**SNA PRACTICALS**

**Practical No.: 1**

**Aim:** Write a program to compute the following for a given network:

1. Number of edges
2. Number of Nodes
3. Degree of Nodes
4. Node with Lowest degree
5. the adjacency list
6. matrix of the graph

**Description:**

1. igraph: igraph is a library and R package for network analysis
2. ecount: To count the number of edges
3. vcount: To count number of vertices/nodes
4. graph.formula: To create a network
5. E(g): Displays a list of edge
6. V(g): Displays a list of vertices/nodes
7. degree(g): The degree of a vertex is its most basic structural property, the number of its adjacent edges
8. mode = “in”: To find the indegree of a directed graph
9. mode = “out”: To find the outdegree of a directes graph
10. $ : To include the output of the code and concatenate it along with the other code
11. adjlist: Create adjacency lists from a graph, either for adjacent edges or for neighboring vertices
12. adjacency: Creates an adjacency matrix

**Code :**

install.packages("igraph")

library(igraph)

g=graph.formula(A-B,B-C,C-D,D-E,E-F,G-H)

g

ecount(g)

vcount(g)

E(g)

V(g)

degree(g)

dg=graph.formula(A-+B,B+-C,C-+B,C++D)

dg

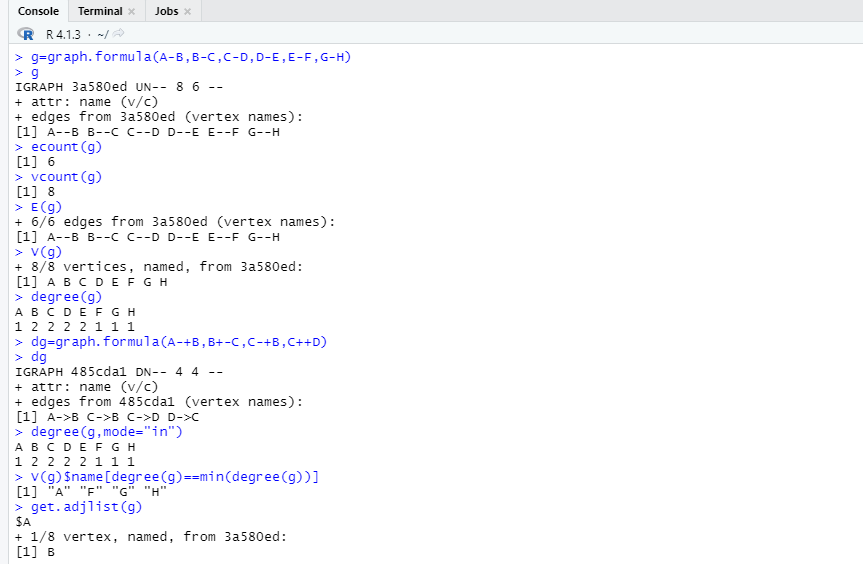
degree(g,mode="in")

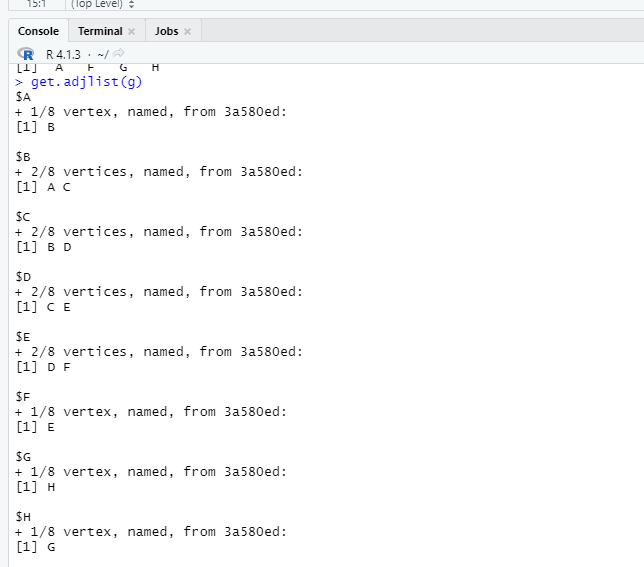
V(g)$name[degree(g)==min(degree(g))]

get.adjlist(g)

get.adjacency(dg)

**Output:**





**Practical No.: 2**

**Aim:** Perform following tasks: (i) View data collection forms and/or import onemode/two-mode datasets; (ii) Basic Networks matrices transformations

**Description:**

getwd()-Get or Set Working Directory. Returns an absolute filepath representing the current working directory of the R process; setwd(dir) is used to set the working directory to dir.

read.csv- Reads a file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the file.

head()-Return the First or Last Parts of an Object. Returns the first or last parts of a vector, matrix, table, data frame or function. Since head() and tail() are generic functions, they may also have been extended to other classes.

graph.data.frame()-Creating igraph graphs from data frames or vice-versa. This function creates an igraph graph from one or two data frames containing the (symbolic) edge list and edge/vertex attributes

as.matrix- attempts to turn its argument into a matrix.

get.adjacency()-Convert a graph to an adjacency matrix. Sometimes it is useful to work with a standard representation of a graph, like an adjacency matrix

**Code:**

library("igraph")

getwd()

setwd("E:/CSV files")

l=read.csv("edges.csv")

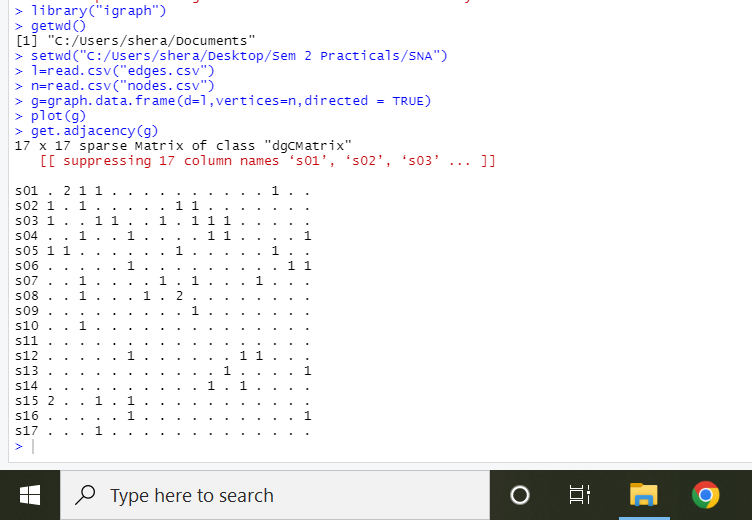
n=read.csv("nodes.csv")

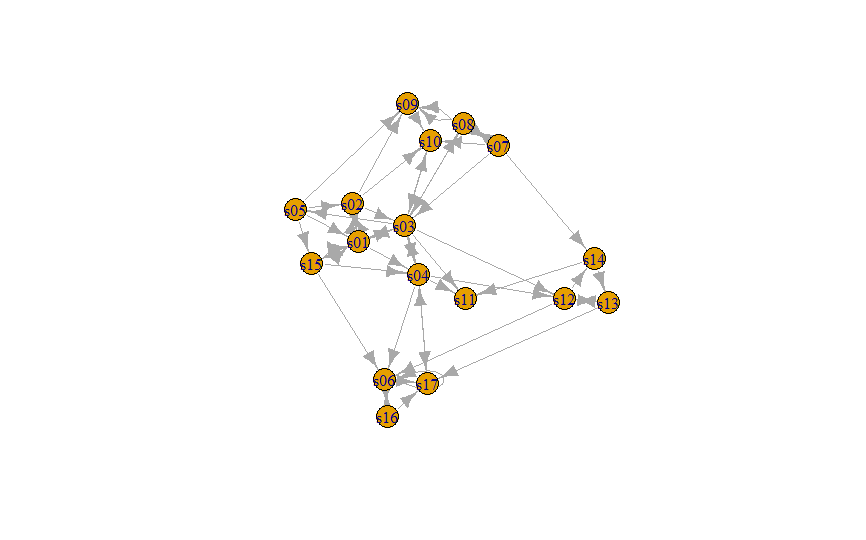
g=graph.data.frame(d=l,vertices=n,directed = TRUE)

plot(g)

get.adjacency(g)

**Output:**





**Practical No.: 3**

**Aim:** Compute the following node level measures: (i) Density; (ii) Degree;

(iii) Reciprocity; (iv) Transitivity; (v) Clustering.

**Description:**

1. Density : It refers to the “connection” between participants. It defined as the number of connections a participant has, divided by the total possible connections a participant could have.
2. vcount() : The size of the graph(number of vertices)

Returns the count of number of vertices in graph

1. ecount() : The size of the graph( number of edges)

Returns the count of number of edges in graph

1. degree : The degree of a vertex is its most basic structural property, the number of its adjacent edges. Degree takes one or more graphs and returns the degree centralities of positions within the graph.
2. Reciprocity: Calculates the reciprocity of a directed graph.

The measure of reciprocity defines the proportion of mutual connections, in a directed graph. It is most commonly defined as the probability that the opposite counterpart of a directed edge is also included in the graph.

1. graph.formula(): Creating (small) graphs via a simple interface

This function is useful if you want to create a small (named) graph quickly, it works for both directed and undirected graphs.

1. plot(): Use to plot any graph

Use to plot the directed or undirected graph.

1. Dyad.census() : Classify dyads in a directed graphs. The relationship between each pair of vertices is measured. It can be in three states: mutual, asymmetric or non-existent.

A named numeric vector with 3 elements:

mut : The number of pairs with mutual connections.

asym : The number of pairs with non-mutual connections.

null : The number of pairs with no connection between them.

1. Transitivity : Transitivity measures the probability that the adjacent vertices of a vertex are connected. This is sometimes also called the clustering coefficient.

10. Adjacent.triangles : Count how many triangles a vertex is part of in a graph or just list the triangles of a graph.

11. barabasi.game : generate a scale free graph according to the barabasi-albert model. The BA-model is a very simple stochastic algorithm for building a graph.

N : number of vertices

P : numeric constant, the contant out-degree of the vertices.

Directed : whether to create directed graph

12. watts.strogatz.game : Generate a graph according to the Watts-Strogatz network model.

dim : Interger constant, the dimension of the starting lattice.

size : Integer constant, the size of the lattice along each dimension.

nei : Interger constant, the neighbourhood within which the vertices of the

lattice will be connected.

p : Real constant between zero and one, the rewiring probability

1. simplify() : Simple graphs are graphs which do not contain loop and multiple edges.

**Code:**

library(igraph)

g <- graph.famous("Krackhardt\_Kite")

vcount(g)

ecount(g)

ecount(g)/(vcount(g)\*(vcount(g)-1)/2)

degree(g)

dg <- graph.formula(1-+2, 1-+3, 2++3)

plot(dg)

reciprocity(dg)

dyad.census(dg)

kite <- graph.famous("Krackhardt\_Kite")

atri <- adjacent.triangles(kite)

plot(kite, vertex.label=atri)

transitivity(kite, type="local")

adjacent.triangles(kite)/(degree(kite)\*(degree(kite)-1)/2)

library(igraph)

g2 <- barabasi.game(50, p=2, directed=F)

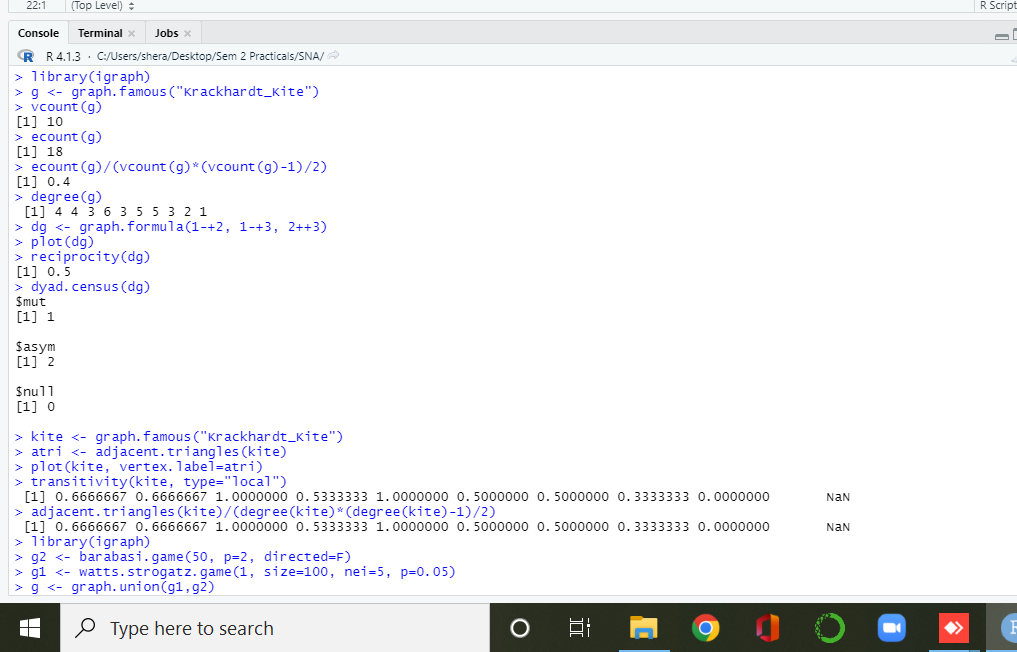
g1 <- watts.strogatz.game(1, size=100, nei=5, p=0.05)

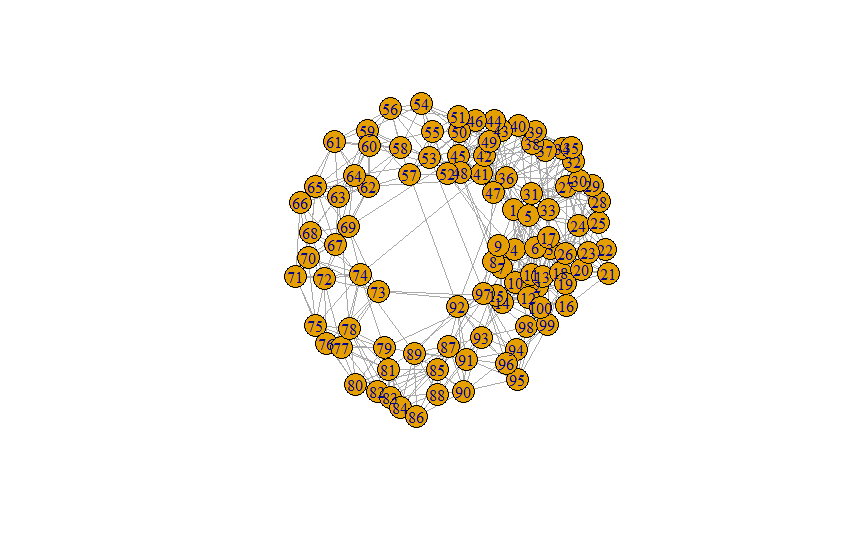
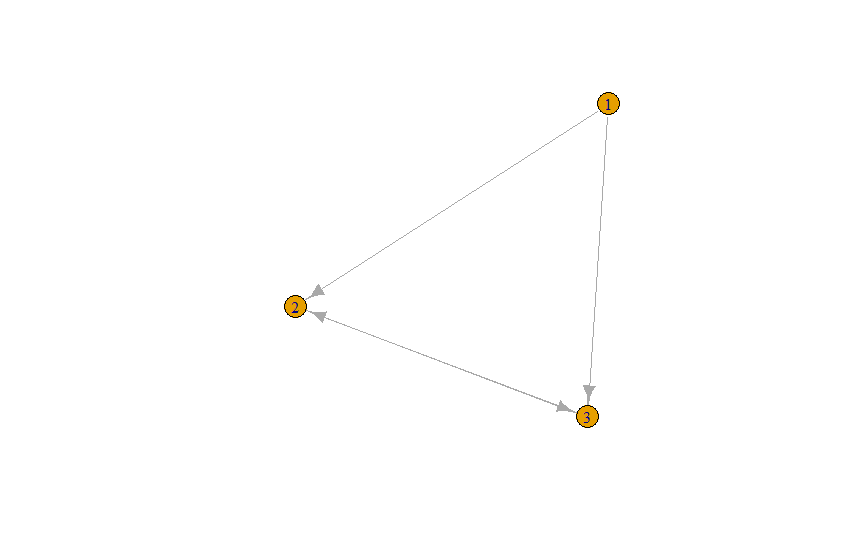
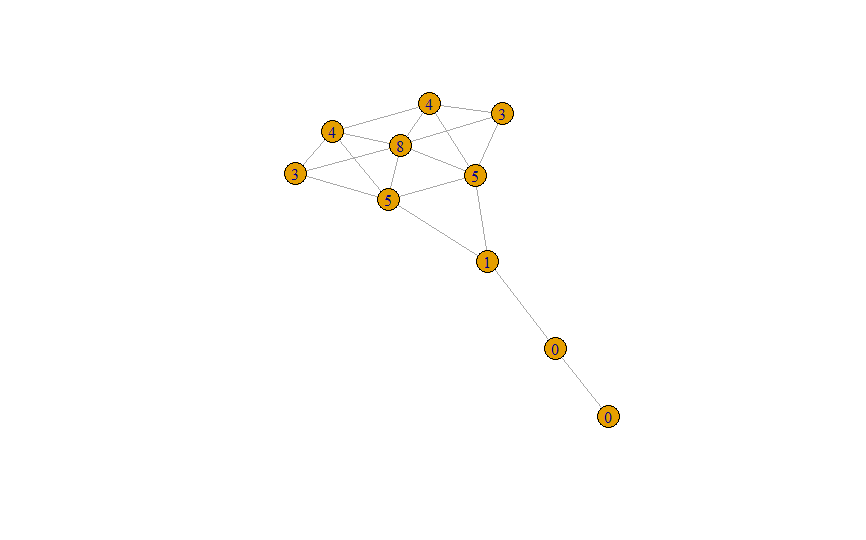
g <- graph.union(g1,g2)

g <- simplify(g)

plot(g)

**Output:**





**Practical No.: 4**

**Aim:** For a given network find the following: (i) Length of the shortest path from a given node to another node; (ii) the density of the graph.

**Description:**

as.matrix- attempts to turn its argument into a matrix.

read.table- Reads a file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the file.

is.na- Not Available’ / Missing Values.

graph.adjacency- a flexible function for creating igraph graphs from adjacency matrices.

shortest.paths- calculates the length of all the shortest paths from or to the vertices in the network. shortest\_paths calculates one shortest path (the path itself, and not just its length) from or to the given vertex.

graph.formula- This function is useful if you want to create a small (named) graph quickly, it works for both directed and undirected graphs.

graph.density- The density of a graph is the ratio of the number of edges and the number of possible edges.

Simplify()-Simple graphs are graphs which do not contain loop and multiple edges.

**Code:**

#(i)Length of the shortest path from a given node to another node

library(igraph)

matt<-as.matrix(read.table(text=

"node R S T U

R 7 5 0 0

S 7 0 0 2

T 0 6 0 0

U 4 0 1 0",header=T))

nms<-matt[,1]

matt<-matt[,-1]

colnames(matt)<-rownames(matt)<-nms

matt[is.na(matt)]<-0

g<-graph.adjacency(matt,weighted=TRUE)

plot(g)

s.paths<-shortest.paths(g,algorithm="dijkstra")

print(s.paths)

shortest.paths(g,v="R", to="S")

plot(g,edge.label=E(g)$weight)

#(ii) the density of the graph

library(igraph)

dg <- graph.formula(1-+2, 1-+3, 2++3)

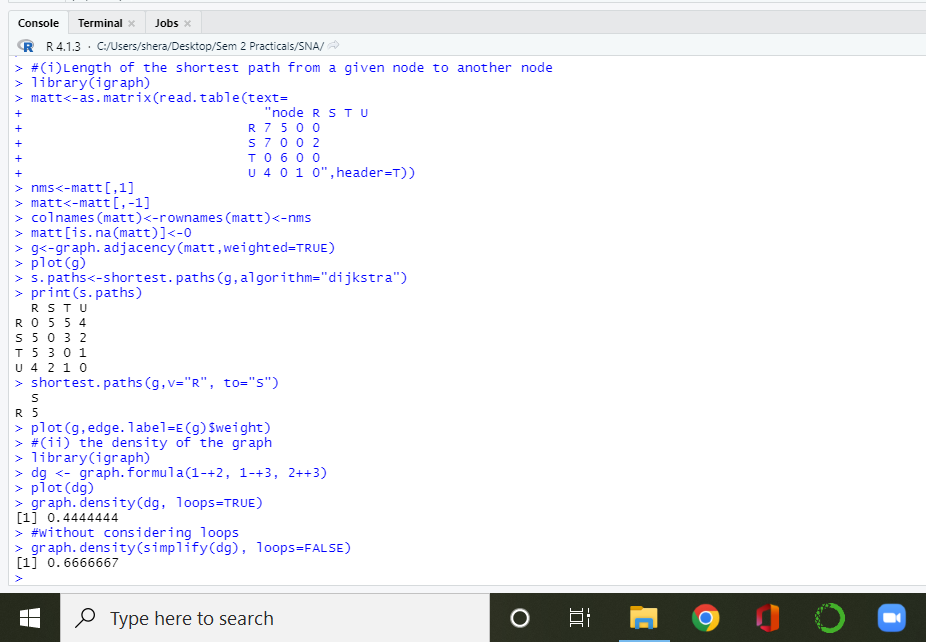
plot(dg)

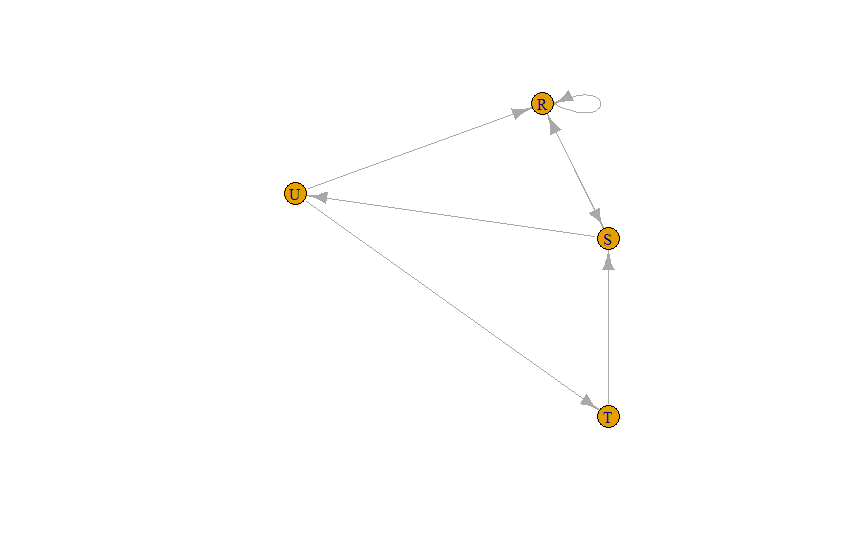
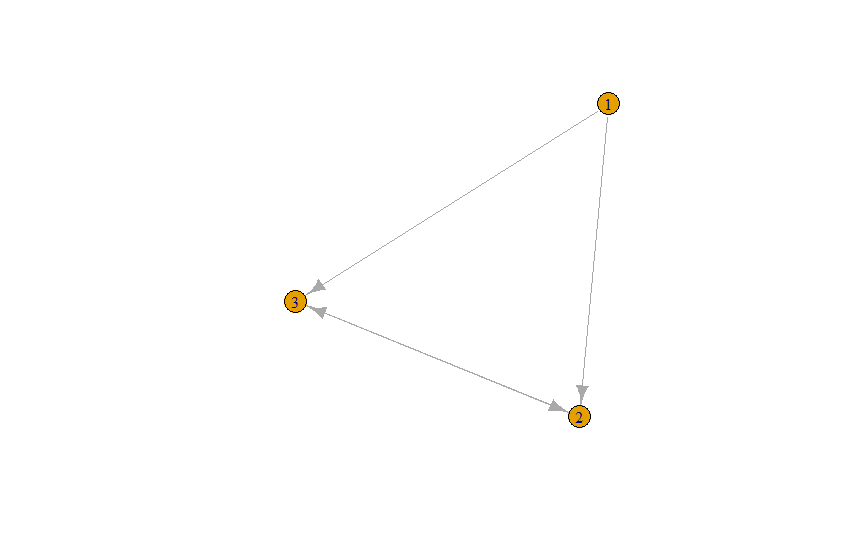
graph.density(dg, loops=TRUE)

#Without considering loops

graph.density(simplify(dg), loops=FALSE)

**Output:**





**Practical No.: 5**

**Aim:** Write a program to distinguish between a network as a matrix, a network as an edge list, and a network as a sociogram (or “network graph”) using 3 distinct networks representatives of each.

**Description:**

**graph.formula-** This function is useful if you want to create a small (named) graph quickly, it works for both directed and undirected graphs.

**get.adjacency-** Convert a graph to an adjacency matrix

**E()-**Edges of a graph. An edge sequence is a vector containing numeric edge ids, with a special class attribute that allows custom operations: selecting subsets of edges based on attributes, or graph structure, creating the intersection, union of edges, etc.

**get.adjedgelist-** Create adjacency lists from a graph, either for adjacent edges or for neighboring vertices.

**Code:**

#1. a network as a sociogram (or "network graph")

library(igraph)

ng=graph.formula(Andy++Garth,Garth-+Bill,Bill-

+Elena,Elena++Frank,Carol-+Andy,Carol-

+Elena,Carol++Dan,Carol++Bill,Dan++Andy,Dan++Bill)

plot(ng)

#2. a network as a matrix

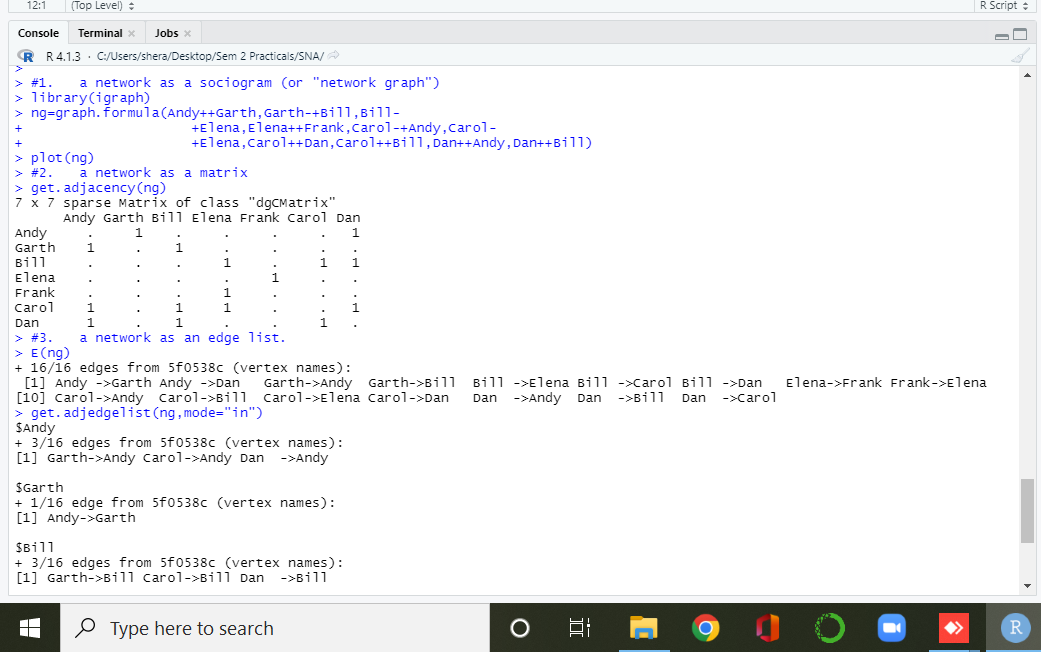
get.adjacency(ng)

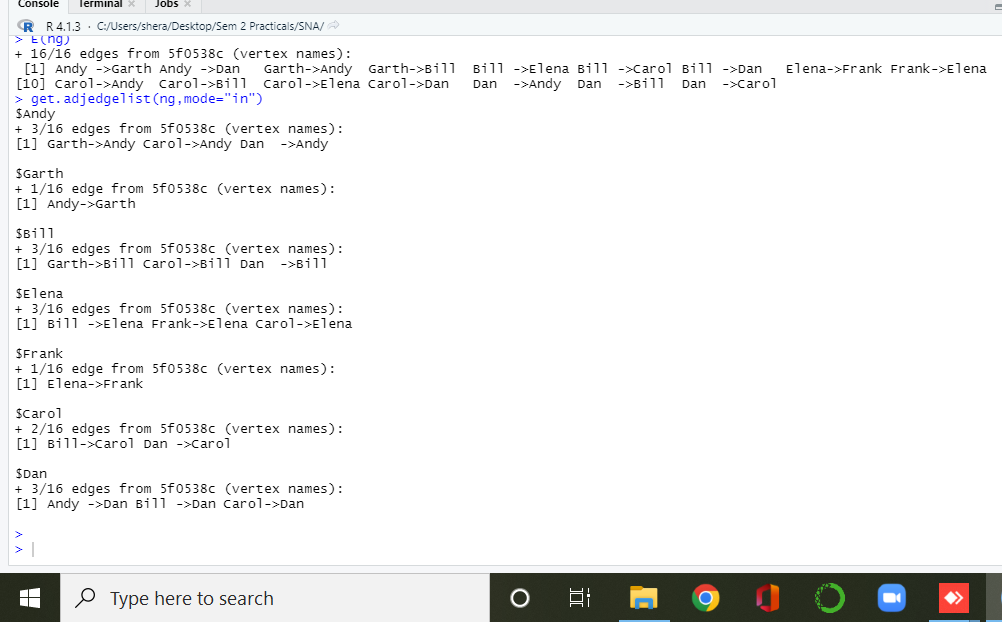
#3. a network as an edge list.

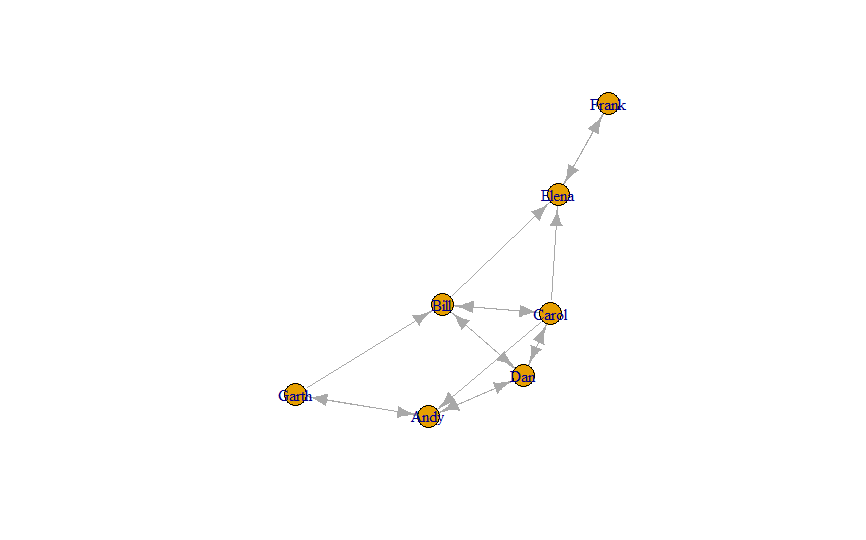
E(ng)

get.adjedgelist(ng,mode="in")

**Output:**







**Practical No.: 6**

**Aim:** Write a program to exhibit structural equivalence, automorphic equivalence, and regular equivalence from a network.

**Code:**

install.packages("sna")

library(sna)

library(igraph)

links2 <- read.csv("C:/Users/micha/Desktop/Sem 2 Practicals/SNA/edges1.csv", header=T, row.names=1)

eq<-equiv.clust(links2)

plot(eq)

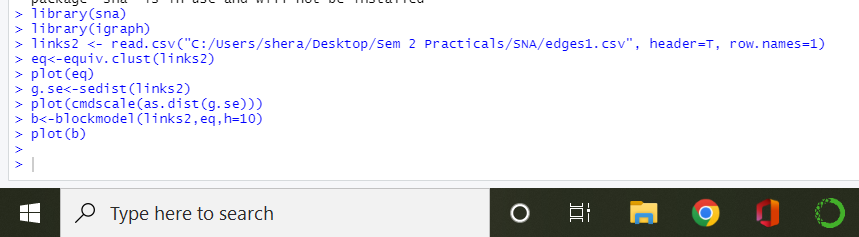
g.se<-sedist(links2)

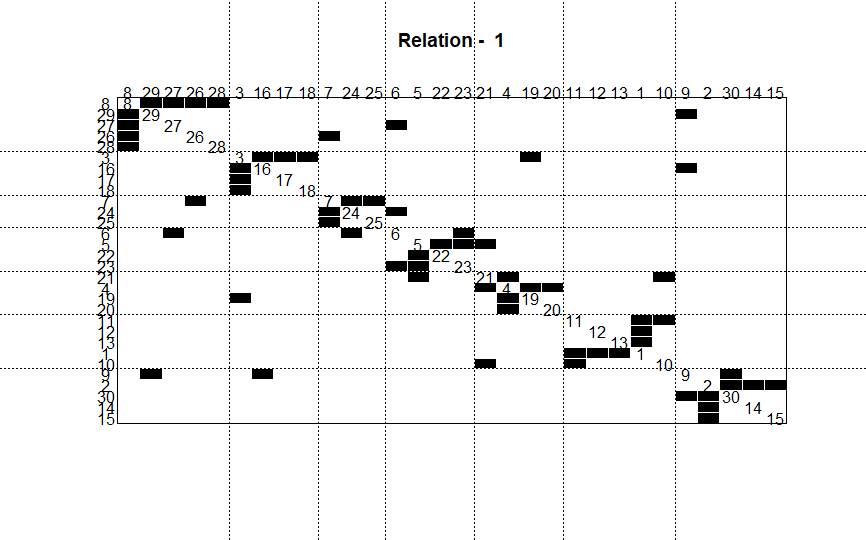
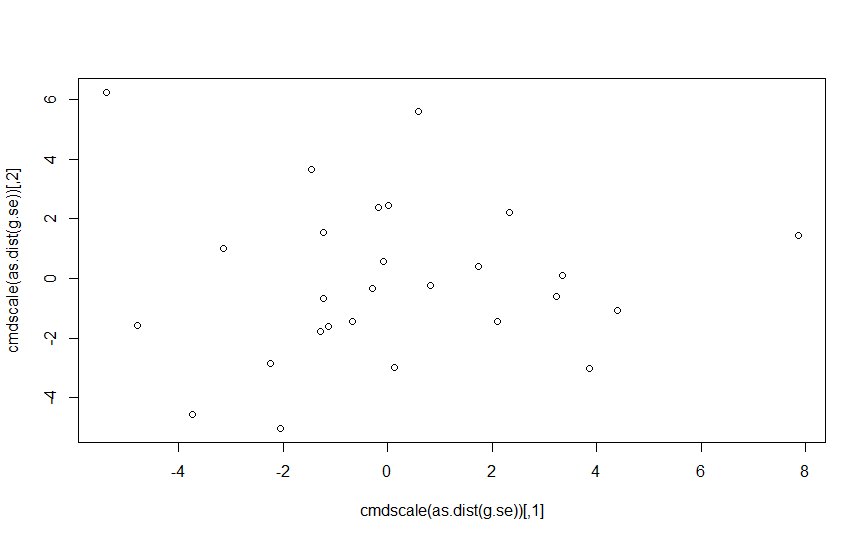
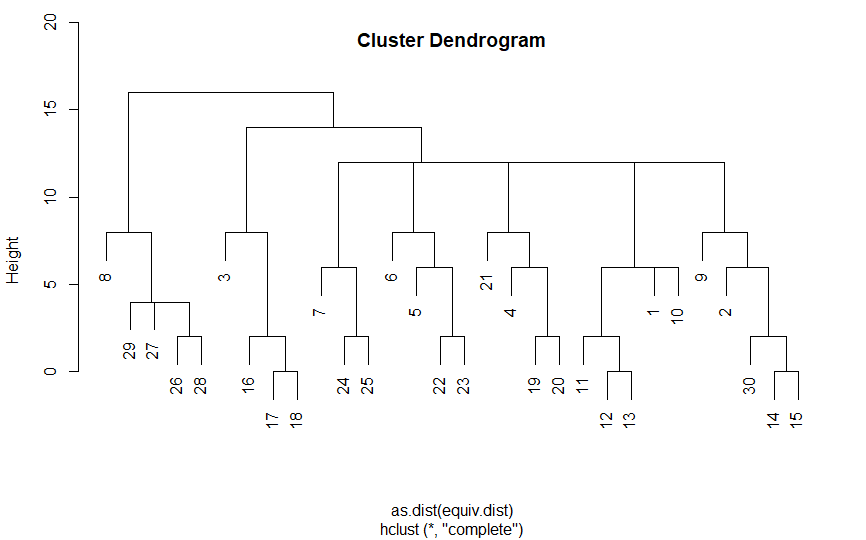
plot(cmdscale(as.dist(g.se)))

b<-blockmodel(links2,eq,h=10)

plot(b)

**Output:**





**Practical No.: 7**

**Aim:** Perform SVD analysis of a network**.**

**Description:**

Matrix- creates a matrix from the given set of values.

Svd()-Singular Value Decomposition of a Matrix. Compute the singular-value decomposition of a rectangular matrix.

**Code:**

install.packages('igraph')

library(igraph)

a<-matrix(c(1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,1,1,1),9,4)

print(a)

svd(a)

**Output:**

