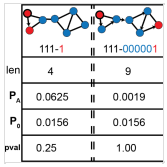


Version 4

Dec 11, 2020

# Reproducibility for Thistlethwaite et al. 2020 V.4

Lillian R Thistlethwaite[1], xiqi.li2 [1], Varduhipetrosyan [1]

[1]Baylor College of Medicine

1 *Works for me* dx.doi.org/10.17504/protocols.io.bpdvmi66

Lillian Thistlethwaite
Baylor College of Medicine

ABSTRACT

We consider the following algorithmic problem that arises in transcriptomics, metabolomics and other fields: given a weighted graph and a subset of its nodes, find subsets of perturbed variables that show significant connectedness within the graph. While solutions to this problem may be discovered by devising a variety of scoring functions, statistically rigorous discovery is obstructed by the high computational cost of permutation testing, required to establish p-values. In this work, we develop CTD, a novel information-theoretic method that evaluates the connectedness of perturbation signatures in a network context and assigns an upper bounds on their p-values without use of permutation testing. We apply CTD to interpret multi-metabolite perturbations due to inborn errors of metabolism and multi-gene perturbations associated with breast cancer in the context of disease-specific Gaussian Markov Random Field networks learned directly from respective molecular profiling data.

THIS PROTOCOL ACCOMPANIES THE FOLLOWING PUBLICATION

Thistlethwaite L.R., Petrosyan V., Li X., Miller M.J., Elsea S.H., Milosavljevic A. CTD: an information-theoretic method to interpret multivariate perturbations in the context of graphical models with applications in metabolomics and transcriptomics. Manuscript in review.

DOI

dx.doi.org/10.17504/protocols.io.bpdvmi66

PROTOCOL CITATION

Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan 2020. Reproducibility for Thistlethwaite et al. 2020. **protocols.io** https://dx.doi.org/10.17504/protocols.io.bpdvmi66 Version created by Lillian Thistlethwaite

MANUSCRIPT CITATION please remember to cite the following publication along with this protocol

Thistlethwaite L.R., Petrosyan V., Li X., Miller M.J., Elsea S.H., Milosavljevic A. CTD: an information-theoretic method to interpret multivariate perturbations in the context of graphical models with applications in metabolomics and transcriptomics. Manuscript in review.

KEYWORDS

CTD, network analysis, interpretation, metabolomics, transcriptomics, feature selection



CREATED

Nov 03, 2020

LAST MODIFIED

Dec 11, 2020

PROTOCOL INTEGER ID

44181

GUIDELINES

R programming language

MATERIALS TEXT

MATERIALS

NONE **Contributed by**

**users Catalog #N/A**

This is a computational workflow protocol, used to help reproduce results published in the publication listed below. Data used are from Miller et al. (2015). See Miller et al (2015) for details about data collection and acquisition.

Thistlethwaite, Li, Petrosyan, Miller, Elsea & Milosavljevic. (2020). CTD: an information-theoretic method to interpret multivariate perturbations in the context of graphical models with applications in metabolomics and transcriptomics. Plos Computational Biology. In revision.

Miller et al. (2015). Untargeted metabolomic analysis for the clinical screening of inborn errors of metabolism. Journal of Inherited Metabolic Disease, 38(6):1029-1039.

SAFETY WARNINGS

None

DISCLAIMER:

> This is a computational method, implemented as an R package.

ABSTRACT

We consider the following algorithmic problem that arises in transcriptomics, metabolomics and other fields: given a weighted graph and a subset of its nodes, find subsets of perturbed variables that show significant connectedness within the graph. While solutions to this problem may be discovered by devising a variety of scoring functions, statistically rigorous discovery is obstructed by the high computational cost of permutation testing, required to establish p-values. In this work, we develop CTD, a novel information-theoretic method that evaluates the connectedness of perturbation signatures in a network context and assigns an upper bounds on their p-values without use of permutation testing. We apply CTD to interpret multi-metabolite perturbations due to inborn errors of metabolism and multi-gene perturbations associated with breast cancer in the context of disease-specific Gaussian Markov Random Field networks learned directly from respective molecular profiling data.

BEFORE STARTING

R package dependencies:

R (>= 4.0)
gmp
igraph
stats
grDevices
graphics

Prepare dataset 1s

1 The CTD R package version 1.0.0 was used. 1s



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

Load the CTD R package

```
require(CTD)
```

Load the Miller et al 2015 dataset

```
data(Miller2015)
```

Remove metabolite annotation columns from dataset

```
data_mx.og = as.matrix(Miller2015[,grep("IEM_", colnames(Miller2015))])
```

One sample per column, one metabolite per row.

Create diagnosis-patient mappings

```
cohorts = list()
diags = data_mx.og[1,]
cohorts$mcc = names(diags[which(diags=="3-methylcrotonyl CoA carboxylase")])
cohorts$arg = names(diags[which(diags=="Argininemia")])
cohorts$cit = names(diags[which(diags=="Citrullinemia")])
cohorts$cob = names(diags[which(diags=="Cobalamin biosynthesis")])
cohorts$ga = names(diags[which(diags=="Glutaric Aciduria")])
cohorts$gamt = names(diags[which(diags=="Guanidinoacetate methyltransferase")])
cohorts$msud = names(diags[which(diags=="Maple syrup urine disease")])
cohorts$mma = names(diags[which(diags=="Methylmalonic aciduria")])
cohorts$otc = names(diags[which(diags=="Ornithine transcarbamoylase")])
cohorts$pa = names(diags[which(diags=="Propionic aciduria")])
cohorts$pku = names(diags[which(diags=="Phenylketonuria")])
cohorts$tmhle = names(diags[which(diags=="Trimethyllysine hydroxylase epsilon")])
cohorts$ref = names(diags[which(diags=="No biochemical genetic diagnosis")])
```

Create a list object that maps patient identifiers to their respective diagnostic class.

Remove diagnosis row and x-compounds from data_mx.og

```
data_mx.og = data_mx.og[-c(1, grep("x -", rownames(data_mx.og))),]
```

Remove diagnosis row from data_mx.og



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

Convert data_mx.og to numeric matrix.

```
data_mx.og = apply(data_mx.og, c(1,2), as.numeric)
```

All elements should be numeric, not character.

Fig 1: Plot individual metabolomics profiles onto biochemical pathway maps. 1m

2 The CTDext R package version 1.0.0 was used. The CTDext R package holds many pre-computed files necessary to reproduce some results from Thistlethwaite et al. 2020, as well as includes some additional pathway visualization and plotting features. 1m

The CTDext R package can be downloaded using the devtools::install_github() function as follows: install_github("BRL-BCM/CTDext")

Load the CTDext R package

```
require(CTDext)
```

Create an output directory

```
dir.create("./pathwayVis", showWarnings = FALSE)
```

Return a list of pathway maps curated by Metabolon's Metabolync.

```
pathway.ListMaps_metabolon()
```

Return a list of pathway maps curated by Metabolon's Metabolync.

Generate pathway map with patient perturbation data superimposed on "all Pathway" map

```
plot.pathwayMap("allPathways", "IEM_1023", data_mx.og[,"IEM_1023"], 2, 1,
out.path="./pathwayVis", SVG = FALSE)
# Display pathway map
require(png)
require(grid)
img <- readPNG('./pathwayVis/allPathways-IEM_1023.png')
options(repr.plot.width=15, repr.plot.height=15)
grid::grid.raster(img)
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

Tuning parameters used in this analysis

3 Before we start the analysis, we need to discuss some tuning parameters.

**Tuning parameters (defaults)**

- **p0:**
  the probability uniformly distributed to all metabolites. default is 0.1.
- **p1:**
  the probability distributed preferentially based on edge weights and connectedness patterns in a network, G.default is 0.9.
- **thresholdDiff:**
  the probability at which the probability diffusion algorithm truncates. default = 0.01.
- **kmx:**
  the number of top-perturbed (up or down) metabolites considered in the network-based interpretation. default in this analysis = 15.

Figure 3: Sensitivity and Specificity of CTD between 3 Network Learning Paradigms 15m 20s

4 **How to Replicate Figure 3**

In this result, we calculate the ROC-AUCs for all 5 CTD-defined diagnostic models. We also plot CTD scores across diagnostic categories as barplots to show the sensitivity and specificity of the models under three different network learning paradigms:

**i)** latent embedding + network pruning ("ind")
**ii)** latent embedding + no network pruning ("noPruning")
**iii)** no latent embedding or network pruning ("noLatent")

4.1 15m The computational work in this step is required for downstream steps. In this analysis step, we estimate the probability and significance of patient-specific metabolite perturbations against different disease-specific network contexts using CTD.

The CTDext package provides precomputed node rankings derived from all 5 disease networks in Thistlethwaite et al. 2020 in order to save time.

- You can find out **how to precompute node rankings** under a different Protocol: Precompute Node Rankings (DOI: dx.doi.org/10.17504/protocols.io.bkecktaw).
- You can find out **how to learn partial correlation disease-specific networks** under a different Protocol: Learn Partial Correlation Disease-Specific Networks (DOI: dx.doi.org/10.17504/protocols.io.bk7xkzpn).

R package versioning:

- CTD v 1.0.0
- CTDext v 1.0.0
- R.utils v 2.9.2
- pROC v 1.16.2
- ggplot2 v 3.3.2
- gridExtra v 2.3.0

Single-node encoding

```
require(R.utils)
p0=0.1
p1=0.9
thresholdDiff=0.01
kmx=15
p=list()
for (type in c("ind", "noPruning", "noLatent")) {
  dir.create(sprintf("./loocv/loocv_%s_runCTD/",type), recursive = TRUE, showW
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
 for (model in c("cit", "msud", "mma", "pa", "pku")) {
  for (fold in 1:length(which(cohorts[[model]] %in% colnames(Miller2015)))) {
    # load corresponding networks.
    if (type=="noPruning") {
      ig = loadToEnv(system.file(sprintf('networks/ind_foldNets/bg_%s_fold%s.R
package='CTDext'))[['ig']]
      # latent embedding + no network pruning
    } else if (type=="noLatent") {
      ig =
loadToEnv(system.file(sprintf('networks/noLatent_foldNets/bg_%s_noLatent_fo
package='CTDext'))[['ig']]
      # no latent embedding + no network pruning
    } else {
      # latent embedding + network pruning
      ig = loadToEnv(system.file(sprintf('networks/ind_foldNets/bg_%s_fold%s.R
package='CTDext'))[['ig_pruned']]
    }
    adj_mat = as.matrix(get.adjacency(ig, attr="weight"))
    G = vector(mode="list", length=length(V(ig)$name))
    names(G) = V(ig)$name

    # load precomputed node ranks that were derived from the loaded graph
    ranks = loadToEnv(system.file(sprintf("ranks/%s_ranks/%s%s-ranks.RData"
fold), package='CTDext'))[["permutationByStartNode"]]
    ranks = lapply(ranks, function(i) tolower(i))

    # p.value matrix derived from z-score
    data_mx = data_mx.og[which(rownames(data_mx.og) %in% V(ig)$name), ]
    data_mx = data_mx[,which(colnames(data_mx) %in% unlist(cohorts))] # cho
include
    data.pvals = apply(data_mx, c(1,2), function(i) 2*pnorm(abs(i), lower.tail =
    data.pvals = t(data.pvals)

    df = data.frame(ptID=character(), S=character(), lenS=numeric(), optT=nu
                    fishers=numeric(), l0=numeric(), lA=numeric(), d=numeric(), stri
    r=1
    ptBSbyK = list()
    for (p in 1:ncol(data_mx)) {
      ptID = colnames(data_mx)[p]
      print(sprintf("Model: %s-fold%d, Type: %s. Patient %d/%d...", model, fold,
ncol(data_mx)))
      if (ptID %in% unlist(cohorts)) {
        diag = names(cohorts)[which(unlist(lapply(cohorts, function(i) ptID %in%
        # using single-node diffusion
        S = data_mx[order(abs(data_mx[,p]), decreasing = TRUE),p][1:kmx]
        ptBSbyK = mle.getPtBSbyK(names(S), ranks, num.misses = log2(length(
        res = mle.getEncodingLength(ptBSbyK, data.pvals, ptID, G)
        for (k in 1:kmx) {
          df[r, "ptID"] = colnames(data_mx)[p]
          df[r, "diag"] = diag # diagnosis
          df[r, "S"] = paste(names(S)[1:k], collapse="/") # node names (metaboli
          df[r, "lenS"] = k # length of diffussion path
          df[r, "optT"] = res[k, "opt.T"]
          df[r, "fishers"] = res[k, "fishers.Info"]
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
        df[r, "l0"] = res[k, "ls.null"]
        df[r, "lA"] = res[k, "ls.alt"]
        df[r, "d"] = res[k, "d.score"]
        r = r + 1
       }
      }
     }
     save(ptBSbyK, df, file=sprintf("./loocv/loocv_%s_runCTD/model_%s_%s_fold
type, model, type, fold, kmx))
   }
  }
}
```

The following cell expanded the diffusion-based encoding followed by decoding process to all samples in data_mx.og for 5 disease models and 3 types of networks. The output dataframes will be saved as RData files. This processmay take a few minutes.

4.2 Now, we can visualize the results that generates Figure 3. 20s

Calculate the ROC-AUCs.

```
require(pROC)
require(ggplot2)
kmx=15
p2=list()
for (model in c("cit", "msud", "mma", "pa", "pku")) {
 for (type in c("ind", "noPruning", "noLatent")) {
  df_all = data.frame(fold=numeric(), pt=numeric(), bits=numeric(),
diag=character(), stringsAsFactors = FALSE)
  for (fold in 1:length(which(cohorts[[model]] %in% colnames(Miller2015))))
{

load(sprintf("./loocv/loocv_%s_runCTD/model_%s_%s_fold%d_kmx%d.RData",
type, model, type, fold, kmx))
    pts = unique(df$ptID)
    df = df[which(df$ptID %in% pts),]
    df_best = data.frame(pt=numeric(), ptID=character(), bits=numeric(),
diag=character(), stringsAsFactors = FALSE)
    for (pt in 1:length(pts)) {
      pt_data = df[which(df$ptID==pts[pt]),]
      ptID = unique(df[which(df$ptID==pts[pt]), "ptID"])
      if (pt_data[1,"diag"]==model) {
       df_best[pt, "pt"] = which(cohorts[[model]]==ptID)
      }
      df_best[pt, "ptID"] = ptID
      df_best[pt, "bits"] = max(df[which(df$ptID==pts[pt]), "d"])-
log2(nrow(pt_data)) # p adjust for kmx
      df_best[pt, "diag"] = unique(pt_data[,"diag"])
    }
    df_best$bits[which(df_best$bits<0)] = 0
    df_best$fold = rep(fold, nrow(df_best))
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
    df_all = rbind(df_all, df_best)
   }
   df_all = df_all[which(df_all$ptID %in% colnames(Miller2015)),]
   df_all$bits = -log2(p.adjust(2^-(df_all$bits), method="fdr"))# p adjust for
samples number

   # Visualize LOOCV signal and compare to off-target diseased test patients
   dff = df_all
   dff$loocv = rep(0, nrow(dff))
   dff$loocv[which(dff$pt==dff$fold)] = 1
   dff = dff[-intersect(which(dff$loocv==0), which(dff$diag==model)), ]
   b_bits = cbind(unique(dff$ptID), sapply(unique(dff$ptID), function(i)
mean(dff[which(dff$ptID==i),"bits"])))
   dff = dff[-which(duplicated(dff$ptID)),]
   dff$diag[which(dff$diag=="ref")] = "z.ref"
   dff = dff[order(dff$ptID),]
   b_bits = b_bits[order(b_bits[,1]),]
   dff$bits = as.numeric(b_bits[,2])
   save(dff,file =
sprintf("./loocv/loocv_%s_runCTD/best_bits_%s_%s_loocv.RData", type, model,
type))
   # Get diagnostic labels
   d = dff$diag
   d[which(d!=model)] = 0
   d[which(d==model)] = 1
   d = as.numeric(d)
   auc = roc(d, dff$bits,quiet = TRUE)

   p2[[type]][[model]] = ggplot(dff, aes(x=diag, y=bits, fill=diag)) +
     geom_boxplot(size=0.5) +
     geom_hline(yintercept=-log2(0.05)) +
     theme(text = element_text(size=15),
          axis.title.x = element_blank(),
          axis.text.x = element_text(angle = 30, hjust = 1,size = 15),
          legend.position = "none") +
     labs(title=sprintf("%s (%s)", model, type),subtitle =
sprintf("AUC=%.2f",auc$auc))

   print(sprintf("AUC (Model: %s Type: %s) = %.3f", toupper(model), type,
auc$auc))
  }
}
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

Barplots of CTD signal across disease cohorts and network contexts.

```
# Visualize barplots and AUC
require(gridExtra)
options(repr.plot.width=15, repr.plot.height=15)
g=unlist(p2,recursive = FALSE)
print(grid.arrange(g[[1]],g[[6]],g[[11]],
             g[[2]],g[[7]],g[[12]],
             g[[3]],g[[8]],g[[13]],
             g[[4]],g[[9]],g[[14]],
             g[[5]],g[[10]],g[[15]],
             ncol = 3))
```

This code chunk generates Figure 3. In the following cell, you should see that patients showed strong significance when interpreted agaisnt the correct disase-specific network and little to no significance when interpreted with incorrect disease-specific networks. Latent variable embedding is associated with higher model sensetivity, whereas network pruning is associated with higher model specificity.

Figure S1: Highly connected patient-specific modules called by CTD. 1m 33s

## 5 How to Replicate Figure S1

1m 33s

In this result, we interpreted disease signatures from selected **individual** patients representing each disease category. We also visualize selected patient's most connected modules in the relevant disease-specific network context.

R package versioning:

- CTD v 1.0.0
- CTDext v 1.0.0
- R.utils v 2.9.2

Estimate probabilities of selected individual patient's metabolite perturbations in disease contexts.

```
# Create an output directory
dir.create("./loocv/blowouts", recursive = TRUE, showWarnings = FALSE)
require(R.utils)
# set global variables
p0 = 0.1
p1 = 0.9
thresholdDiff = 0.01
kmx=15 # Setting kmx=15 will select top 15 perturbed metabolites
ptIDs = c("IEM_1017","IEM_1058","IEM_1051","IEM_1093","IEM_1105")
p.igraph=list()
for (ptID in ptIDs){
  getDiag=sapply(cohorts,function(x) which(x==ptID))
  model=names(getDiag[sapply(getDiag,length)>0])
  fold=getDiag[sapply(getDiag,length)>0]
  # load latent-embedding, pruned network that is learnt from the rest of the patie
diagnosed with the same disease.
  ig =
loadToEnv(system.file(sprintf('networks/ind_foldNets/bg_%s_fold%s.RData',model,fo
package='CTDext'))[['ig_pruned']]
  # get "ig" derived adjacency matrix
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
  G = vector(mode="list", length=length(V(ig)$name))
  names(G) = V(ig)$name
  adj_mat = as.matrix(get.adjacency(ig, attr="weight"))
  data_mx = data_mx.og[which(rownames(data_mx.og) %in% V(ig)$name), ]
  # p.value derived from z-score
  data.pvals = sapply(data_mx[,ptID], function(i) 2*pnorm(abs(i), lower.tail = FALSE)
  data.pvals = t(data.pvals)
  rownames(data.pvals)=ptID

  # using single-node diffusion
  S = data_mx[order(abs(data_mx[,ptID]), decreasing = TRUE),ptID][1:kmx] # top km
perturbed metabolites in ptID's profile
  print(sprintf("%s: Single-node ranking...",ptID))
  ranks = list()
  for (i in 1:length(S)) {
    ind = which(names(G)==names(S)[i])
    ranks[[i]] = singleNode.getNodeRanksN(ind, G, p1, thresholdDiff, adj_mat, names
num.misses = log2(length(G))) # get node ranks
  }
  names(ranks) = names(S)
  ptBSbyK = mle.getPtBSbyK(names(S), ranks) # encode nodes
  res = mle.getEncodingLength(ptBSbyK, data.pvals, ptID, G) # get encoding length
  mets = unique(c(names(S), names(ptBSbyK[[which.max(res[,"d.score"])]]))) # bes
co-perturbed metabolite set is the most compressed subset of nodes
  p.mets=2^-(res[which.max(res[,"d.score"]),"d.score"]-log2(nrow(res))) # p value
this "modular perturbation"
  print(mets)
  print(p.mets)

  # generate igraph for disease-relevant metabolites of the selected patient
  e = delete.vertices(ig, v=V(ig)$name[-which(V(ig)$name %in% mets)])
  reds = intersect(V(e)$name[which(V(e)$name %in% names(S))],
names(S[which(S>0)]))
  blues = intersect(V(e)$name[which(V(e)$name %in% names(S))],
names(S[which(S<0)]))
  V(e)$color = rep("", length(V(e)$name))
  V(e)$color[which(V(e)$name %in% reds)] = "red" # red indicates positive z-score
  V(e)$color[which(V(e)$name %in% blues)] = "light blue" # blue indicates negative
score
  V(e)$color[-which(V(e)$name %in% names(S))] = "grey" # grey verteces are high
connect to "mets"
  cc = cluster_walktrap(e) #find densely connected subgraphs, also called
communities in a graph via random walks.
  weights = ifelse(crossing(cc, e), 1, 5)
  layout = layout_with_fr(e, weights=weights)
  p.igraph[[ptID]]=list(model=model,e=e,layout=layout)
}
```

Now, we are ready to estimate probabilities of individual patient's metabolite perturbations in disease contexts. We choose to highlight one patient per diagnostic category:

Citrullinemia: Patient IEM_1017
Maple syrup urine disease: IEM_1058
Methylmalonic aciduria: IEM_1051
Propionic aciduria: IEM_1093



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

Phenylketonuria: IEM_1105

Plot the igraph object for the most connected metabolite perturbations of the selected patients

```
options(repr.plot.width=15, repr.plot.height=15)
par(mfrow = c(2,2))
for (ptID in ptIDs){
  p=p.igraph[[ptID]]
  png(sprintf("./loocv/blowouts/%s_%s_module.png", p$model, ptID))
  plot.igraph(p$e, layout=p$layout,
edge.width=50*abs(E(e)$weight),vertex.label.cex=2,vertex.label.color="black",ma
%s",toupper(p$model),ptID))
  dev.off()
  plot.igraph(p$e, layout=p$layout,
edge.width=50*abs(E(e)$weight),vertex.label.cex=2,vertex.label.color="black")
  title(main=sprintf("%s: %s",toupper(p$model),ptID),cex.main=3)
}
```

Power analysis 15h 15m

## 6 How to Replicate Figure 4.

In this result, we performed a power analysis on three simulated networks of size 50, with varying levels of connectedness.

Network 1: 10% connected
Network 2: 30% connected
Network 3: 60% connected

The simulation is broken up into 4 steps:
1. Build networks.
2. Get number of edges connecting subsets of size 5.
3. Use CTD to get subset probabilities by enumerating over all >2 million outcomes of choose(50, 5). This enumeration will allow you to establish a ground truth p-values by determining the distribution of probabilities assigned by CTD over all subset outcomes of size 5 in each of the three 50 node networks.
4. For 2,500 node subsets, select equally amongst the number of edges between node subsets to assure that you get a variable level of connected subsets. Then, draw 2,000 permutations to establish a permutation-based p-value estimate.

R package versioning:
- CTD v 1.0.0
- CTDext v 1.0.0
- igraph v 1.2.5
- R.utils v 2.9.2
- gmp v 0.6.0
- ggplot2 v 3.3.2

### 6.1 Build simulation networks: Build 3 undirected networks with 50 nodes and assign edge weights of $1^{1s}$ with a varying level of connectedness between them.



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

Build simulation networks

```
net1 = matrix(0, nrow=50, ncol=50)
net2 = matrix(0, nrow=50, ncol=50)
net3 = matrix(0, nrow=50, ncol=50)

net1[sample(1:nrow(net1), 0.20*nrow(net1), replace = FALSE),
sample(1:nrow(net1), 0.20*nrow(net1), replace = FALSE)] = 1
net2[sample(1:nrow(net2), 0.40*nrow(net2), replace = FALSE),
sample(1:nrow(net2), 0.40*nrow(net2), replace = FALSE)] = 1
net3[sample(1:nrow(net3), 0.60*nrow(net3), replace = FALSE),
sample(1:nrow(net3), 0.60*nrow(net3), replace = FALSE)] = 1

diag(net1) = 0
diag(net2) = 0
diag(net3) = 0

colnames(net1) = sprintf("%s", 1:ncol(net1))
colnames(net2) = sprintf("%s", 1:ncol(net2))
colnames(net3) = sprintf("%s", 1:ncol(net3))

ig_net1 = graph.adjacency(net1, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_net2 = graph.adjacency(net2, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_net3 = graph.adjacency(net3, mode="undirected", weighted=TRUE,
add.colnames = "name")

print("Level of connectedness achieved for each network:")
print(sprintf("Network 1 = %.2f", length(E(ig_net1)$weight)/(50*49/2)))
print(sprintf("Network 2 = %.2f", length(E(ig_net2)$weight)/(50*49/2)))
print(sprintf("Network 3 = %.2f", length(E(ig_net3)$weight)/(50*49/2)))

save.image("nets_setup.RData")
```

Since the simulation networks are generated by random edges, we provide the nets_setup.RData in the CTDext R package (an extension R package to the CTD R package) for perfect replication of this result.

Load nets_setup.RData from CTDext

```
load(system.file("networks/nets_setup.RData", package="CTDext"))
```

As an alternative to building your own networks as in the "Build simulation networks" script, you can load the networks we used in our simulation through the CTDext R package.

6.2 Establish ground truth: Count the number of modules of size 5 with parameter r number of edges between them. 10m



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

Note: We run 10,000 subsets at a time on a computing cluster, a form of parallelization, which significantly speeds up this step compared to enumerating using serial code. The parallel implementation is pasted below, following by a PBS job scheduler launcher script, and a wrapper script which automatically launches the PBS scripts.

Each job running 10,000 subsets took about 1-1.5 minutes, and 100 jobs were running at a given time. It took about 5-7 minutes to run all 212 jobs. Collating the job output .RData files into one .RData files took another 5 minutes.



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

Get number of edges for all subsets in each of the 3 networks

```
args = commandArgs(trailingOnly=TRUE)
itt = as.numeric(args[1])
print(itt)

start = 1+10000*(itt-1)
eend = start+9999

require(igraph)
load("nets_setup.RData")
subsets_k = combn(50, 5)
if (eend>ncol(subsets_k)) {
  eend = ncol(subsets_k)
}

dff_nets = data.frame(network=character(30000), it=numeric(30000),
num.edges = numeric(30000), stringsAsFactors = FALSE)
r = 1
for (it in start:eend) {
  if (it %% 1000 == 0) {
    print(it)
    save.image(sprintf("simulation_%d.RData", itt))
  }
  net1_subset_it = induced_subgraph(ig_net1, v=subsets_k[,it])
  net2_subset_it = induced_subgraph(ig_net2, v=subsets_k[,it])
  net3_subset_it = induced_subgraph(ig_net3, v=subsets_k[,it])

  dff_nets[r, "network"] = "net1"
  dff_nets[r, "it"] = it
  dff_nets[r, "num.edges"] = length(E(net1_subset_it))
  r = r + 1
  dff_nets[r, "network"] = "net2"
  dff_nets[r, "it"] = it
  dff_nets[r, "num.edges"] = length(E(net2_subset_it))
  r = r + 1
  dff_nets[r, "network"] = "net3"
  dff_nets[r, "it"] = it
  dff_nets[r, "num.edges"] = length(E(net3_subset_it))
  r = r + 1
}
save.image(sprintf("simulation_%d.RData", itt))
```

Using number of edges as a heuristic for connectedness (which CTD also estimates using network flow/diffusion) will allow us to draw equally from several different levels of connectedness when we go to estimate power. It's important to test the power of CTD at a full range of connectedness levels, because CTD is most powerful when estimating probabilities of highly connected subsets, and less powerful for sparsely connected subsets. See Figure 4 in Thistlethwaite et al (2020) for details.



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

PBS job launcher for num_edges enumeration.

```
# Request 1 processors on 1 node
#PBS -l nodes=1:ppn=1
#Request 1 hour of walltime
#PBS -l walltime=1:00:00
#Request that regular output and terminal output go to the same file
#PBS -j oe
#PBS -m abe
module load R/3.3
cd metabolomics/9thCommitteeMeeting/conservativeness
itt=${itt}
Rscript get_numedges.r $itt > num_edges:$itt.out
rm num_edges:$itt.out
```

Wrapper script for num_edges.pbs

```
totalN = ceiling(ncol(combn(50, 5))/10000)
for (n in 1:totalN) {
  str = sprintf("qsub -v it=%d num_edges.pbs", n)
  system(str, wait=FALSE)
  system("sleep 0.2")
}
```

You can launch this wrapper script like this:

Rscript wrapper_num_edges.r



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

Collate num_edge simulation results

```
require(igraph)
# collate enumerated subsets into one R object
subsets_k = combn(50, 5)
size_df = 3*ncol(subsets_k)
dfff_nets = data.frame(network=character(size_df), it=numeric(size_df),
num.edges = numeric(size_df), stringsAsFactors = FALSE)
num_jobs = ceiling(ncol(subsets_k)/10000)
for (n in 1:num_jobs) {
  print(n)
  load(sprintf("simulation_%d.RData", n))
  begin_ind = 1+(n-1)*30000
  end_ind = begin_ind+30000
  dfff_nets[begin_ind:end_ind,] = dff_nets
}
dff_net1 = dfff_nets[which(dfff_nets$network=="net1"),]
dff_net2 = dfff_nets[which(dfff_nets$network=="net2"),]
dff_net3 = dfff_nets[which(dfff_nets$network=="net3"),]
save.image(file="simulation_collated.RData")
```

Once all 212 PBS jobs have finished, this means all subsets of size 5 have been enumerated over. Now we can collate those 212 simulation_*.RData files into a single simulated_collated.RData file.

## 6.3 Run CTD on all possible node sets of size 5 for each network, collect p-values for each encoding algorithm (single-node).

21h

Note: We use the same approach to enumeration as we did in Step 6.2, where we launch individual jobs on a cluster that performs the CTD probability estimation for 10,000 subsets at a time. For over 2 million subsets of size 5 in a network of size 50, this amounted to 212 jobs per network. For 3 networks, this amounted to 212*3= 636 jobs.

Each job took a variable amount of time, depending on how connected the 10,000 subsets being processed were and how efficient the cluster node was at the time. Job execution times ranged from 32 minutes to ~6.5 hours, and 100 jobs were running at any given time. 636 jobs were launched in total (212 jobs for 3 networks), resulting in an approximate execution time of about ~21 hours.

Get the CTD probability for 10,000 subsets at a time.

```
args = commandArgs(trailingOnly=TRUE)
network = args[1]
chunk = as.numeric(args[2])

require(CTD)
require(R.utils)
load("nets_setup.RData")
p0=0.1
p1=0.9
thresholdDiff=0.01
kmx=k=5
size_df = 10000
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
subsets_k = combn(50, 5)

dff_res = data.frame(network=character(size_df),
encoder=character(size_df), it=numeric(size_df),
                idx.ig=character(size_df), d.score=numeric(size_df),
stringsAsFactors = FALSE)
r = 1

start_it = 1+size_df*(chunk-1)
end_it = start_it + (size_df-1)
if (end_it > ncol(subsets_k)) {
  end_it = ncol(subsets_k)
}
print(sprintf("Start.it = %d, End.it = %d", start_it, end_it))

require(gmp)
start.time = Sys.time()
for (it in start_it:end_it) {
  print(it)
  if (it %% 1000 == 0) { save.image(sprintf("sn_enumerate_%s_%d.RData",
network, chunk)) }
  sig.nodes = subsets_k[,it]

  # Single-node encoding
  if (network=="net1") {ig=ig_net1} else if(network=="net2"){ig=ig_net2}
else {ig=ig_net3}
  adj_mat = get.adjacency(ig, attr="weight")
  G = vector(mode="list", length=length(V(ig)$name))
  names(G) = V(ig)$name
  ranks = list()
  for (i in 1:length(sig.nodes)) {
    ind = which(names(G)==sig.nodes[i])
    ranks[[i]] = singleNode.getNodeRanksN(ind, G, p1, thresholdDiff, adj_mat,
S=sig.nodes, log2(length(G)))
  }
  names(ranks) = sig.nodes
  ptBSbyK = mle.getPtBSbyK(as.character(sig.nodes), ranks)
  res = mle.getEncodingLength(ptBSbyK, NULL, NULL, G)
  dff_res[r, "network"] = network
  dff_res[r, "encoder"] = "single-node"
  dff_res[r, "it"] = it
  dff_res[r, "idx.ig"] = paste(which(V(ig)$name %in% sig.nodes), sep="-",
collapse="-")
  dff_res[r, "d.score"] = res[k,"d.score"]
  r = r + 1
}
end.time = Sys.time()

print(sprintf("Elapsed time = %.2f seconds.", end.time - start.time))

save(dff_res, file=sprintf("sn_enumerate_%s_%d.RData", network, chunk))
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

PBS script for sn_enumerate_gt.r

```
# Request 1 processors on 1 node
#PBS -l nodes=1:ppn=1
#Request x number of hours of walltime
#PBS -l walltime=15:00:00
#Request that regular output and terminal output go to the same file
#PBS -j oe
#PBS -m abe
module load R/3.3
cd metabolomics/9thCommitteeMeeting/conservativeness

net=${net}
chunk=${chunk}
Rscript sn_enumerate_gt.r $net $chunk > sn_enum:$net-$chunk.out
rm sn_enum:$net-$chunk.out
```

Wrapper script for sn_enum.pbs

```
totalN = ceiling(ncol(combn(50, 5))/10000)
for (net in c("net1", "net2", "net3")) {
  for (chunk in 1:totalN) {
    str = sprintf("qsub -v net=%s,chunk=%d sn_enum.pbs", net, chunk)
    system(str, wait=FALSE)
    system("sleep 0.2")
  }
}
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

Collate sn_enumerate_*.RData files into one per network

```
# Collate Single-node enumerations
subsets_k = combn(50,5)
for (net in c("net1", "net2", "net3")) {
 dfff_res = data.frame(network=character(), encoder=character(),
it=numeric(),
                idx.ig=character(), d.score=numeric(), stringsAsFactors =
FALSE)
 ff = list.files("sn_enum", pattern=sprintf("sn_enumerate_%s", net))
 for (f in ff) {
  load(sprintf("sn_enum/%s", f))
  dfff_res = rbind(dfff_res, dff_res)
  print(dim(dfff_res))
 }
 dfff_res = dfff_res[-which(dfff_res$it==0),]
 dim(dfff_res)
 if (nrow(dfff_res)==ncol(subsets_k)) {
  save(dfff_res, file=sprintf("sn_enumerate_%s.RData", net))
 }
}
```

6.4 Get permutation-based pvalues for 2,500 node sets you calculated a CTD upper bounds estimate for. 5m

Permutation-based p-values, estimate CTD's power.

```
for (net in c("net1", "net2", "net3")) {
 load("simulation_collated.RData")
 if (net=="net1"){numedges_df=dff_net1} else if(net=="net2")
{numedges_df=dff_net2} else {numedges_df=dff_net3}
 # Sample 25000 outcomes, sample equally between the 11 num.edges
categories
 ind.ne0 = sample(which(numedges_df$num.edges==0), 250)
 ind.ne1 = sample(which(numedges_df$num.edges==1), 250)
 ind.ne2 = sample(which(numedges_df$num.edges==2), 250)
 ind.ne3 = sample(which(numedges_df$num.edges==3), 250)
 ind.ne4 = sample(which(numedges_df$num.edges==4), 250)
 ind.ne5 = sample(which(numedges_df$num.edges==5), 250)
 ind.ne6 = sample(which(numedges_df$num.edges==6), 250)
 ind.ne7 = sample(which(numedges_df$num.edges==7), 250)
 ind.ne8 = sample(which(numedges_df$num.edges==8), 250)
 if (sum(numedges_df$num.edges==9)>=250) {
  ind.ne9 = sample(which(numedges_df$num.edges==9), 250)
 }
 if (sum(numedges_df$num.edges==10)>=250) {
  ind.ne10 = sample(which(numedges_df$num.edges==10), 250)
 } else { ind.ne10 = c() }
 ind = c(ind.ne0, ind.ne1, ind.ne2, ind.ne3, ind.ne4, ind.ne5, ind.ne6,
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
ind.ne7, ind.ne8, ind.ne9, ind.ne10)

  # Figure 4a
  load(sprintf("sn_enumerate_%s.RData", net))
  dff_net_sim = data.frame(network=character(), encoder=character(),
it=numeric(), idx.ig=character(),
                           num.edges=numeric(), ctd.pval=numeric(),
perm.pval=numeric(), stringsAsFactors = FALSE)
  r = 1
  for (i in 1:length(ind)) {
    ii = ind[i]
    if (i%% 100==0) { print(i) }
    dff_net_sim[r, "network"] = net
    dff_net_sim[r, "encoder"] = "single-node"
    dff_net_sim[r, "it"] = dfff_res[ii, "it"]
    dff_net_sim[r, "idx.ig"] = dfff_res[ii, "idx.ig"]
    dff_net_sim[r, "ctd.pval"] = 2^-dfff_res[ii,"d.score"]
    perms_df = dfff_res[sample(1:nrow(dfff_res), 2000), "d.score"]
    dff_net_sim[r, "perm.pval"] =
(1+length(which(perms_df>=dfff_res[ii,"d.score"])))/2001
    dff_net_sim[r, "true.pval"] =
length(which(dfff_res[,"d.score"]>=dfff_res[ii,"d.score"]))/nrow(dfff_res)
    r = r + 1
  }
  dff_net_sim$ctd.pval[which(dff_net_sim$ctd.pval>1)] = 1

  # Figure 4b
  dff = data.frame(network=character(), pval.thresh=numeric(),
alpha=numeric(), power=numeric(), spec=numeric(),
estimate_type=character(), encoding_type=character(), stringsAsFactors =
FALSE)
  for (ctd.pval.thresh in seq(0, 1, 0.01)) {
    dff_stats_thresh =
dff_net_sim[which(dff_net_sim$ctd.pval<=ctd.pval.thresh),]
    prob_alpha = c()
    spec_alpha = c()
    r = 1
    for (alpha in 0.05) {
      # CTD-bounds pval
      tp = length(intersect(which(dff_stats_thresh$true.pval<=alpha),
                            which(dff_stats_thresh$ctd.pval<=alpha)))
      fn = length(intersect(which(dff_stats_thresh$true.pval<=alpha),
                            which(dff_stats_thresh$ctd.pval>alpha)))
      tn = length(intersect(which(dff_stats_thresh$true.pval>alpha),
                            which(dff_stats_thresh$ctd.pval>alpha)))
      fp = length(intersect(which(dff_stats_thresh$true.pval>alpha),
                            which(dff_stats_thresh$ctd.pval<=alpha)))
      print(sprintf("Sum TP/FN/FP/TN = %d, should be %d.", tp+fn+fp+tn,
nrow(dff_stats_thresh)))
      prob_alpha[r] = tp/(tp+fn)
      spec_alpha[r] = 1-(tn/(tn+fp))
      r = r + 1
    }
    dff = rbind(dff, data.frame(pval.thresh=rep(ctd.pval.thresh
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
    dff = rbind(dff, data.frame(pval.thresh=rep(ctd.pval.thresh,
length(prob_alpha)),
                                network=net,
                                alpha=0.05,
                                power=prob_alpha,
                                spec = spec_alpha,
                                estimate_type=rep("ctd-upper-bounds",
length(prob_alpha)),
                                encoding_type=rep("single-node", length(prob_alpha))))
  }
  save(dff_net_sim, dff, ind, file=sprintf("power_%s_sn.RData", net))
}
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

Visualize CTD's power

```
require(R.utils)
require(ggplot2)
n1 = loadToEnv("power_net1_sn.RData")[["dff"]]
n2 = loadToEnv("power_net2_sn.RData")[["dff"]]
n3 = loadToEnv("power_net3_sn.RData")[["dff"]]
sn_nets = rbind(n1, n2, n3)
sn_nets$est_type = sprintf("%s-%s", sn_nets$estimate_type,
sn_nets$network)
svg("singleNode_power_v5.svg")
ggplot(sn_nets, aes(x=pval.thresh, y=power, colour=est_type)) +
geom_point(size=2) + geom_line() +
  ggtitle("Simulation: Power of CTD Upper-Bounds\nEstimates (Single-Node)
by Threshold") +
  theme(axis.text.y = element_text(size=12), axis.text.x =
element_text(size=12)) + xlab("CTD p-value threshold")
dev.off()


dff = data.frame(CTD=-log2(dff_net_sim$ctd.pval), Permutation=-
log2(dff_net_sim$perm.pval), True=-log2(dff_net_sim$true.pval))
delta_h = dff$Permutation-dff$CTD
stdev_power = sd(delta_h)
mn_power = mean(delta_h)
max(delta_h)
mn_power
mn_power + 2.03*stdev_power/sqrt(nrow(dff)-1)
mn_power - 2.03*stdev_power/sqrt(nrow(dff)-1)
dff$True = 2^-dff$True
dff$CTD = 2^-dff$CTD
dff$Permutation = 2^-dff$Permutation
dff$True[which(dff$True>1)] = 1
dff$CTD[which(dff$CTD>1)] = 1
dff$Permutation[which(dff$Permutation>1)] = 1
svg("singleNode_conservative_v5.svg")
ggplot(dff) + xlim(0, 1) + ylim(0, 1) +
  geom_point(aes(x=True, y=CTD, color="CTD")) +
  geom_point(aes(x=True, y=Permutation, color="Permutation")) +
  ggtitle("Simulation: CTD Upper-Bounds Estimates\n(Single-Node) vs.
Permutation-based P-values") +
  xlab("True P-value (bits)") + ylab("Estimated P-value (bits)") +
  theme(axis.text.y = element_text(size=12), axis.text.x =
element_text(size=12))
dev.off()
```

Table 5: CTD as a feature selection method 20m 24s

## 7 How to Replicate Table 5



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

In this result, we used CTD as a feature selection method and a covariate in 5 different disease-specific Partial Least Square (PLS) regression models.

Here we compare CTD as a feature selection method to a basic top z-score feature selection method, and the FSFCN algorithm using three different network clustering algorithms: the Greedy Modularity Optimization, InfoMap, and WalkTrap.

R package versioning:

- CTD v 1.0.0
- CTDext v 1.0.0
- igraph v 1.2.5
- pls v 2.7.3
- pROC v 1.16.2
- caret v 6.0.86

7.1 Define the FCFSN algorithm. 1s

The FCFSN algorithm

```
require(entropy)
whichRtoSelect = function(data_mx, classes) {
 rs = c()
 for (f in 1:nrow(data_mx)) {
  y2d = discretize2d(data_mx[f,], classes, ceiling(1+log2(ncol(data_mx))),
ceiling(1+log2(ncol(data_mx))))
  rs = c(rs, mi.empirical(y2d))
 }
 return(quantile(rs, 0.95))
}
prunedIGraph = function(data_mx, classes, R) {
 ig_pruned = make_empty_graph(directed=FALSE)
 fr = c()
 for (f in 1:nrow(data_mx)) {
  y2d = discretize2d(data_mx[f,], classes, ceiling(1+log2(ncol(data_mx))),
ceiling(1+log2(ncol(data_mx))))
  if(mi.empirical(y2d) > R) {
   fr = c(fr, rownames(data_mx)[f])
  }
 }
 ig_pruned = add.vertices(ig_pruned, nv=length(fr), attr=list(name=fr))


 L = data.frame(fi=character(), fj=character(), s=numeric(), stringsAsFactors
= FALSE)
 data_mx_fr = data_mx[which(rownames(data_mx) %in% fr),]
 it = 1
 for (fi in 1:length(fr)) {
  for (fj in fi:length(fr)) {
   if (fi!=fj) {
    L[it,"fi"] = rownames(data_mx_fr)[fi]
    L[it,"fj"] = rownames(data_mx_fr)[fj]
    L[it,"s"] = cor(data_mx_fr[fi,], data_mx_fr[fj,], method="spearman")
    it = it + 1
   }
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
    }
   }
  }
  L = L[order(abs(L$s), decreasing = TRUE),]

  it = 1
  disconnected = TRUE
  while (disconnected) {
   ig_pruned = add.edges(ig_pruned, e=c(L[it,"fi"], L[it, "fj"]), attr =
list(weight=L[it,"s"]))
   it = it + 1
   disconnected = (!is.connected(ig_pruned))
  }

  return(ig_pruned)
}
fsfcn = function(ig_pruned, data_mx, classes, clusters) {
  S = c()
  for (c in 1:length(clusters)) {
   ig_cluster = induced_subgraph(ig_pruned, v=which(V(ig_pruned)$name
%in% clusters[[c]]))
   data_mx_sub = data_mx[which(rownames(data_mx) %in% clusters[[c]]),]
   while (length(V(ig_cluster)$name)>0) {
    if (length(V(ig_cluster)$name)==1) {
     f_mx = V(ig_cluster)$name[1]
     data_mx_sub = as.matrix(data_mx_sub)
    } else {
     mi = c()
     for (f in 1:nrow(data_mx_sub)) {
      y2d = discretize2d(data_mx_sub[f,], classes, 10, 10)
      mi[f] = mi.empirical(y2d)
     }
     f_mx = rownames(data_mx_sub)[which.max(mi)]
    }
    f_n = names(unlist(ego(ig_cluster, 1, nodes=f_mx)))
    ig_cluster = delete.vertices(ig_cluster, v=which(V(ig_cluster)$name %in%
c(f_mx, f_n)))
    data_mx_sub = data_mx_sub[-which(rownames(data_mx_sub) %in%
c(f_mx, f_n)),]
    S = c(S, f_mx)
    print(sprintf("Size S = %d, Size ig_cluster = %d", length(S),
length(V(ig_cluster)$name)))
   }
  }
  return(S)
}
```

Three functions to implement the FCFSN algorithm published in:

Savic M, Kurbalija V, Ivanovic M, Bosnic Z. A Feature Selection Method Based on Feature Correlation Networks. In: Ouhammou Y, Ivanovic M, Abelló A, Bellatreche L, editors. Model and Data Engineering. Springer, Cham: Lecture Notes in Computer Science; 2017. p. 248-61.



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

## 7.2 Apply and compare CTD as a feature selection method.

22s

Apply CTD as a feature selection method.

```
rm(list=setdiff(ls(),c("data_mx.og","cohorts", "whichRtoSelect", "prunedIGraph
"fsfcn")))
module_select=list()
fill.rate = as.numeric(Miller2015$`Times identifed in all 200 samples`[-1])/200
data_mx = data_mx.og[which(fill.rate>0.80), ]
data_mx = data_mx[-grep("x - ", rownames(data_mx)),]
for (model in c("cit", "msud", "mma", "pa", "pku")) {
 # Top Z-score feature selection: Metabolites with an absolute value mean z-
score > 2 will be selected.
 df_mn = apply(data_mx[,which(colnames(data_mx) %in% cohorts[[model]])], 1
function(i) mean(na.omit(i)))
 module_select[["Zscore"]][[model]] = names(df_mn[which(abs(df_mn)>2)])

 # CTD feature selection
 # Iterate through all patients with a known diagnosis to find most connected
metabolites in their
 # z-scored perturbations >2 or <-2
 mdst = c()
 df_ctd = data.frame(ptID=character(), mets=character(), m=numeric(),
stringsAsFactors = FALSE)
 for (fold in 1:length(cohorts[[model]])) {

load(system.file(sprintf('networks/ind_foldNets/bg_%s_fold%s.RData',model,fol
package='CTDext'))
   adjacency_matrix = as.matrix(get.adjacency(ig_pruned, attr="weight"))
   G = vector(mode="list", length=length(V(ig_pruned)$name))
   names(G) = V(ig_pruned)$name
   ranks = loadToEnv(system.file(sprintf('ranks/ind_ranks/%s%d-
ranks.RData',toupper(model), fold), package='CTDext'))
[["permutationByStartNode"]]
   ranks = lapply(ranks, tolower)
   # The patient-who-was-left-out-of-this-network-fold's perturbations will be
scored by CTD.
   ptID = cohorts[[model]][fold]
   diag = names(cohorts)[which(unlist(lapply(cohorts, function(i) ptID %in% i)))
   S = data_mx[order(abs(data_mx[,ptID]), decreasing = TRUE),ptID]
   S = S[which(abs(S)>2)]
   S = S[which(names(S) %in% names(G))]
   ptBSbyK = mle.getPtBSbyK(names(S), ranks, num.misses = log2(length(G)))
   res = mle.getEncodingLength(ptBSbyK, NULL, ptID, G)
   df_ctd[fold, "ptID"] = cohorts[[model]][fold]
   df_ctd[fold, "mets"] =
paste(names(which(ptBSbyK[[which.max(res[,"d.score"])]]==1)), collapse="@"
   df_ctd[fold, "m"] = res[which.max(res[,"d.score"]), "d.score"]-log2(nrow(res
   mdst = c(mdst, names(which(ptBSbyK[[which.max(res[,"d.score"])]]==1)))
 }
 # CTD will select metabolites that are present in at least 50% of patients'
optimally connected subsets
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
  module_select[["CTD"]][[model]] = names(table(mdst)[table(mdst)>
(length(cohorts[[model]])/2)])

  # FSFCN algorithm
  diag_data = data_mx[,which(colnames(data_mx) %in% c(cohorts[[model]],
cohorts$ref))]
  diags = colnames(diag_data)
  diags[-which(diags %in% cohorts[[model]])] = 0
  diags[which(diags %in% cohorts[[model]])] = 1
  diags = as.numeric(diags)
  R = whichRtoSelect(diag_data, diags)
  ig_pruned = prunedIGraph(diag_data, diags, R=R)
  # InfoMap: FSFCN
  cc = cluster_infomap(ig_pruned, e.weights = abs(E(ig_pruned)$weight),
v.weights = NULL, nb.trials = 10, modularity = FALSE)
  module_select[["FSFCN-InfoMap"]][[model]] = fsfcn(ig_pruned, diag_data, diag
communities(cc))
  # Walktrap: FSFCN
  cc = cluster_walktrap(ig_pruned)
  module_select[["FSFCN-WalkTrap"]][[model]] = fsfcn(ig_pruned, diag_data,
diags, communities(cc))
  # GMO: FSFCN
  E(ig_pruned)$weight = abs(E(ig_pruned)$weight)
  cc = cluster_fast_greedy(ig_pruned)
  module_select[["FSFCN-GMO"]][[model]] = fsfcn(ig_pruned, diag_data, diags,
communities(cc))
}
```

Other feature selection methods are also run to compare CTD to:

1. Top z-score method
2. FCFSN with Greedy Modularity Optimization
3. FCFSN with InfoMap
4. FCFSN with WalkTrap

## 7.3 Estimate and plot the variable importance of every feature selected in each feature selection method. 20m

Variable importance calculation.

```
# CTD disease specific PLS models call "yes" or "no" based on CTD-derived
significance
# of patient's top metabolite perturbations.
rm(list=setdiff(ls(),c("data_mx.og","cohorts", "data_mx","module_select")))
require(pls)
require(pROC)
require(caret) # for varImp function
dir.create('./pls', showWarnings = FALSE)
fsmethods=c("Zscore", "CTD", "FSFCN-GMO", "FSFCN-InfoMap", "FSFCN-
WalkTrap")
df_varImp2=list()
dff_model=list()
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
for (model in c("cit", "msud", "mma", "pa", "pku")) {
  # Get CTD LOOCV signal
  load(sprintf("./loocv/loocv_ind_runCTD/best_bits_%s_ind_loocv.RData",
model))
  data_mx2 = data_mx[, dff$ptID]
  # add CTD score as covariate
  data_mx2 = rbind(dff$bits, data_mx2)
  rownames(data_mx2)[1] = "CTD.covariate"
  data_mx2 = t(apply(data_mx2, 1, scale))
  colnames(data_mx2) = dff$ptID
  # Get diagnostic labels
  diag = dff$diag
  diag[which(diag != model)] = 0
  diag[which(diag == model)] = 1
  data_mx2 = rbind(as.numeric(diag), data_mx2)
  rownames(data_mx2)[1] = "diag"
  data_mx2 = apply(data_mx2, c(1,2), as.numeric)

  # compute variant importance of CTD score as a covariate in pls regression
model
  df2_fsmethod=list()
  varImp=list()
  varimp=list()
  dff_model[[model]]=list()
  for (fsmethod in fsmethods){
    df2_fsmethod[[fsmethod]] = data_mx2[which(rownames(data_mx2) %in%
c("CTD.covariate", "diag", module_select[[fsmethod]][[model]])),]
    varImp[[fsmethod]] = vector("list", length=ncol(data_mx2))
    for (it in 1:ncol(data_mx2)){
      isTrain = c(1:ncol(data_mx2))[-it]
      model_res = plsr(diag~., data =
as.data.frame(t(df2_fsmethod[[fsmethod]][,isTrain])))
      varImp[[fsmethod]][[it]] = varImp(model_res)
      tst_data = df2_fsmethod[[fsmethod]][-1,it]
      model_tst =  predict(model_res, ncomp=model_res$ncomp,
newdata=as.data.frame(t(tst_data)))
      dff_model[[model]][[fsmethod]][it] = model_tst
    }
    varimp[[fsmethod]]=apply(as.data.frame(varImp[[fsmethod]]),1,mean)
    names(varimp[[fsmethod]])=gsub("\\`", "", names(varimp[[fsmethod]]))
  }

  for (fsmethod in fsmethods){
    percentile = 1 - length(which(varimp[[fsmethod]]>=varimp[[fsmethod]]
[["CTD.covariate"]]))/length(varimp[[fsmethod]])
    mets_who_beat = names(which(varimp[[fsmethod]]>varimp[[fsmethod]]
[["CTD.covariate"]]))
    print(sprintf("%s: Model %s placed CTD.covariate in %f-th percentile, rank
was %d/%d. Mets who beat were %s", model, fsmethod, percentile,
length(mets_who_beat)+1, length(varimp[[fsmethod]]),
paste(mets_who_beat, collapse=", ")))
  }
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
 df_varImp2[[model]] = data.frame(metabolite=names(Reduce(c,varimp)),
                                  varimp=Reduce(c,varimp),
                                  model=Reduce(c,sapply(names(varimp), function(x)
rep(x,length(varimp[[x]])))))
}
```

May take a few minutes.

Plot variable importance

```
rm(list=setdiff(ls(),c("data_mx.og","cohorts","fsmethods","dff_model","df_varIm
p=list()
for (model in c("cit", "msud", "mma", "pa", "pku")){
  p[[model]]=ggplot(df_varImp2[[model]], aes(x=metabolite, y=varimp,
group=model, colour=model)) +
    geom_point(size=5) +
    geom_line(size=2) +
    theme(text = element_text(size=25), axis.text.x = element_text(angle = 45,
= 1)) +
    ggtitle(sprintf("Variable Importance for %s", toupper(model)))
}
p
```

7.4 Calculate the area under the curve (AUC) for each partial least sqaures (PLS) regression model. 1s



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

Calculate the AUC for PLS.

```
# calculate AUC
pls_auc=list()
pls_auc2=list()
for (model in c("cit", "msud", "mma", "pa", "pku")) {
  print("")
  print(sprintf("For %s model...", toupper(model)))

  # Get diagnostic labels
  load(sprintf("./loocv/loocv_ind_runCTD/best_bits_%s_ind_loocv.RData",
model))
  diag = dff$diag
  diag[which(diag != model)] = 0
  diag[which(diag == model)] = 1

  # Calculate AUC using pROC, since you cannot calculate directly with TP,
TN, FP, FN
  for (fsmethod in fsmethods){
    pls_auc[[fsmethod]]=roc(diag, dff_model[[model]][[fsmethod]], quiet =
TRUE)
    print(sprintf("PLS regression by %s selected features + CTD.covariate AUC
= %.3f", fsmethod, pls_auc[[fsmethod]]$auc))
    pls_auc2[[fsmethod]] = coords(pls_auc[[fsmethod]], "best",
ret=c("threshold", "specificity", "accuracy", "precision", "recall"))
  }
  ctd_auc = roc(diag, dff$bits, quiet = TRUE)
  print(sprintf("PLS regression by CTD.covariate only AUC = %.3f",
pls_auc[[fsmethod]]$auc))
  ctd_auc2 = coords(ctd_auc, "best", ret=c("threshold", "specificity",
"accuracy", "precision", "recall"))
  print(pls_auc2)
  print(ctd_auc2)
}
```

TCGA: topological pathway enrichment analysis 16h 25m

8 Topological Pathway Enrichment Analysis for Breast Cancer Subtypes 1m

TCGA data was downloaded as shared through the UCSC via the Xena browser:

HiSeqV2:
https://xenabrowser.net/datapages/?dataset=TCGA.BRCA.sampleMap%2FHiSeqV2&host=https%3A%2F%2Ftcga.xenahubs.net&removeHub=https%3A%2F%2Fxena.treehouse.gi.ucsc.edu%3A443

BRCA_clinicalMatrix:
https://xenabrowser.net/datapages/?dataset=TCGA.BRCA.sampleMap%2FBRCA_clinicalMatrix&host=https%3A%2F%2Ftcga.xenahubs.net&removeHub=https%3A%2F%2Fxena.treehouse.gi.ucsc.edu%3A443



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

Using the clinicalMatrix, we can select samples from HiSeqV2 that are breast cancer samples and identify the specific subtype of breast cancer for each sample.

**An important note**: Since CePa, SPIA and PRS use permutation testing to estimate p-values, we noticed slightly different p-value estimates between runs. The same general results are reproduced across runs, but p-values outputted by these methods will differ slightly.

R package versioning:

- CTD v 1.0.0
- CTDext v 1.0.0
- graphics (in base R version 4.0.2)
- dplyr v 1.0.2
- DESeq2 v 1.28.1
- graphite v 1.34.0
- igraph v 1.2.5
- graph v 1.66.0
- CePa v 0.7.0
- huge v 1.3.4.1
- org.Hs.eg.db v 3.11.4
- EnrichmentBrowser v 2.18.0
- ToPASeq v 1.22.0
- SPIA v 2.40.0
- DEGraph v 1.40.0
- fgsea v 1.14.0
- reshape2 v 1.4.4
- ggplot2 v 3.3.2

8.1 Normalize TGCA Breast Cancer and Normal Samples with the DESeq2 R package. Secondly, load the pathway knowledge using the graphite R package and build a pathway catalogue R object. 34m

Normalize TCGA breast cancer data

```
require(graphics)
require(dplyr)
require(DESeq2)
BRCA.RNA = read.table("HiSeqV2",header = TRUE)
rownames(BRCA.RNA) = BRCA.RNA$sample
BRCA.RNA$sample = NULL
BRCA.RNA =  (2^BRCA.RNA) - 1
colnames(BRCA.RNA) = gsub(x = colnames(BRCA.RNA), pattern = "\\.",
replacement = "-")
BRCA.table.clinic = read.table("BRCA_clinicalMatrix", header = TRUE, sep = "\t",
row.names = NULL)
subtype = BRCA.table.clinic[,c("sampleID","PAM50Call_RNAseq")]
subtype = subtype[!subtype$PAM50Call_RNAseq == "",]
sample_subtypes = as.character(subtype$sampleID)
normal_samples = BRCA.RNA[,grep("-11",colnames(BRCA.RNA))]
normal_samples_df = as.data.frame(colnames(normal_samples))
normal_samples$genes = rownames(normal_samples)
colnames(normal_samples_df) = "sampleID"
normal_samples_df$group = "normal"
#114 normal samples
BRCA.RNA.tumor = BRCA.RNA[,-grep("-11",colnames(BRCA.RNA))]
#remove the metastatic samples
BRCA.RNA.tumor = BRCA.RNA.tumor[,-grep("-06",colnames(BRCA.RNA.tumor))]
#Remove the normal-like samples
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
#Remove the normal-like samples
subtype = subtype[!subtype$PAM50Call_RNAseq == "Normal", ]
groups = subtype
colnames(groups)[2] = "group"
groups  = rbind(groups,normal_samples_df)
groups = groups[!duplicated(groups$sampleID),]
rownames(groups) = groups$sampleID
groups$sampleID = NULL
rna_subtype = intersect(colnames(BRCA.RNA),rownames(groups))
BRCA.RNA.keep = BRCA.RNA[,rna_subtype]
groups = groups[rna_subtype,,drop = FALSE]
#DeSeq normalization
BRCA.RNA.keep_round = round(BRCA.RNA.keep,digits = 0)
#Remove genes with no expression
BRCA.RNA.keep_round =
BRCA.RNA.keep_round[rowSums(BRCA.RNA.keep_round) > 0, ]
#Remove genes with no variance
require(caret)
nzv <- nearZeroVar(t(BRCA.RNA.keep_round), saveMetrics= TRUE)
nzv = rownames(nzv[nzv$nzv == "TRUE",])
BRCA.RNA.keep_round =
BRCA.RNA.keep_round[!rownames(BRCA.RNA.keep_round) %in% nzv,]
names_counts_all = rownames(BRCA.RNA.keep_round)
BRCA.RNA.keep_round = sapply(BRCA.RNA.keep_round, as.integer)
rownames(BRCA.RNA.keep_round) = names_counts_all
groups$group = as.character(groups$group)
groups$group = as.factor(groups$group)
rownames(groups)
write.table(BRCA.RNA.keep_round,"BRCA.RNA.keep_round.txt", sep = "\t")
write.table(groups,"groups.txt", sep = "\t")
require(BiocParallel)
register(MulticoreParam(6))
dds_groups=  DESeqDataSetFromMatrix(countData = BRCA.RNA.keep_round,
                                    colData = groups,
                                    design= ~ group  )
dds_groups <- estimateSizeFactors(dds_groups)
normalized_counts_groups = counts(dds_groups,normalized = TRUE)
normalized_counts_groups = as.data.frame(normalized_counts_groups)
subtype = BRCA.table.clinic[,c("sampleID","PAM50Call_RNAseq")]
subtype = subtype[!subtype$PAM50Call_RNAseq == "",]
sample_subtypes = as.character(subtype$sampleID)
write.table(normalized_counts_groups,"normal_BRCA_DESeq.txt", sep = "\t")
write.table(groups,"sample_ID_group.txt")
lumA_id =
as.character(subtype$sampleID[which(subtype$PAM50Call_RNAseq=="LumA")
lumB_id =
as.character(subtype$sampleID[which(subtype$PAM50Call_RNAseq=="LumB")
her2_id =
as.character(subtype$sampleID[which(subtype$PAM50Call_RNAseq=="Her2")]
normal_like_id =
as.character(subtype$sampleID[which(subtype$PAM50Call_RNAseq=="Normal
basal_id =
as.character(subtype$sampleID[which(subtype$PAM50Call_RNAseq=="Basal")]
write.table(lumA_id,"lumA_id.txt",sep = "\t")
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
write.table(lumB_id,"lumB_id.txt",sep = "\t")
write.table(her2_id,"her2_id.txt",sep = "\t")
write.table(normal_like_id,"normal_like_id.txt",sep = "\t")
write.table(basal_id,"basal_id.txt",sep = "\t")
```

Load pathways using graphite and pathway catalogue formats

```
require(graphite)
require(igraph)
require(graph)
require(CePa)
# Load pathways using the graphite package (compatible with most
methods)
pwys = pathways("hsapiens","kegg")
# Make a pathway catalogue (PC) object for the KEGG pathway
knowledgebase (compatiable with CePa)
pathList = list()
interactionList = data.frame(interaction.id=numeric(), input=character(),
output=character(), stringsAsFactors = FALSE)
it = 1
for (n in 1:length(pwys)) {
  print(n)
  it.ids = c()
  pwy_graphNEL = pathwayGraph(pwys[[n]])
  ig = igraph.from.graphNEL(pwy_graphNEL)
  e_list = get.edgelist(ig)
  if (nrow(e_list)>0) {
    for (r in 1:nrow(e_list)) {
      interactionList[it, "interaction.id"] = it
      it.ids = c(it.ids, it)
      interactionList[it, "input"] = e_list[r,1]
      interactionList[it, "output"] = e_list[r,2]
      it = it + 1
    }
    pathList[[names(pwys)[n]]] = it.ids
  }
}
# Get ENTREZ ID mappings to Gene SYMBOLS
require(org.Hs.eg.db)
map_df = as.data.frame(mapIds(org.Hs.eg.db,
rownames(normalized_counts_groups), 'ENTREZID', 'SYMBOL'))
colnames(map_df) = "entrez_id"
map_df$gene = rownames(map_df)
mapping = data.frame(node.id=sprintf("ENTREZID:%s", map_df$entrez_id),
symbol=map_df$gene, stringsAsFactors = FALSE)
pc = set.pathway.catalogue(pathList, interactionList, mapping, min.node=5,
max.node=5000, min.gene=5, max.gene=5000)
# optional save state for easy reload
save(pc, file="kegg_pc.RData")
# Keep only pathways from graphite package also found in the KEGG
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

**Pathway Catalogue used for CePa.**
**pwys = pwys[which(names(pwys) %in% names(pc$pathList))]**
graphite is an R package that works with several of the topological pathway enrichment methods we compare CTD to (SPIA, DEGraph, ORA, GSEA).

CePa requires a different format for communicating pathway knowledge: a pathway catalogue object, which is a list object that contains specific values: pathList, interactionList.

We copy information from graphite's pathway knowledge pulled from KEGG, to build a pathway catalogue object compatible with CePa in this code snippet.

## 8.2 CTD for each subtype

4h

CTD as a topological-based pathway enrichment method

```
require(CTD)
require(huge)
require(org.Hs.eg.db)
df_ctd = data.frame(subtype=character(), pathway=character(), ctd.pval=num
stringsAsFactors = FALSE)
r = 1
for (subtype_to_test in c("lumA", "lumB", "her2", "basal")) {
  #Ids
  subtype_id = read.table(sprintf("%s_id.txt",subtype_to_test), sep="\t", header
check.names=FALSE)
  subtype_id = as.character(subtype_id$x)
  subtype_id = gsub("\\-",".",subtype_id)
  subtype_id = subtype_id[-grep("\\.06",subtype_id)]
  print("IDs loaded")
  #R load the data set with the subtype samples and normal samples only
  subtype_normal = as.data.frame(read.table("normal_BRCA_DESeq.txt", header
stringsAsFactors = FALSE))
  subtype_normal = subtype_normal[, c(grep("\\.11", colnames(subtype_normal
which(colnames(subtype_normal) %in% subtype_id))]
  subtype_normal = apply(subtype_normal, c(1,2), as.numeric)
  subtype_normal = subtype_normal[rowSums(subtype_normal) > 0,]
  # map ENTREZ IDs to Gene SYMBOL, because various pathway packages use
and our dataset uses Gene SYMBOLs
  map_df = as.data.frame(mapIds(org.Hs.eg.db, rownames(subtype_normal), 'E
'SYMBOL'))
  colnames(map_df) = "entrez_id"
  map_df$gene = rownames(map_df)
  subtype_normal = subtype_normal[-which(is.na(map_df$entrez_id)),]
  map_df = map_df[-which(is.na(map_df$entrez_id)),]
  subtype_normal = subtype_normal[sort(rownames(subtype_normal)),]
  map_df = map_df[sort(map_df$gene),]
  # Rename Gene SYMBOL rownames in our TCGA dataset to their respective E
  subtype_normal_entrez = subtype_normal
  rownames(subtype_normal_entrez) = map_df$entrez_id
  rownames(subtype_normal_entrez) = sprintf("ENTREZID:%s",
rownames(subtype_normal_entrez))
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
  subtype_normal_entrez = apply(subtype_normal_entrez, c(1,2), as.numeric)
  subtype_normal_entrez = subtype_normal_entrez[rowSums(subtype_normal_
  # Indicator variable to identify the cancer vs normal samples
  phenoData = colnames(subtype_normal)
  phenoData[grep(".01",phenoData)] = 1
  phenoData[grep(".11",phenoData)] = 0
  # Use the deAna function to perform differential expression between subtyp
controls
  # Define differentially expressed (DE) genes between the two conditions
  # ...Using ENTREZ IDs
  subtype_normal_entrez_de = deAna(as.matrix(subtype_normal_entrez),grp =
de.method="limma")
  subtype_normal_entrez_de_fc = subtype_normal_entrez_de$FC
  all = rownames(subtype_normal_entrez_de)
  names(subtype_normal_entrez_de_fc) = all
  # CTD starts here.
  subtype_normal_entrez2 = log2(subtype_normal_entrez+1)
  dif.genes =
names(which(abs(subtype_normal_entrez_de_fc)>quantile(abs(subtype_norma
0.95)))
  paths.hsa = names(pwys)
  for (pathway in 1:length(paths.hsa)) {
    print(sprintf("Pathway %d/%d...", pathway, length(paths.hsa)))
    pwy_graphNEL = pathwayGraph(pwys[[pathway]])
    ig = igraph.from.graphNEL(pwy_graphNEL)
    tmp = t(subtype_normal_entrez2[which(rownames(subtype_normal_entrez2
V(ig)$name),])
    if (ncol(tmp)>0) {
      rrr = apply(tmp, 2, sd)
      if (length(which(rrr==0))>0) {tmp = tmp[,-which(rrr==0)]}
      tmp_ref = t(subtype_normal_entrez2[which(rownames(subtype_normal_en
V(ig)$name), grep(".11", colnames(subtype_normal_entrez2))])
      rrr = apply(tmp_ref, 2, sd)
      if (length(which(rrr==0))>0) {tmp_ref = tmp_ref[,-which(rrr==0)]}
      # Disease+reference interaction network
      inv_covmatt = huge(tmp, method="glasso", lambda=0.1)
      inv_covmat = as.matrix(inv_covmatt$icov[[1]])
      diag(inv_covmat) = 0;
      colnames(inv_covmat) = colnames(tmp)
      ig = graph.adjacency(as.matrix(inv_covmat), mode="undirected", weighted
add.colnames='name')
      V(ig)$name = colnames(inv_covmat)
      # Reference only network
      inv_covmatt_ref = huge(tmp_ref, method="glasso", lambda=0.1)
      inv_covmat_ref = as.matrix(inv_covmatt_ref$icov[[1]])
      diag(inv_covmat_ref) = 0;
      colnames(inv_covmat_ref) = colnames(tmp_ref)
      ig_ref = graph.adjacency(as.matrix(inv_covmat_ref), mode="undirected", w
add.colnames='name')
      V(ig_ref)$name = colnames(inv_covmat_ref)
      if (length(E(ig)$weight)>0) {
        ig_pruned = graph.naivePruning(ig, ig_ref)
        adj_mat = as.matrix(get.adjacency(ig_pruned, attr="weight"))
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
      G = vector(mode="list", length=length(V(ig_pruned)$name))
      names(G) = V(ig_pruned)$name
      if (length(dif.genes[which(dif.genes %in% V(ig_pruned)$name)])>0) {
        for (n in 1:length(dif.genes[which(dif.genes %in% V(ig_pruned)$name)])
          ind = which(names(G)==dif.genes[which(dif.genes %in% V(ig_pruned)
          ranks[[n]] = singleNode.getNodeRanksN(ind, G, p1=0.9, thresholdDiff=
S=dif.genes[which(dif.genes %in% V(ig_pruned)$name)])
        }
        names(ranks) = dif.genes[which(dif.genes %in% V(ig_pruned)$name)]
        ptBSbyK = mle.getPtBSbyK(dif.genes[which(dif.genes %in% V(ig_pruned
ranks)
        res = mle.getEncodingLength(ptBSbyK, NULL, NULL, G)
        df_ctd[r, "subtype"] = subtype_to_test
        df_ctd[r, "pathway"] = paths.hsa[pathway]
        df_ctd[r, "ctd.pval"] = 2^-(max(res$d.score)-log2(length(dif.genes[whic
%in% V(ig)$name)]))
        r = r + 1
      } else {
        df_ctd[r, "subtype"] = subtype_to_test
        df_ctd[r, "pathway"] = paths.hsa[pathway]
        df_ctd[r, "ctd.pval"] = 1
        r = r + 1
      }
    } else {
      df_ctd[r, "subtype"] = subtype_to_test
      df_ctd[r, "pathway"] = paths.hsa[pathway]
      df_ctd[r, "ctd.pval"] = 1
      r = r + 1
    }
  } else {
    df_ctd[r, "subtype"] = subtype_to_test
    df_ctd[r, "pathway"] = paths.hsa[pathway]
    df_ctd[r, "ctd.pval"] = 1
    r = r + 1
  }
 }
 df_ctd[which(df_ctd$ctd.pval>1), "ctd.pval"] = 1
 print("ctd done")
}
save(df_ctd, file="ctd_tbpe.RData")
```

Subtypes:
"lumA" : Luminal A
"lumB": Luminal B
"her2": HER2
"basal": Basal

## 8.3 CePa for each subtype

50m

CePa

```
require(CePa)
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
require(EnrichmentBrowser)
load("kegg_pc.RData")
df_cepa = data.frame(subtype=character(), pathway=character(),
cepa.pval=numeric(), stringsAsFactors = FALSE)
for (subtype_to_test in c("lumA", "lumB", "her2", "basal")) {
  #Ids
  subtype_id = read.table(sprintf("%s_id.txt",subtype_to_test), sep="\t",
header=TRUE, check.names=FALSE)
  subtype_id = as.character(subtype_id$x)
  subtype_id = gsub("\\-",".",subtype_id)
  if (length(grep("\\.06",subtype_id))>0) { subtype_id = subtype_id[-
grep("\\.06",subtype_id)] }
  print(sprintf("IDs loaded for subtype %s", subtype_to_test))
  #R load the data set with the subtype samples and normal samples only
  subtype_normal = as.data.frame(read.table("normal_BRCA_DESeq.txt",
header=TRUE, stringsAsFactors = FALSE))
  subtype_normal = subtype_normal[, c(grep("\\.11",
colnames(subtype_normal)), which(colnames(subtype_normal) %in%
subtype_id))]
  subtype_normal = apply(subtype_normal, c(1,2), as.numeric)
  subtype_normal = subtype_normal[rowSums(subtype_normal) > 0,]
  # Indicator variable to identify the cancer vs normal samples
  phenoData = colnames(subtype_normal)
  phenoData[grep(".01",phenoData)] = 1
  phenoData[grep(".11",phenoData)] = 0
  # Use the deAna function to perform differential expression between
subtype cases and controls
  # Define differentially expressed (DE) genes between the two conditions
  # ...Using Gene SYMBOLS
  subtype_normal_de = deAna(as.matrix(subtype_normal),grp = phenoData,
de.method="limma")
  subtype_normal_de_fc = subtype_normal_de$FC
  all = rownames(subtype_normal_de)
  names(subtype_normal_de_fc) = all
  ## CePa starts here.
  phenoData_cepa = sampleLabel(phenoData,treatment = "1",control = "0")
  dif.genes =
names(which(abs(subtype_normal_de_fc)>quantile(abs(subtype_normal_de_fc)
0.95)))
  cepa_df = cepa.all(dif = dif.genes, bk = names(subtype_normal_de_fc),
pc=pc)
  dff = data.frame(subtype=character(), pathway=character(),
cepa.pval=numeric(), stringsAsFactors = FALSE)
  dff[1:length(cepa_df), "subtype"] = subtype_to_test
  dff[1:length(cepa_df), "pathway"] = names(pc$pathList)
  dff[1:length(cepa_df), "cepa.pval"] = unlist(lapply(cepa_df, function(i)
i$betweenness$p.value))
  df_cepa = rbind(df_cepa, dff)
  print("cepa done")
}
save(df_cepa, file="cepa_tbpe.RData")
```

Subtypes:

"lumA" : Luminal A



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

"lumB": Luminal B
"her2": HER2
"basal": Basal

## 8.4 PRS for each subtype.

10h

If you are unhappy with the amount of time it takes PRS to run, set the nperm parameter to 100 instead of 1000. This will only allow a minimum p-value of 0.01, whereas 1000 permutations can estimate p-values at the resolution of 0.001.

PRS

```
require(ToPASeq)
require(org.Hs.eg.db)
require(EnrichmentBrowser)
df_prs = data.frame(subtype=character(), pathway=character(),
prs.pval=numeric(), stringsAsFactors = FALSE)
r = 1
for (subtype_to_test in c("lumA", "lumB", "her2", "basal")) {
  #Ids
  subtype_id = read.table(sprintf("%s_id.txt",subtype_to_test), sep="\t",
header=TRUE, check.names=FALSE)
  subtype_id = as.character(subtype_id$x)
  subtype_id = gsub("\\-",".",subtype_id)
  if (length(grep("\\.06",subtype_id))>0) { subtype_id = subtype_id[-
grep("\\.06",subtype_id)] }
  print(sprintf("IDs loaded for subtype %s", subtype_to_test))
  #R load the data set with the subtype samples and normal samples only
  subtype_normal = as.data.frame(read.table("normal_BRCA_DESeq.txt",
header=TRUE, stringsAsFactors = FALSE))
  subtype_normal = subtype_normal[, c(grep("\\.11",
colnames(subtype_normal)), which(colnames(subtype_normal) %in%
subtype_id))]
  subtype_normal = apply(subtype_normal, c(1,2), as.numeric)
  subtype_normal = subtype_normal[rowSums(subtype_normal) > 0,]
  # map ENTREZ IDs to Gene SYMBOL, because various pathway packages
use ENTREZ IDs and our dataset uses Gene SYMBOLs
  map_df = as.data.frame(mapIds(org.Hs.eg.db, rownames(subtype_normal),
'ENTREZID', 'SYMBOL'))
  colnames(map_df) = "entrez_id"
  map_df$gene = rownames(map_df)
  subtype_normal = subtype_normal[-which(is.na(map_df$entrez_id)),]
  map_df = map_df[-which(is.na(map_df$entrez_id)),]
  subtype_normal = subtype_normal[sort(rownames(subtype_normal)),]
  map_df = map_df[sort(map_df$gene),]
  # Rename Gene SYMBOL rownames in our TCGA dataset to their respective
ENTREZ IDs
  subtype_normal_entrez = subtype_normal
  rownames(subtype_normal_entrez) = map_df$entrez_id
  rownames(subtype_normal_entrez) = sprintf("ENTREZID:%s",
rownames(subtype_normal_entrez))
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
  subtype_normal_entrez = apply(subtype_normal_entrez, c(1,2), as.numeric)
  subtype_normal_entrez =
subtype_normal_entrez[rowSums(subtype_normal_entrez) > 0, ]
  # Indicator variable to identify the cancer vs normal samples
  phenoData = colnames(subtype_normal)
  phenoData[grep(".01",phenoData)] = 1
  phenoData[grep(".11",phenoData)] = 0
  # Use the deAna function to perform differential expression between
subtype cases and controls
  # Define differentially expressed (DE) genes between the two conditions
  # ...Using ENTREZ IDs
  subtype_normal_entrez_de = deAna(as.matrix(subtype_normal_entrez),grp
= phenoData,  de.method="limma")
  subtype_normal_entrez_de_fc = subtype_normal_entrez_de$FC
  all = rownames(subtype_normal_entrez_de)
  names(subtype_normal_entrez_de_fc) = all
  ## PRS starts here.
  for (n in 1:length(pwys)) {
    print(sprintf("%d / %d...", n, length(pwys)))
    df_prs[r,"subtype"] = subtype_to_test
    df_prs[r,"pathway"] = names(pwys)[n]
    prs_df = try(prs(de = subtype_normal_entrez_de_fc,all =  all, pwys =
pwys[n], nperm=1000))
    if (class(prs_df)=="try-error" || nrow(prs_df)==0) {
      df_prs[r,"prs.pval"] = NA
    } else {
      df_prs[r,"prs.pval"] = prs_df$p.value
    }
    r = r + 1
  }
  print("prs done")
}
save(df_prs, file="prs_tbpe.RData")
```

Subtypes:
"lumA" : Luminal A
"lumB": Luminal B
"her2": HER2
"basal": Basal

## 8.5 SPIA for each subtype

45m

SPIA

```
require(SPIA)
require(org.Hs.eg.db)
require(EnrichmentBrowser)
prepareSPIA(pwys, "baseKEGG")
df_spia = data.frame(subtype=character(), pathway=character(),
spia.pval=numeric(), stringsAsFactors = FALSE)
for (subtype_to_test in c("lumA", "lumB", "her2", "basal")) {
  #Ids
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
  #ids
  subtype_id = read.table(sprintf("%s_id.txt",subtype_to_test), sep="\t",
header=TRUE, check.names=FALSE)
  subtype_id = as.character(subtype_id$x)
  subtype_id = gsub("\\-",".",subtype_id)
  if (length(grep("\\.06",subtype_id))>0) { subtype_id = subtype_id[-
grep("\\.06",subtype_id)] }
  print("IDs loaded")
  #R load the data set with the subtype samples and normal samples only
  subtype_normal = as.data.frame(read.table("normal_BRCA_DESeq.txt",
header=TRUE, stringsAsFactors = FALSE))
  subtype_normal = subtype_normal[, c(grep("\\.11",
colnames(subtype_normal)), which(colnames(subtype_normal) %in%
subtype_id))]
  subtype_normal = apply(subtype_normal, c(1,2), as.numeric)
  subtype_normal = subtype_normal[rowSums(subtype_normal) > 0,]
  # map ENTREZ IDs to Gene SYMBOL, because various pathway packages
use ENTREZ IDs and our dataset uses Gene SYMBOLs
  map_df = as.data.frame(mapIds(org.Hs.eg.db, rownames(subtype_normal),
'ENTREZID', 'SYMBOL'))
  colnames(map_df) = "entrez_id"
  map_df$gene = rownames(map_df)
  subtype_normal = subtype_normal[-which(is.na(map_df$entrez_id)),]
  map_df = map_df[-which(is.na(map_df$entrez_id)),]
  subtype_normal = subtype_normal[sort(rownames(subtype_normal)),]
  map_df = map_df[sort(map_df$gene),]
  # Rename Gene SYMBOL rownames in our TCGA dataset to their respective
ENTREZ IDs
  subtype_normal_entrez = subtype_normal
  rownames(subtype_normal_entrez) = map_df$entrez_id
  rownames(subtype_normal_entrez) = sprintf("ENTREZID:%s",
rownames(subtype_normal_entrez))
  subtype_normal_entrez = apply(subtype_normal_entrez, c(1,2), as.numeric)
  subtype_normal_entrez =
subtype_normal_entrez[rowSums(subtype_normal_entrez) > 0, ]
  # Indicator variable to identify the cancer vs normal samples
  phenoData = colnames(subtype_normal)
  phenoData[grep(".01",phenoData)] = 1
  phenoData[grep(".11",phenoData)] = 0
  # Use the deAna function to perform differential expression between
subtype cases and controls
  # Define differentially expressed (DE) genes between the two conditions
  # ...Using ENTREZ IDs
  subtype_normal_entrez_de = deAna(as.matrix(subtype_normal_entrez),grp
= phenoData,  de.method="limma")
  subtype_normal_entrez_de_fc = subtype_normal_entrez_de$FC
  all = rownames(subtype_normal_entrez_de)
  names(subtype_normal_entrez_de_fc) = all
  ## SPIA starts here.
  spia_df = runSPIA(de=subtype_normal_entrez_de_fc,
names(subtype_normal_entrez_de_fc), pathwaySetName="baseKEGG",
combine="fisher")
  spia_df = spia_df[,c("Name", "pG")]
  colnames(spia_df) = c("pathway", "spia.pval")
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
  spia_df[1:length(spia_df), "subtype"] = subtype_to_test
  df_spia = rbind(df_spia, spia_df)
  print("spia done")
}
save(df_spia, file="spia_tbpe.RData")
```

Subtypes:

"lumA" : Luminal A

"lumB": Luminal B

"her2": HER2

"basal": Basal

## 8.6 DEGraph for each subtype

5m

DEGraph

```
stat.fishersMethod.r = function(x) {
  return (pchisq(-2 * sum(log(x)),df=2*length(x),lower.tail=FALSE))
}
require(DEGraph)
require(org.Hs.eg.db)
df_degraph = data.frame(subtype=character(), pathway=character(),
degraph.pval=numeric(), stringsAsFactors = FALSE)
r = 1
for (subtype_to_test in c("lumA", "lumB", "her2", "basal")) {
  #Ids
  subtype_id = read.table(sprintf("%s_id.txt",subtype_to_test), sep="\t",
header=TRUE, check.names=FALSE)
  subtype_id = as.character(subtype_id$x)
  subtype_id = gsub("\\-",".",subtype_id)
  if (length(grep("\\.06",subtype_id))>0) { subtype_id = subtype_id[-
grep("\\.06",subtype_id)] }
  print(sprintf("IDs loaded for subtype %s", subtype_to_test))
  #R load the data set with the subtype samples and normal samples only
  subtype_normal = as.data.frame(read.table("normal_BRCA_DESeq.txt",
header=TRUE, stringsAsFactors = FALSE))
  subtype_normal = subtype_normal[, c(grep("\\.11",
colnames(subtype_normal)), which(colnames(subtype_normal) %in%
subtype_id))]
  subtype_normal = apply(subtype_normal, c(1,2), as.numeric)
  subtype_normal = subtype_normal[rowSums(subtype_normal) > 0,]
  # map ENTREZ IDs to Gene SYMBOL, because various pathway packages
use ENTREZ IDs and our dataset uses Gene SYMBOLs
  map_df = as.data.frame(mapIds(org.Hs.eg.db, rownames(subtype_normal),
'ENTREZID', 'SYMBOL'))
  colnames(map_df) = "entrez_id"
  map_df$gene = rownames(map_df)
  subtype_normal = subtype_normal[-which(is.na(map_df$entrez_id)),]
  map_df = map_df[-which(is.na(map_df$entrez_id)),]
  subtype_normal = subtype_normal[sort(rownames(subtype_normal)),]
  map_df = map_df[sort(map_df$gene),]
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
  # Rename Gene SYMBOL rownames in our TCGA dataset to their respective
ENTREZ IDs
  subtype_normal_entrez = subtype_normal
  rownames(subtype_normal_entrez) = map_df$entrez_id
  rownames(subtype_normal_entrez) = sprintf("ENTREZID:%s",
rownames(subtype_normal_entrez))
  subtype_normal_entrez = apply(subtype_normal_entrez, c(1,2), as.numeric)
  subtype_normal_entrez =
subtype_normal_entrez[rowSums(subtype_normal_entrez) > 0, ]
  # Indicator variable to identify the cancer vs normal samples
  phenoData = colnames(subtype_normal)
  phenoData[grep(".01",phenoData)] = 1
  phenoData[grep(".11",phenoData)] = 0
  # DEGraph starts here
  for (n in 1:length(names(pwys))) {
    print(sprintf("%d / %d...", n, length(names(pwys))))
    pwy_graphNEL = pathwayGraph(pwys[[n]])
    res = try(testOneGraph(pwy_graphNEL,
subtype_normal_entrez[which(rownames(subtype_normal_entrez) %in%
nodes(pwy_graphNEL)), ],
                      as.numeric(phenoData), useInteractionSigns=FALSE))
    if (class(res)!="try-error") {
      df_degraph[r,"subtype"] = subtype_to_test
      df_degraph[r,"pathway"] = names(pwys)[n]
      df_degraph[r,"degraph.pval"] = stat.fishersMethod(as.numeric(lapply(res,
function(i) i$p.value[2])))
      r = r + 1
    }
  }
  print("degraph done")
}
save(df_degraph, file="degraph_tbpe.RData")
```

Subtypes:
"lumA" : Luminal A
"lumB": Luminal B
"her2": HER2
"basal": Basal

## 8.7 ORA for each subtype

5m

ORA

```
require(org.Hs.eg.db)
require(EnrichmentBrowser)
df_ora = data.frame(pathway=character(), ora.pval=numeric(), subtype=chara
stringsAsFactors = FALSE)
for (subtype_to_test in c("lumA", "lumB", "her2", "basal")) {
  #Ids
  subtype_id = read.table(sprintf("%s_id.txt",subtype_to_test), sep="\t", header
check.names=FALSE)
  subtype_id = as.character(subtype_id$x)
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
  subtype_id = gsub("\\-",".",subtype_id)
  if (length(grep("\\.06",subtype_id))>0) {   subtype_id = subtype_id[-grep("\\.0
}
  print(sprintf("IDs loaded for subtype %s", subtype_to_test))
  #R load the data set with the subtype samples and normal samples only
  subtype_normal = as.data.frame(read.table("normal_BRCA_DESeq.txt", header
stringsAsFactors = FALSE))
  subtype_normal = subtype_normal[, c(grep("\\.11", colnames(subtype_normal
which(colnames(subtype_normal) %in% subtype_id))]
  subtype_normal = apply(subtype_normal, c(1,2), as.numeric)
  subtype_normal = subtype_normal[rowSums(subtype_normal) > 0,]
  # map ENTREZ IDs to Gene SYMBOL, because various pathway packages use
and our dataset uses Gene SYMBOLs
  map_df = as.data.frame(mapIds(org.Hs.eg.db, rownames(subtype_normal), 'E
'SYMBOL'))
  colnames(map_df) = "entrez_id"
  map_df$gene = rownames(map_df)
  subtype_normal = subtype_normal[-which(is.na(map_df$entrez_id)),]
  map_df = map_df[-which(is.na(map_df$entrez_id)),]
  subtype_normal = subtype_normal[sort(rownames(subtype_normal)),]
  map_df = map_df[sort(map_df$gene),]
  # Rename Gene SYMBOL rownames in our TCGA dataset to their respective E
  subtype_normal_entrez = subtype_normal
  rownames(subtype_normal_entrez) = map_df$entrez_id
  rownames(subtype_normal_entrez) = sprintf("ENTREZID:%s",
rownames(subtype_normal_entrez))
  subtype_normal_entrez = apply(subtype_normal_entrez, c(1,2), as.numeric)
  subtype_normal_entrez = subtype_normal_entrez[rowSums(subtype_normal_
  # Indicator variable to identify the cancer vs normal samples
  phenoData = colnames(subtype_normal)
  phenoData[grep(".01",phenoData)] = 1
  phenoData[grep(".11",phenoData)] = 0
  # Use the deAna function to perform differential expression between subtyp
controls
  # Define differentially expressed (DE) genes between the two conditions
  # ...Using ENTREZ IDs
  subtype_normal_entrez_de = deAna(as.matrix(subtype_normal_entrez),grp =
de.method="limma")
  subtype_normal_entrez_de_fc = subtype_normal_entrez_de$FC
  all = rownames(subtype_normal_entrez_de)
  names(subtype_normal_entrez_de_fc) = all
  ## ORA starts here
  dif.genes =
names(which(abs(subtype_normal_entrez_de_fc)>quantile(abs(subtype_norma
0.95)))
  population = intersect(names(subtype_normal_entrez_de_fc), pc$mapping$nc
  paths.hsa = names(pwys)
  ora_df = data.frame(pathway=character(), ora.pval=numeric(), hits=integer()
size=integer(), stringsAsFactors = FALSE)
  for (pathway in 1:length(paths.hsa)) {
    pwy_graphNEL = pathwayGraph(pwys[[pathway]])
    ig = igraph.from.graphNEL(pwy_graphNEL)
    pathway.compounds = V(ig)$name
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
  pathCompIDs = pathway.compounds[which(pathway.compounds %in% popu
  # q (sample successes), m (population successes), n (population failures), k
  sampleSuccesses = length(which(dif.genes %in% pathCompIDs))
  populationSuccesses = length(intersect(pathCompIDs, population))
  N = length(population)
  populationFailures=N-populationSuccesses
  numDraws=length(dif.genes)
  ora_df[pathway, "pathway"] = paths.hsa[pathway]
  ora_df[pathway, "ora.pval"] = phyper(q=sampleSuccesses-1, m=populationS
n=populationFailures, k=numDraws, lower.tail=FALSE)
  ora_df[pathway, "hits"] = sampleSuccesses
  ora_df[pathway, "size"] = populationSuccesses
 }
 ora_df = ora_df[,c("pathway", "ora.pval")]
 ora_df$subtype = rep(subtype_to_test, nrow(ora_df))
 df_ora = rbind(df_ora, ora_df)
 print("ora done")
}
save(df_ora, file="ora_sbpe.RData")
```

Subtypes:
"lumA" : Luminal A
"lumB": Luminal B
"her2": HER2
"basal": Basal

## 8.8 GSEA for each subtype

5m

GSEA

```
require(fgsea)
require(org.Hs.eg.db)
require(EnrichmentBrowser)
df_gsea = data.frame(subtype=character(), pathway=character(),
gsea.pval=numeric(), stringsAsFactors = FALSE)
for (subtype_to_test in c("lumA", "lumB", "her2", "basal")) {
 #Ids
 subtype_id = read.table(sprintf("%s_id.txt",subtype_to_test), sep="\t",
header=TRUE, check.names=FALSE)
 subtype_id = as.character(subtype_id$x)
 subtype_id = gsub("\\-",".",subtype_id)
 if (length(grep("\\.06",subtype_id))>0) { subtype_id = subtype_id[-
grep("\\.06",subtype_id)] }
 print(sprintf("IDs loaded for subtype %s", subtype_to_test))
 #R load the data set with the subtype samples and normal samples only
 subtype_normal = as.data.frame(read.table("normal_BRCA_DESeq.txt",
header=TRUE, stringsAsFactors = FALSE))
 subtype_normal = subtype_normal[, c(grep("\\.11",
colnames(subtype_normal)), which(colnames(subtype_normal) %in%
subtype_id))]
 subtype_normal = apply(subtype_normal, c(1,2), as.numeric)
 subtype_normal = subtype_normal[rowSums(subtype_normal) > 0.1
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
  # map ENTREZ IDs to Gene SYMBOL, because various pathway packages
use ENTREZ IDs and our dataset uses Gene SYMBOLs
  map_df = as.data.frame(mapIds(org.Hs.eg.db, rownames(subtype_normal),
'ENTREZID', 'SYMBOL'))
  colnames(map_df) = "entrez_id"
  map_df$gene = rownames(map_df)
  subtype_normal = subtype_normal[-which(is.na(map_df$entrez_id)),]
  map_df = map_df[-which(is.na(map_df$entrez_id)),]
  subtype_normal = subtype_normal[sort(rownames(subtype_normal)),]
  map_df = map_df[sort(map_df$gene),]
  # Rename Gene SYMBOL rownames in our TCGA dataset to their respective
ENTREZ IDs
  subtype_normal_entrez = subtype_normal
  rownames(subtype_normal_entrez) = map_df$entrez_id
  rownames(subtype_normal_entrez) = sprintf("ENTREZID:%s",
rownames(subtype_normal_entrez))
  subtype_normal_entrez = apply(subtype_normal_entrez, c(1,2), as.numeric)
  subtype_normal_entrez =
subtype_normal_entrez[rowSums(subtype_normal_entrez) > 0, ]
  # Indicator variable to identify the cancer vs normal samples
  phenoData = colnames(subtype_normal)
  phenoData[grep(".01",phenoData)] = 1
  phenoData[grep(".11",phenoData)] = 0
  # Define differentially expressed (DE) genes between the two conditions
  # ...Using ENTREZ IDs
  subtype_normal_entrez_de = deAna(as.matrix(subtype_normal_entrez),grp
= phenoData,  de.method="limma")
  subtype_normal_entrez_de_fc = subtype_normal_entrez_de$FC
  all = rownames(subtype_normal_entrez_de)
  names(subtype_normal_entrez_de_fc) = all
  ## GSEA starts here.
  gsea_df = data.frame(pathway=character(), gsea.pval=numeric(),
stringsAsFactors = FALSE)
  pathways = list()
  for (i in 1:length(pwys)) {
    pwy_graphNEL = pathwayGraph(pwys[[i]])
    ig = igraph.from.graphNEL(pwy_graphNEL)
    pathways[[names(pwys)[i]]] = V(ig)$name
  }
  ranks = subtype_normal_entrez_de_fc
  res = fgsea(pathways, ranks, nperm=1000)
  gsea_df[1:nrow(res), "pathway"] = res$pathway
  gsea_df[1:nrow(res), "gsea.pval"] = res$pval
  gsea_df$subtype = rep(subtype_to_test, nrow(gsea_df))
  df_gsea = rbind(df_gsea, gsea_df)
  print("gsea done")
}
save(df_gsea, file="gsea_tbpe.RData")
```

Subtypes:

"lumA" : Luminal A

"lumB": Luminal B

"her2": HER2

"basal": Basal



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

8.9 Combine all pathway enrichment method's result by subtype 1s



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

Combine pathway enrichment results by subtype

```
#Combine all pathway enrichment results into one data frame, df_all
require(dplyr)
for (subtype_to_test in c("lumA", "lumB", "her2", "basal")) {
  load("ctd_tbpe.RData")
  load("cepa_tbpe.RData")
  load("spia_tbpe.RData")
  load("degraph_tbpe.RData")
  load("prs_tbpe.RData")
  load("ora_sbpe.RData")
  load("gsea_sbpe.RData")
  df_ctd = df_ctd[which(df_ctd$subtype==subtype_to_test), c("pathway",
"ctd.pval")]
  df_cepa = df_cepa[which(df_cepa$subtype==subtype_to_test), c("pathway",
"cepaORA.pval")]
  df_spia = df_spia[which(df_spia$subtype==subtype_to_test), c("pathway",
"spia.pval")]
  df_prs = df_prs[which(df_prs$subtype==subtype_to_test), c("pathway",
"prs.pval")]
  df_degraph = df_degraph[which(df_degraph$subtype==subtype_to_test),
c("pathway", "degraph.pval")]
  df_ora = df_ora[which(df_ora$subtype==subtype_to_test), c("pathway",
"ora.pval")]
  df_gsea = df_gsea[which(df_gsea$subtype==subtype_to_test), c("pathway",
"gsea.pval")]

  df_all = left_join(left_join(left_join(left_join(left_join(df_prs,df_cepa,by
= "pathway"),
                                                    df_spia, by="pathway"),
                                           df_ora, by="pathway"),
                                  df_gsea, by="pathway"),
                         df_ctd, by="pathway"), df_degraph, by="pathway")
  # FDR correct p-values based on number of pathways tested
  df_all$prs.pval = p.adjust(df_all$prs.pval, method="fdr")
  df_all$cepaORA.pval = p.adjust(df_all$cepaORA.pval, method="fdr")
  df_all$spia.pval = p.adjust(df_all$spia.pval, method="fdr")
  df_all$ora.pval = p.adjust(df_all$ora.pval, method="fdr")
  df_all$gsea.pval = p.adjust(df_all$gsea.pval, method="fdr")
  df_all$ctd.pval = p.adjust(df_all$ctd.pval, method="fdr")
  df_all$degraph.pval = p.adjust(df_all$degraph.pval, method="fdr")
  df_all = df_all[order(df_all$pathway), ]
  save(df_all, file=sprintf("topo_%s.RData",subtype_to_test))
}
```

Subtypes:

"lumA" : Luminal A

"lumB": Luminal B

"her2": HER2

"basal": Basal



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

8.10 Generate Figure 5 1s

Generate Figure 5

```
require(reshape2)
require(ggplot2)
dff_all = data.frame(ranks=numeric(), pathway=character(),
method=character(), subtype=character(), stringsAsFactors = FALSE)
for (subtype_to_test in c("lumA", "lumB", "her2", "basal")) {
 load(sprintf("topo_%s.RData", subtype_to_test))
 rankPos.gsea = df_all$pathway[order(df_all$gsea.pval, decreasing =
FALSE)]
 rankPos.ora = df_all$pathway[order(df_all$ora.pval, decreasing = FALSE)]
 rankPos.ctd = df_all$pathway[order(df_all$ctd.pval, decreasing = FALSE)]
 rankPos.spia = df_all$pathway[order(df_all$spia.pval, decreasing = FALSE)]
 rankPos.cepa = df_all$pathway[order(df_all$cepaORA.pval, decreasing =
FALSE)]
 rankPos.prs = df_all$pathway[order(df_all$prs.pval, decreasing = FALSE)]
 rankPos.degraph = df_all$pathway[order(df_all$degraph.pval, decreasing =
FALSE)]
 df_all$gsea.ranks = sapply(df_all$pathway, function(i)
which(rankPos.gsea==i))
 df_all$ora.ranks = sapply(df_all$pathway, function(i) which(rankPos.ora==i))
 df_all$ctd.ranks = sapply(df_all$pathway, function(i) which(rankPos.ctd==i))
 df_all$spia.ranks = sapply(df_all$pathway, function(i)
which(rankPos.spia==i))
 df_all$cepa.ranks = sapply(df_all$pathway, function(i)
which(rankPos.cepa==i))
 df_all$prs.ranks = sapply(df_all$pathway, function(i) which(rankPos.prs==i))
 df_all$degraph.ranks = sapply(df_all$pathway, function(i)
which(rankPos.degraph==i))
 dff = data.frame(ranks=c(df_all$gsea.ranks, df_all$ora.ranks,
df_all$ctd.ranks, df_all$spia.ranks,
                          df_all$cepa.ranks, df_all$prs.ranks, df_all$degraph.ranks),
                  pvalue=c(round(df_all[,"gsea.pval"], 2), round(df_all[,"ora.pval"],
2), round(df_all[,"ctd.pval"], 2),
                           round(df_all[,"spia.pval"], 2), round(df_all[,"cepaORA.pval"],
2), round(df_all[,"prs.pval"], 2),
                           round(df_all[,"degraph.pval"], 2)),
                  pathway=rep(df_all$pathway, 7),
                  method=c(rep("GSEA", length(df_all$gsea.ranks)), rep("ORA",
length(df_all$ora.ranks)),
                           rep("CTD", length(df_all$ctd.ranks)), rep("SPIA",
length(df_all$spia.ranks)),
                           rep("CePa", length(df_all$cepa.ranks)), rep("PRS",
length(df_all$prs.ranks)),
                           rep("DEGraph", length(df_all$degraph.ranks))),
                  subtype=rep(subtype_to_test, 7*nrow(df_all)))
 dff_all = rbind(dff_all, dff)
}
ind.pos = c("PI3K-Akt signaling pathway", "Pathways in cancer", "TGF-beta
signaling pathway", "Breast cancer", "Estrogen signaling pathway",
       "MAPK signaling pathway", "JAK-STAT signaling pathway", "Wnt signaling
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
pathway")
ind.neg = c("Alzheimer disease", "Cushing syndrome", "Inflammatory bowel
disease (IBD)",
            "Inositol phosphate metabolism", "Morphine addiction", "Olfactory
transduction",
            "Pertussis", "Porphyrin and chlorophyll metabolism")
ind = c(ind.pos, ind.neg)
dff_all = dff_all[which(dff_all$pathway %in% ind),]
# Which methods called which pathways significant (FDR p-value < 0.05)?
dff_all[which(dff_all$pvalue<0.15),]
# Plot ranks
dff_all$subtype = factor(dff_all$subtype, levels=c("lumA", "lumB", "her2",
"basal"))
svg("fig5_pathway_enrichment.svg")
ggplot(dff_all) + geom_bar(aes(x=method, y=ranks, fill=subtype),
stat="identity") +
  facet_wrap(~pathway) + ggtitle("Comparison of Pathway Enrichment
Methods") + ylim(c(0,1000))
dev.off()
```

Set-based pathway enrichment methods for metabolomics 1m 57s

9 Two set-based pathway enrichment methods are applied to the metabolomics data from Miller et al 2015, for 5 diagnostic categories (citrullinemia, maple syrup urine disease, methylmalonic aciduria, propionic aciduria, phenylketonuria).

R package versioning:
- CTD v 1.0.0
- CTDext v 1.0.0

9.1 Over-representation analysis (ORA) 3s

Over-representation analysis (ORA)

```
# CIT patients: selected IEM_1017 for visualization
stat.getORA_Metabolon(data_mx.og[,"IEM_1017"], threshold = 2, type =
"zscore")
# MSUD patients: Selected IEM_1058 for visualization
stat.getORA_Metabolon(data_mx.og[,"IEM_1058"], threshold = 2, type =
"zscore")
# MMA patients: Selected IEM_1051 for visualization
stat.getORA_Metabolon(data_mx.og[,"IEM_1051"], threshold = 2, type =
"zscore")
# PA patients: Selected IEM_1093 for visualization
stat.getORA_Metabolon(data_mx.og[,"IEM_1093"], threshold = 2, type =
"zscore")
# PKU patients: selected IEM_1105 for visualization
stat.getORA_Metabolon(data_mx.og[,"IEM_1105"], threshold = 2, type =
"zscore")
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

## 9.2 Metabolite set enrichment analysis (MSEA)

1m 54s

Metabolite set enrichment analysis (MSEA)

```
fill.rate = as.numeric(Miller2015$`Times identifed in all 200 samples`[-1])/200
# CIT
data_mx = data_mx.og[which(fill.rate>0.80),which(colnames(data_mx.og)
%in% c(cohorts$cit, cohorts$ref))]
data_mx = data_mx[-grep("x -", rownames(data_mx)),]
diag.ind = colnames(data_mx)
diag.ind[-which(diag.ind %in% cohorts$cit)] = 0
diag.ind[which(diag.ind %in% cohorts$cit)] = 1
diag.ind = as.numeric(diag.ind)
cit_msea = stat.getMSEA_Metabolon(data_mx, diag.ind,
pathway_knowledgebase = "Metabolon")
cit_msea = cit_msea[which(cit_msea$`NOM\npval`<0.05),c("Pathway", "Size",
"NES", "NOM\npval", "FDR\nqval")]
cit_msea[order(cit_msea$`NOM\npval`, decreasing = FALSE), ]

# MSUD
data_mx = data_mx.og[which(fill.rate>0.80),which(colnames(data_mx.og)
%in% c(cohorts$msud, cohorts$ref))]
data_mx = data_mx[-grep("x -", rownames(data_mx)),]
diag.ind = colnames(data_mx)
diag.ind[-which(diag.ind %in% cohorts$msud)] = 0
diag.ind[which(diag.ind %in% cohorts$msud)] = 1
diag.ind = as.numeric(diag.ind)
msud_msea = stat.getMSEA_Metabolon(data_mx, diag.ind,
pathway_knowledgebase = "Metabolon")
msud_msea =
msud_msea[which(msud_msea$`NOM\npval`<0.05),c("Pathway", "Size",
"NES", "NOM\npval", "FDR\nqval")]
msud_msea[order(msud_msea$`NOM\npval`, decreasing = FALSE), ]

# MMA
data_mx = data_mx.og[which(fill.rate>0.80),which(colnames(data_mx.og)
%in% c(cohorts$mma, cohorts$ref))]
data_mx = data_mx[-grep("x -", rownames(data_mx)),]
diag.ind = colnames(data_mx)
diag.ind[-which(diag.ind %in% cohorts$mma)] = 0
diag.ind[which(diag.ind %in% cohorts$mma)] = 1
diag.ind = as.numeric(diag.ind)
mma_msea = stat.getMSEA_Metabolon(data_mx, diag.ind,
pathway_knowledgebase = "Metabolon")
mma_msea = mma_msea[which(mma_msea$`NOM\npval`<0.05),c("Pathway",
"Size", "NES", "NOM\npval", "FDR\nqval")]
mma_msea[order(mma_msea$`NOM\npval`, decreasing = FALSE), ]

# PA
data_mx = data_mx.og[which(fill.rate>0.80),which(colnames(data_mx.og)
%in% c(cohorts$pa, cohorts$ref))]
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
data_mx = data_mx[-grep("x -", rownames(data_mx)),]
diag.ind = colnames(data_mx)
diag.ind[-which(diag.ind %in% cohorts$pa)] = 0
diag.ind[which(diag.ind %in% cohorts$pa)] = 1
diag.ind = as.numeric(diag.ind)
pa_msea = stat.getMSEA_Metabolon(data_mx, diag.ind,
pathway_knowledgebase = "Metabolon")
pa_msea = pa_msea[which(pa_msea$`NOM\npval`<0.05),c("Pathway", "Size",
"NES", "NOM\npval", "FDR\nqval")]
pa_msea[order(pa_msea$`NOM\npval`, decreasing = FALSE), ]

# PKU
data_mx = data_mx.og[which(fill.rate>0.80),which(colnames(data_mx.og)
%in% c(cohorts$pku, cohorts$ref))]
data_mx = data_mx[-grep("x -", rownames(data_mx)),]
diag.ind = colnames(data_mx)
diag.ind[-which(diag.ind %in% cohorts$pku)] = 0
diag.ind[which(diag.ind %in% cohorts$pku)] = 1
diag.ind = as.numeric(diag.ind)
pku_msea = stat.getMSEA_Metabolon(data_mx, diag.ind,
pathway_knowledgebase = "Metabolon")
pku_msea = pku_msea[which(pku_msea$`NOM\npval`<0.05),c("Pathway",
"Size", "NES", "NOM\npval", "FDR\nqval")]
pku_msea[order(pku_msea$`NOM\npval`, decreasing = FALSE), ]
```

Several things can affect the pathway enrichment results returned by metabolite set enrichment analysis (MSEA). These include

1. The GMT pathway knowledgebase you use can differ in the coverage of metabolites profiled by untargeted metabolomics. Make sure the knowledgebase you're using has the highest percentage of metabolites that can map back to your patient profiling data.

2. The fill rate threshold you select for your profiling data. If you include metabolite data that was highly imputed, your results may show pathways that include the metabolites associated with imputed values, depending on your imputation strategy. We have found that using a fill rate threshold outputs more disease relevant pathways using MSEA compared to including all metabolites. In our use of MSEA, we use an 80% fill rate threshold when deciding which metabolites to keep in our dataset.

Figure 7: Node set probability is based on network context. 1h 3m 20s

10 In this result, we show that the probability assigned to a node set varies significantly based on the network's specificity. For example, the top 5 perturbed metabolites in patients with citrullinemia are assigned the highest probability in the Citrullinemia disease context, compared to the top 5 metabolites perturbed in methylmalonic aciduria or phenylketonuria.

**Note, because surrogate profiles (generated by data.surrogateProfiles()) are generated using random draws from the standard normal distribution, results will vary when you run the code in this section, but the same general trends will be consistent.**

R package versioning:

- CTD v 1.0.0
- CTDext v 1.0.0
- huge v 1.3.4.1
- ggplot2 v 3.3.2



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

10.1 Separate data into cohorts 17s

LOOCV data matrix construction by disease

```
fill.rate = as.numeric(Miller2015$`Times identifed in all 200 samples`[-1])/200
data_mx = data_mx.og[which(fill.rate>0.80),]
data_mx = data_mx[-grep("x -", rownames(data_mx)),]
cit_data = data_mx[,which(diags=="Citrullinemia")]
mma_data = data_mx[,which(diags=="Methylmalonic aciduria")]
pku_data = data_mx[,which(diags=="Phenylketonuria")]
ref_data = data_mx[,which(diags=="No biochemical genetic diagnosis")]
# Sort metabolite perturbations by mean z-score across all reference
profiles.
cit_tmp = apply(abs(cit_data), 1, function(i) mean(na.omit(i)))
cit_tmp = cit_tmp[order(abs(cit_tmp), decreasing = TRUE)]
mma_tmp = apply(abs(mma_data), 1, function(i) mean(na.omit(i)))
mma_tmp = mma_tmp[order(abs(mma_tmp), decreasing = TRUE)]
pku_tmp = apply(abs(pku_data), 1, function(i) mean(na.omit(i)))
pku_tmp = pku_tmp[order(abs(pku_tmp), decreasing = TRUE)]
ref_tmp = apply(abs(ref_data), 1, function(i) mean(na.omit(i)))
ref_tmp = pku_tmp[order(abs(ref_tmp), decreasing = TRUE)]
# 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.
cit_data1 = data.surrogateProfiles(cit_data[,-1], 1, ref_data = ref_data)
cit_data2 = data.surrogateProfiles(cit_data[,-2], 1, ref_data = ref_data)
cit_data3 = data.surrogateProfiles(cit_data[,-3], 1, ref_data = ref_data)
cit_data4 = data.surrogateProfiles(cit_data[,-4], 1, ref_data = ref_data)
cit_data5 = data.surrogateProfiles(cit_data[,-5], 1, ref_data = ref_data)
cit_data6 = data.surrogateProfiles(cit_data[,-6], 1, ref_data = ref_data)
cit_data7 = data.surrogateProfiles(cit_data[,-7], 1, ref_data = ref_data)
cit_data8 = data.surrogateProfiles(cit_data[,-8], 1, ref_data = ref_data)
cit_data9 = data.surrogateProfiles(cit_data[,-9], 1, ref_data = ref_data)
cit_data = data.surrogateProfiles(cit_data, 1, ref_data = ref_data)
dim(cit_data)
mma_data1 = data.surrogateProfiles(mma_data[,-1], 1, ref_data = ref_data)
mma_data2 = data.surrogateProfiles(mma_data[,-2], 1, ref_data = ref_data)
mma_data3 = data.surrogateProfiles(mma_data[,-3], 1, ref_data = ref_data)
mma_data4 = data.surrogateProfiles(mma_data[,-4], 1, ref_data = ref_data)
mma_data5 = data.surrogateProfiles(mma_data[,-5], 1, ref_data = ref_data)
mma_data6 = data.surrogateProfiles(mma_data[,-6], 1, ref_data = ref_data)
mma_data7 = data.surrogateProfiles(mma_data[,-7], 1, ref_data = ref_data)
mma_data8 = data.surrogateProfiles(mma_data[,-8], 1, ref_data = ref_data)
mma_data9 = data.surrogateProfiles(mma_data[,-9], 1, ref_data = ref_data)
mma_data = data.surrogateProfiles(mma_data, 1, ref_data = ref_data)
dim(mma_data)
pku_data1 = data.surrogateProfiles(pku_data[,-1], 1, ref_data = ref_data)
pku_data2 = data.surrogateProfiles(pku_data[,-2], 1, ref_data = ref_data)
pku_data3 = data.surrogateProfiles(pku_data[,-3], 1, ref_data = ref_data)
pku_data4 = data.surrogateProfiles(pku_data[,-4], 1, ref_data = ref_data)
pku_data5 = data.surrogateProfiles(pku_data[,-5], 1, ref_data = ref_data)
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
pku_data6 = data.surrogateProfiles(pku_data[,-6], 1, ref_data = ref_data)
pku_data7 = data.surrogateProfiles(pku_data[,-7], 1, ref_data = ref_data)
pku_data8 = data.surrogateProfiles(pku_data[,-8], 1, ref_data = ref_data)
pku_data = data.surrogateProfiles(pku_data, 1, ref_data = ref_data)
dim(pku_data)
ref_data = data.surrogateProfiles(ref_data, 1, ref_data = ref_data)
dim(ref_data)
```

Build data matrices with surrogate disease and control profiles using a leave one out cross validation data paradigm. Do this for three disease states, separately:

1. Citrullinemia
2. Methylmalonic aciduria
3. Phenylketonuria

## 10.2 Learn disease-specific network folds.

2m

Learn network folds

```
# Learn a Gaussian Markov Random Field model using the Graphical LASSO in the R package "huge".
# Use a regularization parameter of 0.25 for all graphs.
require(huge)
# cit graph
cit_ig = huge(t(cit_data), method="glasso", lambda = 0.25)
cit_ig1 = huge(t(cit_data1), method="glasso", lambda = 0.25)
cit_ig2 = huge(t(cit_data2), method="glasso", lambda = 0.25)
cit_ig3 = huge(t(cit_data3), method="glasso", lambda = 0.25)
cit_ig4 = huge(t(cit_data4), method="glasso", lambda = 0.25)
cit_ig5 = huge(t(cit_data5), method="glasso", lambda = 0.25)
cit_ig6 = huge(t(cit_data6), method="glasso", lambda = 0.25)
cit_ig7 = huge(t(cit_data7), method="glasso", lambda = 0.25)
cit_ig8 = huge(t(cit_data8), method="glasso", lambda = 0.25)
cit_ig9 = huge(t(cit_data9), method="glasso", lambda = 0.25)
plot(cit_ig)
# mma graph
mma_ig = huge(t(mma_data), method="glasso", lambda = 0.25)
mma_ig1 = huge(t(mma_data1), method="glasso", lambda = 0.25)
mma_ig2 = huge(t(mma_data2), method="glasso", lambda = 0.25)
mma_ig3 = huge(t(mma_data3), method="glasso", lambda = 0.25)
mma_ig4 = huge(t(mma_data4), method="glasso", lambda = 0.25)
mma_ig5 = huge(t(mma_data5), method="glasso", lambda = 0.25)
mma_ig6 = huge(t(mma_data6), method="glasso", lambda = 0.25)
mma_ig7 = huge(t(mma_data7), method="glasso", lambda = 0.25)
mma_ig8 = huge(t(mma_data8), method="glasso", lambda = 0.25)
mma_ig9 = huge(t(mma_data9), method="glasso", lambda = 0.25)
plot(mma_ig)
# pku graph
pku_ig = huge(t(pku_data), method="glasso", lambda = 0.25)
pku_ig1 = huge(t(pku_data1), method="glasso", lambda = 0.25)
pku_ig2 = huge(t(pku_data2), method="glasso", lambda = 0.25)
pku_ig3 = huge(t(pku_data3), method="glasso", lambda = 0.25)
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
pku_ig4 = huge(t(pku_data4), method="glasso", lambda = 0.25)
pku_ig5 = huge(t(pku_data5), method="glasso", lambda = 0.25)
pku_ig6 = huge(t(pku_data6), method="glasso", lambda = 0.25)
pku_ig7 = huge(t(pku_data7), method="glasso", lambda = 0.25)
pku_ig8 = huge(t(pku_data8), method="glasso", lambda = 0.25)
plot(pku_ig)

ref_ig = huge(t(ref_data), method="glasso", lambda=0.25)
plot(ref_ig)

# Get the adjacency matrices, set the diagonal edges (self-edges) to 0, as
we are not interested in
# "selfed" edge weights.
cit_ig = as.matrix(cit_ig$icov[[1]])
cit_ig1 = as.matrix(cit_ig1$icov[[1]])
cit_ig2 = as.matrix(cit_ig2$icov[[1]])
cit_ig3 = as.matrix(cit_ig3$icov[[1]])
cit_ig4 = as.matrix(cit_ig4$icov[[1]])
cit_ig5 = as.matrix(cit_ig5$icov[[1]])
cit_ig6 = as.matrix(cit_ig6$icov[[1]])
cit_ig7 = as.matrix(cit_ig7$icov[[1]])
cit_ig8 = as.matrix(cit_ig8$icov[[1]])
cit_ig9 = as.matrix(cit_ig9$icov[[1]])
mma_ig = as.matrix(mma_ig$icov[[1]])
mma_ig1 = as.matrix(mma_ig1$icov[[1]])
mma_ig2 = as.matrix(mma_ig2$icov[[1]])
mma_ig3 = as.matrix(mma_ig3$icov[[1]])
mma_ig4 = as.matrix(mma_ig4$icov[[1]])
mma_ig5 = as.matrix(mma_ig5$icov[[1]])
mma_ig6 = as.matrix(mma_ig6$icov[[1]])
mma_ig7 = as.matrix(mma_ig7$icov[[1]])
mma_ig8 = as.matrix(mma_ig8$icov[[1]])
mma_ig9 = as.matrix(mma_ig9$icov[[1]])
pku_ig = as.matrix(pku_ig$icov[[1]])
pku_ig1 = as.matrix(pku_ig1$icov[[1]])
pku_ig2 = as.matrix(pku_ig2$icov[[1]])
pku_ig3 = as.matrix(pku_ig3$icov[[1]])
pku_ig4 = as.matrix(pku_ig4$icov[[1]])
pku_ig5 = as.matrix(pku_ig5$icov[[1]])
pku_ig6 = as.matrix(pku_ig6$icov[[1]])
pku_ig7 = as.matrix(pku_ig7$icov[[1]])
pku_ig8 = as.matrix(pku_ig8$icov[[1]])
ref_ig = as.matrix(ref_ig$icov[[1]])
diag(cit_ig) = 0
diag(cit_ig1) = 0
diag(cit_ig2) = 0
diag(cit_ig3) = 0
diag(cit_ig4) = 0
diag(cit_ig5) = 0
diag(cit_ig6) = 0
diag(cit_ig7) = 0
diag(cit_ig8) = 0
diag(cit_ig9) = 0
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
diag(mma_ig) = 0
diag(mma_ig1) = 0
diag(mma_ig2) = 0
diag(mma_ig3) = 0
diag(mma_ig4) = 0
diag(mma_ig5) = 0
diag(mma_ig6) = 0
diag(mma_ig7) = 0
diag(mma_ig8) = 0
diag(mma_ig9) = 0
diag(pku_ig) = 0
diag(pku_ig1) = 0
diag(pku_ig2) = 0
diag(pku_ig3) = 0
diag(pku_ig4) = 0
diag(pku_ig5) = 0
diag(pku_ig6) = 0
diag(pku_ig7) = 0
diag(pku_ig8) = 0
diag(ref_ig) = 0
colnames(cit_ig) = rownames(cit_data)
colnames(cit_ig1) = rownames(cit_data1)
colnames(cit_ig2) = rownames(cit_data2)
colnames(cit_ig3) = rownames(cit_data3)
colnames(cit_ig4) = rownames(cit_data4)
colnames(cit_ig5) = rownames(cit_data5)
colnames(cit_ig6) = rownames(cit_data6)
colnames(cit_ig7) = rownames(cit_data7)
colnames(cit_ig8) = rownames(cit_data8)
colnames(cit_ig9) = rownames(cit_data9)
colnames(mma_ig) = rownames(mma_data)
colnames(mma_ig1) = rownames(mma_data1)
colnames(mma_ig2) = rownames(mma_data2)
colnames(mma_ig3) = rownames(mma_data3)
colnames(mma_ig4) = rownames(mma_data4)
colnames(mma_ig5) = rownames(mma_data5)
colnames(mma_ig6) = rownames(mma_data6)
colnames(mma_ig7) = rownames(mma_data7)
colnames(mma_ig8) = rownames(mma_data8)
colnames(mma_ig9) = rownames(mma_data9)
colnames(pku_ig) = rownames(pku_data)
colnames(pku_ig1) = rownames(pku_data1)
colnames(pku_ig2) = rownames(pku_data2)
colnames(pku_ig3) = rownames(pku_data3)
colnames(pku_ig4) = rownames(pku_data4)
colnames(pku_ig5) = rownames(pku_data5)
colnames(pku_ig6) = rownames(pku_data6)
colnames(pku_ig7) = rownames(pku_data7)
colnames(pku_ig8) = rownames(pku_data8)
colnames(ref_ig) = rownames(ref_data)
# Convert adjacency matrices to igraph objects for all three graphs.
ig_cit1 = graph.adjacency(cit_ig1, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_cit2 = graph.adjacency(cit_ig2, mode="undirected", weighted=TRUE
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
ig_cit2 = graph.adjacency(cit_ig2, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_cit3 = graph.adjacency(cit_ig3, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_cit4 = graph.adjacency(cit_ig4, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_cit5 = graph.adjacency(cit_ig5, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_cit6 = graph.adjacency(cit_ig6, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_cit7 = graph.adjacency(cit_ig7, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_cit8 = graph.adjacency(cit_ig8, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_cit9 = graph.adjacency(cit_ig9, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_cit = graph.adjacency(cit_ig, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_cit
ig_mma1 = graph.adjacency(mma_ig1, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_mma2 = graph.adjacency(mma_ig2, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_mma3 = graph.adjacency(mma_ig3, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_mma4 = graph.adjacency(mma_ig4, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_mma5 = graph.adjacency(mma_ig5, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_mma6 = graph.adjacency(mma_ig6, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_mma7 = graph.adjacency(mma_ig7, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_mma8 = graph.adjacency(mma_ig8, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_mma9 = graph.adjacency(mma_ig9, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_mma = graph.adjacency(mma_ig, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_mma
ig_pku1 = graph.adjacency(pku_ig1, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_pku2 = graph.adjacency(pku_ig2, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_pku3 = graph.adjacency(pku_ig3, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_pku4 = graph.adjacency(pku_ig4, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_pku5 = graph.adjacency(pku_ig5, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_pku6 = graph.adjacency(pku_ig6, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_pku7 = graph.adjacency(pku_ig7, mode="undirected", weighted=TRUE,
add.colnames = "name")
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
ig_pku8 = graph.adjacency(pku_ig8, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_pku = graph.adjacency(pku_ig, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_pku

ig_ref = graph.adjacency(ref_ig, mode="undirected", weighted=TRUE,
add.colnames = "name")
ig_ref
# Create a random graph based on permuting node labels of the learned
reference graph.
ig_rand = ig_ref
V(ig_rand)$name = V(ig_ref)$name[sample(1:length(V(ig_ref)$name),
length(V(ig_ref)$name), replace = FALSE)]
ig_rand
```

We learn 1 network per patient left out. So if we have 9 Citrullinemia patients, we have 9 network folds, where each network fold corresponds to 1 Citrullinemia patient being left out of network learning. We repeat this for 9 methylmalonic aciduria patients, with 8 network folds, and 8 phenylketonuria patients, with 8 network folds. We also learn a "reference only" network, used in network pruning, and a "random network", which is a copy of the reference only network with node labels scrambled.

10.3 Prune disease+control network folds to output disease-specific network folds. 25m



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

Prune network folds

```
# Naive pruning method: Prune edges in disease networks that are
associated with significant modules in reference network.
ig_cit_naive = graph.naivePruning(ig_cit, ig_ref)
ig_cit1_naive = graph.naivePruning(ig_cit1, ig_ref)
ig_cit2_naive = graph.naivePruning(ig_cit2, ig_ref)
ig_cit3_naive = graph.naivePruning(ig_cit3, ig_ref)
ig_cit4_naive = graph.naivePruning(ig_cit4, ig_ref)
ig_cit5_naive = graph.naivePruning(ig_cit5, ig_ref)
ig_cit6_naive = graph.naivePruning(ig_cit6, ig_ref)
ig_cit7_naive = graph.naivePruning(ig_cit7, ig_ref)
ig_cit8_naive = graph.naivePruning(ig_cit8, ig_ref)
ig_cit9_naive = graph.naivePruning(ig_cit9, ig_ref)

ig_mma_naive = graph.naivePruning(ig_mma, ig_ref)
ig_mma1_naive = graph.naivePruning(ig_mma1, ig_ref)
ig_mma2_naive = graph.naivePruning(ig_mma2, ig_ref)
ig_mma3_naive = graph.naivePruning(ig_mma3, ig_ref)
ig_mma4_naive = graph.naivePruning(ig_mma4, ig_ref)
ig_mma5_naive = graph.naivePruning(ig_mma5, ig_ref)
ig_mma6_naive = graph.naivePruning(ig_mma6, ig_ref)
ig_mma7_naive = graph.naivePruning(ig_mma7, ig_ref)
ig_mma8_naive = graph.naivePruning(ig_mma8, ig_ref)
ig_mma9_naive = graph.naivePruning(ig_mma9, ig_ref)

ig_pku_naive = graph.naivePruning(ig_pku, ig_ref)
ig_pku1_naive = graph.naivePruning(ig_pku1, ig_ref)
ig_pku2_naive = graph.naivePruning(ig_pku2, ig_ref)
ig_pku3_naive = graph.naivePruning(ig_pku3, ig_ref)
ig_pku4_naive = graph.naivePruning(ig_pku4, ig_ref)
ig_pku5_naive = graph.naivePruning(ig_pku5, ig_ref)
ig_pku6_naive = graph.naivePruning(ig_pku6, ig_ref)
ig_pku7_naive = graph.naivePruning(ig_pku7, ig_ref)
ig_pku8_naive = graph.naivePruning(ig_pku8, ig_ref)

ig_rand_naive = graph.naivePruning(ig_rand, ig_ref)
```

10.4 This is the main driver script for this experiment. Now that we have all disease-specific network folds learned, we can calculate the probabilities of the top 5 perturbed metabolites for each disease state (citrullinemia, methylmalonic aciduria, phenylketonuria) in each network context. 34m

Probability based on network context main driver script

```
p1=0.9
thresholdDiff=0.01
docit = TRUE
domma = TRUE
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
domma = TRUE
dopku = TRUE
res = data.frame(subset_size=numeric(), graph_name=character(),
                 IA_top_cit=numeric(), IA_top_mma=numeric(),
IA_top_pku=numeric(),
                 IA_top_ref=numeric(), IA_top_rand=numeric(), stringsAsFactors =
FALSE)
r_row = 1
for (subset_size in c(5,10,15,20)) {
  # Set the top K metabolite perturbations based on each cohort dataset. K =
subset_size in the for loop.
  # Top K metabolites from cit cohort
  met_set1 = tolower(names(head(cit_tmp, n=subset_size)))
  # Top K metabolites from mma cohort
  met_set2 = tolower(names(head(mma_tmp, n=subset_size)))
  # Top K metabolites from pku cohort
  met_set3 = tolower(names(head(pku_tmp, n=subset_size)))
  # Top K metabolties from reference cohort
  met_set4 = tolower(names(head(ref_tmp, n=subset_size)))
  # K random metabolites
  met_set5 = tolower(sample(V(ig_rand)$name, size=subset_size, replace =
FALSE))

  if (docit) {
    # ig_cit: More global parameters
    G = vector(mode="list", length=length(V(ig_cit_naive)$name))
    names(G) = V(ig_cit_naive)$name
    adj_mat = as.matrix(get.adjacency(ig_cit_naive, attr="weight"))
    # Use single node encoding to get node ranks
    ranks1 = list()
    for (n in 1:length(met_set1)) {
      ind = which(names(G)==met_set1[n])
      ranks1[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set1, num.misses = log2(length(G)))
    }
    names(ranks1) = met_set1
    ranks2 = list()
    for (n in 1:length(met_set2)) {
      ind = which(names(G)==met_set2[n])
      ranks2[[n]] = singleNode.getNodeRanksN(n=ind, G=G,  p1, thresholdDiff,
adj_mat, S=met_set2, num.misses = log2(length(G)))
    }
    names(ranks2) = met_set2
    ranks3 = list()
    for (n in 1:length(met_set3)) {
      ind = which(names(G)==met_set3[n])
      ranks3[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set3, num.misses = log2(length(G)))
    }
    names(ranks3) = met_set3
    ranks4 = list()
    for (n in 1:length(met_set4)) {
      ind = which(names(G)==met_set4[n])
      ranks4[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
adj_mat, S=met_set4, num.misses = log2(length(G)))
  }
  names(ranks4) = met_set4
  ranks5 = list()
  for (n in 1:length(met_set5)) {
    ind = which(names(G)==met_set5[n])
    ranks5[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set5, num.misses = log2(length(G)))
  }
  names(ranks5) = met_set5
  # Get bitstrings from node ranks
  set1_cit_bs = mle.getPtBSbyK(met_set1, ranks1)
  set2_cit_bs = mle.getPtBSbyK(met_set2, ranks2)
  set3_cit_bs = mle.getPtBSbyK(met_set3, ranks3)
  set4_cit_bs = mle.getPtBSbyK(met_set4, ranks4)
  set5_cit_bs = mle.getPtBSbyK(met_set5, ranks5)
  # Then use the bitstrings to convert into encoding length.
  set1_res = mle.getEncodingLength(set1_cit_bs, NULL, "set1", G)
[length(met_set1),]
  set2_res = mle.getEncodingLength(set2_cit_bs, NULL, "set2", G)
[length(met_set2),]
  set3_res = mle.getEncodingLength(set3_cit_bs, NULL, "set3", G)
[length(met_set3),]
  set4_res = mle.getEncodingLength(set4_cit_bs, NULL, "set4", G)
[length(met_set4),]
  set5_res = mle.getEncodingLength(set5_cit_bs, NULL, "set5", G)
[length(met_set5),]
  print(set1_res) # look at IS.alt column.
  print(set2_res) # look at IS.alt column.
  print(set3_res) # look at IS.alt column.
  print(set4_res) # look at IS.alt column.
  print(set5_res) # look at IS.alt column.
  res[r_row, "subset_size"] = subset_size
  res[r_row, "graph_name"] = "cit"
  res[r_row, "IA_top_cit"] = set1_res[,"IS.alt"]
  res[r_row, "IA_top_mma"] = set2_res[,"IS.alt"]
  res[r_row, "IA_top_pku"] = set3_res[,"IS.alt"]
  res[r_row, "IA_top_ref"] = set4_res[,"IS.alt"]
  res[r_row, "IA_top_rand"] = set5_res[,"IS.alt"]
  r_row = r_row + 1

  # ig_cit1: More global parameters
  G = vector(mode="list", length=length(V(ig_cit1_naive)$name))
  names(G) = V(ig_cit1_naive)$name
  adj_mat = as.matrix(get.adjacency(ig_cit1_naive, attr="weight"))
  # Use single node encoding to get node ranks.
  ranks1 = list()
  for (n in 1:length(met_set1)) {
    ind = which(names(G)==met_set1[n])
    ranks1[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set1, num.misses = log2(length(G)))
  }
  names(ranks1) = met_set1
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
  ranks2 = list()
  for (n in 1:length(met_set2)) {
    ind = which(names(G)==met_set2[n])
    ranks2[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set2, num.misses = log2(length(G)))
  }
  names(ranks2) = met_set2
  ranks3 = list()
  for (n in 1:length(met_set3)) {
    ind = which(names(G)==met_set3[n])
    ranks3[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set3, num.misses = log2(length(G)))
  }
  names(ranks3) = met_set3
  ranks4 = list()
  for (n in 1:length(met_set4)) {
    ind = which(names(G)==met_set4[n])
    ranks4[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set4, num.misses = log2(length(G)))
  }
  names(ranks4) = met_set4
  ranks5 = list()
  for (n in 1:length(met_set5)) {
    ind = which(names(G)==met_set5[n])
    ranks5[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set5, num.misses = log2(length(G)))
  }
  names(ranks5) = met_set5
  # Then use the node permutations to outputted to convert them into
bitstrings.
  set1_cit_bs = mle.getPtBSbyK(met_set1, ranks1)
  set2_cit_bs = mle.getPtBSbyK(met_set2, ranks2)
  set3_cit_bs = mle.getPtBSbyK(met_set3, ranks3)
  set4_cit_bs = mle.getPtBSbyK(met_set4, ranks4)
  set5_cit_bs = mle.getPtBSbyK(met_set5, ranks5)
  # Then use the bitstrings to convert into encoding length.
  set1_res = mle.getEncodingLength(set1_cit_bs, NULL, "set1", G)
[length(met_set1),]
  set2_res = mle.getEncodingLength(set2_cit_bs, NULL, "set2", G)
[length(met_set2),]
  set3_res = mle.getEncodingLength(set3_cit_bs, NULL, "set3", G)
[length(met_set3),]
  set4_res = mle.getEncodingLength(set4_cit_bs, NULL, "set4", G)
[length(met_set4),]
  set5_res = mle.getEncodingLength(set5_cit_bs, NULL, "set5", G)
[length(met_set5),]
  print(set1_res) # look at IS.alt column.
  print(set2_res) # look at IS.alt column.
  print(set3_res) # look at IS.alt column.
  print(set4_res) # look at IS.alt column.
  print(set5_res) # look at IS.alt column.
  res[r_row, "subset_size"] = subset_size
  res[r_row, "graph_name"] = "cit1"
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
  res[r_row, "IA_top_cit"] = set1_res[,"IS.alt"]
  res[r_row, "IA_top_mma"] = set2_res[,"IS.alt"]
  res[r_row, "IA_top_pku"] = set3_res[,"IS.alt"]
  res[r_row, "IA_top_ref"] = set4_res[,"IS.alt"]
  res[r_row, "IA_top_rand"] = set5_res[,"IS.alt"]
  r_row = r_row + 1

  # ig_cit2: More global parameters
  G = vector(mode="list", length=length(V(ig_cit2_naive)$name))
  names(G) = V(ig_cit2_naive)$name
  adj_mat = as.matrix(get.adjacency(ig_cit2_naive, attr="weight"))
  # Use single node encoding to get node ranks.
  ranks1 = list()
  for (n in 1:length(met_set1)) {
    ind = which(names(G)==met_set1[n])
    ranks1[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set1, num.misses = log2(length(G)))
  }
  names(ranks1) = met_set1
  ranks2 = list()
  for (n in 1:length(met_set2)) {
    ind = which(names(G)==met_set2[n])
    ranks2[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set2, num.misses = log2(length(G)))
  }
  names(ranks2) = met_set2
  ranks3 = list()
  for (n in 1:length(met_set3)) {
    ind = which(names(G)==met_set3[n])
    ranks3[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set3, num.misses = log2(length(G)))
  }
  names(ranks3) = met_set3
  ranks4 = list()
  for (n in 1:length(met_set4)) {
    ind = which(names(G)==met_set4[n])
    ranks4[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set4, num.misses = log2(length(G)))
  }
  names(ranks4) = met_set4
  ranks5 = list()
  for (n in 1:length(met_set5)) {
    ind = which(names(G)==met_set5[n])
    ranks5[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set5, num.misses = log2(length(G)))
  }
  names(ranks5) = met_set5
  # Then use the node permutations to outputted to convert them into
bitstrings.
  set1_cit_bs = mle.getPtBSbyK(met_set1, ranks1)
  set2_cit_bs = mle.getPtBSbyK(met_set2, ranks2)
  set3_cit_bs = mle.getPtBSbyK(met_set3, ranks3)
  set4_cit_bs = mle.getPtBSbyK(met_set4, ranks4)
  set5_cit_bs = mle.getPtBSbyK(met_set5, ranks5)
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
    # Then use the bitstrings to convert into encoding length.
    set1_res = mle.getEncodingLength(set1_cit_bs, NULL, "set1", G)
[length(met_set1),]
    set2_res = mle.getEncodingLength(set2_cit_bs, NULL, "set2", G)
[length(met_set2),]
    set3_res = mle.getEncodingLength(set3_cit_bs, NULL, "set3", G)
[length(met_set3),]
    set4_res = mle.getEncodingLength(set4_cit_bs, NULL, "set4", G)
[length(met_set4),]
    set5_res = mle.getEncodingLength(set5_cit_bs, NULL, "set5", G)
[length(met_set5),]
    print(set1_res) # look at IS.alt column.
    print(set2_res) # look at IS.alt column.
    print(set3_res) # look at IS.alt column.
    print(set4_res) # look at IS.alt column.
    print(set5_res) # look at IS.alt column.
    res[r_row, "subset_size"] = subset_size
    res[r_row, "graph_name"] = "cit2"
    res[r_row, "IA_top_cit"] = set1_res[,"IS.alt"]
    res[r_row, "IA_top_mma"] = set2_res[,"IS.alt"]
    res[r_row, "IA_top_pku"] = set3_res[,"IS.alt"]
    res[r_row, "IA_top_ref"] = set4_res[,"IS.alt"]
    res[r_row, "IA_top_rand"] = set5_res[,"IS.alt"]
    r_row = r_row + 1

    # ig_cit3: More global parameters
    G = vector(mode="list", length=length(V(ig_cit3_naive)$name))
    names(G) = V(ig_cit3_naive)$name
    adj_mat = as.matrix(get.adjacency(ig_cit3_naive, attr="weight"))
    # Use single node encoding to get node ranks.
    ranks1 = list()
    for (n in 1:length(met_set1)) {
      ind = which(names(G)==met_set1[n])
      ranks1[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set1, num.misses = log2(length(G)))
    }
    names(ranks1) = met_set1
    ranks2 = list()
    for (n in 1:length(met_set2)) {
      ind = which(names(G)==met_set2[n])
      ranks2[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set2, num.misses = log2(length(G)))
    }
    names(ranks2) = met_set2
    ranks3 = list()
    for (n in 1:length(met_set3)) {
      ind = which(names(G)==met_set3[n])
      ranks3[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set3, num.misses = log2(length(G)))
    }
    names(ranks3) = met_set3
    ranks4 = list()
    for (n in 1:length(met_set4)) {
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
    ind = which(names(G)==met_set4[n])
    ranks4[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set4, num.misses = log2(length(G)))
  }
  names(ranks4) = met_set4
  ranks5 = list()
  for (n in 1:length(met_set5)) {
    ind = which(names(G)==met_set5[n])
    ranks5[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set5, num.misses = log2(length(G)))
  }
  names(ranks5) = met_set5
  # Then use the node permutations to outputted to convert them into
bitstrings.
  set1_cit_bs = mle.getPtBSbyK(met_set1, ranks1)
  set2_cit_bs = mle.getPtBSbyK(met_set2, ranks2)
  set3_cit_bs = mle.getPtBSbyK(met_set3, ranks3)
  set4_cit_bs = mle.getPtBSbyK(met_set4, ranks4)
  set5_cit_bs = mle.getPtBSbyK(met_set5, ranks5)
  # Then use the bitstrings to convert into encoding length.
  set1_res = mle.getEncodingLength(set1_cit_bs, NULL, "set1", G)
[length(met_set1),]
  set2_res = mle.getEncodingLength(set2_cit_bs, NULL, "set2", G)
[length(met_set2),]
  set3_res = mle.getEncodingLength(set3_cit_bs, NULL, "set3", G)
[length(met_set3),]
  set4_res = mle.getEncodingLength(set4_cit_bs, NULL, "set4", G)
[length(met_set4),]
  set5_res = mle.getEncodingLength(set5_cit_bs, NULL, "set5", G)
[length(met_set5),]
  print(set1_res) # look at IS.alt column.
  print(set2_res) # look at IS.alt column.
  print(set3_res) # look at IS.alt column.
  print(set4_res) # look at IS.alt column.
  print(set5_res) # look at IS.alt column.
  res[r_row, "subset_size"] = subset_size
  res[r_row, "graph_name"] = "cit3"
  res[r_row, "IA_top_cit"] = set1_res[,"IS.alt"]
  res[r_row, "IA_top_mma"] = set2_res[,"IS.alt"]
  res[r_row, "IA_top_pku"] = set3_res[,"IS.alt"]
  res[r_row, "IA_top_ref"] = set4_res[,"IS.alt"]
  res[r_row, "IA_top_rand"] = set5_res[,"IS.alt"]
  r_row = r_row + 1

  # ig_cit4: More global parameters
  G = vector(mode="list", length=length(V(ig_cit4_naive)$name))
  names(G) = V(ig_cit4_naive)$name
  adj_mat = as.matrix(get.adjacency(ig_cit4_naive, attr="weight"))
  # Use single node encoding to get node ranks.
  ranks1 = list()
  for (n in 1:length(met_set1)) {
    ind = which(names(G)==met_set1[n])
    ranks1[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
adj_mat, S=met_set1, num.misses = log2(length(G)))
  }
  names(ranks1) = met_set1
  ranks2 = list()
  for (n in 1:length(met_set2)) {
    ind = which(names(G)==met_set2[n])
    ranks2[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set2, num.misses = log2(length(G)))
  }
  names(ranks2) = met_set2
  ranks3 = list()
  for (n in 1:length(met_set3)) {
    ind = which(names(G)==met_set3[n])
    ranks3[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set3, num.misses = log2(length(G)))
  }
  names(ranks3) = met_set3
  ranks4 = list()
  for (n in 1:length(met_set4)) {
    ind = which(names(G)==met_set4[n])
    ranks4[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set4, num.misses = log2(length(G)))
  }
  names(ranks4) = met_set4
  ranks5 = list()
  for (n in 1:length(met_set5)) {
    ind = which(names(G)==met_set5[n])
    ranks5[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set5, num.misses = log2(length(G)))
  }
  names(ranks5) = met_set5
  # Then use the node permutations to outputted to convert them into
bitstrings.
  set1_cit_bs = mle.getPtBSbyK(met_set1, ranks1)
  set2_cit_bs = mle.getPtBSbyK(met_set2, ranks2)
  set3_cit_bs = mle.getPtBSbyK(met_set3, ranks3)
  set4_cit_bs = mle.getPtBSbyK(met_set4, ranks4)
  set5_cit_bs = mle.getPtBSbyK(met_set5, ranks5)
  # Then use the bitstrings to convert into encoding length.
  set1_res = mle.getEncodingLength(set1_cit_bs, NULL, "set1", G)
[length(met_set1),]
  set2_res = mle.getEncodingLength(set2_cit_bs, NULL, "set2", G)
[length(met_set2),]
  set3_res = mle.getEncodingLength(set3_cit_bs, NULL, "set3", G)
[length(met_set3),]
  set4_res = mle.getEncodingLength(set4_cit_bs, NULL, "set4", G)
[length(met_set4),]
  set5_res = mle.getEncodingLength(set5_cit_bs, NULL, "set5", G)
[length(met_set5),]
  print(set1_res) # look at IS.alt column.
  print(set2_res) # look at IS.alt column.
  print(set3_res) # look at IS.alt column.
  print(set4_res) # look at IS.alt column.
  print(set5_res) # look at IS.alt column.
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
    res[r_row, "subset_size"] = subset_size
    res[r_row, "graph_name"] = "cit4"
    res[r_row, "IA_top_cit"] = set1_res[,"IS.alt"]
    res[r_row, "IA_top_mma"] = set2_res[,"IS.alt"]
    res[r_row, "IA_top_pku"] = set3_res[,"IS.alt"]
    res[r_row, "IA_top_ref"] = set4_res[,"IS.alt"]
    res[r_row, "IA_top_rand"] = set5_res[,"IS.alt"]
    r_row = r_row + 1

    # ig_cit5: More global parameters
    G = vector(mode="list", length=length(V(ig_cit5_naive)$name))
    names(G) = V(ig_cit5_naive)$name
    adj_mat = as.matrix(get.adjacency(ig_cit5_naive, attr="weight"))
    # Use single node encoding to get node ranks.
    ranks1 = list()
    for (n in 1:length(met_set1)) {
      ind = which(names(G)==met_set1[n])
      ranks1[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set1, num.misses = log2(length(G)))
    }
    names(ranks1) = met_set1
    ranks2 = list()
    for (n in 1:length(met_set2)) {
      ind = which(names(G)==met_set2[n])
      ranks2[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set2, num.misses = log2(length(G)))
    }
    names(ranks2) = met_set2
    ranks3 = list()
    for (n in 1:length(met_set3)) {
      ind = which(names(G)==met_set3[n])
      ranks3[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set3, num.misses = log2(length(G)))
    }
    names(ranks3) = met_set3
    ranks4 = list()
    for (n in 1:length(met_set4)) {
      ind = which(names(G)==met_set4[n])
      ranks4[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set4, num.misses = log2(length(G)))
    }
    names(ranks4) = met_set4
    ranks5 = list()
    for (n in 1:length(met_set5)) {
      ind = which(names(G)==met_set5[n])
      ranks5[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set5, num.misses = log2(length(G)))
    }
    names(ranks5) = met_set5
    # Then use the node permutations to outputted to convert them into
bitstrings.
    set1_cit_bs = mle.getPtBSbyK(met_set1, ranks1)
    set2_cit_bs = mle.getPtBSbyK(met_set2, ranks2)
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
    set3_cit_bs = mle.getPtBSbyK(met_set3, ranks3)
    set4_cit_bs = mle.getPtBSbyK(met_set4, ranks4)
    set5_cit_bs = mle.getPtBSbyK(met_set5, ranks5)
    # Then use the bitstrings to convert into encoding length.
    set1_res = mle.getEncodingLength(set1_cit_bs, NULL, "set1", G)
[length(met_set1),]
    set2_res = mle.getEncodingLength(set2_cit_bs, NULL, "set2", G)
[length(met_set2),]
    set3_res = mle.getEncodingLength(set3_cit_bs, NULL, "set3", G)
[length(met_set3),]
    set4_res = mle.getEncodingLength(set4_cit_bs, NULL, "set4", G)
[length(met_set4),]
    set5_res = mle.getEncodingLength(set5_cit_bs, NULL, "set5", G)
[length(met_set5),]
    print(set1_res) # look at IS.alt column.
    print(set2_res) # look at IS.alt column.
    print(set3_res) # look at IS.alt column.
    print(set4_res) # look at IS.alt column.
    print(set5_res) # look at IS.alt column.
    res[r_row, "subset_size"] = subset_size
    res[r_row, "graph_name"] = "cit5"
    res[r_row, "IA_top_cit"] = set1_res[,"IS.alt"]
    res[r_row, "IA_top_mma"] = set2_res[,"IS.alt"]
    res[r_row, "IA_top_pku"] = set3_res[,"IS.alt"]
    res[r_row, "IA_top_ref"] = set4_res[,"IS.alt"]
    res[r_row, "IA_top_rand"] = set5_res[,"IS.alt"]
    r_row = r_row + 1

    # ig_cit6: More global parameters
    G = vector(mode="list", length=length(V(ig_cit6_naive)$name))
    names(G) = V(ig_cit6_naive)$name
    adj_mat = as.matrix(get.adjacency(ig_cit6_naive, attr="weight"))
    # Use single node encoding to get node ranks.
    ranks1 = list()
    for (n in 1:length(met_set1)) {
      ind = which(names(G)==met_set1[n])
      ranks1[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set1, num.misses = log2(length(G)))
    }
    names(ranks1) = met_set1
    ranks2 = list()
    for (n in 1:length(met_set2)) {
      ind = which(names(G)==met_set2[n])
      ranks2[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set2, num.misses = log2(length(G)))
    }
    names(ranks2) = met_set2
    ranks3 = list()
    for (n in 1:length(met_set3)) {
      ind = which(names(G)==met_set3[n])
      ranks3[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set3, num.misses = log2(length(G)))
    }
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
  names(ranks3) = met_set3
  ranks4 = list()
  for (n in 1:length(met_set4)) {
    ind = which(names(G)==met_set4[n])
    ranks4[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set4, num.misses = log2(length(G)))
  }
  names(ranks4) = met_set4
  ranks5 = list()
  for (n in 1:length(met_set5)) {
    ind = which(names(G)==met_set5[n])
    ranks5[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set5, num.misses = log2(length(G)))
  }
  names(ranks5) = met_set5
  # Then use the node permutations to outputted to convert them into
bitstrings.
  set1_cit_bs = mle.getPtBSbyK(met_set1, ranks1)
  set2_cit_bs = mle.getPtBSbyK(met_set2, ranks2)
  set3_cit_bs = mle.getPtBSbyK(met_set3, ranks3)
  set4_cit_bs = mle.getPtBSbyK(met_set4, ranks4)
  set5_cit_bs = mle.getPtBSbyK(met_set5, ranks5)
  # Then use the bitstrings to convert into encoding length.
  set1_res = mle.getEncodingLength(set1_cit_bs, NULL, "set1", G)
[length(met_set1),]
  set2_res = mle.getEncodingLength(set2_cit_bs, NULL, "set2", G)
[length(met_set2),]
  set3_res = mle.getEncodingLength(set3_cit_bs, NULL, "set3", G)
[length(met_set3),]
  set4_res = mle.getEncodingLength(set4_cit_bs, NULL, "set4", G)
[length(met_set4),]
  set5_res = mle.getEncodingLength(set5_cit_bs, NULL, "set5", G)
[length(met_set5),]
  print(set1_res) # look at IS.alt column.
  print(set2_res) # look at IS.alt column.
  print(set3_res) # look at IS.alt column.
  print(set4_res) # look at IS.alt column.
  print(set5_res) # look at IS.alt column.
  res[r_row, "subset_size"] = subset_size
  res[r_row, "graph_name"] = "cit6"
  res[r_row, "IA_top_cit"] = set1_res[,"IS.alt"]
  res[r_row, "IA_top_mma"] = set2_res[,"IS.alt"]
  res[r_row, "IA_top_pku"] = set3_res[,"IS.alt"]
  res[r_row, "IA_top_ref"] = set4_res[,"IS.alt"]
  res[r_row, "IA_top_rand"] = set5_res[,"IS.alt"]
  r_row = r_row + 1

  # ig_cit7: More global parameters
  G = vector(mode="list", length=length(V(ig_cit7_naive)$name))
  names(G) = V(ig_cit7_naive)$name
  adj_mat = as.matrix(get.adjacency(ig_cit7_naive, attr="weight"))
  # Use single node encoding to get node ranks.
  ranks1 = list()
  for (n in 1:length(met_set1)) {
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
    for (n in 1:length(met_set2)) {
      ind = which(names(G)==met_set1[n])
      ranks1[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set1, num.misses = log2(length(G)))
    }
    names(ranks1) = met_set1
    ranks2 = list()
    for (n in 1:length(met_set2)) {
      ind = which(names(G)==met_set2[n])
      ranks2[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set2, num.misses = log2(length(G)))
    }
    names(ranks2) = met_set2
    ranks3 = list()
    for (n in 1:length(met_set3)) {
      ind = which(names(G)==met_set3[n])
      ranks3[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set3, num.misses = log2(length(G)))
    }
    names(ranks3) = met_set3
    ranks4 = list()
    for (n in 1:length(met_set4)) {
      ind = which(names(G)==met_set4[n])
      ranks4[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set4, num.misses = log2(length(G)))
    }
    names(ranks4) = met_set4
    ranks5 = list()
    for (n in 1:length(met_set5)) {
      ind = which(names(G)==met_set5[n])
      ranks5[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set5, num.misses = log2(length(G)))
    }
    names(ranks5) = met_set5
    # Then use the node permutations to outputted to convert them into
bitstrings.
    set1_cit_bs = mle.getPtBSbyK(met_set1, ranks1)
    set2_cit_bs = mle.getPtBSbyK(met_set2, ranks2)
    set3_cit_bs = mle.getPtBSbyK(met_set3, ranks3)
    set4_cit_bs = mle.getPtBSbyK(met_set4, ranks4)
    set5_cit_bs = mle.getPtBSbyK(met_set5, ranks5)
    # Then use the bitstrings to convert into encoding length.
    set1_res = mle.getEncodingLength(set1_cit_bs, NULL, "set1", G)
[length(met_set1),]
    set2_res = mle.getEncodingLength(set2_cit_bs, NULL, "set2", G)
[length(met_set2),]
    set3_res = mle.getEncodingLength(set3_cit_bs, NULL, "set3", G)
[length(met_set3),]
    set4_res = mle.getEncodingLength(set4_cit_bs, NULL, "set4", G)
[length(met_set4),]
    set5_res = mle.getEncodingLength(set5_cit_bs, NULL, "set5", G)
[length(met_set5),]
    print(set1_res) # look at IS.alt column.
    print(set2_res) # look at IS.alt column.
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
  print(set3_res) # look at IS.alt column.
  print(set4_res) # look at IS.alt column.
  print(set5_res) # look at IS.alt column.
  res[r_row, "subset_size"] = subset_size
  res[r_row, "graph_name"] = "cit7"
  res[r_row, "IA_top_cit"] = set1_res[,"IS.alt"]
  res[r_row, "IA_top_mma"] = set2_res[,"IS.alt"]
  res[r_row, "IA_top_pku"] = set3_res[,"IS.alt"]
  res[r_row, "IA_top_ref"] = set4_res[,"IS.alt"]
  res[r_row, "IA_top_rand"] = set5_res[,"IS.alt"]
  r_row = r_row + 1

  # ig_cit8: More global parameters
  G = vector(mode="list", length=length(V(ig_cit8_naive)$name))
  names(G) = V(ig_cit8_naive)$name
  adj_mat = as.matrix(get.adjacency(ig_cit8_naive, attr="weight"))
  # Use single node encoding to get node ranks.
  ranks1 = list()
  for (n in 1:length(met_set1)) {
    ind = which(names(G)==met_set1[n])
    ranks1[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set1, num.misses = log2(length(G)))
  }
  names(ranks1) = met_set1
  ranks2 = list()
  for (n in 1:length(met_set2)) {
    ind = which(names(G)==met_set2[n])
    ranks2[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set2, num.misses = log2(length(G)))
  }
  names(ranks2) = met_set2
  ranks3 = list()
  for (n in 1:length(met_set3)) {
    ind = which(names(G)==met_set3[n])
    ranks3[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set3, num.misses = log2(length(G)))
  }
  names(ranks3) = met_set3
  ranks4 = list()
  for (n in 1:length(met_set4)) {
    ind = which(names(G)==met_set4[n])
    ranks4[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set4, num.misses = log2(length(G)))
  }
  names(ranks4) = met_set4
  ranks5 = list()
  for (n in 1:length(met_set5)) {
    ind = which(names(G)==met_set5[n])
    ranks5[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set5, num.misses = log2(length(G)))
  }
  names(ranks5) = met_set5
  # Then use the node permutations to outputted to convert them into
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
bitstrings.
   set1_cit_bs = mle.getPtBSbyK(met_set1, ranks1)
   set2_cit_bs = mle.getPtBSbyK(met_set2, ranks2)
   set3_cit_bs = mle.getPtBSbyK(met_set3, ranks3)
   set4_cit_bs = mle.getPtBSbyK(met_set4, ranks4)
   set5_cit_bs = mle.getPtBSbyK(met_set5, ranks5)
   # Then use the bitstrings to convert into encoding length.
   set1_res = mle.getEncodingLength(set1_cit_bs, NULL, "set1", G)
[length(met_set1),]
   set2_res = mle.getEncodingLength(set2_cit_bs, NULL, "set2", G)
[length(met_set2),]
   set3_res = mle.getEncodingLength(set3_cit_bs, NULL, "set3", G)
[length(met_set3),]
   set4_res = mle.getEncodingLength(set4_cit_bs, NULL, "set4", G)
[length(met_set4),]
   set5_res = mle.getEncodingLength(set5_cit_bs, NULL, "set5", G)
[length(met_set5),]
   print(set1_res) # look at IS.alt column.
   print(set2_res) # look at IS.alt column.
   print(set3_res) # look at IS.alt column.
   print(set4_res) # look at IS.alt column.
   print(set5_res) # look at IS.alt column.
   res[r_row, "subset_size"] = subset_size
   res[r_row, "graph_name"] = "cit8"
   res[r_row, "IA_top_cit"] = set1_res[,"IS.alt"]
   res[r_row, "IA_top_mma"] = set2_res[,"IS.alt"]
   res[r_row, "IA_top_pku"] = set3_res[,"IS.alt"]
   res[r_row, "IA_top_ref"] = set4_res[,"IS.alt"]
   res[r_row, "IA_top_rand"] = set5_res[,"IS.alt"]
   r_row = r_row + 1

   # ig_cit9: More global parameters
   G = vector(mode="list", length=length(V(ig_cit9_naive)$name))
   names(G) = V(ig_cit9_naive)$name
   adj_mat = as.matrix(get.adjacency(ig_cit9_naive, attr="weight"))
   # Use single node encoding to get node ranks.
   ranks1 = list()
   for (n in 1:length(met_set1)) {
     ind = which(names(G)==met_set1[n])
     ranks1[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set1, num.misses = log2(length(G)))
   }
   names(ranks1) = met_set1
   ranks2 = list()
   for (n in 1:length(met_set2)) {
     ind = which(names(G)==met_set2[n])
     ranks2[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set2, num.misses = log2(length(G)))
   }
   names(ranks2) = met_set2
   ranks3 = list()
   for (n in 1:length(met_set3)) {
     ind = which(names(G)==met_set3[n])
     ranks3[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
    ranks3[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set3, num.misses = log2(length(G)))
   }
   names(ranks3) = met_set3
   ranks4 = list()
   for (n in 1:length(met_set4)) {
    ind = which(names(G)==met_set4[n])
    ranks4[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set4, num.misses = log2(length(G)))
   }
   names(ranks4) = met_set4
   ranks5 = list()
   for (n in 1:length(met_set5)) {
    ind = which(names(G)==met_set5[n])
    ranks5[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set5, num.misses = log2(length(G)))
   }
   names(ranks5) = met_set5
   # Then use the node permutations to outputted to convert them into
bitstrings.
   set1_cit_bs = mle.getPtBSbyK(met_set1, ranks1)
   set2_cit_bs = mle.getPtBSbyK(met_set2, ranks2)
   set3_cit_bs = mle.getPtBSbyK(met_set3, ranks3)
   set4_cit_bs = mle.getPtBSbyK(met_set4, ranks4)
   set5_cit_bs = mle.getPtBSbyK(met_set5, ranks5)
   # Then use the bitstrings to convert into encoding length.
   set1_res = mle.getEncodingLength(set1_cit_bs, NULL, "set1", G)
[length(met_set1),]
   set2_res = mle.getEncodingLength(set2_cit_bs, NULL, "set2", G)
[length(met_set2),]
   set3_res = mle.getEncodingLength(set3_cit_bs, NULL, "set3", G)
[length(met_set3),]
   set4_res = mle.getEncodingLength(set4_cit_bs, NULL, "set4", G)
[length(met_set4),]
   set5_res = mle.getEncodingLength(set5_cit_bs, NULL, "set5", G)
[length(met_set5),]
   print(set1_res) # look at IS.alt column.
   print(set2_res) # look at IS.alt column.
   print(set3_res) # look at IS.alt column.
   print(set4_res) # look at IS.alt column.
   print(set5_res) # look at IS.alt column.
   res[r_row, "subset_size"] = subset_size
   res[r_row, "graph_name"] = "cit9"
   res[r_row, "IA_top_cit"] = set1_res[,"IS.alt"]
   res[r_row, "IA_top_mma"] = set2_res[,"IS.alt"]
   res[r_row, "IA_top_pku"] = set3_res[,"IS.alt"]
   res[r_row, "IA_top_ref"] = set4_res[,"IS.alt"]
   res[r_row, "IA_top_rand"] = set5_res[,"IS.alt"]
   r_row = r_row + 1
  }

  if (domma) {
   # ig_mma: Re-set global parameters for mma network.
   G = vector(mode="list", length=length(V(ig_mma_naive)$name))
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
  names(G) = V(ig_mma_naive)$name
  adj_mat = as.matrix(get.adjacency(ig_mma_naive, attr="weight"))
  # Use single node encoding to get node ranks
  ranks1 = list()
  for (n in 1:length(met_set1)) {
    ind = which(names(G)==met_set1[n])
    ranks1[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set1, num.misses = log2(length(G)))
  }
  names(ranks1) = met_set1
  ranks2 = list()
  for (n in 1:length(met_set2)) {
    ind = which(names(G)==met_set2[n])
    ranks2[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set2, num.misses = log2(length(G)))
  }
  names(ranks2) = met_set2
  ranks3 = list()
  for (n in 1:length(met_set3)) {
    ind = which(names(G)==met_set3[n])
    ranks3[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set3, num.misses = log2(length(G)))
  }
  names(ranks3) = met_set3
  ranks4 = list()
  for (n in 1:length(met_set4)) {
    ind = which(names(G)==met_set4[n])
    ranks4[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set4, num.misses = log2(length(G)))
  }
  names(ranks4) = met_set4
  ranks5 = list()
  for (n in 1:length(met_set5)) {
    ind = which(names(G)==met_set5[n])
    ranks5[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set5, num.misses = log2(length(G)))
  }
  names(ranks5) = met_set5
  # Then use the node permutations to outputted to convert them into
bitstrings.
  set1_mma_bs = mle.getPtBSbyK(met_set1, ranks1)
  set2_mma_bs = mle.getPtBSbyK(met_set2, ranks2)
  set3_mma_bs = mle.getPtBSbyK(met_set3, ranks3)
  set4_mma_bs = mle.getPtBSbyK(met_set4, ranks4)
  set5_mma_bs = mle.getPtBSbyK(met_set5, ranks5)
  # Then use the bitstrings to convert into encoding length.
  set1_res = mle.getEncodingLength(set1_mma_bs, NULL, "set1", G)
[length(met_set1),]
  set2_res = mle.getEncodingLength(set2_mma_bs, NULL, "set2", G)
[length(met_set2),]
  set3_res = mle.getEncodingLength(set3_mma_bs, NULL, "set3", G)
[length(met_set3),]
  set4_res = mle.getEncodingLength(set4_mma_bs, NULL, "set4", G)
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
[length(met_set4),]
   set5_res = mle.getEncodingLength(set5_mma_bs, NULL, "set5", G)
[length(met_set5),]
   print(set1_res) # look at IS.alt column.
   print(set2_res) # look at IS.alt column.
   print(set3_res) # look at IS.alt column.
   print(set4_res) # look at IS.alt column.
   print(set5_res) # look at IS.alt column.
   res[r_row, "subset_size"] = subset_size
   res[r_row, "graph_name"] = "mma"
   res[r_row, "IA_top_cit"] = set1_res[,"IS.alt"]
   res[r_row, "IA_top_mma"] = set2_res[,"IS.alt"]
   res[r_row, "IA_top_pku"] = set3_res[,"IS.alt"]
   res[r_row, "IA_top_ref"] = set4_res[,"IS.alt"]
   res[r_row, "IA_top_rand"] = set5_res[,"IS.alt"]
   r_row = r_row + 1

   # ig_mma1: Re-set global parameters for mma network.
   G = vector(mode="list", length=length(V(ig_mma1_naive)$name))
   names(G) = V(ig_mma1_naive)$name
   adj_mat = as.matrix(get.adjacency(ig_mma1_naive, attr="weight"))
   # Use single node encoding to get node ranks
   ranks1 = list()
   for (n in 1:length(met_set1)) {
     ind = which(names(G)==met_set1[n])
     ranks1[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set1, num.misses = log2(length(G)))
   }
   names(ranks1) = met_set1
   ranks2 = list()
   for (n in 1:length(met_set2)) {
     ind = which(names(G)==met_set2[n])
     ranks2[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set2, num.misses = log2(length(G)))
   }
   names(ranks2) = met_set2
   ranks3 = list()
   for (n in 1:length(met_set3)) {
     ind = which(names(G)==met_set3[n])
     ranks3[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set3, num.misses = log2(length(G)))
   }
   names(ranks3) = met_set3
   ranks4 = list()
   for (n in 1:length(met_set4)) {
     ind = which(names(G)==met_set4[n])
     ranks4[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set4, num.misses = log2(length(G)))
   }
   names(ranks4) = met_set4
   ranks5 = list()
   for (n in 1:length(met_set5)) {
     ind = which(names(G)==met_set5[n])
     ranks5[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
    ranks5[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set5, num.misses = log2(length(G)))
    }
    names(ranks5) = met_set5
    # Then use the node permutations to outputted to convert them into
bitstrings.
    set1_mma_bs = mle.getPtBSbyK(met_set1, ranks1)
    set2_mma_bs = mle.getPtBSbyK(met_set2, ranks2)
    set3_mma_bs = mle.getPtBSbyK(met_set3, ranks3)
    set4_mma_bs = mle.getPtBSbyK(met_set4, ranks4)
    set5_mma_bs = mle.getPtBSbyK(met_set5, ranks5)
    # Then use the bitstrings to convert into encoding length.
    set1_res = mle.getEncodingLength(set1_mma_bs, NULL, "set1", G)
[length(met_set1),]
    set2_res = mle.getEncodingLength(set2_mma_bs, NULL, "set2", G)
[length(met_set2),]
    set3_res = mle.getEncodingLength(set3_mma_bs, NULL, "set3", G)
[length(met_set3),]
    set4_res = mle.getEncodingLength(set4_mma_bs, NULL, "set4", G)
[length(met_set4),]
    set5_res = mle.getEncodingLength(set5_mma_bs, NULL, "set5", G)
[length(met_set5),]
    print(set1_res) # look at IS.alt column.
    print(set2_res) # look at IS.alt column.
    print(set3_res) # look at IS.alt column.
    print(set4_res) # look at IS.alt column.
    print(set5_res) # look at IS.alt column.
    res[r_row, "subset_size"] = subset_size
    res[r_row, "graph_name"] = "mma1"
    res[r_row, "IA_top_cit"] = set1_res[,"IS.alt"]
    res[r_row, "IA_top_mma"] = set2_res[,"IS.alt"]
    res[r_row, "IA_top_pku"] = set3_res[,"IS.alt"]
    res[r_row, "IA_top_ref"] = set4_res[,"IS.alt"]
    res[r_row, "IA_top_rand"] = set5_res[,"IS.alt"]
    r_row = r_row + 1

    # ig_mma2: Re-set global parameters for mma network.
    G = vector(mode="list", length=length(V(ig_mma2_naive)$name))
    names(G) = V(ig_mma2_naive)$name
    adj_mat = as.matrix(get.adjacency(ig_mma2_naive, attr="weight"))
    # Use single node encoding to get node ranks
    ranks1 = list()
    for (n in 1:length(met_set1)) {
      ind = which(names(G)==met_set1[n])
      ranks1[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set1, num.misses = log2(length(G)))
    }
    names(ranks1) = met_set1
    ranks2 = list()
    for (n in 1:length(met_set2)) {
      ind = which(names(G)==met_set2[n])
      ranks2[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set2, num.misses = log2(length(G)))
    }
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
  names(ranks2) = met_set2
  ranks3 = list()
  for (n in 1:length(met_set3)) {
    ind = which(names(G)==met_set3[n])
    ranks3[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set3, num.misses = log2(length(G)))
  }
  names(ranks3) = met_set3
  ranks4 = list()
  for (n in 1:length(met_set4)) {
    ind = which(names(G)==met_set4[n])
    ranks4[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set4, num.misses = log2(length(G)))
  }
  names(ranks4) = met_set4
  ranks5 = list()
  for (n in 1:length(met_set5)) {
    ind = which(names(G)==met_set5[n])
    ranks5[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set5, num.misses = log2(length(G)))
  }
  names(ranks5) = met_set5
  # Then use the node permutations to outputted to convert them into
bitstrings.
  set1_mma_bs = mle.getPtBSbyK(met_set1, ranks1)
  set2_mma_bs = mle.getPtBSbyK(met_set2, ranks2)
  set3_mma_bs = mle.getPtBSbyK(met_set3, ranks3)
  set4_mma_bs = mle.getPtBSbyK(met_set4, ranks4)
  set5_mma_bs = mle.getPtBSbyK(met_set5, ranks5)
  # Then use the bitstrings to convert into encoding length.
  set1_res = mle.getEncodingLength(set1_mma_bs, NULL, "set1", G)
[length(met_set1),]
  set2_res = mle.getEncodingLength(set2_mma_bs, NULL, "set2", G)
[length(met_set2),]
  set3_res = mle.getEncodingLength(set3_mma_bs, NULL, "set3", G)
[length(met_set3),]
  set4_res = mle.getEncodingLength(set4_mma_bs, NULL, "set4", G)
[length(met_set4),]
  set5_res = mle.getEncodingLength(set5_mma_bs, NULL, "set5", G)
[length(met_set5),]
  print(set1_res) # look at IS.alt column.
  print(set2_res) # look at IS.alt column.
  print(set3_res) # look at IS.alt column.
  print(set4_res) # look at IS.alt column.
  print(set5_res) # look at IS.alt column.
  res[r_row, "subset_size"] = subset_size
  res[r_row, "graph_name"] = "mma2"
  res[r_row, "IA_top_cit"] = set1_res[,"IS.alt"]
  res[r_row, "IA_top_mma"] = set2_res[,"IS.alt"]
  res[r_row, "IA_top_pku"] = set3_res[,"IS.alt"]
  res[r_row, "IA_top_ref"] = set4_res[,"IS.alt"]
  res[r_row, "IA_top_rand"] = set5_res[,"IS.alt"]
  r_row = r_row + 1
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
    # ig_mma3: Re-set global parameters for mma network.
    G = vector(mode="list", length=length(V(ig_mma3_naive)$name))
    names(G) = V(ig_mma3_naive)$name
    adj_mat = as.matrix(get.adjacency(ig_mma3_naive, attr="weight"))
    # Use single node encoding to get node ranks
    ranks1 = list()
    for (n in 1:length(met_set1)) {
      ind = which(names(G)==met_set1[n])
      ranks1[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set1, num.misses = log2(length(G)))
    }
    names(ranks1) = met_set1
    ranks2 = list()
    for (n in 1:length(met_set2)) {
      ind = which(names(G)==met_set2[n])
      ranks2[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set2, num.misses = log2(length(G)))
    }
    names(ranks2) = met_set2
    ranks3 = list()
    for (n in 1:length(met_set3)) {
      ind = which(names(G)==met_set3[n])
      ranks3[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set3, num.misses = log2(length(G)))
    }
    names(ranks3) = met_set3
    ranks4 = list()
    for (n in 1:length(met_set4)) {
      ind = which(names(G)==met_set4[n])
      ranks4[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set4, num.misses = log2(length(G)))
    }
    names(ranks4) = met_set4
    ranks5 = list()
    for (n in 1:length(met_set5)) {
      ind = which(names(G)==met_set5[n])
      ranks5[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set5, num.misses = log2(length(G)))
    }
    names(ranks5) = met_set5
    # Then use the node permutations to outputted to convert them into
bitstrings.
    set1_mma_bs = mle.getPtBSbyK(met_set1, ranks1)
    set2_mma_bs = mle.getPtBSbyK(met_set2, ranks2)
    set3_mma_bs = mle.getPtBSbyK(met_set3, ranks3)
    set4_mma_bs = mle.getPtBSbyK(met_set4, ranks4)
    set5_mma_bs = mle.getPtBSbyK(met_set5, ranks5)
    # Then use the bitstrings to convert into encoding length.
    set1_res = mle.getEncodingLength(set1_mma_bs, NULL, "set1", G)
[length(met_set1),]
    set2_res = mle.getEncodingLength(set2_mma_bs, NULL, "set2", G)
[length(met_set2),]
    set3_res = mle.getEncodingLength(set3_mma_bs, NULL, "set3", G)
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
   set3_res = mle.getEncodingLength(set3_mma_bs, NULL, "set3", G)
[length(met_set3),]
   set4_res = mle.getEncodingLength(set4_mma_bs, NULL, "set4", G)
[length(met_set4),]
   set5_res = mle.getEncodingLength(set5_mma_bs, NULL, "set5", G)
[length(met_set5),]
   print(set1_res) # look at IS.alt column.
   print(set2_res) # look at IS.alt column.
   print(set3_res) # look at IS.alt column.
   print(set4_res) # look at IS.alt column.
   print(set5_res) # look at IS.alt column.
   res[r_row, "subset_size"] = subset_size
   res[r_row, "graph_name"] = "mma3"
   res[r_row, "IA_top_cit"] = set1_res[,"IS.alt"]
   res[r_row, "IA_top_mma"] = set2_res[,"IS.alt"]
   res[r_row, "IA_top_pku"] = set3_res[,"IS.alt"]
   res[r_row, "IA_top_ref"] = set4_res[,"IS.alt"]
   res[r_row, "IA_top_rand"] = set5_res[,"IS.alt"]
   r_row = r_row + 1

   # ig_mma4: Re-set global parameters for mma network.
   G = vector(mode="list", length=length(V(ig_mma4_naive)$name))
   names(G) = V(ig_mma4_naive)$name
   adj_mat = as.matrix(get.adjacency(ig_mma4_naive, attr="weight"))
   # Use single node encoding to get node ranks
   ranks1 = list()
   for (n in 1:length(met_set1)) {
     ind = which(names(G)==met_set1[n])
     ranks1[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set1, num.misses = log2(length(G)))
   }
   names(ranks1) = met_set1
   ranks2 = list()
   for (n in 1:length(met_set2)) {
     ind = which(names(G)==met_set2[n])
     ranks2[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set2, num.misses = log2(length(G)))
   }
   names(ranks2) = met_set2
   ranks3 = list()
   for (n in 1:length(met_set3)) {
     ind = which(names(G)==met_set3[n])
     ranks3[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set3, num.misses = log2(length(G)))
   }
   names(ranks3) = met_set3
   ranks4 = list()
   for (n in 1:length(met_set4)) {
     ind = which(names(G)==met_set4[n])
     ranks4[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set4, num.misses = log2(length(G)))
   }
   names(ranks4) = met_set4
   ranks5 = list()
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
  for (n in 1:length(met_set5)) {
    ind = which(names(G)==met_set5[n])
    ranks5[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set5, num.misses = log2(length(G)))
  }
  names(ranks5) = met_set5
  # Then use the node permutations to outputted to convert them into
bitstrings.
  set1_mma_bs = mle.getPtBSbyK(met_set1, ranks1)
  set2_mma_bs = mle.getPtBSbyK(met_set2, ranks2)
  set3_mma_bs = mle.getPtBSbyK(met_set3, ranks3)
  set4_mma_bs = mle.getPtBSbyK(met_set4, ranks4)
  set5_mma_bs = mle.getPtBSbyK(met_set5, ranks5)
  # Then use the bitstrings to convert into encoding length.
  set1_res = mle.getEncodingLength(set1_mma_bs, NULL, "set1", G)
[length(met_set1),]
  set2_res = mle.getEncodingLength(set2_mma_bs, NULL, "set2", G)
[length(met_set2),]
  set3_res = mle.getEncodingLength(set3_mma_bs, NULL, "set3", G)
[length(met_set3),]
  set4_res = mle.getEncodingLength(set4_mma_bs, NULL, "set4", G)
[length(met_set4),]
  set5_res = mle.getEncodingLength(set5_mma_bs, NULL, "set5", G)
[length(met_set5),]
  print(set1_res) # look at IS.alt column.
  print(set2_res) # look at IS.alt column.
  print(set3_res) # look at IS.alt column.
  print(set4_res) # look at IS.alt column.
  print(set5_res) # look at IS.alt column.
  res[r_row, "subset_size"] = subset_size
  res[r_row, "graph_name"] = "mma4"
  res[r_row, "IA_top_cit"] = set1_res[,"IS.alt"]
  res[r_row, "IA_top_mma"] = set2_res[,"IS.alt"]
  res[r_row, "IA_top_pku"] = set3_res[,"IS.alt"]
  res[r_row, "IA_top_ref"] = set4_res[,"IS.alt"]
  res[r_row, "IA_top_rand"] = set5_res[,"IS.alt"]
  r_row = r_row + 1

  # ig_mma5: Re-set global parameters for mma network.
  G = vector(mode="list", length=length(V(ig_mma5_naive)$name))
  names(G) = V(ig_mma5_naive)$name
  adj_mat = as.matrix(get.adjacency(ig_mma5_naive, attr="weight"))
  # Use single node encoding to get node ranks
  ranks1 = list()
  for (n in 1:length(met_set1)) {
    ind = which(names(G)==met_set1[n])
    ranks1[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set1, num.misses = log2(length(G)))
  }
  names(ranks1) = met_set1
  ranks2 = list()
  for (n in 1:length(met_set2)) {
    ind = which(names(G)==met_set2[n])
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
    ranks2[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set2, num.misses = log2(length(G)))
  }
  names(ranks2) = met_set2
  ranks3 = list()
  for (n in 1:length(met_set3)) {
    ind = which(names(G)==met_set3[n])
    ranks3[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set3, num.misses = log2(length(G)))
  }
  names(ranks3) = met_set3
  ranks4 = list()
  for (n in 1:length(met_set4)) {
    ind = which(names(G)==met_set4[n])
    ranks4[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set4, num.misses = log2(length(G)))
  }
  names(ranks4) = met_set4
  ranks5 = list()
  for (n in 1:length(met_set5)) {
    ind = which(names(G)==met_set5[n])
    ranks5[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set5, num.misses = log2(length(G)))
  }
  names(ranks5) = met_set5
  # Then use the node permutations to outputted to convert them into
bitstrings.
  set1_mma_bs = mle.getPtBSbyK(met_set1, ranks1)
  set2_mma_bs = mle.getPtBSbyK(met_set2, ranks2)
  set3_mma_bs = mle.getPtBSbyK(met_set3, ranks3)
  set4_mma_bs = mle.getPtBSbyK(met_set4, ranks4)
  set5_mma_bs = mle.getPtBSbyK(met_set5, ranks5)
  # Then use the bitstrings to convert into encoding length.
  set1_res = mle.getEncodingLength(set1_mma_bs, NULL, "set1", G)
[length(met_set1),]
  set2_res = mle.getEncodingLength(set2_mma_bs, NULL, "set2", G)
[length(met_set2),]
  set3_res = mle.getEncodingLength(set3_mma_bs, NULL, "set3", G)
[length(met_set3),]
  set4_res = mle.getEncodingLength(set4_mma_bs, NULL, "set4", G)
[length(met_set4),]
  set5_res = mle.getEncodingLength(set5_mma_bs, NULL, "set5", G)
[length(met_set5),]
  print(set1_res) # look at IS.alt column.
  print(set2_res) # look at IS.alt column.
  print(set3_res) # look at IS.alt column.
  print(set4_res) # look at IS.alt column.
  print(set5_res) # look at IS.alt column.
  res[r_row, "subset_size"] = subset_size
  res[r_row, "graph_name"] = "mma5"
  res[r_row, "IA_top_cit"] = set1_res[,"IS.alt"]
  res[r_row, "IA_top_mma"] = set2_res[,"IS.alt"]
  res[r_row, "IA_top_pku"] = set3_res[,"IS.alt"]
  res[r_row, "IA_top_ref"] = set4_res[,"IS.alt"]
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
  res[r_row, "IA_top_ref"] = set4_res[,"IS.alt"]
  res[r_row, "IA_top_rand"] = set5_res[,"IS.alt"]
  r_row = r_row + 1

  # ig_mma6: Re-set global parameters for mma network.
  G = vector(mode="list", length=length(V(ig_mma6_naive)$name))
  names(G) = V(ig_mma6_naive)$name
  adj_mat = as.matrix(get.adjacency(ig_mma6_naive, attr="weight"))
  # Use single node encoding to get node ranks
  ranks1 = list()
  for (n in 1:length(met_set1)) {
    ind = which(names(G)==met_set1[n])
    ranks1[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set1, num.misses = log2(length(G)))
  }
  names(ranks1) = met_set1
  ranks2 = list()
  for (n in 1:length(met_set2)) {
    ind = which(names(G)==met_set2[n])
    ranks2[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set2, num.misses = log2(length(G)))
  }
  names(ranks2) = met_set2
  ranks3 = list()
  for (n in 1:length(met_set3)) {
    ind = which(names(G)==met_set3[n])
    ranks3[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set3, num.misses = log2(length(G)))
  }
  names(ranks3) = met_set3
  ranks4 = list()
  for (n in 1:length(met_set4)) {
    ind = which(names(G)==met_set4[n])
    ranks4[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set4, num.misses = log2(length(G)))
  }
  names(ranks4) = met_set4
  ranks5 = list()
  for (n in 1:length(met_set5)) {
    ind = which(names(G)==met_set5[n])
    ranks5[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set5, num.misses = log2(length(G)))
  }
  names(ranks5) = met_set5
  # Then use the node permutations to outputted to convert them into
bitstrings.
  set1_mma_bs = mle.getPtBSbyK(met_set1, ranks1)
  set2_mma_bs = mle.getPtBSbyK(met_set2, ranks2)
  set3_mma_bs = mle.getPtBSbyK(met_set3, ranks3)
  set4_mma_bs = mle.getPtBSbyK(met_set4, ranks4)
  set5_mma_bs = mle.getPtBSbyK(met_set5, ranks5)
  # Then use the bitstrings to convert into encoding length.
  set1_res = mle.getEncodingLength(set1_mma_bs, NULL, "set1", G)
[length(met_set1),]
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
   set2_res = mle.getEncodingLength(set2_mma_bs, NULL, "set2", G)
[length(met_set2),]
   set3_res = mle.getEncodingLength(set3_mma_bs, NULL, "set3", G)
[length(met_set3),]
   set4_res = mle.getEncodingLength(set4_mma_bs, NULL, "set4", G)
[length(met_set4),]
   set5_res = mle.getEncodingLength(set5_mma_bs, NULL, "set5", G)
[length(met_set5),]
   print(set1_res) # look at IS.alt column.
   print(set2_res) # look at IS.alt column.
   print(set3_res) # look at IS.alt column.
   print(set4_res) # look at IS.alt column.
   print(set5_res) # look at IS.alt column.
   res[r_row, "subset_size"] = subset_size
   res[r_row, "graph_name"] = "mma6"
   res[r_row, "IA_top_cit"] = set1_res[,"IS.alt"]
   res[r_row, "IA_top_mma"] = set2_res[,"IS.alt"]
   res[r_row, "IA_top_pku"] = set3_res[,"IS.alt"]
   res[r_row, "IA_top_ref"] = set4_res[,"IS.alt"]
   res[r_row, "IA_top_rand"] = set5_res[,"IS.alt"]
   r_row = r_row + 1

   # ig_mma7: Re-set global parameters for mma network.
   G = vector(mode="list", length=length(V(ig_mma7_naive)$name))
   names(G) = V(ig_mma7_naive)$name
   adj_mat = as.matrix(get.adjacency(ig_mma7_naive, attr="weight"))
   # Use single node encoding to get node ranks
   ranks1 = list()
   for (n in 1:length(met_set1)) {
     ind = which(names(G)==met_set1[n])
     ranks1[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set1, num.misses = log2(length(G)))
   }
   names(ranks1) = met_set1
   ranks2 = list()
   for (n in 1:length(met_set2)) {
     ind = which(names(G)==met_set2[n])
     ranks2[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set2, num.misses = log2(length(G)))
   }
   names(ranks2) = met_set2
   ranks3 = list()
   for (n in 1:length(met_set3)) {
     ind = which(names(G)==met_set3[n])
     ranks3[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set3, num.misses = log2(length(G)))
   }
   names(ranks3) = met_set3
   ranks4 = list()
   for (n in 1:length(met_set4)) {
     ind = which(names(G)==met_set4[n])
     ranks4[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set4, num.misses = log2(length(G)))
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
  }
  names(ranks4) = met_set4
  ranks5 = list()
  for (n in 1:length(met_set5)) {
    ind = which(names(G)==met_set5[n])
    ranks5[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set5, num.misses = log2(length(G)))
  }
  names(ranks5) = met_set5
  # Then use the node permutations to outputted to convert them into
bitstrings.
  set1_mma_bs = mle.getPtBSbyK(met_set1, ranks1)
  set2_mma_bs = mle.getPtBSbyK(met_set2, ranks2)
  set3_mma_bs = mle.getPtBSbyK(met_set3, ranks3)
  set4_mma_bs = mle.getPtBSbyK(met_set4, ranks4)
  set5_mma_bs = mle.getPtBSbyK(met_set5, ranks5)
  # Then use the bitstrings to convert into encoding length.
  set1_res = mle.getEncodingLength(set1_mma_bs, NULL, "set1", G)
[length(met_set1),]
  set2_res = mle.getEncodingLength(set2_mma_bs, NULL, "set2", G)
[length(met_set2),]
  set3_res = mle.getEncodingLength(set3_mma_bs, NULL, "set3", G)
[length(met_set3),]
  set4_res = mle.getEncodingLength(set4_mma_bs, NULL, "set4", G)
[length(met_set4),]
  set5_res = mle.getEncodingLength(set5_mma_bs, NULL, "set5", G)
[length(met_set5),]
  print(set1_res) # look at IS.alt column.
  print(set2_res) # look at IS.alt column.
  print(set3_res) # look at IS.alt column.
  print(set4_res) # look at IS.alt column.
  print(set5_res) # look at IS.alt column.
  res[r_row, "subset_size"] = subset_size
  res[r_row, "graph_name"] = "mma7"
  res[r_row, "IA_top_cit"] = set1_res[,"IS.alt"]
  res[r_row, "IA_top_mma"] = set2_res[,"IS.alt"]
  res[r_row, "IA_top_pku"] = set3_res[,"IS.alt"]
  res[r_row, "IA_top_ref"] = set4_res[,"IS.alt"]
  res[r_row, "IA_top_rand"] = set5_res[,"IS.alt"]
  r_row = r_row + 1

  # ig_mma8: Re-set global parameters for mma network.
  G = vector(mode="list", length=length(V(ig_mma8_naive)$name))
  names(G) = V(ig_mma8_naive)$name
  adj_mat = as.matrix(get.adjacency(ig_mma8_naive, attr="weight"))
  # Use single node encoding to get node ranks
  ranks1 = list()
  for (n in 1:length(met_set1)) {
    ind = which(names(G)==met_set1[n])
    ranks1[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set1, num.misses = log2(length(G)))
  }
  names(ranks1) = met_set1
  ranks2 = list()
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
  ranks2 = list()
  for (n in 1:length(met_set2)) {
    ind = which(names(G)==met_set2[n])
    ranks2[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set2, num.misses = log2(length(G)))
  }
  names(ranks2) = met_set2
  ranks3 = list()
  for (n in 1:length(met_set3)) {
    ind = which(names(G)==met_set3[n])
    ranks3[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set3, num.misses = log2(length(G)))
  }
  names(ranks3) = met_set3
  ranks4 = list()
  for (n in 1:length(met_set4)) {
    ind = which(names(G)==met_set4[n])
    ranks4[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set4, num.misses = log2(length(G)))
  }
  names(ranks4) = met_set4
  ranks5 = list()
  for (n in 1:length(met_set5)) {
    ind = which(names(G)==met_set5[n])
    ranks5[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set5, num.misses = log2(length(G)))
  }
  names(ranks5) = met_set5
  # Then use the node permutations to outputted to convert them into
bitstrings.
  set1_mma_bs = mle.getPtBSbyK(met_set1, ranks1)
  set2_mma_bs = mle.getPtBSbyK(met_set2, ranks2)
  set3_mma_bs = mle.getPtBSbyK(met_set3, ranks3)
  set4_mma_bs = mle.getPtBSbyK(met_set4, ranks4)
  set5_mma_bs = mle.getPtBSbyK(met_set5, ranks5)
  # Then use the bitstrings to convert into encoding length.
  set1_res = mle.getEncodingLength(set1_mma_bs, NULL, "set1", G)
[length(met_set1),]
  set2_res = mle.getEncodingLength(set2_mma_bs, NULL, "set2", G)
[length(met_set2),]
  set3_res = mle.getEncodingLength(set3_mma_bs, NULL, "set3", G)
[length(met_set3),]
  set4_res = mle.getEncodingLength(set4_mma_bs, NULL, "set4", G)
[length(met_set4),]
  set5_res = mle.getEncodingLength(set5_mma_bs, NULL, "set5", G)
[length(met_set5),]
  print(set1_res) # look at IS.alt column.
  print(set2_res) # look at IS.alt column.
  print(set3_res) # look at IS.alt column.
  print(set4_res) # look at IS.alt column.
  print(set5_res) # look at IS.alt column.
  res[r_row, "subset_size"] = subset_size
  res[r_row, "graph_name"] = "mma8"
  res[r_row, "IA_top_cit"] = set1_res["IS.alt"]
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
  res[r_row, "IA_top_mma"] = set2_res[,"IS.alt"]
  res[r_row, "IA_top_pku"] = set3_res[,"IS.alt"]
  res[r_row, "IA_top_ref"] = set4_res[,"IS.alt"]
  res[r_row, "IA_top_rand"] = set5_res[,"IS.alt"]
  r_row = r_row + 1

  # ig_mma9: Re-set global parameters for mma network.
  G = vector(mode="list", length=length(V(ig_mma9_naive)$name))
  names(G) = V(ig_mma9_naive)$name
  adj_mat = as.matrix(get.adjacency(ig_mma9_naive, attr="weight"))
  # Use single node encoding to get node ranks
  ranks1 = list()
  for (n in 1:length(met_set1)) {
    ind = which(names(G)==met_set1[n])
    ranks1[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set1, num.misses = log2(length(G)))
  }
  names(ranks1) = met_set1
  ranks2 = list()
  for (n in 1:length(met_set2)) {
    ind = which(names(G)==met_set2[n])
    ranks2[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set2, num.misses = log2(length(G)))
  }
  names(ranks2) = met_set2
  ranks3 = list()
  for (n in 1:length(met_set3)) {
    ind = which(names(G)==met_set3[n])
    ranks3[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set3, num.misses = log2(length(G)))
  }
  names(ranks3) = met_set3
  ranks4 = list()
  for (n in 1:length(met_set4)) {
    ind = which(names(G)==met_set4[n])
    ranks4[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set4, num.misses = log2(length(G)))
  }
  names(ranks4) = met_set4
  ranks5 = list()
  for (n in 1:length(met_set5)) {
    ind = which(names(G)==met_set5[n])
    ranks5[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set5, num.misses = log2(length(G)))
  }
  names(ranks5) = met_set5
  # Then use the node permutations to outputted to convert them into
bitstrings.
  set1_mma_bs = mle.getPtBSbyK(met_set1, ranks1)
  set2_mma_bs = mle.getPtBSbyK(met_set2, ranks2)
  set3_mma_bs = mle.getPtBSbyK(met_set3, ranks3)
  set4_mma_bs = mle.getPtBSbyK(met_set4, ranks4)
  set5_mma_bs = mle.getPtBSbyK(met_set5, ranks5)
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
    # Then use the bitstrings to convert into encoding length.
    set1_res = mle.getEncodingLength(set1_mma_bs, NULL, "set1", G)
[length(met_set1),]
    set2_res = mle.getEncodingLength(set2_mma_bs, NULL, "set2", G)
[length(met_set2),]
    set3_res = mle.getEncodingLength(set3_mma_bs, NULL, "set3", G)
[length(met_set3),]
    set4_res = mle.getEncodingLength(set4_mma_bs, NULL, "set4", G)
[length(met_set4),]
    set5_res = mle.getEncodingLength(set5_mma_bs, NULL, "set5", G)
[length(met_set5),]
    print(set1_res) # look at IS.alt column.
    print(set2_res) # look at IS.alt column.
    print(set3_res) # look at IS.alt column.
    print(set4_res) # look at IS.alt column.
    print(set5_res) # look at IS.alt column.
    res[r_row, "subset_size"] = subset_size
    res[r_row, "graph_name"] = "mma9"
    res[r_row, "IA_top_cit"] = set1_res[,"IS.alt"]
    res[r_row, "IA_top_mma"] = set2_res[,"IS.alt"]
    res[r_row, "IA_top_pku"] = set3_res[,"IS.alt"]
    res[r_row, "IA_top_ref"] = set4_res[,"IS.alt"]
    res[r_row, "IA_top_rand"] = set5_res[,"IS.alt"]
    r_row = r_row + 1
  }

  if (dopku) {
    # ig_pku: Re-set global parameters for pku network.
    G = vector(mode="list", length=length(V(ig_pku_naive)$name))
    names(G) = V(ig_pku_naive)$name
    adj_mat = as.matrix(get.adjacency(ig_pku_naive, attr="weight"))
    # Use single node encoding to get node ranks
    ranks1 = list()
    for (n in 1:length(met_set1)) {
      ind = which(names(G)==met_set1[n])
      ranks1[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set1, num.misses = log2(length(G)))
    }
    names(ranks1) = met_set1
    ranks2 = list()
    for (n in 1:length(met_set2)) {
      ind = which(names(G)==met_set2[n])
      ranks2[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set2, num.misses = log2(length(G)))
    }
    names(ranks2) = met_set2
    ranks3 = list()
    for (n in 1:length(met_set3)) {
      ind = which(names(G)==met_set3[n])
      ranks3[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set3, num.misses = log2(length(G)))
    }
    names(ranks3) = met_set3
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
  ranks4 = list()
  for (n in 1:length(met_set4)) {
    ind = which(names(G)==met_set4[n])
    ranks4[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set4, num.misses = log2(length(G)))
  }
  names(ranks4) = met_set4
  ranks5 = list()
  for (n in 1:length(met_set5)) {
    ind = which(names(G)==met_set5[n])
    ranks5[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set5, num.misses = log2(length(G)))
  }
  names(ranks5) = met_set5
  # Then use the node permutations to outputted to convert them into
bitstrings.
  set1_pku_bs = mle.getPtBSbyK(met_set1, ranks1)
  set2_pku_bs = mle.getPtBSbyK(met_set2, ranks2)
  set3_pku_bs = mle.getPtBSbyK(met_set3, ranks3)
  set4_pku_bs = mle.getPtBSbyK(met_set4, ranks4)
  set5_pku_bs = mle.getPtBSbyK(met_set5, ranks5)
  # Then use the bitstrings to convert into encoding length.
  set1_res = mle.getEncodingLength(set1_pku_bs, NULL, "set1", G)
[length(met_set1),]
  set2_res = mle.getEncodingLength(set2_pku_bs, NULL, "set2", G)
[length(met_set2),]
  set3_res = mle.getEncodingLength(set3_pku_bs, NULL, "set3", G)
[length(met_set3),]
  set4_res = mle.getEncodingLength(set4_pku_bs, NULL, "set4", G)
[length(met_set4),]
  set5_res = mle.getEncodingLength(set5_pku_bs, NULL, "set5", G)
[length(met_set5),]
  print(set1_res) # look at IS.alt column.
  print(set2_res) # look at IS.alt column.
  print(set3_res) # look at IS.alt column.
  print(set4_res) # look at IS.alt column.
  print(set5_res) # look at IS.alt column.
  res[r_row, "subset_size"] = subset_size
  res[r_row, "graph_name"] = "pku"
  res[r_row, "IA_top_cit"] = set1_res[,"IS.alt"]
  res[r_row, "IA_top_mma"] = set2_res[,"IS.alt"]
  res[r_row, "IA_top_pku"] = set3_res[,"IS.alt"]
  res[r_row, "IA_top_ref"] = set4_res[,"IS.alt"]
  res[r_row, "IA_top_rand"] = set5_res[,"IS.alt"]
  r_row = r_row + 1

  # ig_pku1: Re-set global parameters for pku network.
  G = vector(mode="list", length=length(V(ig_pku1_naive)$name))
  names(G) = V(ig_pku1_naive)$name
  adj_mat = as.matrix(get.adjacency(ig_pku1_naive, attr="weight"))
  # Use single node encoding to get node ranks
  ranks1 = list()
  for (n in 1:length(met_set1)) {
    ind = which(names(G)==met_set1[n])
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
    ranks1[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set1, num.misses = log2(length(G)))
  }
  names(ranks1) = met_set1
  ranks2 = list()
  for (n in 1:length(met_set2)) {
    ind = which(names(G)==met_set2[n])
    ranks2[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set2, num.misses = log2(length(G)))
  }
  names(ranks2) = met_set2
  ranks3 = list()
  for (n in 1:length(met_set3)) {
    ind = which(names(G)==met_set3[n])
    ranks3[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set3, num.misses = log2(length(G)))
  }
  names(ranks3) = met_set3
  ranks4 = list()
  for (n in 1:length(met_set4)) {
    ind = which(names(G)==met_set4[n])
    ranks4[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set4, num.misses = log2(length(G)))
  }
  names(ranks4) = met_set4
  ranks5 = list()
  for (n in 1:length(met_set5)) {
    ind = which(names(G)==met_set5[n])
    ranks5[[n]] = singleNode.getNodeRanksN(n=ind, G=G, p1, thresholdDiff,
adj_mat, S=met_set5, num.misses = log2(length(G)))
  }
  names(ranks5) = met_set5
  # Then use the node permutations to outputted to convert them into
bitstrings.
  set1_pku_bs = mle.getPtBSbyK(met_set1, ranks1)
  set2_pku_bs = mle.getPtBSbyK(met_set2, ranks2)
  set3_pku_bs = mle.getPtBSbyK(met_set3, ranks3)
  set4_pku_bs = mle.getPtBSbyK(met_set4, ranks4)
  set5_pku_bs = mle.getPtBSbyK(met_set
```

10.5 Plot the results.

3s

Plot the results

```
require(ggplot2)
bits_to_prob = function(bits) { return(2^-bits) }
ratio_cit_to_rand = log2(bits_to_prob(res$IA_top_cit) /
bits_to_prob(res$IA_top_rand))
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
ratio_mma_to_rand = log2(bits_to_prob(res$IA_top_mma) /
bits_to_prob(res$IA_top_rand))
ratio_pku_to_rand = log2(bits_to_prob(res$IA_top_pku) /
bits_to_prob(res$IA_top_rand))
df = data.frame(subset_size = as.factor(rep(sprintf("K=%d",
res$subset_size), 3)),
                graph = rep(res$graph_name, 3),
                Log2.Ratio.to.Random = c(ratio_cit_to_rand, ratio_mma_to_rand,
ratio_pku_to_rand),
                Metabolite.Set = c(rep("TopCIT", nrow(res)), rep("TopMMA",
nrow(res)), rep("TopPKU", nrow(res))))
df$subset_size = factor(df$subset_size, levels=c("K=5", "K=10", "K=15",
"K=20"))
levels(df$subset_size)
# CIT LOOCV plot
df_cit = df[grep("cit", df$graph),]
err = df_cit[grep("cit[[:digit:]]", df_cit$graph),]
error_bars = data.frame(k=numeric(), mn=numeric(), low = numeric(), high =
numeric(), Metabolite.Set=character(), stringsAsFactors = FALSE)
error_bars[1, "k"] = 5
error_bars[1, "mn"] = mean(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[1, "low"] = min(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[1, "high"] = max(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[1, "Metabolite.Set"] = "TopCIT"
error_bars[2, "k"] = 5
error_bars[2, "mn"] = mean(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[2, "low"] = min(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[2, "high"] = max(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[2, "Metabolite.Set"] = "TopMMA"
error_bars[3, "k"] = 5
error_bars[3, "mn"] = mean(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[3, "low"] = min(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[3, "high"] = max(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[3, "Metabolite.Set"] = "TopPKU"
error_bars[4, "k"] = 10
error_bars[4, "mn"] = mean(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[4, "low"] = min(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[4, "high"] = max(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[4, "Metabolite.Set"] = "TopCIT"
error_bars[5, "k"] = 10
error_bars[5, "mn"] = mean(err[intersect(which(err$subset_size=="K=10"),
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[5, "low"] = min(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[5, "high"] = max(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[5, "Metabolite.Set"] = "TopMMA"
error_bars[6, "k"] = 10
error_bars[6, "mn"] = mean(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[6, "low"] = min(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[6, "high"] = max(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[6, "Metabolite.Set"] = "TopPKU"
error_bars[7, "k"] = 15
error_bars[7, "mn"] = mean(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[7, "low"] = min(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[7, "high"] = max(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[7, "Metabolite.Set"] = "TopCIT"
error_bars[8, "k"] = 15
error_bars[8, "mn"] = mean(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[8, "low"] = min(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[8, "high"] = max(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[8, "Metabolite.Set"] = "TopMMA"
error_bars[9, "k"] = 15
error_bars[9, "mn"] = mean(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[9, "low"] = min(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[9, "high"] = max(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[9, "Metabolite.Set"] = "TopPKU"
error_bars[10, "k"] = 20
error_bars[10, "mn"] = mean(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[10, "low"] = min(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[10, "high"] = max(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[10, "Metabolite.Set"] = "TopCIT"
error_bars[11, "k"] = 20
error_bars[11, "mn"] = mean(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[11, "low"] = min(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[11, "high"] = max(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[11, "Metabolite.Set"] = "TopMMA"
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
error_bars[11, "Metabolite.Set"] = "TopMMA"
error_bars[12, "k"] = 20
error_bars[12, "mn"] = mean(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[12, "low"] = min(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[12, "high"] = max(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[12, "Metabolite.Set"] = "TopPKU"
svg("probbasedon_cit_LOOCV.svg", width=5, height = 2, pointsize=12)
ggplot(error_bars, aes(x=Metabolite.Set, y=mn, fill=Metabolite.Set)) +
geom_bar(stat="identity", position = "dodge") +
  geom_errorbar(aes(ymin=error_bars$low, ymax=error_bars$high),
width=0.1) + theme_bw() +
  ggtitle("Probability Assigned to Metabolites Sets\nUsing an Citrullinemia-
Specific Disease Network") + facet_wrap(~k, nrow=1)
dev.off()

# MMA LOOCV plot
df_mma = df[grep("mma", df$graph),]
err = df_mma[grep("mma[[:digit:]]", df_mma$graph),]
error_bars = data.frame(k=numeric(), mn=numeric(), low = numeric(), high =
numeric(), Metabolite.Set=character(), stringsAsFactors = FALSE)
error_bars[1, "k"] = 5
error_bars[1, "mn"] = mean(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[1, "low"] = min(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[1, "high"] = max(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[1, "Metabolite.Set"] = "TopCIT"
error_bars[2, "k"] = 5
error_bars[2, "mn"] = mean(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[2, "low"] = min(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[2, "high"] = max(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[2, "Metabolite.Set"] = "TopMMA"
error_bars[3, "k"] = 5
error_bars[3, "mn"] = mean(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[3, "low"] = min(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[3, "high"] = max(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[3, "Metabolite.Set"] = "TopPKU"
error_bars[4, "k"] = 10
error_bars[4, "mn"] = mean(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[4, "low"] = min(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[4, "high"] = max(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
error_bars[4, "Metabolite.Set"] = "TopCIT"
error_bars[5, "k"] = 10
error_bars[5, "mn"] = mean(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[5, "low"] = min(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[5, "high"] = max(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[5, "Metabolite.Set"] = "TopMMA"
error_bars[6, "k"] = 10
error_bars[6, "mn"] = mean(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[6, "low"] = min(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[6, "high"] = max(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[6, "Metabolite.Set"] = "TopPKU"
error_bars[7, "k"] = 15
error_bars[7, "mn"] = mean(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[7, "low"] = min(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[7, "high"] = max(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[7, "Metabolite.Set"] = "TopCIT"
error_bars[8, "k"] = 15
error_bars[8, "mn"] = mean(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[8, "low"] = min(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[8, "high"] = max(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[8, "Metabolite.Set"] = "TopMMA"
error_bars[9, "k"] = 15
error_bars[9, "mn"] = mean(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[9, "low"] = min(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[9, "high"] = max(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[9, "Metabolite.Set"] = "TopPKU"
error_bars[10, "k"] = 20
error_bars[10, "mn"] = mean(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[10, "low"] = min(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[10, "high"] = max(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[10, "Metabolite.Set"] = "TopCIT"
error_bars[11, "k"] = 20
error_bars[11, "mn"] = mean(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[11, "low"] = min(err[intersect(which(err$subset_size=="K=20"),
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[11, "high"] = max(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[11, "Metabolite.Set"] = "TopMMA"
error_bars[12, "k"] = 20
error_bars[12, "mn"] = mean(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[12, "low"] = min(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[12, "high"] = max(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[12, "Metabolite.Set"] = "TopPKU"
svg("probbasedon_mma_LOOCV.svg", width=5, height = 2, pointsize=12)
ggplot(error_bars, aes(x=Metabolite.Set, y=mn, fill=Metabolite.Set)) +
geom_bar(stat="identity", position = "dodge") +
  geom_errorbar(aes(ymin=error_bars$low, ymax=error_bars$high),
width=0.1) + theme_bw() +
  ggtitle("Probability Assigned to Metabolites Sets\nUsing an
Guanidinoacetate Methyltransferase Deficiency-Specific Disease Network") +
facet_wrap(~k, nrow=1)
dev.off()

# PKU LOOCV plot
df_pku = df[grep("pku", df$graph),]
err = df_pku[grep("pku[[:digit:]]", df_pku$graph),]
error_bars = data.frame(k=numeric(), mn=numeric(), low = numeric(), high =
numeric(), Metabolite.Set=character(), stringsAsFactors = FALSE)
error_bars[1, "k"] = 5
error_bars[1, "mn"] = mean(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[1, "low"] = min(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[1, "high"] = max(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[1, "Metabolite.Set"] = "TopCIT"
error_bars[2, "k"] = 5
error_bars[2, "mn"] = mean(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[2, "low"] = min(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[2, "high"] = max(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[2, "Metabolite.Set"] = "TopMMA"
error_bars[3, "k"] = 5
error_bars[3, "mn"] = mean(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[3, "low"] = min(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[3, "high"] = max(err[intersect(which(err$subset_size=="K=5"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[3, "Metabolite.Set"] = "TopPKU"
error_bars[4, "k"] = 10
error_bars[4, "mn"] = mean(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[4, "low"] = min(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[4, "high"] = max(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[4, "Metabolite.Set"] = "TopCIT"
error_bars[5, "k"] = 10
error_bars[5, "mn"] = mean(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[5, "low"] = min(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[5, "high"] = max(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[5, "Metabolite.Set"] = "TopMMA"
error_bars[6, "k"] = 10
error_bars[6, "mn"] = mean(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[6, "low"] = min(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[6, "high"] = max(err[intersect(which(err$subset_size=="K=10"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[6, "Metabolite.Set"] = "TopPKU"
error_bars[7, "k"] = 15
error_bars[7, "mn"] = mean(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[7, "low"] = min(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[7, "high"] = max(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[7, "Metabolite.Set"] = "TopCIT"
error_bars[8, "k"] = 15
error_bars[8, "mn"] = mean(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[8, "low"] = min(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[8, "high"] = max(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[8, "Metabolite.Set"] = "TopMMA"
error_bars[9, "k"] = 15
error_bars[9, "mn"] = mean(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[9, "low"] = min(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[9, "high"] = max(err[intersect(which(err$subset_size=="K=15"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[9, "Metabolite.Set"] = "TopPKU"
error_bars[10, "k"] = 20
error_bars[10, "mn"] = mean(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[10, "low"] = min(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[10, "high"] = max(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopCIT")), "Log2.Ratio.to.Random"])
error_bars[10, "Metabolite.Set"] = "TopCIT"
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66

```
error_bars[11, "k"] = 20
error_bars[11, "mn"] = mean(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[11, "low"] = min(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[11, "high"] = max(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopMMA")), "Log2.Ratio.to.Random"])
error_bars[11, "Metabolite.Set"] = "TopMMA"
error_bars[12, "k"] = 20
error_bars[12, "mn"] = mean(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[12, "low"] = min(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[12, "high"] = max(err[intersect(which(err$subset_size=="K=20"),
which(err$Metabolite.Set=="TopPKU")), "Log2.Ratio.to.Random"])
error_bars[12, "Metabolite.Set"] = "TopPKU"
svg("probbasedon_pku_LOOCV.svg", width=5, height = 2, pointsize=12)
ggplot(error_bars, aes(x=Metabolite.Set, y=mn, fill=Metabolite.Set)) +
geom_bar(stat="identity", position = "dodge") +
  geom_errorbar(aes(ymin=error_bars$low, ymax=error_bars$high),
width=0.1) + theme_bw() +
  ggtitle("Probability Assigned to Metabolites Sets\nUsing an Phenylketonuria-
Specific Disease Network") + facet_wrap(~k, nrow=1)
dev.off()
```



**Citation:** Lillian R Thistlethwaite, xiqi.li2 , Varduhipetrosyan (12/11/2020). Reproducibility for Thistlethwaite et al. 2020. https://dx.doi.org/10.17504/protocols.io.bpdvmi66