GOstats

October 25, 2011

`GOHyperG` *(DEPRECATED) Hypergeometric Tests for GO*

Description

Use `hyperGTest` instead.

Given a set of unique Entrez Gene Identifiers, a microarray annotation data package name, and the GO category of interest, this function will compute Hypergeomtric p-values for overrepresentation of each GO term in the specified category among the GO annotations for the interesting genes (as indicated by the Entrez Gene ids).

Usage

```
GOHyperG(x, lib, what="MF", universe=NULL)
```

Arguments

`x`	A character vector of unique Entrez Gene identifiers.
`lib`	The name of the annotation data package for the chip that was used or `"YEAST"`, see details for more information.
`what`	One of "MF", "BP", or "CC" indicating which of the GO categories to use for the computation. In `GOKEGGHyperG`, what can also be "KEGG"
`universe`	A character vector of unique Entrez Gene identifiers or `NULL`. This is the population (the urn) of the Hypergeometric test. When `NULL` (default), the population is all Entrez Gene ids in the annotation package that have a GO term annotation in the specified GO category (see details).

Details

The Entrez Gene ids given in `x` define the selected set of genes. The universe of Entrez Gene ids is determined by the chip annotation data package (`lib`) or specified by the `universe` argument which must be a subset of the Entrez Gene ids represented on the chip. Both the selected genes and the universe are reduced by removing Entrez Gene ids that do not have any annotations in the specified GO category.

For each GO term in the specified category that has at least one annotation in the selected gene set (`x`), we determine how many of its Entrez Gene annotations are in the universe set and how many are in the selected set. With these counts we perform a Hypergeometric test using `phyper`. This is equivalent to using Fisher's exact test.

It is important that the correct chip annotation data package be identified as it determines the GO term to Entrez Gene id mapping as well as the universe of Entrez Gene ids in the case that the `universe` argument is omitted.

For S. cerevisiae if the `lib` argument is set to `"YEAST"` then comparisons and statistics are computed using common names and are with respect to all genes annotated in the S. cerevisiae genome not with respect to any microarray chip. This will **not** be the right thing to do if you are working with a yeast microarray.

Value

The returned value is a list with components:

`pvalues`	The ordered p-values.
`goCounts`	The vector of counts of Entrez Gene ids from the universe at each node.
`intCounts`	The vector of counts of the supplied Entrez Gene ids annotated at each GO term.
`numLL`	The number of unique Entrez Gene ids in the universe that are mapped to some term in the specified GO category.
`numInt`	The number of unique Entrez Gene ids in the selected gene set, `x`, that are mapped to some term in the specified GO category.
`chip`	A string identifying the chip annotation data package used.
`intLLs`	The input vector `x`.
`go2Affy`	A list with one element for each GO term tested, containing the Affymetrix identifiers associated with that node, for the whole chip (not just the interesting genes). This is the same as extracting the tested GO ids from the annotation package's GO2ALLPROBES environment.

Note

Typically, one has a set of interesting genes/probes obtained from a microarray experiment and is interested in determining whether there is an overrepresentation of these genes at particular GO terms. `GOHyperG` carries out simple Hypergeometric tests to assess the overrepresentation of GO terms.

Two substantial issues arise. First, it is not clear how to do any form of p-value correction. The tests are not independent and the underlying structure of the GO graph presents certain problems that need to be addressed. The second substantial issue is that not all probes on a microarray map to a unique Entrez Gene identifer. In `GOHyperG` every attempt to appropriately correct for non-uniqueness of mappings has been made.

Author(s)

R. Gentleman

See Also

`hyperGTest`, `geneKeggHyperGeoTest`,

`phyper`

Examples

```
## Not run:
library("hgu95av2.db")
library("GO.db")
w1<-as.list(hgu95av2ENTREZID)
w2<-unique(unlist(w1))
set.seed(123)
## pick a 25 interesting genes
myLL <- sample(w2, 25)
xx <- GOHyperG(myLL, lib="hgu95av2.db", what="CC")
xx$numLL
xx$numInt
sum(xx$pvalues < 0.01)

## End(Not run)
```

`GOHyperGResult-class` *Class "GOHyperGResult"*

Description

This class represents the results of a test for overrepresentation of GO categories among genes in a selected gene set based upon the Hypergeometric distribution.

For details on extracting information from this object, be sure to read the accessor documentation in the Category package: HyperGResult-accessors.

Objects from the Class

Objects can be created by calls of the form `new("GOHyperGResult", ...)`.

Slots

`goDag`**:** Object of class `"graph"` representing the DAG of GO terms tested.

`pvalue.order`**:** Object of class `"integer"`. The sort order of the computed p-values.

`annotation`**:** Object of class `"character"`. The name of the annotation data package used in the analysis.

`geneIds`**:** Object of class `"ANY"`. The intersection of the gene identifiers given as input and the computed gene universe.

`testName`**:** Object of class `"character"`. Identifies the testing method used to produce this result instance.

`pvalueCutoff`**:** Object of class `"numeric"`. The cutoff for significance used for some testing methods. Also used for pretty display in the `show` method.

`conditional`**:** A logical indicating whether the calculation should condition on the GO structure.

`testDirection`**:** A string which can be either "over" or "under". This determines whether the test performed detects over or under represented GO terms.

Extends

Class `"HyperGResultBase"`, directly.

Methods

goDag `signature(r = "GOHyperGResult")`: return the `graph` instance representing the DAG of the GO terms that were tested.

condGeneIdUniverse `signature(r = "GOHyperGResult")`: return a list named by the tested GO terms. Each element of the list is a vector of the remaining gene identifiers (from the gene universe) annotated at the corresponding GO term after the identifiers from significant children were removed. This is only available when the calculation performed used the conditional Hypergeometric algorithm.

summary `signature(r = "GOHyperGResult")`: Returns a `data.frame` summarizing the test result. Optional arguments `pvalue` and `categorySize` allow specification of maximum p-value and minimum categorySize, respectively. Optional argument `htmlLinks` is a logical value indicating whether to add HTML links (useful in conjunction with xtables print method with `type` set to `"html"`).

htmlReport `signature(r = "GOHyperGResult")`: Write an HTML version of the table produced by the `summary` method. The path of a file to write the report to can be specified using the `file` argument. The default is `file=""` which will cause the report to be printed to the screen. If you wish to create a single report comprising multiple results you can set `append=TRUE`. The default is `FALSE` (overwrite preexisting report file). You can specify a string to use as an identifier for each table by providing a value for the `label` argument. Additional named arguments will be passed to the `summary` method.

description `signature(object = "GOHyperGResult")`: Return a string giving a one-line description of the result.

Author(s)

Seth Falcon

See Also

HyperGResult-accessors

`GOLeaves` *DEPRECATED Identify the leaves in a GO Graph*

Description

This function is DEPRECATED. Use the `leaves` function in the graph package instead. See the example below.

Given a GO graph this function returns the node labels for all leaves in the graph. A leaf is defined to be a node with only out-edges and no in-edges.

Usage

```
GOLeaves(inG)
```

Arguments

`inG`	An instance of a GO graph.

Details

All nodes in `inG` are inspected for in-edges and those with none are returned.

This should probably be replaced by a function in the graph package that identifies leaves, there is nothing special about GO here.

Value

A vector of the node labels for those nodes with no in-edges.

Author(s)

R. Gentleman

See Also

`makeGOGraph`

Examples

```
library("GO.db")
g1 <- oneGOGraph("GO:0003680", GOMFPARENTS)
g2 <- oneGOGraph("GO:0003701", GOMFPARENTS)
g3 <- join(g1, g2)
leaves(g3, "in")
```

`GOstats-defunct` *Defunct Functions in GOstats Package*

Description

The functions or variables listed here are no longer part of GOstats as they are not needed (any more).

Usage

```
combGOGraph(g1, g2)
hyperGtable(probids, lib, type="MF", pvalue=0.05,
                        min.count=10, save = FALSE, output = TRUE,
                        filename = NULL, universe = NULL)
hyperG2Affy(probids, lib, type="MF", pvalue=0.05,
                         min.count=10, universe = NULL)
```

Arguments

`g1`	A graph
`g2`	A graph
`probids`	A vector of Affymetrix probe IDs
`lib`	An annotation package (e.g., `hgu95av2.db`)
`type`	One of "MF", "CC", "BP", indicating molecular function, cellular component, or biological process, respectively.
`pvalue`	The significance level used to choose GO terms
`min.count`	The minimum number of a given GO term that must be on the chip in order to choose that GO term. This protects against very low p-values that result from the situation where there are very few genes with a given GO term on the chip, but one or two are found in the set of significant genes.
`universe`	A character vector of unique Entrez Gene identifiers or `NULL`. This is the population (the urn) of the Hypergeometric test. When `NULL` (default), the population is all Entrez Gene ids in the annotation package that have a GO term annotation in the specified GO category (see `GOHyperG` for more details).
`save`	Boolean - Set to `TRUE` to save the resulting `data.frame`.
`output`	Boolean - Set to `TRUE` to output the resulting `data.frame` as a text file.
`filename`	If output is set to `TRUE`, give the file name for the output file.

Details

`combGOGraph` was replaced by `join`. `hyperGtable` was replaced by `summary`. `hyperG2Affy` was replaced by `probeSetSummary`.

`GOstats-package` *Tools for manipulating GO and microarrays.*

Description

A set of tools for interacting with GO and microarray data. A variety of basic manipulation tools for graphs, hypothesis testing and other simple calculations.

Details

Package:	GOstats
Version:	1.7.4
Date:	23-08-2006
biocViews:	Statistics, Annotation, GO, MultipleComparisons
Depends:	graph (>= 1.9.25), GO, annotate, RBGL, xtable, Biobase, genefilter, multtest, Category (>= 1.3.7), methods
Imports:	methods, Category
Suggests:	hgu95av2.db (>= 1.6.0)
License:	Artistic

Index:

`ALL`	`Acute Lymphoblastic Leukemia Data from the Ritz Laboratory`
`GOLeaves`	`Identify the leaves in a GO Graph`
`GOstats-defunct`	`Defunct Functions in GOstats Package`
`Ndists`	`Distance matrices for the BCR/ABL and NEG subgroups.`
`compCorrGraph`	`A function to compute a correlation based graph from Gene Expression Data`
`compGdist`	`A function to compute the distance between pairs of nodes in a graph.`
`dropECode`	`Drop GO labels for specified Evidence Codes`
`getEvidence`	`Get the Evidence codes for a set of GO terms.`
`getGOTerm`	`Functions to Access GO data.`
`getOntology`	`Get GO terms for a specified ontology`
`hasGOannote`	`Check for GO annotation`
`hyperG2Affy`	`A List of Affymetrix IDs from GOIDs`
`hyperGtable`	`Table from GOHyperG`
`idx2dimnames`	`Index to Dimnames`
`makeGOGraph`	`Construct a GO Graph`
`notConn`	`Find genes that are not connected to the others.`
`oneGOGraph`	`Construct the GO graph given a set of leaves.`
`shortestPath`	`Shortest Path Analysis`
`simLL`	`Functions to compute similarities between GO graphs and also between Entrez Gene IDs based on their induced GO graphs.`
`triadCensus`	`Triad Functions`

Further information is available in the following vignettes:

`GOstats`	Using GOstats (source, pdf)
`GOusage`	Basic GO Usage (source, pdf)
`GOvis`	Visualizing Data Using GOstats (source, pdf)

Author(s)

R. Gentleman with contributions from S. Falcon

Maintainer: R. Gentleman <rgentlem@fhcrc.org>

`Ndists` *Distance matrices for the BCR/ABL and NEG subgroups.*

Description

These are precomputed distance matrices between all transcription factors selected. In the future they will be computed on the fly but currently that takes about 3 hours and so precomputed versions are supplied.

Usage

```
data(Ndists)
  data(Bdists)
```

Format

These are both distance matrices.

Source

They are based on the ALL data, `ALL`.

Examples

```
data(Ndists)
data(Bdists)
```

`compCorrGraph` *A function to compute a correlation based graph from Gene Expression*

Description

Given a set of gene expression data (an instance of the `ExpressionSet` class) this function computes a graph based on correlations between the probes.

Usage

```
compCorrGraph(eSet, k = 1, tau = 0.6)
```

Arguments

`eSet`	An instance of the `ExpressionSet` class.
`k`	The power to raise the correlations to.
`tau`	The lower cutoff for absolute correlations.

Details

Zhou et al. describe a method of computing a graph between probes (genes) based on estimated correlations between probes. This function implements some of their methods.

Pearson correlations between probes are computed and then these are raised to the power `k`. Any of the resulting estimates that are less than `tau` in absolute value are set to zero.

Value

An instance of the `graph` class. With edges and edge weights determined by applying the algorithm described previously.

Author(s)

R. Gentleman

References

Zhou et al., Transitive functional annotation by shortest-path analysis of gene expression data.

See Also

`compGdist`

Examples

```
## Create an ExpressionSet to work with
set.seed(123)
exprMat <- matrix(runif(50 * 5), nrow=50)
genData <- new("ExpressionSet", exprs=exprMat)

corrG = compCorrGraph(genData)
```

`compGdist` *A function to compute the distance between pairs of nodes in a graph.*

Description

Given a graph, `g`, and a set of nodes in the graph, `whNodes`, Dijkstra's shortest path algorithm is used to compute the distance between all pairs of nodes in `whNodes`.

Usage

```
compGdist(g, whNodes, verbose = FALSE)
```

Arguments

`g`	An instance of the `graph` class.
`whNodes`	A vector of lables of the nodes in `g` for which distances are to be computed.
`verbose`	If `TRUE` then output reporting the progress will be reported.

Details

This function can be quite slow, computation of the pairwise distances is not especially fast and if `whNodes` is long then there are many of them to compute.

Value

A matrix containing the pairwise distances. It might be worth making this an instance of the `dist` class at some point.

Author(s)

R. Gentleman

See Also

compCorrGraph

Examples

```
example(compCorrGraph)
compGdist(corrG, nodes(corrG)[1:5])
```

`hyperGTest` *Hypergeometric Tests for GO term association*

Description

Given a `GOHyperGParams` instance containing a set of unique Entrez Gene Identifiers, a microarray annotation data package name, and the GO ontology of interest, this function will compute Hypergeomtric p-values for over or under-representation of each GO term in the specified ontology among the GO annotations for the interesting genes. The computations can be done conditionally based on the structure of the GO graph.

Arguments

`p`	A `GOHyperGParams` instance

Details

When `conditional(p) == TRUE`, the `hyperGTest` function uses the structure of the GO graph to estimate for each term whether or not there is evidence beyond that which is provided by the term's children to call the term in question statistically overrepresented.

The algorithm conditions on all child terms that are themselves significant at the specified p-value cutoff. Given a subgraph of one of the three GO ontologies, the terms with no child categories are tested first. Next the nodes whose children have already been tested are tested. If any of a given node's children tested significant, the appropriate conditioning is performed.

Value

A `GOHyperGResult` instance.

Author(s)

Seth Falcon

References

FIXME

See Also

GOHyperGResult-class,

geneGoHyperGeoTest, geneKeggHyperGeoTest

`idx2dimnames` *Index to Dimnames*

Description

A function to map from integer offsets in an array to the corresponding values of the row and column names. There is probably a better way but I didn't find it.

Usage

```
idx2dimnames(x, idx)
```

Arguments

`x`	a `matrix` or `data.frame`.
`idx`	An integer vector of offsets into the matrix (values between 1 and the `length` of the matrix.

Value

A list with two components. If it is a LIST, use

`rowNames`	The row names corresponding to the integer index.
`colNames`	The column names corresponding to the integer index.

Author(s)

R. Gentleman

See Also

`dimnames`

Examples

```
data(Ndists)
ltInf = is.finite(Ndists)
xx = idx2dimnames(Ndists, ltInf)
```

`makeGOGraph` *Construct a GO Graph*

Description

The directed acyclic graph (DAG) based on finding the most specific terms for the supplied Entrez Gene IDs is constructed and returned. The constructuion is per GO ontology (there are three, MF, BP and CC) and once the most specific terms have been identified then all less specific terms are found (these are the parents of the terms) and then their parents and so on, until the root is encountered.

Usage

```
makeGOGraph(x, Ontology = "MF", removeRoot = TRUE, mapfun = NULL,
            chip = NULL)
```

Arguments

`x`	A vector of Entrez Gene IDs.
`Ontology`	Which GO ontology to use (CC, BP, or MF).
`removeRoot`	A logical value indicating whether the GO root node should be removed or not.
`mapfun`	A function taking a character vector of Entrez Gene IDs as its only argument and returning a list of "GO lists" matching the structure of the lists in the GO maps of annotation data packages. The function should behave similarly to `mget(x, eg2gomap, ifnotfound=NA)`, that is, `NA` should be returned if a specified Entrez ID has no GO mapping. See details for the interaction of `mapfun` and `chip`.
`chip`	The name of a DB-based annotation data package (the name will end in ".db"). This package will be used to generate an Entrez ID to GO ID mapping instead of `mapfun`.

Details

For each supplied Entrez Gene identifier all the GO annotations (in the specified ontology) are found. The mapping is achieved in one of three ways:

1. If `mapfun` is provided, it will be used to perform the needed lookups. In this case, `chip` will be ignored.
2. If `chip` is provided and `mapfun=NULL`, then the needed lookups will be done based on the Entrez to GO mappings encapsulated in the specified annotation data package. This is the recommended usage.
3. If `mapfun` and `chip` are `NULL` or missing, then the function will attempt to load the GO package (the environment-based package, distinct from GO.db). This package contains a legacy environment mapping Entrez IDs to GO IDs. If the GO package is not available, an error will be raised. Omitting both `mapfun` and `chip` is not recommended as it is not compatible with the DB-based annotation data packages.

The mappings are different for the different ontologies. Typically a GO indentifier is used only in one specific ontology.

The resulting structure is stored in a graph using the `graph` package, again from Bioconductor.

Value

An object that inherits from the `graph` class. The particular implementation is not specified.

Author(s)

R. Gentleman

References

The Gene Ontology Consortium

See Also

`oneGOGraph`

Examples

```
 library("hgu95av2.db")
 set.seed(321)
 uniqun <- function(x) unique(unlist(x))
 gN <- uniqun(sample(hgu95av2ENTREZID, 4))
 gg1 <- makeGOGraph(gN, "BP", chip="hgu95av2.db")
```

`notConn` *Find genes that are not connected to the others.*

Description

A function that takes as input a distance matrix and finds those entries that are not connected to any others (ie. those with distance `Inf`.

Usage

```
notConn(dists)
```

Arguments

`dists`	A distance matrix.

Details

It is a very naive implementation. It presumes that not connected entries are not connected to any other entries, and this might not be true. Using the `connComp` function from the `graph` package or the `RBGL` package might be a better approach.

Value

A vector of the names of the items that are not connected.

Author(s)

R. Gentleman

See Also

`connComp`

Examples

```
 data(Ndists)
 notConn(Ndists)
```

oneGOGraph *Construct the GO graph given a set of leaves.*

Description

Given one or more GO identifiers (which indicate the leaves in the graph) and a set of mappings to the less specific sets of nodes this function will construct the graph that includes that node and all children down to the root node for the ontology.

Usage

```
oneGOGraph(x, dataenv)
GOGraph(x, dataenv)
getGoGraph(p, goIds)
```

Arguments

x	A character vector of GO identifiers.
dataenv	An environment for finding the parents of that term.
p	A GOHyperGParams instance.
goIds	A character vector providing the GO term IDs.

Details

For any set of GO identifiers (from a common ontology) we define the induced GO graph to be that graph, based on the DAG structure (child - parent) of the GO ontology of terms, which takes the most specific set of GO terms that apply (for that ontology) and then joins these to all less specific terms. These functions help construct such graphs.

Value

The induced GO graph (or NULL) for the given GO identifier.

Author(s)

R. Gentleman

See Also

makeGOGraph

Examples

```
 library("GO.db")
 g1 <- oneGOGraph("GO:0003680", GOMFPARENTS)
 g2 <- oneGOGraph("GO:0003701", GOMFPARENTS)
 g3 <- join(g1, g2)

 g4 <- GOGraph(c("GO:0003680", "GO:0003701"), GOMFPARENTS)
if( require("Rgraphviz") && interactive() )
  plot(g3)
```

`probeSetSummary` *Summarize Probe Sets Associated with a hyperGTest Result*

Description

Given the result of a `hyperGTest` run (an instance of `GOHyperGResult`), this function lists all Probe Set IDs associated with the selected Entrez IDs annotated at each significant GO term in the test result.

Usage

```
probeSetSummary(result, pvalue, categorySize, sigProbesets)
```

Arguments

`result`	A `GOHyperGResult` instance. This is the output of the `hyperGTest` function when testing the GO category.
`pvalue`	Optional p-value cutoff. Only results for GO terms with a p-value less than the specified value will be returned. If omitted, `pvalueCutoff(result)` is used.
`categorySize`	Optional minimum size (number of annotations) for the GO terms. Only results for GO terms with `categorySize` or more annotations will be returned. If omitted, no category size criteria will be used.
`sigProbesets`	Optional vector of probeset IDs. See details for more information.

Details

Usually the goal of doing a Fisher's exact test on a set of significant probesets is to find pathways or cellular activities that are being perturbed in an experiment. After doing the test, one usually gets a list of significant GO terms, and the next logical step might be to determine which probesets contributed to the significance of a certain term.

Because the input for the Fisher's exact test consists of a vector of unique Entrez Gene IDs, and there may be multiple probesets that interrogate a particular transcript, the ouput for this function lists all of the probesets that map to each Entrez Gene ID, along with an indicator that shows which of the probesets were used as input.

The rationale for this is that one might not be able to assume a given probeset actually interrogates the intended transcript, so it might be useful to be able to check to see what other similar probesets are doing.

Because one of the first steps before running `hyperGTest` is to subset the input vectors of geneIds and universeGeneIds, any information about probeset IDs that interrogate the same gene transcript is lost. In order to recover this information, one can pass a vector of probeset IDs that were considered significant. This vector will then be used to indicate which of the probesets that map to a given GO term were significant in the original analysis.

Value

A `list` of `data.frame`. Each element of the list corresponds to one of the GO terms (the term is provides as the name of the element). Each `data.frame` has three columns: the Entrez Gene ID (`EntrezID`), the probe set ID (`ProbeSetID`), and a 0/1 indicator of whether the probe set ID was provided as part of the initial input (`selected`)

Note that this 0/1 indicator will only be correct if the 'geneId' vector used to construct the `GOHyperGParams` object was a named vector (where the names are probeset IDs), or if a vector of 'sigProbesets' was passed to this function.

Author(s)

S. Falcon and J. MacDonald

Examples

```
## Fake up some data
library("hgu95av2.db")
library("annotate")
prbs <- ls(hgu95av2GO)[1:300]
## Only those with GO ids
hasGO <- sapply(mget(prbs, hgu95av2GO), function(ids)
if(!is.na(ids) && length(ids) > 1) TRUE else FALSE)
prbs <- prbs[hasGO]
prbs <- getLL(prbs, "hgu95av2")
## remove duplicates, but keep named vector
prbs <- prbs[!duplicated(prbs)]
## do the same for universe
univ <- ls(hgu95av2GO)[1:5000]
hasUnivGO <- sapply(mget(univ, hgu95av2GO), function(ids)
if(!is.na(ids) && length(ids) > 1) TRUE else FALSE)
univ <- univ[hasUnivGO]
univ <- unique(getLL(univ, "hgu95av2"))

p <- new("GOHyperGParams", geneIds=prbs, universeGeneIds=univ,
ontology="BP", annotation="hgu95av2", conditional=TRUE)
## this part takes time...
if(interactive()){
  hyp <- hyperGTest(p)
  ps <- probeSetSummary(hyp, 0.05, 10)
}
```

`shortestPath` *Shortest Path Analysis*

Description

The shortest path analysis was proposed by Zhou et. al. The basic computation is to find the shortest path in a supplied graph between two Entrez Gene IDs. Zhou et al claim that other genes annotated along that path are likely to have the same GO annotation as the two end points.

Usage

```
shortestPath(g, GOnode, mapfun=NULL, chip=NULL)
```

Arguments

`g`	An instance of the `graph` class.
`GOnode`	A length one character vector specifying the GO node of interest.
`mapfun`	A function taking a character vector of GO IDs as its only argument and returning a list of character vectors of Enterz Gene IDs annotated at each corresponding GO ID. The function should behave similarly to `mget(x, go2egmap, ifnotfound=NA)`, that is, `NA` should be returned if a specified GO ID has no Entrez ID mappings. See details for the interaction of `mapfun` and `chip`.
`chip`	The name of a DB-based annotation data package (the name will end in ".db"). This package will be used to generate an Entrez ID to GO ID mapping instead of `mapfun`.

Details

The algorithm implemented here is quite simple. All Entrez Gene identifiers that are annotated at the GO node of interest are obtained. Those that are found as nodes in the graph are retained and used for the computation. For every pair of nodes at the GO term the shortest path between them is computed using `sp.between` from the RBGL package.

There is a presumption that the graph is `undirected`. This restriction could probably be lifted if there was some reason for it - a patch would be gratefully accepted.

The mapping of GO node to Entrez ID is achieved in one of three ways:

1. If `mapfun` is provided, it will be used to perform the needed lookups. In this case, `chip` will be ignored.
2. If `chip` is provided and `mapfun=NULL`, then the needed lookups will be done based on the GO to Entrez mappings encapsulated in the specified annotation data package. This is the recommended usage.
3. If `mapfun` and `chip` are `NULL` or missing, then the function will attempt to load the GO package (the environment-based package, distinct from GO.db). This package contains a legacy environment mapping GO IDs to Entrez IDs. If the GO package is not available, an error will be raised. Omitting both `mapfun` and `chip` is not recommended as it is not compatible with the DB-based annotation data packages.

Value

The return values is a list with the following components:

`shortestpaths`	A list of the ouput from `sp.between`. The names are the names of the nodes used as the two endpoints
`nodesUsed`	A vector of the Entrez Gene IDs that were both found at the GO term of interest and were nodes in the supplied graph, `g`. These were used to compute the shortest paths.
`nodesNotUsed`	A vector of Entrez Gene IDs that were annotated at the GO term, but were not found in the graph `g`.

Author(s)

R. Gentleman

References

Transitive functional annotation by shortest-path analysis of gene expression data, by X. Zhou and M-C J. Kao and W. H. Wong, PNAS, 2002

See Also

sp.between

Examples

```
library("hgu95av2.db")
library("RBGL")

set.seed(321)
uniqun <- function(x) unique(unlist(x))

goid <- "GO:0005778"
egIds <- uniqun(mget(uniqun(hgu95av2GO2PROBE[[goid]]),
                             hgu95av2ENTREZID))

v1 <- randomGraph(egIds, 1:10, .3, weights=FALSE)
## Since v1 is random, it might be disconnected and we need a
## connected graph to guarantee the existence of a path.
c1 <- connComp(v1)
largestComp <- c1[[which.max(sapply(c1, length))]]
v2 <- subGraph(largestComp, v1)

a1 <- shortestPath(v2, goid, chip="hgu95av2.db")
```

`simLL` *Functions to compute similarities between GO graphs and also between*

Description

Both `simUI` and `simLP` compute a similarity measure between two GO graphs. For `simLL`, first the induced GO graph for each of its arguments is found and then these are passed to one of `simUI` or `simLP`.

Usage

```
simLL(ll1, ll2, Ontology = "MF", measure = "LP", dropCodes = NULL,
      mapfun = NULL, chip = NULL)
simUI(g1, g2)
simLP(g1, g2)
```

Arguments

`ll1`	A Entrez Gene ID as a character vector.
`ll2`	A Entrez Gene ID as a character vector.
`Ontology`	Which ontology to use ("MF", "BP", "CC").

`measure`	Which measure to use ("LP", "UI").
`dropCodes`	A set of evidence codes to be ignored in constructing the induced GO graphs.
`mapfun`	A function taking a character vector of Entrez Gene IDs as its only argument and returning a list of "GO lists" matching the structure of the lists in the GO maps of annotation data packages. The function should behave similarly to `mget(x, eg2gomap, ifnotfound=NA)`, that is, `NA` should be returned if a specified Entrez ID has no GO mapping. See details for the interaction of `mapfun` and `chip`.
`chip`	The name of a DB-based annotation data package (the name will end in ".db"). This package will be used to generate an Entrez ID to GO ID mapping instead of `mapfun`.
`g1`	An instance of the `graph` class.
`g2`	An instance of the `graph` class.

Details

For each of `ll1` and `ll2` the set of most specific GO terms within the ontology specified (`Ontology`) that are not based on any excluded evidence code (`dropCodes`) are found. The mapping is achieved in one of three ways:

1. If `mapfun` is provided, it will be used to perform the needed lookups. In this case, `chip` will be ignored.
2. If `chip` is provided and `mapfun=NULL`, then the needed lookups will be done based on the Entrez to GO mappings encapsulated in the specified annotation data package. This is the recommended usage.
3. If `mapfun` and `chip` are `NULL` or missing, then the function will attempt to load the GO package (the environment-based package, distinct from GO.db). This package contains a legacy environment mapping Entrez IDs to GO IDs. If the GO package is not available, an error will be raised. Omitting both `mapfun` and `chip` is not recommended as it is not compatible with the DB-based annotation data packages.

Next, the induced GO graphs are computed.

Finally these graphs are passed to one of `simUI`, (union intersection), or `simLP` (longest path). For `simUI` the distance is the size of the intersection of the node sets divided by the size of the union of the node sets. Large values indicate more similarity. These similarities are between 0 and 1.

For `simLP` the length of the longest path in the intersection graph of the two supplied graph. Again, large values indicate more similarity. Similarities are between 0 and the maximum leaf depth of the graph for the specified ontology.

Value

A list with:

`sim`	The numeric similarity measure.
`measure`	Which measure was used.
`g1`	The graph induced by `ll1`.
`g2`	The graph induced by `ll2`.

If one of the supplied Gene IDs does not have any GO terms associated with it, in the selected ontology and with the selected evidence codes then `NA` is returned.

Author(s)

R. Gentleman

See Also

`makeGOGraph`

Examples

```
library("hgu95av2.db")
eg1 = c("9184", "3547")

bb = simLL(eg1[1], eg1[2], "BP", chip="hgu95av2.db")
```

`termGraphs` *Extraction and Plotting of GO Terms from a GOHyperGResult Object*

Description

These functions extract and plot `graph` instances representing the relationships among GO terms tested using `hyperGTest`.

Usage

```
termGraphs(r, id = NULL, pvalue = NULL, use.terms = TRUE)
inducedTermGraph(r, id, children = TRUE, parents = TRUE)
plotGOTermGraph(g, r = NULL, add.counts = TRUE, max.nchar = 20,
                node.colors=c(sig="lightgray", not="white"),
                node.shape="plaintext", ...)
```

Arguments

`r`	A `GOHyperGResult` object as returned by `hyperGTest` when given a `GOHyperGParams` object as input.
`id`	A character vector of category IDs that specifies which terms should be included in the graph.
`pvalue`	Numeric p-value cutoff to use for selecting category terms to include. Will be ignored if `id` is present.
`use.terms`	Logical value indicating whether a `"term"` node attribute should be added to the returned graph providing the more descriptive, but possibly much longer, GO Terms.
`children`	A logical value indicating whether to include direct child terms of the terms specified by `id`.
`parents`	A logical value indicating whether to include direct parent terms of the terms specified by `id`.
`g`	A `graph` object as returned by `inducedTermGraph` or `termGraphs`.
`add.counts`	A logical value indicating whether category size counts should be added to the node labels when plotting.
`max.nchar`	The maximum character length for node labels in the plot.

`node.colors`	A named character vector of length two with compoents `sig` and `not`, giving color names for the significant and non-significant nodes, respectively.
`node.shape`	This argument controls the shape of the plotted nodes and must take on a value allowed by Rgraphviz.
`...`	For `plotGOTermGraph`, extra arguments are passed to the `plot` function.

Details

termGraphs returns a list of `graph` objects each representing one of the connected components of the subgraph of the GO ontology induced by selecting the specified GO IDs (if `id` is present) or by selecting the GO IDs that have a p-value less that `pvalue`. If `use.terms` is `TRUE` the GO IDs will be translated into GO Term names and attached to the nodes as node attributes (see `nodeData`). Edges in the graphs go from child (more specific) to parent (less specific).

inducedTermGraph returns a `graph` object representing the GO graph induced by the terms specified by `id`. The `children` and `parent` arguments control whether direct children and/or direct parents of the terms specified by `id` are added to the graph (at least one of the two must be `TRUE`).

plotGOTermGraph Create a plot using Rgraphviz of a `graph` object as returned by either `termGraphs` or `inducedTermGraph`. If a `GOHyperGResult` object is provided, then the nodes will be colored according to significance (based on the result object's `pvalueCutoff`) and counts will be added to show the size of the categories.

Author(s)

Seth Falcon

`triadCensus` *Triad Functions*

Description

These functions provide some tools for finding triads in an undirected graph. A triad is a clique of size 3. The function `triadCensus` returns a list of all triads.

Usage

```
triadCensus(graph)
isTriad(x, y, z, elz, ely)
reduce2Degreek(graph, k)
enumPairs(iVec)
```

Arguments

`graph`	An instance of the `graph` class.
`k`	An integer indicating the minimum degree wanted.
`x`	A node
`y`	A node
`z`	A node

`elz`	The edgelist for `z`
`ely`	The edgelist for `y`
`iVec`	A vector of unique values

Details

`enumPairs` takes a vector as input and returns a list of length `choose(length(iVec),2)/2` containing all unordered pairs of elements.

`isTriad` takes three nodes as arguments. It is already known that `x` has edges to both `y` and `z` and we want to determine whether these are reciprocated. This is determined by examining `elz` for both `x` and `y` and then examining `ely` for both `x` and `z`.

`reduce2Degreek` is a function that takes an undirected graph as input and removes all nodes of degree less than `k`. This process is iterated until there are no nodes left (an error is thrown) or all nodes remaining have degree at least `k`. The resultant subgraph is returned. It is used here because to be in a triad all nodes must have degree 2 or more.

`triadCensus` makes use of the helper functions described above and finds all triads in the graph.

Value

A list where each element is a triple indicating the members of the triad. Order is not important and all triads are reported in alphabetic order.

Note

See the graph package, RBGL and Rgraphviz for more details and alternatives.

Author(s)

R. Gentleman

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
```

Index