Package ‘CTD’

January 12, 2020

Title CTD method for “connecting the dots” in weighted graphs

Version 0.0.0.9000

Date 2017-05-25

Maintainer Lillian Thistlethwaite <lillian.thistlethwaite@bcm.edu>

Description An R package for pattern discovery in weighted graphs. Two use cases are achieved: 1) Given a weighted graph and a subset of its nodes; do the nodes show significant connectedness? 2) Given a weighted graph and two subsets of its nodes; do the subsets show significant similarity?

Depends R (>= 3.3.0),
igraph,
plotly,
gplots,
RColorBrewer,
gmp

License MIT License

Encoding UTF-8

LazyData true

RoxygenNote 6.1.0.9000

R topics documented:

`data.HMDBtoKEGG` *Convert HMDB IDs to KEGG compound IDs.*

Description

A function that converts HMDB IDs to KEGG compounds.

Usage

```
data.HMDBtoKEGG(hmdb.ids)
```

Arguments

`hmdb.ids`	- A character vector of HMDB IDs.

Examples

```
kegg.ids = data.HMDBtoKEGG(hmdb.ids)
```

`data.LabeltoKEGG` *Convert compound names to KEGG compound IDs.*

Description

A function that converts HMDB IDs to KEGG compounds.

Usage

```
data.LabeltoKEGG(compound.names)
```

Arguments

`compound.names`	- A character vector of metabolite (compound) names.

Examples

```
kegg.ids = data.HMDBtoKEGG(compound.names)
```

---

data.surrogateProfiles *Surrogate profiles*

Description

Fill in a data matrix rank, when your data is low n, high p. Fill in rank with surrogate profiles.

Usage

```
data.surrogateProfiles(data, std = 1, useMnDiseaseProfile = FALSE,
  addHealthyControls = TRUE, ref_data = NULL)
```

Arguments

`data` - Data matrix with observations as rows, features as columns.

`std` - The level of variability (standard deviation) around each feature's mean you want to add in surrogate profiles.

`useMnDiseaseProfile` - Boolean. For disease cohorts not showing homogeneity, mean across disease profiles and generate disease surrogates around this mean.

`addHealthyControls` - Boolean. Add healthy control profiles to data?

Value

data_mx - Data matrix with added surrogate profiles.

---

data.zscoreData *Z-transform normalized, imputed metabolomics data*

Description

Z-transform normalized, imputed metabolomics data

Usage

```
data.zscoreData(data, ref)
```

Arguments

`data` - Normalized, imputed data. Data matrix with observations as rows, features as columns.

`ref` - Reference samples normalized, imputed data.

Value

zscore.data - Z-transformed data.

`graph.diffuseP1` *Diffuse Probability P1 from a starting node.*

Description

Recursively diffuse probability from a starting node based on the connectivity of the background knowledge graph, representing the likelihood that a variable will be most influenced by a perturbation in the starting node.

Usage

```
graph.diffuseP1(p1, startNode, G, visitedNodes, graphNumber = 1,
  verbose = FALSE)
```

Arguments

`p1` - The probability being dispersed from the starting node, startNode.

`startNode` - The first variable drawn in the node ranking, from which p1 gets dispersed.

`G` - A list of probabilities, with names of the list being the node names in the background knowledge graph.

`visitedNodes` - The history of previous draws in the node ranking sequence.

`graphNumber` - If testing against multiple background knowledge graphs, this is the index associated with the adjacency matrix that codes for G. Default value is 1.

`verbose` - If debugging or tracking a diffusion event, verbose=TRUE will activate print statements. Default is FALSE.

Value

G - A list of returned probabilities after the diffusion of probability has truncated, with names of the list being the node names in the background knowledge graph.

Examples

```
# Read in any network via its adjacency matrix
tmp=matrix(1, nrow=100, ncol=100)
for (i in 1:100) {
  for (j in 1:100) {
    tmp[i, j]=rnorm(1, mean=0, sd=1)
  }
}
colnames(tmp)=sprintf("MolPheno%d", 1:100)
ig=graph.adjacency(tmp, mode="undirected", weighted=TRUE, add.colnames="name")
V(ig)$name=tolower(V(ig)$name)
adjacency_matrix=list(as.matrix(get.adjacency(ig, attr="weight"))) # Must have this declared as a GLOBAL var
# Set other tuning parameters
p0=0.1  # 10% of probability distributed uniformly
p1=0.9  # 90% of probability diffused based on edge weights in networks
thresholdDiff=0.01
G=vector(mode="list", length=length(V(ig)$name))
names(G)=V(ig)$name
G=lapply(G, function(i) i[[1]]=0)
```

```
startNode=names(G)[1]
visitedNodes=G[1]
probs_afterCurrDraw=graph.diffuseP1(p1, startNode, G, visitedNodes, 1, TRUE)
```

---

graph.diffuseP1Movie *Make a movie of the diffusion of probability, P1, from a starting node.*

Description

Recursively diffuse probability from a starting node based on the connectivity of the background knowledge graph, representing the likelihood that a variable will be most influenced by a perturbation in the starting node.

Usage

```
graph.diffuseP1Movie(p1, startNode, G, visitedNodes, ig,
  recursion_level = 1, output_dir = getwd())
```

Arguments

`p1`	- The probability being dispersed from the starting node, startNode.
`startNode`	- The first variable drawn in the node ranking, from which p1 gets dispersed.
`G`	- A list of probabilities, with names of the list being the node names in the background knowledge graph.
`visitedNodes`	- A character vector of node names, storing the history of previous draws in the node ranking.
`graphNumber`	- If testing against multiple background knowledge graphs, this is the index associated with the adjacency matrix that codes for G. Default value is 1.

Value

G - A list of returned probabilities after the diffusion of probability has truncated, with names of the list being the node names in the background knowledge graph.

Examples

```
# 7 node example graph illustrating diffusion of probability based on network connectivity
# from Thistlethwaite et al., 2019.
adj_mat = rbind(c(0,2,1,0,0,0,0), # A
                c(2,0,1,0,0,0,0), # B
                c(1,0,0,1,0,0,0), # C
                c(0,0,1,0,2,0,0), # D
                c(0,0,0,2,0,2,1), # E
                c(0,0,0,1,2,0,1), # F
                c(0,0,0,0,1,1,0)  # G
                )
rownames(adj_mat) = c("A", "B", "C", "D", "E", "F", "G")
colnames(adj_mat) = c("A", "B", "C", "D", "E", "F", "G")
adjacency_matrix = list(adj_mat)
ig = graph.adjacency(as.matrix(adj_mat), mode="undirected", weighted=TRUE)
G=vector(mode="list", length=7)
G[1:length(G)] = 0
```

```
names(G) = c("A", "B", "C", "D", "E", "F", "G")
startNode = "A"
visitedNodes = startNode
# Diffuse 100% of probability from startNode "A"
p1 = 1.0
# Probability diffusion truncates at
thresholdDiff=0.01
coords = layout.fruchterman.reingold(ig)
V(ig)$x = coords[,1]
V(ig)$y = coords[,2]
# Global variable imgNum
imgNum=1
G_new = graph.diffuseP1Movie(p1, startNode, G, visitedNodes, ig, 1, getwd())
```

graph.naivePruning *Pruning edges from disease differential network that also occur in reference-only network.*

Description

Recursively diffuse probability from a starting node based on the connectivity of the background knowledge graph, representing the likelihood that a variable will be most influenced by a perturbation in the starting node.

Usage

```
graph.naivePruning(ig_dis, ig_ref)
```

Arguments

ig_dis	- The igraph object associated with the disease+reference trained differential interaction network.
ig_ref	- The igraph object associated with the reference-only trained interaction network.

Value

ig_pruned - The pruned igraph object of the disease+reference differential interaction network, with reference edges subtracted.

Examples

```
ig_pruned=graph.naivePruning(ig_dis, ig_ref)
```

mle.blowoutSim *Module that best explains the patient similarity assigned between a set of patients.*

Description

Module that best explains the patient similarity assigned between a set of patients.

Usage

```
mle.blowoutSim(patientSim, data_mx, ptIDs, ig_pruned, kmx)
```

Arguments

patientSim	- A similarity matrix, where row and columns are patient identifiers.
data_mx	- The matrix that gives the perturbation strength (z-scores) for all variables (columns) for each patient (rows).
ptIDs	- The identifier associated with patient 1's sample.
ig_pruned	- The list of igraph objects associated with the integrated, pruned disease+reference differential interaction networks.
kmx	- The maximum metabolite set size probed when assessing patient similarity.

Value

ptsim_blowout - An igraph object showing the module blowout describing the similarity between patients in ptIDs.

Examples

```
require(CTD)
data(Thistlethwaite2019)
data_mx = as.matrix(data_mx)
data_mx = suppressWarnings(apply(data_mx, c(1,2), as.numeric))
data_mx = data_mx[,-c(1,2,3,4,5,6,7,8)]
# Load your background network, ig_pruned and your computed patientSim matrix
kmns.clust = kmeans(patientSim, centers=4)
table(kmns.clust$cluster)
ptIDs = names(kmns.clust$cluster[which(kmns.clust$cluster==1)])
ptsim_blowout = mle.blowoutSim(patientSim, data_mx, ptIDs, ig_pruned, kmx=15)
plot.igraph(ptsim_blowout, layout=layout.circle, edge.width=50*abs(E(ptsim_blowout)$weight))
```

`mle.getEncodingLength` *Minimum encoding length (MLE)*

Description

This function calculates the mininmum encoding length associated with a subset of variables given a background knowledge graph.

Usage

```
mle.getEncodingLength(bs, pvals, ptID, G)
```

Arguments

`bs`	- A list of bitstrings associated with a given patient's perturbed variables.
`pvals`	- The matrix that gives the perturbation strength significance for all variables (columns) for each patient (rows)
`ptID`	- The row name in data.pvals corresponding to the patient you specifically want encoding information for.
`G`	- A list of probabilities with list names being the node names of the background graph.

Value

df - a data.frame object, for every bitstring provided in bs input parameter, a row is returned with the following data: the patientID; the bitstring evaluated where T denotes a hit and 0 denotes a miss; the subsetSize, or the number of hits in the bitstring; the individual p-values associated with the variable's perturbations, delimited by '/'; the combined p-value of all variables in the set using Fisher's method; Shannon's entropy, IS.null; the minimum encoding length IS.alt; and IS.null-IS.alt, the d.score.

Examples

```
# Identify the most significant subset per patient, given the background graph
data_mx.pvals = t(apply(data_mx, c(1,2), function(i) 2*pnorm(abs(i), lower.tail = FALSE)))
for (pt in 1:ncol(data_mx)) {
    ptID = colnames(data_mx)[pt]
    res = mle.getEncodingLength(ptBSbyK[[ptID]], data_mx.pvals, ptID, G)
    res = res[order(res[,"d.score"], decreasing=TRUE),]
    print(res)
}
```

`mle.getMinPtDistance` *Metabolite set enrichment analysis (MSEA) using pathway knowledge curated by Metabolon*

Description

A function that returns the pathway enrichment score for all perturbed metabolites in a patient's full metabolomic profile.

Usage

```
mle.getMinPtDistance(allSimMatrices)
```

Arguments

`allSimMatrices`	- A list of all similarity matrices, across all k for a given graph, or across many graphs.

Examples

```
# Get patient distances
data_mx.pvals = apply(data_mx, c(1,2), function(i) 2*pnorm(abs(i), lower.tail = FALSE))
res = list()
t = list(ncd=matrix(NA, nrow=ncol(data_mx), ncol=ncol(data_mx)))
rownames(t$ncd) = colnames(data_mx)
colnames(t$ncd) = colnames(data_mx)
for (i in 1:kmx) {
  res[[i]] = t
}
for (pt in 1:ncol(data_mx)) {
  print(pt)
  ptID = colnames(data_mx)[pt]
  for (pt2 in pt:ncol(data_mx)) {
    ptID2 = colnames(data_mx)[pt2]
    tmp = mle.getPtSim(ptBSbyK[[ptID]], ptID, ptBSbyK[[ptID2]], ptID2, data_mx, ranks)
    for (k in 1:kmx) {
      res[[k]]$ncd[ptID, ptID2] = tmp$NCD[k]
      res[[k]]$ncd[ptID2, ptID] = tmp$NCD[k]
    }
  }
}
patientSimilarity = mle.getMinPtDistance(res)
```

`mle.getPtSim` *Patient similarity using mutual information MLE metric of patients' most modular, perturbed subsets.*

Description

This function calculates the universal distance between patients, using a mutual information metric, where self-information comes from the minimum encoding length of each patient's encoded modular perturbations in the background knowledge graph.

Usage

```
mle.getPtSim(p1.optBS, ptID, p2.optBS, ptID2, data_mx, ranks)
```

Arguments

`p1.optBS` - The optimal bitstring associated with patient 1.

`ptID` - The identifier associated with patient 1's sample.

`p2.optBS` - The optimal bitstring associated with patient 2.

`data_mx` - The matrix that gives the perturbation strength (z-scores) for all variables (columns) for each patient (rows).

`ptID` - The identifier associated with patient 2's sample.

Value

patientSim - a similarity matrix, where row and columns are patient identifiers.

Examples

```
# Get patient distances
data_mx.pvals = apply(data_mx, c(1,2), function(i) 2*pnorm(abs(i), lower.tail = FALSE))
res = list()
tt = list(ncd=matrix(NA, nrow=ncol(data_mx), ncol=ncol(data_mx)))
rownames(t$ncd) = colnames(data_mx)
colnames(t$ncd) = colnames(data_mx)
for (i in 1:kmx) {
  res[[i]] = tt
}
for (pt in 1:ncol(data_mx)) {
  print(pt)
  ptID = colnames(data_mx)[pt]
  for (pt2 in pt:ncol(data_mx)) {
    ptID2 = colnames(data_mx)[pt2]
    for (k in 1:kmx) {
     tmp = mle.getPtSim(ptBSbyK[[ptID]][k], ptID, ptBSbyK[[ptID2]][k], ptID2, data_mx, ranks)
      res[[k]]$ncd[ptID, ptID2] = tmp$NCD
      res[[k]]$ncd[ptID2, ptID] = tmp$NCD
    }
  }
}
```

mle.kraftMcMillan *Apply the Kraft-McMillian Inequality using a specific encoding algorithm.*

Description

A power analysis of the encoding algorithm using to encode subsets of S in G.

Usage

```
mle.kraftMcMillan(G, k, multiNode = FALSE)
```

Arguments

`G` - A character vector of all node names in the background knowledge graph.

`k` - The size of the node name subsets of G.

`multiNode` - Boolean, indicating whether to use the multi-node diffusion encoding algorithm (TRUE) or the single-node diffusion encoding algorithm (FALSE). Default is FALSE.

Value

IA - a list of bitlengths associated with all outcomes in the N choose K outcome space, with the names of the list elements the node names of the encoded nodes

Examples

```
G = list(A=0, B=0, C=0, D=0, E=0, F=0, G=0)
names(G) = tolower(names(G))
adj_mat = rbind(c(0,2,1,0,0,0,0), #A's neighbors
                c(2,0,1,0,0,0,0), #B's neighbors
                c(1,1,0,1,0,0,0), #C's neighbors
                c(0,0,1,0,2,1,0), #D's neighbors
                c(0,0,0,2,0,2,1), #E's neighbors
                c(0,0,0,1,2,0,1), #F's neighbors
                c(0,0,0,0,1,1,0)  #G's neighbors
                )
rownames(adj_mat) = names(G)
colnames(adj_mat) = names(G)
adjacency_matrix = list(adj_mat)
IA = mle.kraftMcMillian(G, 2)
# Power to find effects is
sum(2^-unlist(IA))
```

multiNode.getNodeRanks *Generate the "adaptive walk" node rankings, starting from a given perturbed variable*

Description

This function calculates the node rankings starting from a given perturbed variable in a subset of variables in the background knowledge graph.

Usage

```
multiNode.getNodeRanks(S, G, num.misses = NULL, verbose = FALSE)
```

Arguments

`S` - A character vector of the node names for the subset of nodes you want to encode.

`G` - A list of probabilities with list names being the node names of the background graph.

num.misses - The number of "misses" the network walker will tolerate before switching to fixed length codes for remaining nodes to be found.

verbose - If TRUE, print statements will execute as progress is made. Default is FALSE.

Value

ranks - A list of character vectors of node names in the order they were drawn by the probability diffusion algorithm, from each starting node in S.

Examples

```
# Read in any network via its adjacency matrix
tmp = matrix(1, nrow=100, ncol=100)
for (i in 1:100) {
  for (j in 1:100) {
    tmp[i, j] = rnorm(1, mean=0, sd=1)
  }
}
colnames(tmp) = sprintf("Compound%d", 1:100)
ig = graph.adjacency(tmp, mode="undirected", weighted=TRUE, add.colnames="name")
V(ig)$name = tolower(V(ig)$name)
# Get node rankings for graph
G = vector(mode="list", length=length(V(ig)$name))
names(G) = V(ig)$name
S = names(G)[1:3]
ranks = multiNode.getNodeRanks(S, G)
```

multiNode.getNodeRanksMovie

Capture the movement of the adaptive walk of the diffusion probability method.

Description

Make a movie of the adaptive walk the diffusion probability method makes in search of a given patient's perturbed variables.

Usage

```
multiNode.getNodeRanksMovie(subset.nodes, ig, output_filepath,
  movie = TRUE, zoomIn = FALSE)
```

Arguments

subset.nodes - The subset of variables, S, in a background graph, G.

ig - The igraph object associated with the background knowledge graph.

output_filepath - The local directory at which you want still images to be saved.

movie - If you want to make a movie, set to TRUE. This will produce a set of still images that you can stream together to make a movie. Default is TRUE. Alternatively (movie=FALSE), you could use this function to get the node labels returned for each node ranking starting with a perturbed variable.

zoomIn - Boolean. Delete nodes outside of node subset's order 1 neighborhood?. Default is FALSE.

Value

ranksByStartNode - a list object of node rankings Each element is based on a different startNode. Images are also generated in the output_directory specified.

Examples

```
# Read in any network via its adjacency matrix
tmp = matrix(1, nrow=100, ncol=100)
for (i in 1:100) {
  for (j in 1:100) {
    tmp[i, j] = rnorm(1, mean=0, sd=1)
  }
}
colnames(tmp) = sprintf("Compound%d", 1:100)
ig = graph.adjacency(tmp, mode="undirected", weighted=TRUE, add.colnames="name")
V(ig)$name = tolower(V(ig)$name)
# Set other tuning parameters
p0=0.1  # 10% of probability distributed uniformly
p1=0.9  # 90% of probability diffused based on edge weights in networks
thresholdDiff=0.01
G = vector(mode="list", length=length(V(ig)$name))
names(G) = V(ig)$name
subset.nodes = names(G)[sample(1:length(G), 3)]
multiNode.getNodeRanksMovie(subset.nodes, ig, output_filepath = getwd(), movie=TRUE)
```

`multiNode.getPtBSbyK` *Generate patient-specific bitstrings from adaptive network walk.*

Description

This function calculates the bitstrings (1 is a hit; 0 is a miss) associated with the adaptive network walk made by the diffusion algorithm trying to find the variables in the encoded subset, given the background knowledge graph.

Usage

```
multiNode.getPtBSbyK(S, ranks)
```

Arguments

`S`	- A character vector of node names describing the node subset to be encoded.
`ranks`	- The list of node ranks calculated over all possible nodes, starting with each node in subset of interest.

Value

pt.byK - a list of bitstrings, with the names of the list elements the node names of the encoded nodes

Examples

```
# Get bitstrings associated with each patient's top kmx variable subsets
kmx = 15
ptBSbyK = list()
for (pt in 1:ncol(data_mx)) {
  S = data_mx[order(abs(data_mx[,pt]), decreasing=TRUE),pt][1:kmx]
  ptBSbyK[[ptID]] = multiNode.getPtBSbyK(S, ranks)
}
```

---

pathway.ListMaps_metabolon

Get All Metabolites In Metabolon's Pathway Knowledgebase

Description

Get All Metabolites In Metabolon's Pathway Knowledgebase

Usage

```
pathway.ListMaps_metabolon()
```

Value

pwys - List of pathway maps curated by Metabolon's Metabolync.

Examples

```
pwys = pathway.ListMaps_metabolon()
print(pwys)
```

---

pathway.ListMetabolites_metabolon

Get All Metabolites In Metabolon's Pathway Knowledgebase

Description

Get All Metabolites In Metabolon's Pathway Knowledgebase

Usage

```
pathway.ListMetabolites_metabolon()
```

Value

mets - a character vector of unique metabolites and enzymes found in at least 1 pathway in Metabolon's pathway knowledgebase.

Examples

```
mets = pathway.ListMetabolites_metabolon()
print(mets)
```

`plot.getPathwayIgraph` *plot.getPathwayIgraph*

Description

plot.getPathwayIgraph

Usage

```
plot.getPathwayIgraph(input, Pathway.Name)
```

Arguments

`input`	- A list object of parameters (esp. from R shiny app). Required parameters are ptIDs, diagClass and pathwayMapId.
`Pathway.Name`	- The name of the pathway map for which you want the topological information.

Value

template.ig - Igraph object of selected pathway map.

Examples

```
data(Miller2015)
# Input is supplied by R shiny app, but you can hard code parameters as a list object, too, to test functionali
input = list()
input$ptIDs = colnames(Miller2015)[4]
input$diagClass = "paa"
input$pathwayMapId = "All"
ig = plot.getPathwayIgraph(input, Miller2015)
# Returns a blank template for selected pathway.
plot.igraph(ig, edge.arrow.size = 0.01)
```

`plot.hmSim` *Generate heatmap plot of patient similarity matrix.*

Description

This function plots a heatmap of a patient similarity matrix.

Usage

```
plot.hmSim(patientSim, path=getwd(), diagnoses=NULL)
```

Arguments

`patientSim`	- The patient similarity matrix.
`path`	- The filepath to a directory in which you want to store the .png file.
`diagnoses`	- A character vector of diagnostic labels associated with the rownames of patientSim.

Examples

```
plot.hmSim(patientSim, path=getwd(), diagnoses)
```

plot.knnSim — *Visualize the confusion matrix using nearest neighbor as classification model.*

Description

Visualize the confusion matrix using nearest neighbor as classification model.

Usage

```
plot.knnSim(patientSim, diagnoses, diag)
```

Arguments

`patientSim` - The patient similarity matrix.

`diagnoses` - A character vector of diagnostic labels associated with the rownames of patientSim. The names of this vector are patient IDs, and values are diagnostic labels.

`diag` - The diagnosis associated with positive controls in your data.

Value

p - a plotly scatter plot colored by provided diagnostic labels.

Examples

```
# if you have diagnostic labels associated with the colnames(data_mx), send them using diagnoses parameter
p = plot.knnSim(patientSim, diagnoses, diag="diseased")
p
```

plot.mdsSim — *View patient clusters using multi-dimensional scaling.*

Description

This function plots the provided patient similarity matrix in a lower dimensional space using multidimensional scaling, which is well suited for similarity metrics.

Usage

```
plot.mdsSim(patientSim, diagnoses, k, diag)
```

Arguments

`patientSim`	- The patient similarity matrix.
`diagnoses`	- A character vector of diagnostic labels associated with the rownames of patientSim.
`k`	- The number of dimension you want to plot your data using multi-dimensional scaling.
`diag`	- The diagnosis associated with positive controls in your data.

Value

p - a plotly scatter plot colored by provided diagnostic labels.

Examples

```
# if you have diagnostic labels associated with the colnames(data_mx), send them using diagnoses parameter
p = plot.mdsSim(patientSim, diagnoses, k=2, diag="diseased")
p
p = plot.mdsSim(patientSim, diagnoses, k=3, diag="diseased")
p
```

---

plot.pathwayMap *Generate pathway map with patient perturbation data superimposed.*

Description

Generate pathway map with patient perturbation data superimposed.

Usage

```
plot.pathwayMap(Pathway, ptID, pt.zscore, zscore.threshold, scale, out.path, SVG=TRUE)
```

Arguments

`Pathway`	- The name of the pathway map you want to plot patient data on.
`ptID`	- An identifier string associated with the patient.
`pt.zscore`	- A named vector of metabolites with corresponding z-scores.
`zscore.threshold`	- Plot all z-scores > or < this threshold.
`scale`	- Integer associated with increase in node size.
`out.path`	- The directory in which you want to store image files.
`SVG`	- Save as SVG or PNG? If SVG is TRUE, then an SVG image is saved. If FALSE, a PNG is saved.

Examples

```
Pathway = pathway.ListMaps_metabolon()
data(Miller2015)
Miller2015 = Miller2015[,grep("IEM", colnames(Miller2015))]
ptID = colnames(Miller2015)[1]
pt.zscore = Miller2015[,1]
plot.pathwayMap(Pathway[1], ptID, pt.zscore, zscore.threshold, scale=1, out.path=getwd(), SVG=TRUE)
```

singleNode.getNodeRanksMovie

Capture the movement of the fixed, single-node walk of the diffusion probability method.

Description

Make a movie of the fixed, single-node walk the diffusion probability method makes in search of a given patient's perturbed variables.

Usage

```
singleNode.getNodeRanksMovie(subset.nodes, ig, output_filepath,
  num.misses = NULL, zoomIn = FALSE)
```

Arguments

subset.nodes	- The subset of variables, S, in a background graph, G.
ig	- The igraph object associated with the background knowledge graph.
output_filepath	- The local directory at which you want still images to be saved.
zoomIn	- Boolean. Delete nodes outside of node subset's order 1 neighborhood?. Default is FALSE.
movie	- If you want to make a movie, set to TRUE. This will produce a set of still images that you can stream together to make a movie. Default is TRUE. Alternatively (movie=FALSE), you could use this function to get the node labels returned for each node ranking starting with a perturbed variable.

Value

ranksByStartNode - a list object of node rankings Each element is based on a different startNode. Images are also generated in the output_directory specified.

Examples

```
# Read in any network via its adjacency matrix
tmp = matrix(1, nrow=100, ncol=100)
for (i in 1:100) {
  for (j in 1:100) {
    tmp[i, j] = rnorm(1, mean=0, sd=1)
  }
}
colnames(tmp) = sprintf("Compound%d", 1:100)
ig = graph.adjacency(tmp, mode="undirected", weighted=TRUE, add.colnames="name")
V(ig)$name = tolower(V(ig)$name)
adjacency_matrix = list(tmp) # MUST BE GLOBAL VARIABLE
# Set other tuning parameters
p0=0.1  # 10% of probability distributed uniformly
p1=0.9  # 90% of probability diffused based on edge weights in networks
thresholdDiff=0.01
G = vector(mode="list", length=length(V(ig)$name))
```

```
names(G) = V(ig)$name
subset.nodes = names(G)[sample(1:length(G), 3)]
singleNode.getNodeRanksMovie(subset.nodes, ig, output_filepath = getwd())
```

singleNode.getNodeRanksN

Generate the fixed , single-node diffusion node rankings, starting from a given perturbed variable.

Description

This function calculates the node rankings starting from a given perturbed variable in a subset of variables in the background knowledge graph.

Usage

```
singleNode.getNodeRanksN(n, G, S = NULL, num.misses = NULL,
  verbose = FALSE)
```

Arguments

n	- The index (out of a vector of node names) of the node ranking you want to calculate.
G	- A list of probabilities with list names being the node names of the background graph.
S	- A character vector of node names in the subset you want the network walker to find.
num.misses	- The number of "misses" the network walker will tolerate before switching to fixed length codes for remaining nodes to be found.
verbose	- If TRUE, print statements will execute as progress is made. Default is FALSE.

Value

current_node_set - A character vector of node names in the order they were drawn by the probability diffusion algorithm.

Examples

```
# Get node rankings for graph
ranks = list()
for (n in 1:length(G)) {
  print(sprintf("Generating node rankings starting with node %s", names(G)[n]))
  ranks[[n]] = singleNode.getNodeRanksN(n, G)
}
names(ranks) = names(G)
```

singleNode.getPtBSbyK *Generate patient-specific bitstrings from the fixed, single-node network walker.*

Description

This function calculates the bitstrings (1 is a hit; 0 is a miss) associated with the fixed, single-node network walker trying to find the variables in the encoded subset, given the background knowledge graph.

Usage

```
singleNode.getPtBSbyK(S, ranks, num.misses = NULL)
```

Arguments

S	- A character vector of node names describing the node subset to be encoded.
ranks	- The list of node ranks calculated over all possible nodes, starting with each node in subset of interest.
num.misses	- The number of misses tolerated by the network walker before path truncation occurs.

Value

pt.byK - a list of bitstrings, with the names of the list elements the node names of the encoded nodes

Examples

```
# Load in your profiling data (rows are compounds, columns are samples)
data_mx = read.table("your_profiling_data.txt", sep="\t", header=TRUE)
# Get bitstrings associated with each patient's top kmx variable subsets
kmx = 15
ptBSbyK = list()
for (pt in 1:ncol(data_mx)) {
  S = data_mx[order(abs(data_mx[,pt]), decreasing=TRUE),pt][1:kmx]
  ptBSbyK[[ptID]] = singleNode.getPtBSbyK(S, ranks)
}
```

stats.entropyFunction *Entropy of a bit-string*

Description

The entropy of a bitstring (ex: 1010111000) is calculated.

Usage

```
stats.entropyFunction(bitString)
```

Arguments

x - A vector of 0's and 1's.

Value

e - a floating point percentage, between 0 and 1.

Examples

```
stats.entropyFunction(c(1,0,0,0,1,0,0,0,0,0,0,0,0))
> 0.6193822
stats.entropyFunction(c(1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0))
> 1
stats.entropyFunction(c(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1))
> 0
```

`stats.fishersMethod` *Fisher's Combined P-value*

Description

Fisher's combined p-value, used to combine the results of individual statistical tests into an overall hypothesis.

Usage

```
stats.fishersMethod(x)
```

Arguments

x - A vector of p-values (floating point numbers).

Value

a floating point number, a combined p-value using Fisher's method.

Examples

```
stats.fishersMethod(c(0.2,0.1,0.3))
> 0.1152162
```

stats.getMSEA_Metabolon

Metabolite set enrichment analysis (MSEA) using pathway knowledge curated by Metabolon

Description

A function that returns the pathway enrichment score for all perturbed metabolites in a patient's full metabolomic profile.

Usage

```
stats.getMSEA_Metabolon(abs_filename_dataset, abs_filename_classes,
  pathway_knowledgebase = "Metabolon", output_dir = getwd(),
  expt_name = "msea_results")
```

Arguments

`abs_filename_dataset`
- Relative or absolute path to relevant .gct file. A .gct file contains profiling data, rows are compounds and columns are sample IDs.

`abs_filename_classes`
- Relative or absolute path to relevant .cls file. A .cls file contains a mapping of class labels to columns in the .gct file.

`output_dir` - The path associated with the folder in which MSEA results will be saved.

`expt_name` - A name to be associated with the experiment you are analyzing. This name will be used in filestems of results rendered in output_dir.

`pathway.knowledgebase`
- The filename of the .gmt file associated with the pathway knowledge desired. Currently only "Metabolon" is offered, though "KEGG", "WikiPathways", "SM-PDB" and/or "Reactome" can be added in future versions.

Examples

```
data(Miller2015)
Miller2015 = Miller2015[,grep("IEM", colnames(Miller2015))]
# Generate a .cls file for your data.
diagnoses = gsub("[[:digit:]]", "", colnames(Miller2015))
diag.ind = diagnoses
diag.ind[which(diag.ind!="Argininemia")] = 0
diag.ind[which(diag.ind=="Argininemia")] = 1
diag.ind = as.numeric(diag.ind)
# Manually add the following text to 1st line of .cls,
  where num_samples is the length of diag.ind: #num_samples 1 2
# Manually add the following text to 2nd line of .cls: #disease control
write.table(diag.ind, file=system.file("extdata/MSEA_Datasets/Miller2015_arg.cls", package="CTD"),
            sep=" ", quote=FALSE, row.names = FALSE, col.names = FALSE)

# Create a .gct file.
data_mx = Miller2015
data_mx = data_mx[, order(diags.ind)]
data_mx = cbind(rep(NA, nrow(data_mx)), data_mx)
```

```
colnames(data_mx)[1] = "DESCRIPTION"
write.table(data_mx, file=system.file("extdata/MSEA_Datasets/Miller2015.gct", package="CTD"),
            sep="\t", quote=FALSE, row.names = TRUE)

# Generate a .gmt file.
population = names(met.profile)
paths.hsa = list.dirs(path="../inst/extdata", full.names = FALSE)
paths.hsa = paths.hsa[-which(paths.hsa %in% c("", "RData", "allPathways"))]
sink(system.file("extdata/Pathway_GMTs/Metabolon.gmt", package="CTD"))
for (p in 1:length(paths.hsa)) {
  load(sprintf("../inst/extdata/RData/%s.RData", paths.hsa[p]))
  pathway.compounds = V(ig)$label[which(V(ig)$shape=="circle")]
  pathCompIDs = unique(tolower(pathway.compounds[which(pathway.compounds %in% population)]))
  print(sprintf("%s       %s", paths.hsa[p], paste(pathCompIDs, collapse="    ")), quote=FALSE)
}
sink()
print("test")
abs_filename_dataset = system.file("extdata/MSEA_Datasets/Miller2015.gct", package="CTD")
abs_filename_classes = system.file("extdata/MSEA_Datasets/Miller2015_arg.cls", package="CTD")
pathway.data = stats.getMSEA_Metabolon(abs_filename_dataset, abs_filename_classes, pathway_knowledgebase = '
                                       output_dir = getwd(), expt_name="msea_results")
```

stats.getORA_Metabolon *Metabolite set enrichment analysis (MSEA) (using a hypergeometric test) using pathway knowledge curated by Metabolon*

Description

A function that returns the pathway enrichment score for all perturbed metabolites in a patient's full metabolomic profile.

Usage

```
stats.getORA_Metabolon(met.profile, threshold = 3, type = "zscore",
  gene.profile = NULL)
```

Arguments

`met.profile`	- A character vector of a patient's metabolomic profile, including KEGG IDs and the associated z-score or p-value describing the level of the metabolite compared to controls.
`threshold`	- A cutoff to select metabolites with a zscore > threshold or < -1*threshold.
`type`	- Either "p-value" or "z-score".
`gene.profile`	- Default set to NULL, meaning the default enrichment analysis only considers metabolites. However, if you have gene data, too, set this parameter to a character vector of the gene names with found variants in the patient's record. Gene IDs must be converted to Entrez Identifiers.

Examples

```
pathway.data = stats.getORA_Metabolon(met.profile, threhold=3, "z-score", NULL)
```

Index