Introduction to the mcgraph package

Detlef Groth

13. Februar 2022

The package *mcgraph* can be used to create different types of directed and undirected graphs and data for those graphs where the correlations between the nodes, which can be seen as well as variables, represent the graph structure. The data are generated using Monte Carlo simulations. The average strength of the associations between the nodes can be modulated by changing the number of iterations or the amount of noise. Finally the package contains a few convenience functions to visualize the graphs and the correlations between the nodes.

Introduction

The *mcgraph* package can be used to create various variations of random or defined graph types and to create data for those graphs where the correlations between individual nodes/variables resemble the graph topology. The user can further create directed from undirected graphs, either using the graph creation functions provided by this package or by their own input graphs. So in comparison to other graph and graph data generation packages such as the *huge* package,[1] the user can create directed from undirected graphs and has control over the input nodes from those graphs. The package *mcgraph* is not thought to be used as a tool for analysing graphs in great detail, however some standard methods are provided to allow basic inspection and analysis of the generated graphs and their data. For more sophisticated analysis of the graphs and their data, packages like *igraph*[2] or *sna*[3] should be used.

The resulting data can be used to reconstruct the graph by using the simple correlation threshold method, a pruning method which just hilights strong correlations between nodes. This method is just given for illustration purposes. It works only well for simple and small graphs, more complex graphs should be reconstructed using methods like linear model approaches with greedy algorithms such as stepwise forward selection or more efficient best subset selection methods such as the *Lasso*,[4] the latter as well applicable for larger graphs. Therefor the R packages like *qgraph*[5] or *huge*[1] should be used. Further there are also partial correlation approaches which are implemented for instance in the package *PCIT*.[6]

Generating Graphs

Graphs can be created based on an user defined adjacency matrix or by using the random graph generators provided by the *mcgraph* package. Let's start with a simple graph made step by step. We first create an empty adjacency matrix, thereafter, name the node with letters and add edges between some of the nodes. In an adjacency graph nodes connected by an edge have a matrix value of one, nodes which are unconnected have an matrix value of zero. In a directed graph the direction is read in form of from row-node-name to column-node-name. The adjacency matrix created in the following R code represents a directed graph.

```
library(mcgraph)
```

```
## Loading required package: MASS
```

```
## Loading required package: rpart
```

```
## Loading required package: Rcpp
```

```
G=matrix(0,nrow=8,ncol=8)
rownames(G)=colnames(G)=LETTERS[1:8]
G['A','C']=1
G['B','C']=1
G['C','D']=1
G['D','E']=1
G['D','F']=1
G['E','F']=1
G
```

```
##   A B C D E F G H
## A 0 0 1 0 0 0 0 0
## B 0 0 1 0 0 0 0 0
## C 0 0 0 1 0 0 0 0
## D 0 0 0 0 1 1 0 0
## E 0 0 0 0 0 1 0 0
## F 0 0 0 0 0 0 0 0
## G 0 0 0 0 0 0 0 0
## H 0 0 0 0 0 0 0 0
```

The matrix has now 8 rows and 8 columns. We can convert such an adjacency matrix to a *mcgraph* object by using the function *mcg.new*. Optionally it is possible to specify a type name to indicate which type of graph we have.

```
G=mcg.new(G,type="mygraph")
class(G)
```

```
## [1] "mcgraph"
```

With the new graph object we have a few S3 class methods for a *mcgraph* object to our use:

```
methods(class=class(G))
```

```
## [1] as.matrix degree    density   plot      summary
## see '?methods' for accessing help and source code
```

```
degree(G)
```

```
## A B C D E F G H
## 1 1 3 3 2 2 0 0
```

```
degree(G,mode="out")
```

```
## A B C D E F G H
## 1 1 1 2 1 0 0 0
```

```
degree(G,mode="in")
```

```
## A B C D E F G H
## 0 0 2 1 1 2 0 0
```

```
par(mai=c(0.1,0.1,0.2,0.0),mfrow=c(2,2))
plot(G,vertex.size=2,layout="mds",main="mds")
plot(G,vertex.size=2,layout='circle',main="circle")
plot(G,vertex.size=2,layout='sam',main="sam(mon)")
p=recordPlot()
```

The *degree* function returns the number of edges for each graph, with the *plot* function we can visualize our graph. The default layout for plotting is 'mds' where classical multidimensional scaling (MDS) is used to calculate the graph coordinates using the first two eigenvectors with largest eigenvalues. The data which are used are this shortest path data for the graph, so the longer the path between two nodes, the longer the distance between the nodes in the plot. This MDS approach however, is here not optimal as two nodes are overlapping (see figure **??**). We switch therefor to *circle* and *sam* layout. The latter layout is based on a MDS variant, "Sammon's Non-Linear Mapping" implemented in the *sammon* function of the *MASS* package which, as a recommended package, should be part of every standard R installation. In case of directed graphs, input nodes, nodes with no incoming edge are colored by default red and output nodes, nodes with no outgoing edge but input edges are colored blue. Nodes with incoming and outcoming edges are colored gray.

If you would like to provide your own plotting layout, you can achieve this by setting the layout attribute of your graph. For this you have to create matrix with two columns, the first columns contains the x, the second the y values for the node layout in two dimensions. The layout matrix must have the same number of rows as the graph has nodes.

```
lay=matrix(c(1,2, 1,1, 2,1.5, 3,1.5, 4,2, 4,1, 2,2, 3,2),
    ncol=2,byrow=TRUE)
lay
```

```
##      [,1] [,2]
## [1,]    1  2.0
## [2,]    1  1.0
## [3,]    2  1.5
## [4,]    3  1.5
## [5,]    4  2.0
## [6,]    4  1.0
## [7,]    2  2.0
## [8,]    3  2.0
```

```
attr(G,'layout')=lay
par(mfrow=c(2,2),mai=c(0.1,0.1,0.1,0.1))
replayPlot(p)
plot(G,vertex.size=2,main="custom")
```

For more sophisticated plotting methods and other graph layouts you should use the *igraph* package. Instead of creating our own graphs with a defined topology, we can as well use some of the provided graph generators of the *mcgraph* package. Here an example with a random graph giving the number of edges and the number of nodes as an argument.

```
set.seed(12345)
ang=mcg.angie(nodes=12,edges=18)
```

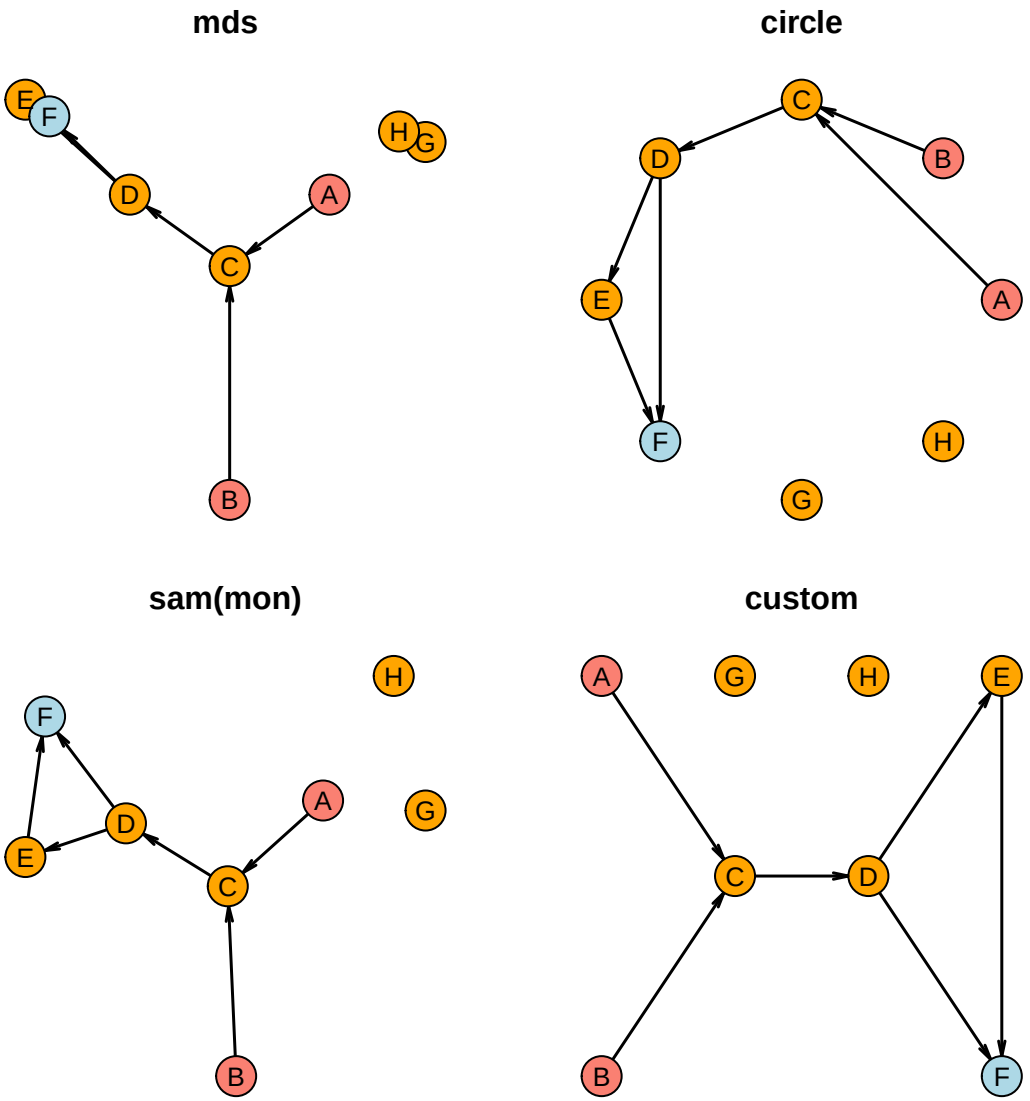


Figure 1: Plotting graph with different layouts.

The resulting graph is shown in figure 2 in the upper left. The *angie* graph algorithm at first constructs a tree structure by adding nodes to existing nodes chosing one of them randomly. When all nodes are added to the graph, the remaining edges are added randomly to connect two existing nodes. In comparison to the similar Erdos-Renyi graph creation algorithm we always end up with a single graph component if the number of edges plus two is larger than the number of nodes: $|E|+2 > |N|$.

Creating directed from undirected graphs

The *mcgraph* package allows you to generate directed from undirected graphs. This is done specifying 1 or more input nodes. The user can either specify defined input nodes by providing node names(s) or by giving a number where one or more nodes are randomly chosen as input nodes.

Let's try both approches on the graph *ang* created before. First we specify two input nodes by their node names: 'A1' and 'G1', thereafer we let the algorithm randomly choose 3 inout nodes:

```
par(mfrow=c(2,2),mai=c(0.0,0.1,0.1,0.0))
plot(ang,vertex.size=2,layout="sam")
anu=mcg.u2d(ang,input=c("A1","G1"))
plot(anu,vertex.size=2,layout="sam")
anu=mcg.u2d(ang,input=3)
plot(anu,vertex.size=2,layout="sam")
anu=mcg.u2d(ang,input=4)
plot(anu,vertex.size=2,layout="sam")
```

As you can see, edges are starting from the input node and spread in the same direction to other nodes. If the edge has already a direction the spread of input signals stops.

We can find out which graphs are input and output nodes by using the degree function using the mode argument:

```
degree(anu,mode="in")==0
```

```
##    A1    B1    C1    D1    E1    F1    G1    H1    I1    J1    K1    L1
## FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE
```

The three nodes which are input nodes have no incoming edges and therefore their value here is *TRUE*. We can further as well use mode *out* to check for output nodes, those ones which have no outgoing edges.

```
degree(anu,mode="out")==0
```

```
##    A1    B1    C1    D1    E1    F1    G1    H1    I1    J1    K1    L1
##  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
```

Exploring some other graph types of the mcgraph package

Let's now demonstrate some other graph topologies which can be created with the *mcgraph* package. We start with some graphs having a defined graph structure.

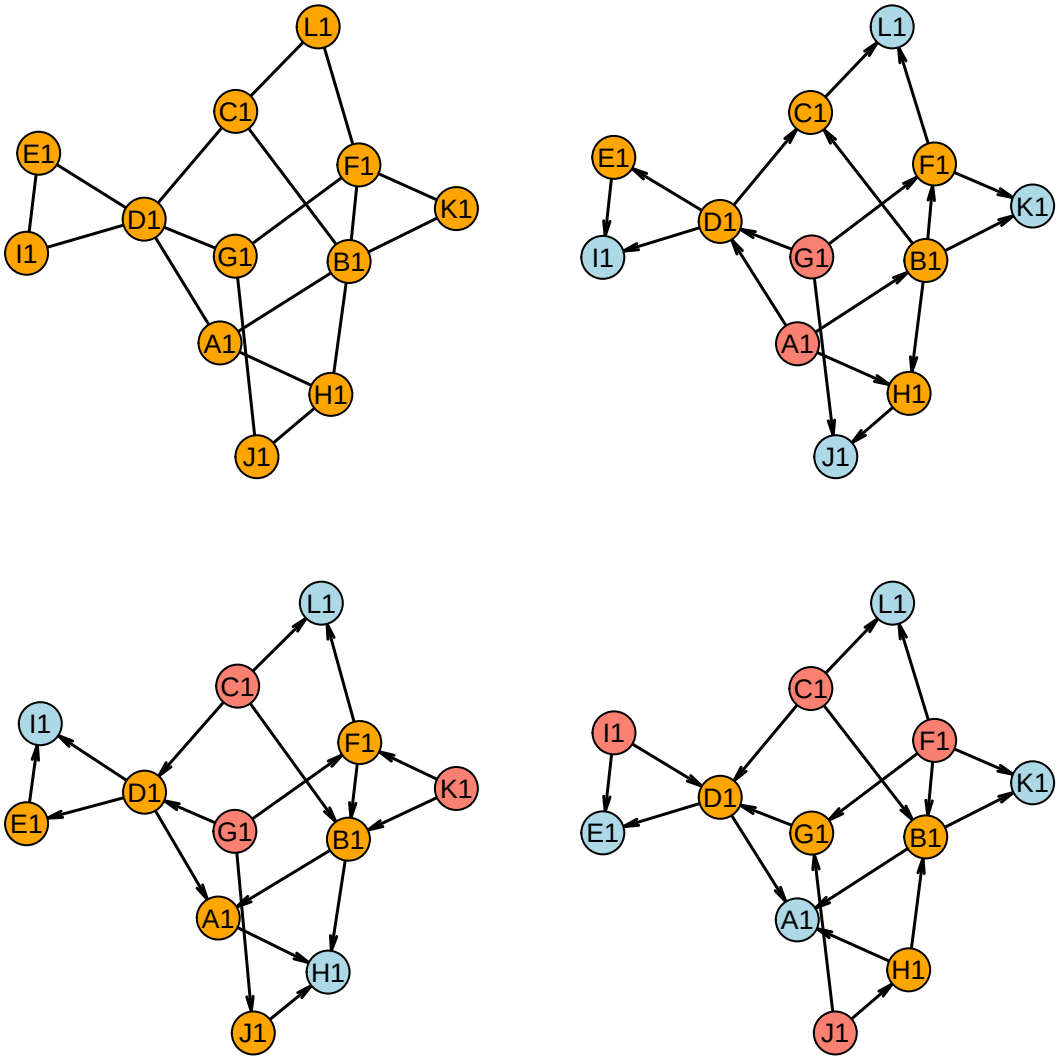


Figure 2: Random undirected and directed graphs made using the angie algorithm and using the mcg.u2d function by choosing defined or random input nodes .

```
par(mfrow=c(3,2),mai=c(0.1,0.1,0.1,0.1))
banu=mcg.band(nodes=12)
band=mcg.u2d(banu,input=1)
plot(band,vertex.size=2,layout="circle")
ciru=mcg.circular(nodes=12)
cird=mcg.u2d(ciru,input=1)
plot(cird,vertex.size=2,layout="circle")
cross=mcg.cross(bands=5,length=4)
crosd=mcg.u2d(cross,input=1)
plot(crosd,vertex.size=2,layout="star")
hubs=mcg.hubs(nodes=12,hubs=2)
hubd=mcg.u2d(hubs,input=c("A1","H1"))
plot(hubd,vertex.size=2,layout="circle")
latu=mcg.lattice(dim=5)
latd=mcg.u2d(latu,input=1)
plot(latd,vertex.size=2,layout="grid")
latu=mcg.lattice(dim=6,centralize=2)
latd=mcg.u2d(latu,input=3)
plot(latd,vertex.size=2,layout="grid")
```

There are as well a few random graph generation functions. Please note, that for the last graph, the cluster graph we have to ensure, that in every cluster is at least one inout node.

```
par(mfrow=c(2,2),mai=c(0.1,0.1,0.1,0.1))
ang=mcg.angie(nodes=12,edges=18)
and=mcg.u2d(ang,input=1)
plot(and,vertex.size=2,layout="circle")
bar=mcg.barabasi(nodes=12,m=1)
bad=mcg.u2d(bar,input=1)
plot(bad,vertex.size=2,layout="circle")
ran=mcg.random(nodes=12,edges=18)
rad=mcg.u2d(ran,input=1)
plot(rad,vertex.size=2,layout="circle")
clu=mcg.cluster(nodes=15,cluster=3,edges=24)
cld=mcg.u2d(clu,input=c("A1","F1","K1","M1"))
plot(cld,vertex.size=2,layout="circle")
```

Creating data for directed and undirected graphs

For given graphs data belonging to the graph topology can be generated. The nodes can be seen as variables for which several sample measurements are available. Correlations of the node data should then match the given graph structure. If an undirected input graph is given it is converted internally to an directed graph with one random input node and for this directed graph data are created. Data creation is based on a Monte Carlo simulation as follows:

For each node at first random values for a normal distribution with mean 100 and a standard deviation of 2 is created. Thereafter for each directed edge a new value for the node with incoming edges, the target node, a new values is generated where the new value of this node is the weighted mean, 95% of the new value comes from the target node and 5% of the value comes the outgoing source node. Thereafter some noise is added to the data. The order in which the nodes influence is each other is randomized in each iteration. This is repeated for all edges and several times. Finally the data for each node are returned. Let's now create some data, here 100 vectors representing simulations of

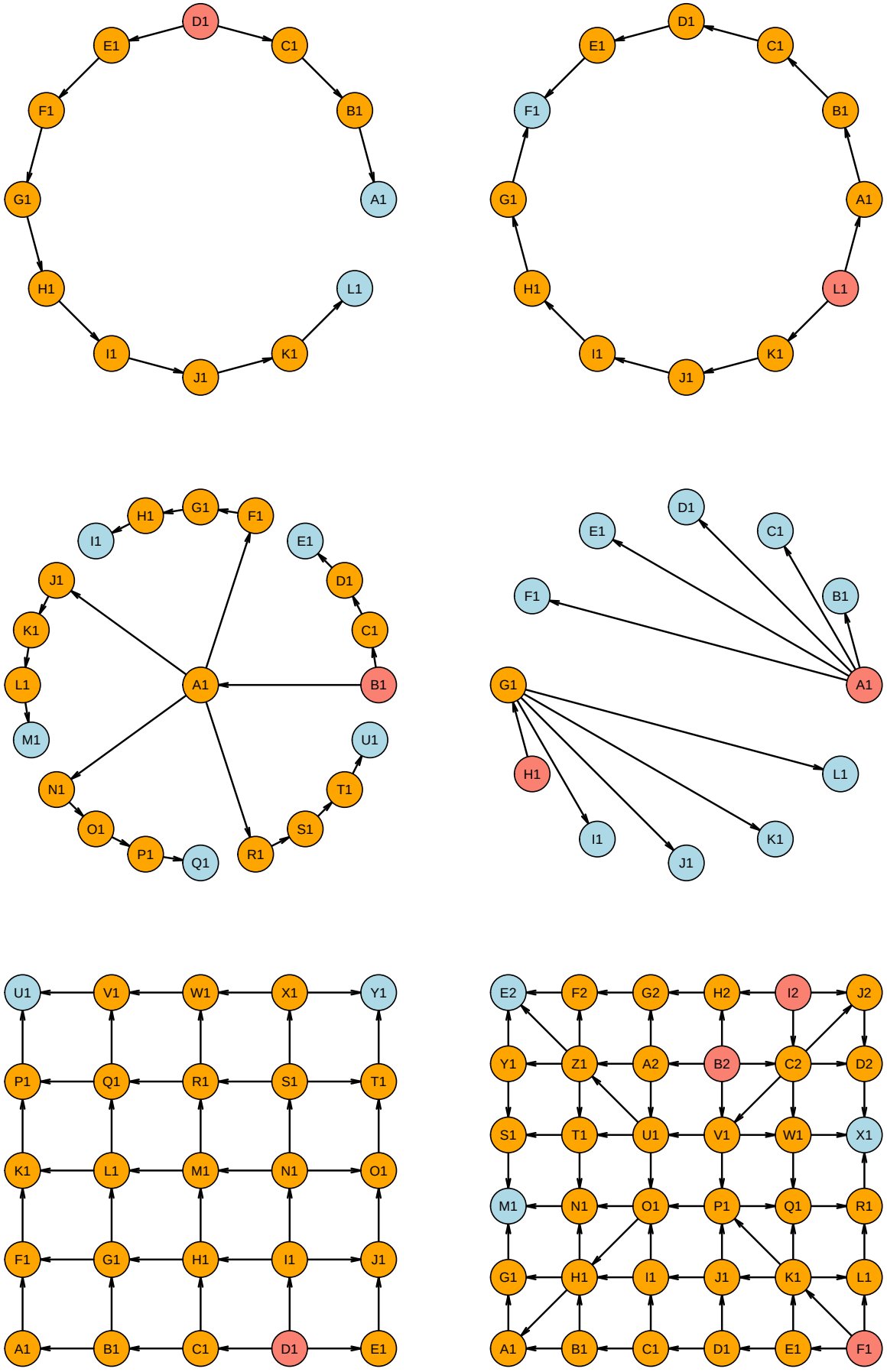


Figure 3: Regular graph topologies

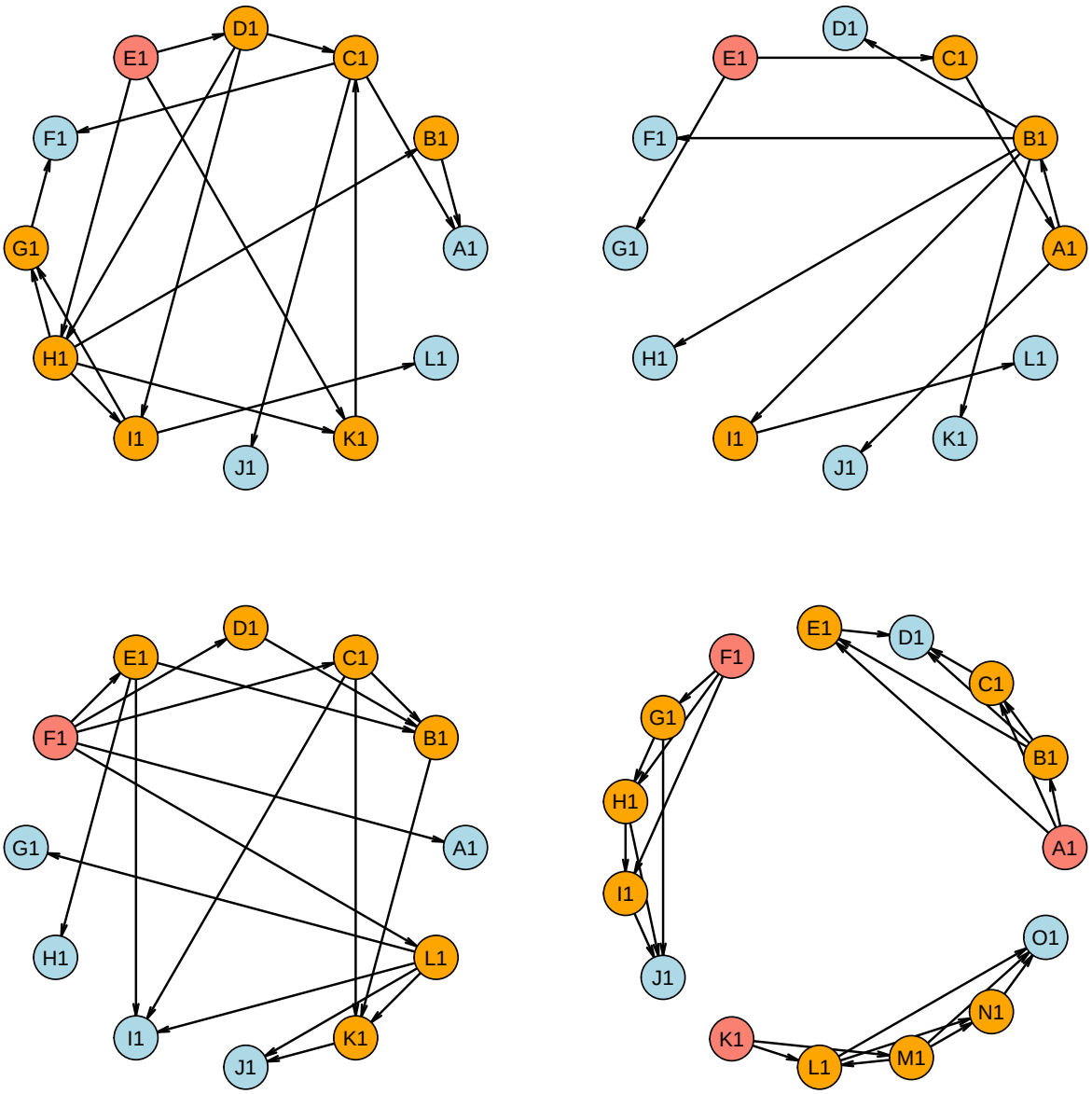


Figure 4: Random graph topologies

100 experiments for the given set of variables, for the graph G with the variables A-F created at the beginning of this vignette:

```
set.seed(12345)
G.data=mcg.graph2data(G,n=100,iter=15,noise=0.41)
G.data[,1:5]
```

```
##        [,1]      [,2]     [,3]      [,4]      [,5]
## A 102.33121 100.74060 96.06913 100.76824  97.55173
## B 101.18910 102.96359 99.23977 106.70512 103.79380
## C 101.57205 100.52525 98.37525 104.63316 100.77369
## D 103.94836 101.09887 97.21639 101.76710 102.67711
## E 101.01607 100.49450 98.69186 103.56880  97.44582
## F  99.26722  99.69670 98.97887 101.85766  97.36206
## G 102.81596 101.42695 96.33849 100.72826  99.63801
## H 100.17403  99.90669 99.96257  98.74632  98.64264
```

Let's now calculate the correlations between the variables A-F:

```
round(cor(t(G.data)),2)
```

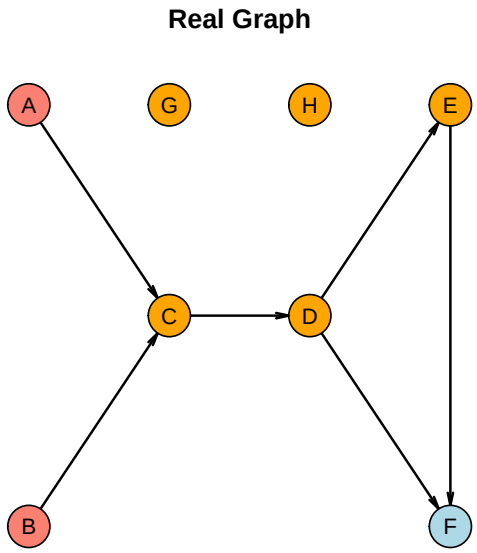
```
##       A     B     C     D    E     F     G     H
## A  1.00  0.08  0.48  0.13 0.02  0.17  0.02 -0.03
## B  0.08  1.00  0.59  0.21 0.04 -0.10  0.00 -0.17
## C  0.48  0.59  1.00  0.43 0.11  0.15 -0.02 -0.15
## D  0.13  0.21  0.43  1.00 0.43  0.42  0.05 -0.12
## E  0.02  0.04  0.11  0.43 1.00  0.57  0.14  0.07
## F  0.17 -0.10  0.15  0.42 0.57  1.00  0.10  0.12
## G  0.02  0.00 -0.02  0.05 0.14  0.10  1.00  0.09
## H -0.03 -0.17 -0.15 -0.12 0.07  0.12  0.09  1.00
```

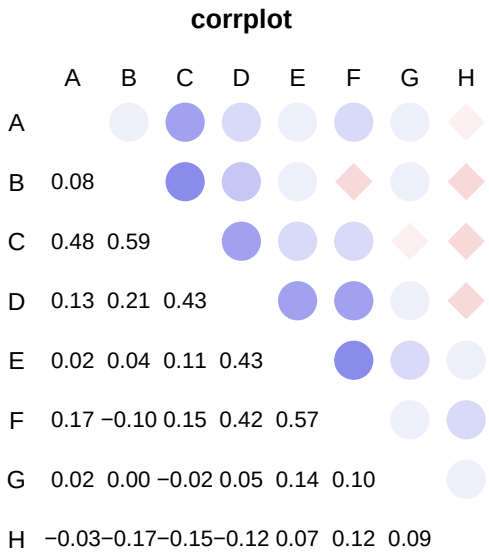
As you can see nodes connected by a directed edge, such as A with C, are strongly correlated with each other. In contrast A and B are here uncorrelated as this is a directed graph and both nodes are influencing C independently. If the edges would be in reverse direction, C would influence A and B then A and B would be correlated. Exploring such pairwise correlations is quite a common task. Therefor the package *mcgraph* offers a simple pairwise correlation plot function *mcg.corrplot*, below is an usage example. For more sophisticated correlation plots we highly recommend the nice *corrplot* package.[7]

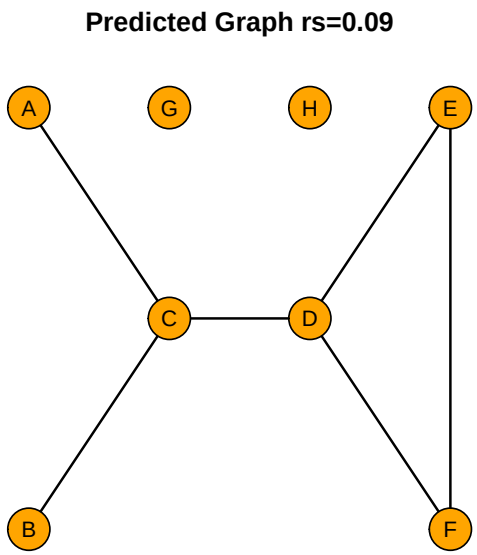
As network reconstruction based on correlation matrices is a common research problem we added the basic correlation threshold method to the package to construct a graph based on a given correlation matrix. This is a simple pruning approach just visualizing high associations without distinction of primary and secondary associations. In the resulting undirected graph every two nodes will be connected if the association between both nodes exceeds the given R-square value. The default R-square for the method is 0.04, representing positive or negative correlation higher than 0.2 or smaller than -0.2. As this produces far too many edges in the graph we thereafter increase the threshold to 0.25 representing absolute correlation values of more than 0.5, so strong associations. With this setting the original graph can be restored demonstrating the usefulness of the method.

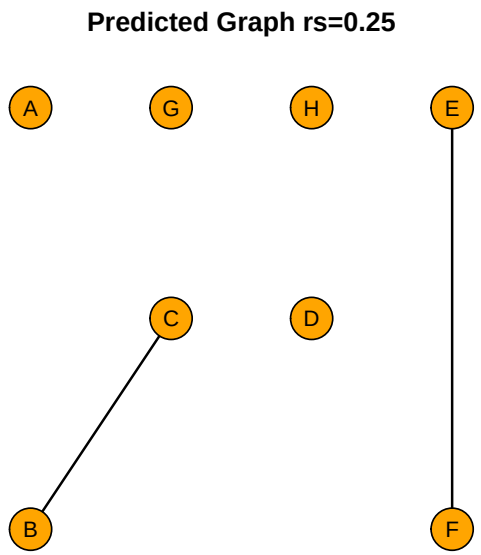
```
par(mfrow=c(2,2),mai=rep(0.3,4))
plot(G,vertex.size=2,main="Real Graph")
mcg.corrplot(cor(t(G.data)),text.lower=TRUE,pch.minus=18,main="corrplot")
G2=mcg.ct(cor(t(G.data)),rs=0.09)
```

```
attr(G2,"layout")=attr(G,"layout")
plot(G2,vertex.size=2,main="Predicted Graph rs=0.09")
G2=mcg.ct(cor(t(G.data)),rs=0.25)
attr(G2,"layout")=attr(G,"layout")
plot(G2,vertex.size=2,main="Predicted Graph rs=0.25")
```









In this small graph the correlation thresholding method can reconstruct the original graph topology if a "good" threshold is choosen. The package provides a few more network reconstruction methods such as `mcg.lvs` using linear regression fwith stepwise forward selection, `mvg.rpart` using regression trees and `mvg.glmnet` using Ridge, elastic net and Lasso regression. See below for more details on these methods. For more complex graphs R packages with more advanced graph reconstruction methods such as *huge*,[1] *qgraph* {[5]} or *PCIT*[6] should be used. These packages provide more elaborated methods to distinguish between direct (like A-C) and spurious associations (like A-D) in the network.

Weighted graphs

Since version 0.4.3 of the *mcgraph* package it is as well possible to weight the edges of the graphs. The higher the edge weights between the nodes of the graph, the higher the corresponding associations. Let's increase the edge weight for the connections between A and C and between E and F by 1.5, which should increase the correlations.

```
par(mfrow=c(1,2),mai=rep(0.3,4), pty="s")
mcg.corrplot(cor(t(G.data)),text.lower=TRUE,pch.minus=18,cex.coeff=0.8,
    main="Unweighted Graph")
G['A','C']=1.5
G['E','F']=1.5
G.data=mcg.graph2data(G,n=50,iter=15,noise=0.4)
mcg.corrplot(cor(t(G.data)),text.lower=TRUE,pch.minus=18,cex.coeff=0.8,
    main="Weighted Graph")
```

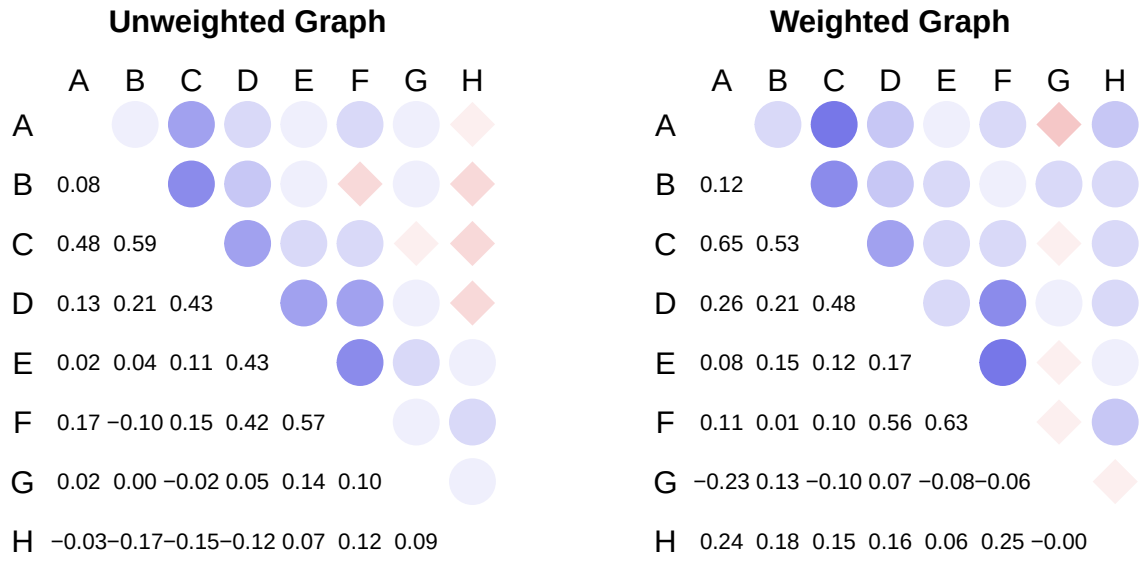


Figure 5: Unweighted vs weighted graph correlations

As you can see the correlation between A and C is much higher in the weighted graph than the correlation between B and C. The gain in strength is dependent on the number of iterations, so how often the neighbor nodes are influencing the target nodes. Below an example with the correlations after 5, 10, 15 and 20 iterations.

```
par(mfrow=c(3,2),mai=rep(0.3,4), pty="s")
for (i in c(3,5,10,15,30,50)) {
    G.data=mcg.graph2data(G,n=50,iter=i,noise=0.4)
    mcg.corrplot(cor(t(G.data)),text.lower=TRUE,pch.minus=18,cex.coeff=1,
        main=paste("Weighted Graph",i,"iterations"))
}
```

The result shows that after some iterations the strength of the associations does only marginally increase as there is a balance between the noise added in each iteration and the influence of the neighbor nodes.

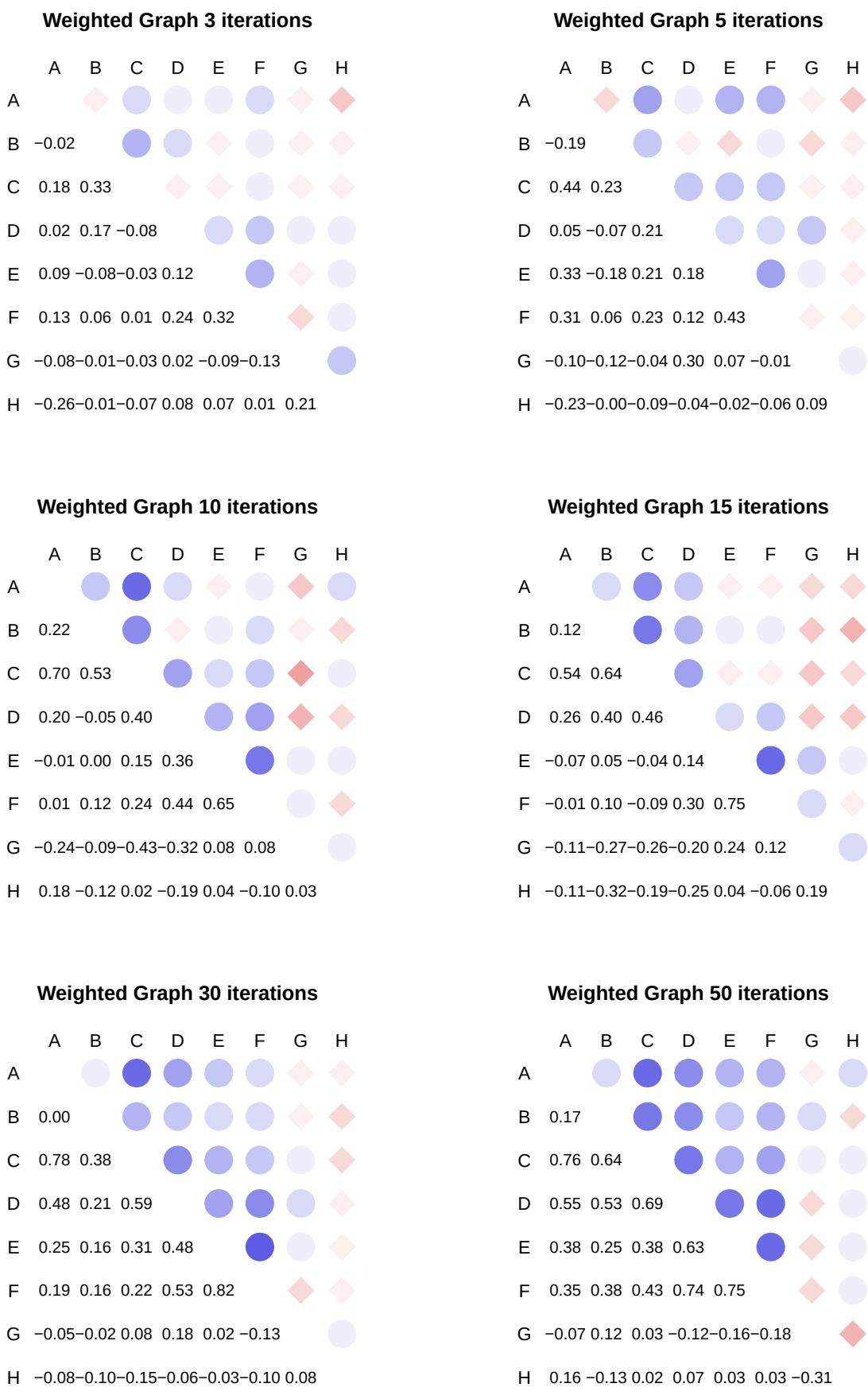


Figure 6: Monitoring of graph correlations after different iterations.

Own data input matrix

Since version 0.4.3 it is as well possible to start the iterations with your own input data. This feature can be as well used to monitor the increase in correlations over the iterations. Here an example which monitors the increase in correlation for the correlations between A and C as well as the correlations between B and C. As the weight for the A-C association is larger than that for the B-C association, the association A-C should be stronger than that of B-C.

```
par(mfrow=c(1,2),mai=c(0.9,0.9,0.7,0.1))
set.seed(1234)
for (n in c(0.3,1.0)) {
    g.data=matrix(rnorm(800,mean=100,sd=3),nrow=8)
    corsAC=c(cor(t(g.data))[1,3])
    corsBC=c(cor(t(g.data))[2,3])
    corsAB=c(cor(t(g.data))[1,2])
    corsCD=c(cor(t(g.data))[3,4])
    for (i in 1:50) {
        g.data=mcg.graph2data(G,iter=1,n=100,init=g.data,noise=n,code="C++");
        corsAC=c(corsAC,cor(t(g.data))[1,3])
        corsAB=c(corsAB,cor(t(g.data))[1,2])
        corsBC=c(corsBC,cor(t(g.data))[2,3])
        corsCD=c(corsCD,cor(t(g.data))[3,4])
    }
    plot(0:50,corsAC,type="l",ylim=c(0,1),xlab="iteration",ylab="r",
                col="red",lwd=2,main=paste("noise:",n))
    points(0:50,corsBC,type="l",col="blue",lwd=2)
    points(0:50,corsAB,type="l",col="grey",lwd=2)
    points(0:50,corsCD,type="l",col="darkgreen",lwd=2)
    legend("topright",legend=c("AC", "BC", "AB","CD"),
               col=c('red','blue','grey','darkgreen'),pch=15)
}
```

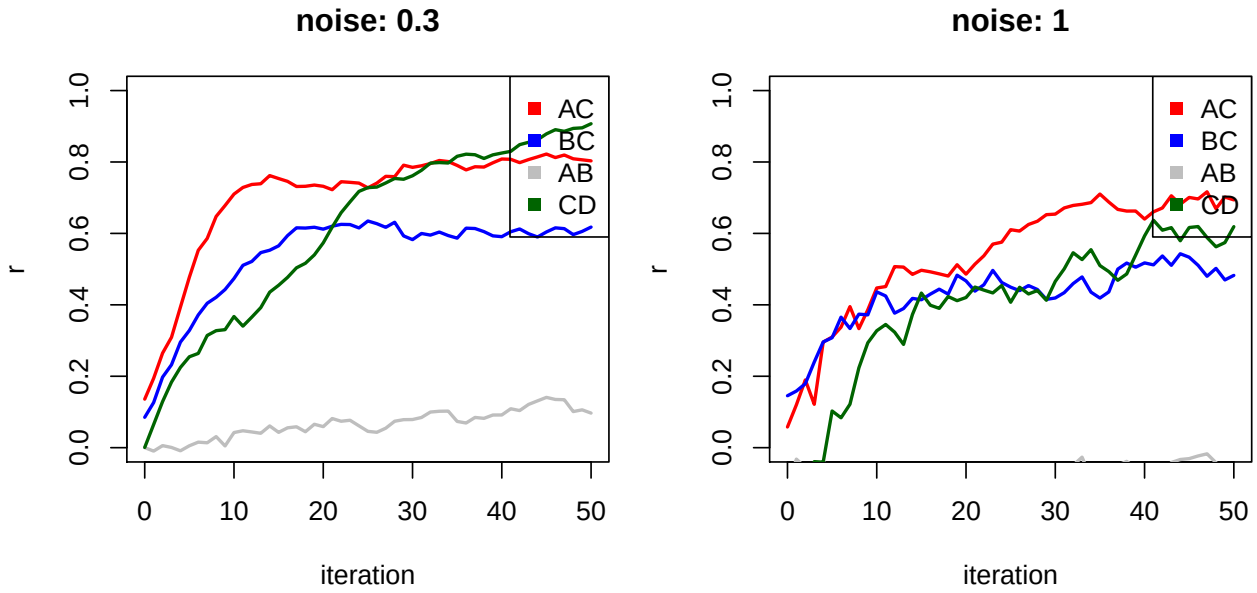


Figure 7: Correlation changes at different iterations and different noise settings. The weight for the edge A-C was set to 1.5 in this case, between A and B was no edge.

Graphs with negative correlations

Since version 0.4 of the *mcgraph* package, it is as well possible to use -1 in an adjacency matrix to indicate negative associations. In the example below our graph G is extended with a new node *G* which is negatively associated with A. This as side effect leads to an imbalance of the influence of A and B to C. For instance if A is constantly down-regulated by G, C depends very much on the low values of A, but not on the much more available B.

In the plots the layout of the on the left plot does not look nice, as letter A is shown right, so again here a own layout using a matrix of x and y coordinates is created:

```
par(mfrow=c(2,1),mai=rep(0.3,4))
set.seed(123)
G=matrix(0,nrow=7,ncol=7)
rownames(G)=colnames(G)=LETTERS[1:7]
G['A','C']=1
G['B','C']=1
G['C','D']=1
G['D','E']=1
G['D','F']=1
G['E','F']=1
G['G','A']=-1
G
```

```
##    A B C D E F G
## A  0 0 1 0 0 0 0
## B  0 0 1 0 0 0 0
## C  0 0 0 1 0 0 0
## D  0 0 0 0 1 1 0
## E  0 0 0 0 0 1 0
## F  0 0 0 0 0 0 0
## G -1 0 0 0 0 0 0
```

```
g=mcg.new(G)
plot(g,vertex.size=2,layout='sam',main="sam(mon) layout")
```

```
lay=matrix(c(1,3, 1,1, 2,2, 3,2, 4,3, 4,1, 2.5,3),byrow=TRUE,ncol=2)
colnames(lay)=c('x','y')
rownames(lay)=LETTERS[1:7]
lay
```

```
##     x y
## A 1.0 3
## B 1.0 1
## C 2.0 2
## D 3.0 2
## E 4.0 3
## F 4.0 1
## G 2.5 3
```

```
plot(g,vertex.size=2,layout=lay,main="custom layout")
```

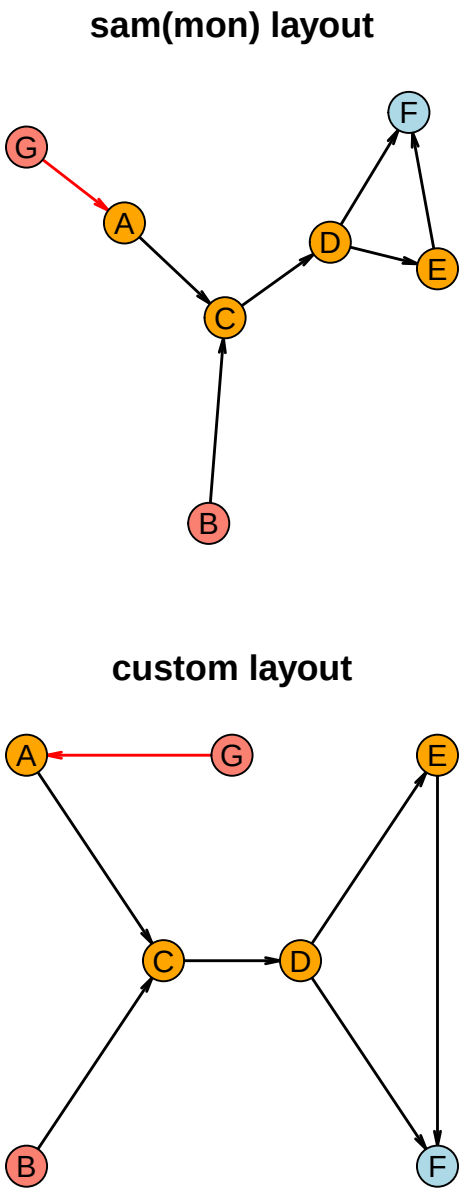


Figure 8: Graph with negative correlations

The more iterations are done, the stronger are the associations between the nodes/variables. If you add more noise the weaker the associations. In the example above with 10 iterations, the correlation for connected nodes are: -0.3, 0.55, 0.37, 0.43, 0.24, 0.47, 0.44, the average for those absolute values is: 0.4. See below an example with stronger associations due to more iterations and reduced noise.

```
data=mcg.graph2data(g,iter=25,noise=0.5)
kable(round(cor(t(data)),2), format="markdown",align="ccccccc",
    caption="Correlations with iter=25, noise=0.5")
```

Table 1: Correlations with iter=25, noise=0.5

	A	B	C	D	E	F	G
A	1.00	-0.02	0.87	0.54	0.18	0.14	-0.49
B	-0.02	1.00	0.30	0.18	-0.11	0.09	-0.08
C	0.87	0.30	1.00	0.60	0.15	0.17	-0.38
D	0.54	0.18	0.60	1.00	0.46	0.48	-0.18
E	0.18	-0.11	0.15	0.46	1.00	0.63	0.00
F	0.14	0.09	0.17	0.48	0.63	1.00	0.04
G	-0.49	-0.08	-0.38	-0.18	0.00	0.04	1.00

In the example above the correlations for connected nodes are: -0.49, 0.87, 0.3, 0.6, 0.46, 0.48, 0.63, the average for those absolute values is: 0.55.

Correlations can be weakend again by adding more noise to the data:

```
data=as.matrix(mcg.graph2data(g,iter=25,noise=1.2))
kable(round(cor(t(data)),2), format="markdown",align="ccccccc",
    caption="Correlations with iter=25, noise=1.2")
```

Table 2: Correlations with iter=25, noise=1.2

	A	B	C	D	E	F	G
A	1.00	0.02	0.83	0.41	0.09	0.18	-0.42
B	0.02	1.00	0.32	0.22	0.09	0.02	-0.10
C	0.83	0.32	1.00	0.50	0.08	0.16	-0.36
D	0.41	0.22	0.50	1.00	0.28	0.37	-0.08
E	0.09	0.09	0.08	0.28	1.00	0.56	-0.01
F	0.18	0.02	0.16	0.37	0.56	1.00	-0.08
G	-0.42	-0.10	-0.36	-0.08	-0.01	-0.08	1.00

Now the correlation for the connected nodes are: -0.42, 0.83, 0.32, 0.5, 0.28, 0.37, 0.56, the average for those absolute values drops down now to: 0.47.

Graph reconstruction methods

The *mcgraph* package provides currently four methods to construct network graphs from data

- `mcg.ct` - simple edge pruning method based on correlation threshold
- `mcg.lvs` - linear regression with greedy stepwise forward variable selection as long as the model improves

- `mcg.glmnet` - using Ridge, elastic net and Lasso regression, the latter is prefered as it reduces insignficant coefficients directly to zero
- `mvg.rpart` - using regression trees and connecting nodes which are required for prediction in one or the other direction

Let's apply those methods on a simple random graph where we now the structure:

```
par(mfrow=c(2,1), mai=rep(0.1,4))
set.seed(123)
ang=mcg.u2d(mcg.angie(nodes=10,edges=16),input=c("B1","F1"))
plot(ang,vertex.size=2)
ang
```

```
##    A1 B1 C1 D1 E1 F1 G1 H1 I1 J1
## A1  0  0  1  0  0  0  1  0  0  0
## B1  1  0  0  0  0  0  1  1  0  0
## C1  0  0  0  1  1  0  0  0  1  0
## D1  0  0  0  0  0  0  0  0  0  0
## E1  0  0  0  0  0  0  0  0  0  0
## F1  0  0  1  1  0  0  0  0  1  0
## G1  0  0  0  0  0  0  0  0  0  1
## H1  0  0  1  0  0  0  0  0  0  0
## I1  0  0  0  0  1  0  0  0  0  1
## J1  0  0  0  0  1  0  0  0  0  0
## attr(,"class")
## [1] "mcgraph"
## attr(,"type")
## [1] "angie"
## attr(,"mode")
## [1] "directed"
```

```
data=mcg.graph2data(ang,n=100,iter=10,noise=0.5)
mcg.corrplot(cor(t(data)),text.lower=TRUE,cex.coeff=0.8)
```

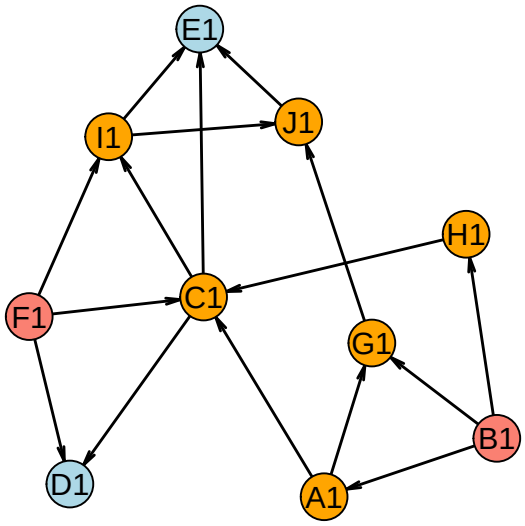
Let's now predict the graph using for instance `mcg.rpart`.

```
data[1:4,1:4]
```

```
##         [,1]      [,2]      [,3]      [,4]
## A1 102.3918  98.47514  98.33408 100.49694
## B1 102.0469 102.53960 104.19593  99.53687
## C1 103.1618 101.44118  99.59157  99.32007
## D1 100.1389  99.79450  99.58305 101.50329
```

```
ang.rpart=mcg.rpart(t(data),rs=0.04)
as.matrix(ang.rpart)
```

```
##    A1 B1 C1 D1 E1 F1 G1 H1 I1 J1
## A1  0  1  0  0  0  0  1  1  0  0
## B1  1  0  0  0  0  0  1  1  0  0
## C1  0  0  0  0  0  1  0  1  0  0
## D1  0  0  0  0  0  1  0  0  1  0
## E1  0  0  0  0  0  0  1  0  0  1
```



	A1	B1	C1	D1	E1	F1	G1	H1	I1	J1
A1										
B1	0.38									
C1	0.30	0.35								
D1	0.02	0.06	0.35							
E1	0.08	0.13	0.15	0.09						
F1	−0.02	0.18	0.50	0.53	0.20					
G1	0.52	0.60	0.30	−0.08	0.01	0.10				
H1	0.18	0.51	0.49	0.23	0.12	0.16	0.25			
I1	−0.02	0.13	0.41	0.38	0.23	0.48	−0.06	0.16		
J1	0.13	0.20	0.18	0.09	0.56	0.11	0.30	0.11	0.14	

Figure 9: A plot made by the `mcg.angie` function with according correlation matrix of generated synthetic data.

```
## F1  0  0  1  1  0  0  0  1  1  0
## G1  1  1  0  0  1  0  0  0  0  1
## H1  1  1  1  0  0  1  0  0  0  0
## I1  0  0  0  1  0  1  0  0  0  0
## J1  0  0  0  0  1  0  1  0  0  0
```

Let's calculate the number of correctly predicted edges (TP) and the number of wrongly predicted edges (FP):

```
args(mcg.accuracy)
```

```
## function (g.true, g.pred)
## NULL
```

```
round(unlist(mcg.accuracy(ang,mcg.rpart(t(data),rs=0.04))),3)
```

```
##       TP       FP       TN       FN     Sens     Spec      BCR       F1
##   10.000    4.000   25.000    6.000    0.625    0.862    0.744    0.667
##      MCC norm_MCC
##    0.504    0.752
```

```
round(unlist(mcg.accuracy(ang,mcg.glmnet(t(data),rs=0.04))),3)
```

```
##       TP       FP       TN       FN     Sens     Spec      BCR       F1
##   11.000    0.000   29.000    5.000    0.688    1.000    0.844    0.815
##      MCC norm_MCC
##    0.766    0.883
```

```
round(unlist(mcg.accuracy(ang,mcg.lvs(t(data),rs=0.04))),3)
```

```
##       TP       FP       TN       FN     Sens     Spec      BCR       F1
##    9.000    0.000   29.000    7.000    0.562    1.000    0.781    0.720
##      MCC norm_MCC
##    0.673    0.837
```

```
round(unlist(mcg.accuracy(ang,mcg.ct(t(data),rs=0.04))),3)
```

```
##       TP       FP       TN       FN     Sens     Spec      BCR       F1
##   14.000    6.000   23.000    2.000    0.875    0.793    0.834    0.778
##      MCC norm_MCC
##    0.644    0.822
```

As the mcgraph object created from these graph generators returns have as well a r.squared attribute we can apply the r-square threshold rs as well on those graphs from low to high. Here an example:

```
ang.rpart=mcg.rpart(t(data),rs=0.001)
round(attr(ang.rpart,"r.squared"),3)
```

```
##       A1    B1    C1    D1    E1    F1    G1    H1    I1    J1
## A1 0.000 0.050 0.000 0.000 0.000 0.009 0.311 0.105 0.000 0.000
## B1 0.043 0.000 0.000 0.000 0.000 0.000 0.401 0.114 0.000 0.000
## C1 0.000 0.000 0.000 0.017 0.000 0.155 0.034 0.327 0.000 0.000
## D1 0.000 0.070 0.031 0.000 0.000 0.428 0.025 0.000 0.019 0.000
## E1 0.000 0.000 0.000 0.000 0.000 0.000 0.122 0.000 0.000 0.372
## F1 0.017 0.000 0.041 0.345 0.000 0.000 0.000 0.000 0.178 0.000
## G1 0.030 0.464 0.000 0.000 0.000 0.065 0.000 0.000 0.000 0.000
## H1 0.000 0.384 0.118 0.000 0.000 0.078 0.000 0.000 0.000 0.000
## I1 0.000 0.000 0.000 0.043 0.000 0.345 0.000 0.000 0.000 0.000
## J1 0.000 0.000 0.000 0.000 0.447 0.000 0.074 0.000 0.000 0.000
```

By starting from a r-square threshold from 0.001 to 0.01, 0.04,0.09, 0.15,0.2,..0.5 we can measure the Specificity and the Sensitivity and generate a ROC curve and determine the AUC.

Here some example code to get started (from the posting: https://stats.stackexchange.com/questions/145566/how-to-calculate-area-under-the-curve-auc-or-the-c-statistic-by-hand):

```
n <- 100L

x1 <- rnorm(n, 2.0, 0.5)
x2 <- rnorm(n, -1.0, 2)
y <- rbinom(n, 1L, plogis(-0.4 + 0.5 * x1 + 0.1 * x2))
tab=table(y)
tab
```

```
## y
##  0  1
## 40 60
```

```
mod <- glm(y ~ x1 + x2, "binomial")

probs <- predict(mod, type = "response")
head(data.frame(probs=probs,y=y))
```

```
##       probs y
## 1 0.8360806 1
## 2 0.3955266 1
## 3 0.5667065 0
## 4 0.7410231 0
## 5 0.5833361 0
## 6 0.9369849 1
```

```
head(probs[y == 1L])
```

```
##         1         2         6         7         8         9
## 0.8360806 0.3955266 0.9369849 0.6349456 0.7395156 0.4973986
```

```
head(probs[y == 0L])
```

```
##         3         4         5        10        15        16
## 0.5667065 0.7410231 0.5833361 0.3916902 0.3335158 0.4563482
```

```
combinations <- expand.grid(positiveProbs = probs[y == 1L],
        negativeProbs = probs[y == 0L])
head(combinations)
```

```
##   positiveProbs negativeProbs
## 1     0.8360806     0.5667065
## 2     0.3955266     0.5667065
## 3     0.9369849     0.5667065
## 4     0.6349456     0.5667065
## 5     0.7395156     0.5667065
## 6     0.4973986     0.5667065
```

```
mean(combinations$positiveProbs > combinations$negativeProbs)
```

```
## [1] 0.71
```

```
dim(combinations)
```

```
## [1] 2400    2
```

```
tab[1]*tab[2]
```

```
##    0
## 2400
```

```
tail(combinations)
```

```
##      positiveProbs negativeProbs
## 2395     0.7723537     0.4078179
## 2396     0.6256036     0.4078179
## 2397     0.6265723     0.4078179
## 2398     0.7397870     0.4078179
## 2399     0.8276149     0.4078179
## 2400     0.6135327     0.4078179
```

Imputation of missing values

Often data are incomplete and we have missing values. Many mathematical procedures however require that we have complete data. In this case we have to guess/impute the missing values. Recommended approaches in this case are suing regression for numerical and classification for categorical data to guess the missing values. Simple replacement of the missing values for instance with the median or the mean which is often done does not produce vera good results. Better approaches are decision trees and the k-nearest neighbour approach where the missing values are replaced with the mean of exisiting values for the k-most similar samples were the value is missing. Below an example where we introduce some NA's in the iris data frame and then re-impute the values and compare the outcomes.

First lets introduce some NA's:

```
data(iris)
set.seed(123)
ir=as.matrix(iris[,1:4])
ir.mv=ir
# introduce 5 percent NA's
mv=sample(1:length(ir),as.integer(0.05*length(ir)))
ir.mv[mv]=NA
summary(ir.mv)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.400   Min.   :2.200   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.300   Median :1.300
##  Mean   :5.857   Mean   :3.054   Mean   :3.745   Mean   :1.205
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.200   Max.   :6.900   Max.   :2.500
##  NA's   :7       NA's   :10      NA's   :7       NA's   :6
```

Let's now impute the missing values using the median, regression trees and the knn approach.

```
ir.imp.med=mcg.impute(ir.mv,method='median') # not good
ir.imp.rpart=mcg.impute(ir.mv) # method rpart (default)
ir.imp.knn=mcg.impute(ir.mv,method='knn')
```

We can then measure the quality of the imputations using the Root mean squared error, the smaller the error, the better and as well the correlation, the higher the better:

```
rmse = function (x,y) { return(sqrt(sum((x-y)^2))) }
rmse(ir[mv],ir.imp.med[mv]) # should be high
```

```
## [1] 6.09508
```

```
rmse(ir[mv],ir.imp.rpart[mv]) # should be low!
```

```
## [1] 1.845312
```

```
rmse(ir[mv],ir.imp.knn[mv]) # should be low!
```

```
## [1] 1.764426
```

```
cor(ir[mv],ir.imp.med[mv])
```

```
## [1] 0.806656
```

```
cor(ir[mv],ir.imp.rpart[mv])
```

```
## [1] 0.9843221
```

```
cor(ir[mv],ir.imp.knn[mv]) # should be high!
```

```
## [1] 0.9852723
```

The rpart library can be as well used to impute factor variables using classification trees.

```
# factor variables
data(iris)
ciris=iris
idx=sample(1:nrow(ciris),15) # 10 percent NA's
ciris$Species[idx]=NA
summary(ciris)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :47
##  versicolor:41
##  virginica :47
##  NA's      :15
##
##
```

```
ciris=mcg.impute(ciris,method="rpart")
table(ciris$Species[idx],iris$Species[idx])
```

```
##
##              setosa versicolor virginica
##   setosa          3          0         0
##   versicolor      0          9         0
##   virginica       0          0         3
```

As you can see the result is quite good.

Benchmarking

As speed is an important issue for larger data sets the package contains as well a simple method, *mcg.timeit*, to measure the executation time of a R expression. As you can see below the standard C++ coded method *mcg.lvs* is much faster than the R counterparts.

```
data(swiss)
options(warn=-1)
mcg.timeit(expression(mcg.lvs(swiss, rs=0.1, output='mcgraph')))
```

```
## [1] 0.0007963181 0.0002861023 0.0002639294 0.0002691746 0.0004389286
```

```
mean(mcg.timeit(expression(mcg.lvs(swiss, rs=0.1, output='mcgraph'))))
```

```
## [1] 0.0002918243
```

```
mean(mcg.timeit(expression(mcg.rpart(swiss, rs=0.1))))
```

```
## [1] 0.0486814
```

```
mean(mcg.timeit(expression(mcg.lvs(swiss, rs=0.1, output='mcgraph',code="R"))))
```

```
## [1] 0.02755203
```

```
options(warn=0)
```

Summary

This vignette illustrates a few use cases for the package *mcgraph*, how you can create regular or random graphs and how to create data for those graphs where the correlations of the nodes/variables represent the graph structure. The methods for visualizing graphs and the data correlations are as well shown. For more details on the methods have a look at the R help packages provided with this package.

For much more sophisticated methods dealing with graphs and correlations I recommend to use the *igraph* and the *corrplot* R packages.

Overview on functions of the mcgraph package

The following functions are available in the *mcgraph* package:

1. functions to generated graphs with a defined topology
    - *mcg.band* create a band/chain graph
    - *mcg.circular* create a circular graph
    - *mcg.cross* create hub like graph where non-hub nodes are in chains
    - *mcg.lattice* create a grid like graph, optionally with some centralizing edges pointing towards the graph center
    - *mcg.hubs* create one or more graph components with a central hub node and nodes directly attached to this hub
    - *mcg.new* create graph based on a given adjacency matrix
2. functions to generate random graphs
    - *mcg.angie* create random graph with all nodes in one component
    - *mcg.barabasi* create random graph following the Barabasi-Albert model
    - *mcg.random* create random graph were all edges have equal probabilities to be generated
    - *mcg.cluster* create graph with several components, where nodes of each component are densely connected to each other
3. functions to create directed from undirected graphs and vice versa
    - *mcg.u2d* convert undirected into directed graph using defined or random input nodes
    - *mcg.d2u* convert directed into undirected graph where any directed edge becomes an undirected one
4. function to generate data for graphs
    - *mcg.graph2data* create data for graphs
5. S3 class methods to analyze and visualize *mcgraph* objects
    - *plot* plot graphs

- *degree* get degree centralities for graph nodes
- *density* determine ratio of edges to all possible edges
- *summary* summarize basic graph properties
- *as.matrix* extract the adjacency matrix of a *mcgraph* object useful for use the graph with other R packages

6. network construction functions

- *mcg.ct* - creates mcgraphs from correlation matrices with a given threshold of R-square
- *mcg.lvs* - creates mcgraphs using greedy selection of linear model variables
- *mcg.glmnet* - create mcgraphs using Ridge, elastic net or Lasso regression
- *mcg.rpart* - create mcgraphs using rpart regression trees

7. general purpose functions

- *is.mcgraph* checks if a given object is a *mcgraph*
- *mcg.autonames* create names for nodes automatically
- *mcg.corrplot* create a simple correlation plot of pairwise correlation
- *mcg.components* extract graph components, unconnected groups of nodes
- *mcg.impute* impute missing values for instance using decision trees or knn
- *mcg.shortest.paths* determine path lengths between nodes
- *mcg.timeit* measure executation time for a given R-expression

1. Jiang, H. *et al. Huge: High-dimensional undirected graph estimation.* (2020).

2. Csardi, G. & Nepusz, T. The igraph software package for complex network research. *InterJournal* **Complex Systems**, 1695 (2006).

3. Butts, C. T. Social network analysis with sna. *Journal of Statistical Software, Articles* **24**, 1–51 (2008).

4. Tibshirani, R. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)* **58**, 267–288 (1996).

5. Epskamp, S., Cramer, A. O. J., Waldorp, L. J., Schmittmann, V. D. & Borsboom, D. qgraph: Network visualizations of relationships in psychometric data. *Journal of Statistical Software* **48**, 1–18 (2012).

6. Watson-Haigh, N. S., Kadarmideen, H. N. & Reverter, A. PCIT: an R package for weighted gene co-expression networks based on partial correlation and information theory approaches. *Bioinformatics* **26**, 411–413 (2010).

7. Wei, T. & Simko, V. *R package "corrplot": Visualization of a correlation matrix.* (2017).