I. Generate background knowledge graph.

I.I: Manually build graphs from adjacency matrices.

```r
adj_mat = rbind(c(0,3,1,0,0,0,0,0,0),  # A's neighbors
               c(3,0,2,2,0,0,0,0,0),  # B's neighbors
               c(1,2,0,0,2,0,0,0,0),  # C's neighbors
               c(0,2,0,1,0,1,1,0,0),  # D's neighbors
               c(0,0,2,1,0,2,0,2,0),  # E's neighbors
               c(0,0,0,0,2,0,0,0,0),  # F's neighbors
               c(0,0,0,1,0,0,0,1,0),  # G's neighbors
               c(0,0,0,1,2,0,1,0,1),  # H's neighbors
               c(0,0,0,0,0,0,0,1,0))  # I's neighbors
colnames(adj_mat) = c("A", "B", "C", "D", "E", "F", "G", "H", "I")

# Convert adjacency matrices to igraph objects for all three graphs.
ig = graph.adjacency(adj_mat, mode="undirected", weighted=TRUE,
                      add.colnames = "name")
print(ig)

plot.igraph(ig, edge.width=5*E(ig)$weight)
```

# Multiply edge weights by 5 to amplify edge width
# differences
plot.igraph(ig, edge.width=5*E(ig)$weight)
I.II: Learn a graph from data.

Note all code chunks in sections I - IV may rely on lines in previous code chunks, so do not empty your environment between code chunks.

```r
# Load the Miller2015_Heparin dataset
data(Miller2015)
# Only include metabolites that are present in >90% reference samples.
fil.rate = as.numeric(Miller2015$`Times identified in all 200 samples`[-1])/200
names(fil.rate) = rownames(Miller2015)[-1]
data_mx = Miller2015[,grep("IEM_", colnames(Miller2015))]
data_mx = data_mx[which(fil.rate>0.90),]
dim(data_mx)
## [1] 620 186
# Remove any metabolites where any profile has a z-score > 1000. 
# These are likely imputed raw values that were not z-scored.
rmMets = names(which(apply(data_mx, 1, function(i) any(i>20))))
if (length(rmMets)>0) {
  data_mx = data_mx[-which(rownames(data_mx) %in% rmMets),]
}
dim(data_mx)
## [1] 196 186
# Get data from all patients with Argininemia
diags = Miller2015["diagnosis", grep("IEM", colnames(Miller2015))]
arg_data = data_mx[,which(diags=="Argininemia")]
# Add surrogate disease and surrogate reference profiles based on 1 standard 
# deviation around profiles from real patients to improve rank of matrix when 
# learning Gaussian Markov Random Field network on data. While surrogate 
# profiles are not required, they tend to learn less complex networks 
# (i.e., networks with less edges) and in faster time.
ind = which(diags="No biochemical genetic diagnosis")
arg_data = data.surrogateProfiles(arg_data, 1, ref_data = data_mx[,ind])
dim(arg_data)
```

---

![Graph Diagram](image_url)

**Graph Diagram Description:**

The graph consists of nodes labeled A, B, C, D, E, F, G, and H, connected by edges.

- Edge A connects to B and C.
- Edge B connects to C and D.
- Edge C connects to D and E.
- Edge D connects to E and F.
- Edge E connects to F and G.
- Edge F connects to G and H.
- Edge G connects to H.
- Edge H is isolated.

The graph illustrates the connections and relationships between the variables (nodes) within the dataset.
Learn a Gaussian Markov Random Field model using the Graphical LASSO in the R package "huge".
Select the regularization parameter based on the "STARS" stability estimate.
This will take 30 seconds - 1 minute.

```r
arg = huge(t(arg_data), method="glasso")
```

## Conducting the graphical lasso (glasso) with lossless screening....in progress: 9%
Conducting the graphical lasso (glasso)....done.

```r
plot(arg)
```

```
Sparsity vs. Regularization

Regularization Parameter

lambda = 0.341
lambda = 0.264
lambda = 0.204

Solution path sparsity levels

Regularization Parameter
```

This will take several minutes. For a faster option, you can use the "ebic" criterion instead of "stars", but we recommend "stars".

```r
arg.select = huge.select(arg, criterion="stars")
```

## Conducting Subsampling....in progress:5% Conducting Subsampling....in progress:10% Conducting Subsampling....done.

```r
plot(arg.select)
```
# This is the regularization parameter the STARS method selected.
print(arg.select$opt.lambda)

## [1] 0.2639634

# This is the corresponding inverse of the covariance matrix that corresponds
# to the selected regularization level.
arg_icov = as.matrix(arg.select$opt.icov)
# Remove all "self" edges, as we are not interested in self-relationships.
diag(arg_icov) = 0
rownames(arg_icov) = rownames(arg_data)
colnames(arg_icov) = rownames(arg_data)
# Convert adjacency matrices to an igraph object.
ig_arg = graph.adjacency(arg_icov, mode="undirected", weighted=TRUE,
                           add.colnames="name")
print(ig_arg)

## IGRAPH 99fcc76 UNW- 196 996 --
## + attr: name (v/c), weight (e/n)
## + edges from 99fcc76 (vertex names):
## [1] 1,5-anhydroglucitol (1,5-ag)--1-docosahexaenoyl-gpc (22:6)
## [2] 1,5-anhydroglucitol (1,5-ag)--indolepropionate
## [3] 1,5-anhydroglucitol (1,5-ag)--n4-acetylcytidine
## [4] 1,5-anhydroglucitol (1,5-ag)--pantothenate (vitamin b5)
## [5] 1,5-anhydroglucitol (1,5-ag)--prolylhydroxyproline
## [6] 1,5-anhydroglucitol (1,5-ag)--taurocholate
## [7] 1,5-anhydroglucitol (1,5-ag)--x - 10346
## [8] 1,5-anhydroglucitol (1,5-ag)--x - 18920
## + ... omitted several edges

II. The Probability Diffusion Algorithm

II.I From a starting node.

Run the following code, then go to the directory, and open all diffusionP1Movie*.png files all at once (sorted
by timestamp). Starting from the first image, view how the probability diffusion algorithm works to diffuse
100% probability to the rest of the graph. Be sure to pay attention to the recursion level listed in the title of
each image, to imagine where in the call stack the algorithm is at the captured time the image was generated.

if (!dir.exists(sprintf("%s/images", getwd()))) {
    dir.create(sprintf("%s/images", getwd()))
}
G=vector(mode="list", length=length(V(ig)$name))
G[1:length(G)] = 0
names(G) = c("A", "B", "C", "D", "E", "F", "G", "H", "I")
startNode = "A"
visitedNodes = startNode
coords = layout.fruchterman.reingold(ig)
# Uncomment next lines if you want to generate movie images
#G_new = graph.diffuseP1(p1=1.0, sn=startNode, G=G, vNodes=visitedNodes,
II.II From a starting node, after visiting previous nodes.

Now, delete all diffusionP1Movie*.png files from your current directory, and run the following code. View the new image stack in the same way we did previously.

```r
# Now let's see how the probability diffusion algorithm diffuses probability
# after B has been "stepped" into.
visitedNodes = c("A", "B")
startNode = "B"
# Uncomment the next lines if you want to generate movie images
#G_new = graph.diffuseP1(p1=1.0, sn=startNode, G, vNodes=visitedNodes,
# thresholdDiff=0.01, adj_mat, TRUE,
# out_dir = sprintf("%s/images", getwd()),
# r_level = 1, coords = coords)
# Inherited probabilities across all nodes should add to 1.
#sum(unlist(G_new))
# Which node inherited the highest probability from startNode?
#G_new[which.max(G_new)]
```

II.III Diffusing through visited nodes, based on connectivity.

Sometimes the startNode is “stranded” by a bunch of visited nodes. The diffusion algorithm diffuses “through” visited nodes, so that nodes in the same connected component can be prioritized over nodes in a different connected component, or “island nodes” (e.g. in the below code snippet, “I” is an island node). This only works currently for nodes 2 hops away from the current startNode, however.

```r
adj_mat = rbind(c(0,1,2,0,0,0,0,0,0), #A's neighbors
c(1,0,3,0,0,0,0,0,0), #B's neighbors
c(2,3,0,1,0,0,0,0,0), #C's neighbors
c(0,0,0,0,0,0,1,1,0), #D's neighbors
c(0,0,1,0,1,0,0,0,0), #E's neighbors
c(0,0,0,1,0,0,0,0,0), #F's neighbors
c(0,0,0,1,0,0,0,1,0), #G's neighbors
c(0,0,0,1,0,0,1,0,0), #H's neighbors
c(0,0,0,0,0,0,0,0,0) #I's neighbors
)
colnames(adj_mat) = c("A", "B", "C", "D", "E", "F", "G", "H", "I")
# Convert adjacency matrices to igraph objects for all three graphs.
ig = graph.adjacency(adj_mat, mode="undirected", weighted=TRUE,
add.colnames = "name")
coords = layout.fruchterman.reingold(ig)
print(ig)
```
# Now let's see how the probability diffusion algorithm diffuses probability
# after B has been "stepped" into "C" and then "A". As you can see, startNode
# "A" is surrounded by visited nodes "B" and "C". It needs to be smart enough
# to weigh "E" and "F" before "D", "H", "G" and "I".
visitedNodes = c("B", "C", "A")
startNode = "A"
# Uncomment the next lines if you want to generate movie images
#G_new = graph.diffuseP1(p1=1.0, sn=startNode, G, vNodes=visitedNodes,
#                           thresholdDiff=0.01, adj_mat, TRUE,
#                           out_dir = sprintf("%s/images", getwd()),
#                           1, coords)
# Inherited probabilities across all nodes should add to 1.
#sum(unlist(G_new))
# Which node inherited the highest probability from startNode?
#G_new[which.max(G_new)]

III. The Network Encoding Algorithms

III.I Multi-Node Diffusion Encoding

# The multi-node network walker tends to overfit on the network and is more
# computationally intensive/slow compared to the single-node network walker.
# It is therefore not recommended that you use this network walker over the
# single-node network walker. However, it is unclear if this network walker
# can be beneficial in some circumstances or application areas. We include
# it as an experimental feature only.
adj_mat = rbind(c(0,3,1,0,0,0,0,0,0),
               c(3,0,2,2,0,0,0,0,0),
               c(1,2,0,0,2,0,0,0,0),
               c(0,2,0,0,1,0,1,1,0),
               c(0,0,2,1,0,2,0,2,0),
               c(0,0,0,0,2,0,0,0,0),
               c(0,0,0,1,0,0,0,1,0),
               c(0,0,0,1,2,0,1,0,1),
               c(0,0,0,0,0,0,0,1,0))
colnames(adj_mat) = c("A", "B", "C", "D", "E", "F", "G", "H", "I")
# Convert adjacency matrices to igrpah objects for all three graphs.
ig = graph.adjacency(adj_mat, mode="undirected", weighted=TRUE,
                      add.colnames = "name")
coords = layout.fruchterman.reingold(ig)
print(ig)
## + attr: name (v/c), weight (e/n)
## + edges from 9238347 (vertex names):

```r
# Uncomment following line to generate PNGs and animate the
# multi-node encoding node ranks.
#ranks = multiNode.getNodeRanks(S = c("A", "B"), G, p1=1.0,
# thresholdDiff=0.01, adj_mat,
# log2(length(G)), FALSE,
# out_dir = sprintf("%s/images", getwd()),
# TRUE, coords)
# Get node ranks as list object, with no images generated
ranks = multiNode.getNodeRanks(S = c("A", "B"), G, p1=1.0,
thresholdDiff=0.01, adj_mat,
log2(length(G)), FALSE)
```

### III.II Single-Node Diffusion Encoding

```r
# This network walker tends to avoid overfitting. Of further note, since
# single-node network walker is not parameterized by the subset being
# encoded, you can pre-compute node rankings using dynamic programming.
# Pre-computing node ranks enables quick encoding of thousands of subsets
# at a time (see The Encoding Process).
S = c("A", "B")
#out_dir=sprintf("%s/images",getwd())
# Uncomment the following lines to generate PNGs and animate the
# single-node encoding node ranks.
#ranks = list()
#for (n in seq_len(length(S))) {
#  ind = which(names(G)==S[n])
#  ranks[[n]]=singleNode.getNodeRanksN(ind,G,p1=1.0,thresholdDiff=0.01,
#  adj_mat,S=S,num.misses=log2(length(G)),
#  FALSE,out_dir,TRUE,coords)
#}
#names(ranks) = S
# Get node ranks as list object, with no images generated
S = c("A", "B")
ranks = list()
for (n in 1:length(S)) {
  ind = which(names(G)==S[n])
  ranks[[n]]=singleNode.getNodeRanksN(ind,G,p1=1.0,thresholdDiff=0.01,
  adj_mat,S=S,
  num.misses=log2(length(G)),FALSE)
}
names(ranks) = S
```

### IV. The Encoding Process

We’re going to go back to our data using the Arginase deficiency network model, and the Miller et al (2015) dataset. 

```r
## IV.0 Re-define the Arginase deficiency network model
```
print(ig_arg)

# IGRAPH 99fcc76 UNW- 196 996 --
# + attr: name (v/c), weight (e/n)
# + edges from 99fcc76 (vertex names):
# [1] 1,5-anhydroglucitol (1,5-ag)--1-docosahexaenoyl-gpc (22:6)
# [2] 1,5-anhydroglucitol (1,5-ag)--indolepropionate
# [3] 1,5-anhydroglucitol (1,5-ag)--n4-acetylcytidine
# [4] 1,5-anhydroglucitol (1,5-ag)--pantothenate (vitamin b5)
# [5] 1,5-anhydroglucitol (1,5-ag)--prolylhydroxyproline
# [6] 1,5-anhydroglucitol (1,5-ag)--taurocholate
# [7] 1,5-anhydroglucitol (1,5-ag)--x - 10346
# [8] 1,5-anhydroglucitol (1,5-ag)--x - 18920
# + ... omitted several edges

adj_mat = as.matrix(get.adjacency(ig_arg, attr="weight"))
G=vector(mode="list", length=length(V(ig_arg)$name))
G[1:length(G)] = 0
names(G) = V(ig_arg)$name

IV.I Choose your node subset.

# Maximum subset size to inspect
kmx=15
# Get our node subset associated with the $KMX highest perturbed (up or down)
# in our first Arginase deficiency sample.
S_arg = sort(abs(arg_data[,1]), decreasing=TRUE)[1:kmx]
print(S_arg)

# x - 11381 x - 13481
# 4.197347 3.510904
# 2-aminobutyrate dihydroorotate
# 3.122232 3.117244
# 2-arachidonoyl-gpc (20:4) 2-linoleoylglycerol (2-monolinolein)
# 2.942611 2.862771
# 1-docosahexaenoyl-gpc (22:6) x - 13478
# 2.754610 2.741754
# x - 12792 gamma-glutamyl-2-aminobutyrate
# 2.707036 2.661916
# 1-arachidonoyl-gpc (20:4) x - 17001
# 2.653048 2.586215
# x - 18914 prolylhydroxyproline
# 2.573498 2.537495
# 1-stearoyl-gpc (18:0)
# 2.246071

IV.II Get k node permutations.
# Get the single-node encoding node ranks starting from each node in the subset # S_arg.\nranks = list()
for (n in 1:length(S_arg)) {
    ind = which(names(G)==names(S_arg)[n])
    ranks[[n]]=singleNode.getNodeRanksN(ind,G,p1=1.0,thresholdDiff=0.01, adj_mat,S=names(S_arg), num.misses=log2(length(G)),TRUE)
}

## [1] "Node ranking 128 of 196."
## [1] "Node ranking 19 of 196."
## [1] "Node ranking 60 of 196."
## [1] "Node ranking 21 of 196."
## [1] "Node ranking 28 of 196."
## [1] "Node ranking 3 of 196."
## [1] "Node ranking 156 of 196."
## [1] "Node ranking 149 of 196."
## [1] "Node ranking 63 of 196."
## [1] "Node ranking 2 of 196."
## [1] "Node ranking 170 of 196."
## [1] "Node ranking 188 of 196."
## [1] "Node ranking 103 of 196."
## [1] "Node ranking 18 of 196."

names(ranks) = names(S_arg)

IV.III Convert to bitstrings.

# Get the bitstrings associated with the patient's perturbed metabolites in # "S_arg" based on the node ranks calculated in the previous step, "ranks".
ptBSbyK = mle.getPtBSbyK(names(S_arg), ranks)

IV.IV Get encoding length of minimum length codeword.

ind = which(colnames(arg_data) %in% names(diags))
data_mx.pvals=apply(arg_data[,ind], c(1,2),
    function(i) 2*pnorm(abs(i), lower.tail=FALSE))
ptID = "IEM_1006"
res = mle.getEncodingLength(ptBSbyK, t(data_mx.pvals), ptID, G)
ind.mx = which.max(res$d.score)
res[ind.mx,]

## patientID   optimalBS subsetSize opt.T
## 8  IEM_1006 TT00TTTT0TT  8   8
## 8 6.8e-03/1.8e-03/7.8e-03/1.8e-03/2.7e-05/4.5e-04/1.0e-02/1.1e-02     44.47243
IV.V Get probability of node subset.

```r
# This is the lower bound of the probability associated with the metabolites
# in S_arg. The higher the probability relative to a random set of the same
# size, the more tightly connected the metabolite set is.
2^-res[ind.mx, "IS.alt"]
```

## [1] 4.982462e-06

```r
# Note the probability printed above may seem low, but there are
# log2(length(G), kmx) outcomes that probability is assigned between.
# We should expect a probability for a node set of size kmx in a length(G)
# network to have probability:
2^-((log2(choose(length(G), kmx)))
```

## [1] 9.359992e-23

```r
# You'll notice the probability associated with the metabolite set we encoded,
# S_arg, is orders of magnitude higher than a uniform probability model. This
# implies the metabolites in S_arg are connected in the network ig_arg more
# than is expected by chance.
```

IV.V Get p-value of variable length encoding vs. fixed length encoding.

```r
# You can interpret the probability assigned to this metabolite set by
# comparing it to a null encoding algorithm, which uses fixed-length codes
# for all metabolites in the set. The "d.score" is the difference in bitlength
# between the null and alternative encoding models. Using the Algorithmic
# Significance theorem, we can estimate the upper bounds on a p-value by
# 2^-d.score.
2^-res[ind.mx, "d.score"]
```

## [1] 9.215059e-14

```r
# All metabolites in S_arg
names(S_arg)
```

## [1] "x - 11381"
## [2] "x - 13481"
## [3] "2-aminobutyrate"
## [4] "dihydroorotate"
## [5] "2-arachidonoyl-gpc (20:4)"
## [6] "2-linoleoylglycerol (2-monolinolein)"
V. Patient Distances, Single-Node Encoding Recommended
ranks=list()
for (n in 1:length(S_arg)) {
    ind=which(names(G)==S_arg[n])
    ranks[[n]]=getNodeRanksN(ind,G,p1=1.0,thresholdDiff=0.01,
                            adj_mat,S=S_arg,
                            num.misses=log2(length(G)),FALSE)
}

names(ranks)=S_arg

# Calculate patient bitstrings
ptBSbyK=list()
for (pt in 1:ncol(data_mx)) {
    ptID=colnames(data_mx)[pt]
    S_pt=names(sort(abs(data_mx[,pt]),decreasing=TRUE)[1:kmx])
    ptBSbyK[[ptID]]=getPtBSbyK(S_pt, ranks)
}

# Now perform mutual information-based patient similarity scoring
res = list()
t = list(ncd=matrix(NA, nrow=ncol(data_mx), ncol=ncol(data_mx)))
rownames(t$ncd) = colnames(data_mx)
colnames(t$ncd) = colnames(data_mx)
for (i in 1:kmx) { res[[i]] = t }
for (pt in 1:ncol(data_mx)) {
    ptID=colnames(data_mx)[pt]
    for (pt2 in pt:ncol(data_mx)) {
        ptID2=colnames(data_mx)[pt2]
        tmp = getPtDist(ptBSbyK[[ptID]],ptID,ptBSbyK[[ptID2]],ptID2,
                        data_mx,ranks,p1=1.0,thresholdDiff=0.01,adj_mat)
        for (k in 1:kmx) {
            res[[k]]$ncd[ptID, ptID2] = tmp$NCD[k]
            res[[k]]$ncd[ptID2, ptID] = tmp$NCD[k]
        }
    }
}

VI. Visualizations

# Multi-dimensional scaling
plot.mdsDist = function(patientDist, diagnoses, diag) {
    if (is.null(diagnoses)) {
        print("To view patient clusters, please provide clinical labels.")
        return(0)
    }
    fitDist = cmdscale(patientDist, eig=FALSE, k=2)
    x = round(fitDist[,1], 2)
    y = round(fitDist[,2], 2)
    df=data.frame(x=x,y=y,color=as.factor(diagnoses),label=colnames(patientDist))
    p=ggplot(df,aes(x,y,label=label))+geom_point(aes(colour=color),size=5)+
    geom_label(nudge_x = 0.1)
    return(p)
# K-nearest neighbors

```r
plot.knnDist = function(patientDist, diagnoses, diag) {
  diagnoses = diagnoses[colnames(patientDist)]
  # Add a GREEN edge between patient nodes if k nearest neighbor is
  # correct diagnosis (either TP or TN)
  # Add a RED edge between patient nodes if k nearest neighbor is
  # incorrect diagnosis (either FP or FN)
  tp = 0
  fp = 0
  tn = 0
  fn = 0
  ig = make_empty_graph(n=ncol(patientDist), directed=TRUE)
  V(ig)$name = colnames(patientDist)
  for (pt1 in 1:length(diagnoses)) {
    diag_pt1 = diagnoses[pt1]
    ind = sort(patientDist[pt1,-pt1], decreasing = FALSE)
    ind = ind[which(ind==min(ind))]
    diag_pt_ind = diagnoses[which(names(diagnoses) %in% names(ind))]
    if (any(diag_pt_ind==diag) && diag_pt1==diag) { # True positive
      tp=tp + 1
      ind=ind[which(diag_pt_ind==diag)]
      ig=add.edges(ig,
        edges=c(colnames(patientDist)[pt1],names(ind[1])),
        attr=list(color="green", lty=1))
    } else if (diag_pt_ind!=diag && diag_pt1!=diag) { # True negative
      tn=tn + 1
      ig=add.edges(ig,
        edges=c(colnames(patientDist)[pt1],names(ind[1])),
        attr=list(color="green", lty=1))
    } else if (diag_pt_ind==diag && diag_pt1!=diag) { # False positive
      fp=fp + 1
      ig=add.edges(ig,
        edges=c(colnames(patientDist)[pt1],names(ind[1])),
        attr=list(color="red", lty=1))
    } else { # False negative
      fn=fn + 1
      ig=add.edges(ig,
        edges=c(colnames(patientDist)[pt1],names(ind[1])),
        attr=list(color="red", lty=1))
    }
  }
  print(sprintf("Tp = %d, Tn= %d, Fp = %d, Fn=%d", tp, tn, fp, fn))
  sens = tp / (tp+fn)
  spec = tn / (tn+fp)
  print(sprintf("Sens = %.2f, Spec= %.2f", sens, spec))
  V(ig)$label = rep("", length(V(ig)$name))
  return(ig)
}
```

# If you have diagnostic labels associated with the colnames(data_mx),
# send them using diagnoses parameter

res_ncd = lapply(res, function(i) i$ncd)
ncd = mle.getMinPtDistance(res_ncd)

dd = colnames(data_mx)

dd[which(dd %in% names(diags)[which(diags=="Argininemia")])]) = "ARG"

dd[which(dd %in% names(diags)[which(diags!="Argininemia")])]) = "negCntl"

names(dd) = colnames(res[[1]]$ncd)

colnames(ncd)=colnames(res[[1]]$ncd)

rownames(ncd)=colnames(res[[1]]$ncd)

p=plot.mdsDist(ncd, dd, NULL)

p

## Hierarchical clustering

dd_f = as.numeric(as.factor(as.character(dd)))

heatmap.2(x=ncd,dendrogram="both", Rowv=TRUE,Colv=TRUE,
        ColSideColors=c(brewer.pal(12,"Set3"),brewer.pal(9,"BrBG"))[dd_f],
        RowSideColors=c(brewer.pal(12,"Set3"),brewer.pal(9,"BrBG"))[dd_f],
        cexRow=0.75,cexCol=1, margins=c(12,12), trace="none", key=TRUE,
        col=bluered, notecol="black")

legend("left", legend=unique(sort(as.character(dd))),
        fill=c(brewer.pal(12,"Set3"), brewer.pal(9,"BrBG")), cex=2)
# K-NN

ig=plot.knnDist(ncd, dd, diag="ARG")

## [1] "Tp = 4, Tn= 1, Fp = 3, Fn=0"
## [1] "Sens = 1.00, Spec= 0.25"

grps=list()
grps[[1]]=names(dd)[which(dd=="ARG")]
grps[[2]]=names(dd)[which(dd!="ARG")]
names(grps)=names(table(dd))
plot.igraph(ig, mark.groups=grps, mark.col=c("white", "black"),
            layout=layout.circle, edge.width=3, edge.arrow.size=0.5)