmembers)  print(sizeofmembersGCC) |

Different from previous problem 2 and 3. We use ‘walktrap‘ method to find the community structure. The reason why we use ‘walktrap’ method instead of ‘fast greedy’ method is that the fast greedy method can be applied only for undirected graph. So it can not be used for the directed graph.

|  |
| --- |
| **[Result - Modularity]** |
|  |

The modularity value was 0.1944502.

|  |
| --- |
| **[Result – Community Structure]** |
|  |
|  |