Package 'SANTA'

October 9, 2013

Type Package

Title Spatial Analysis of Network Associations

Version 1.0.0

Date 24 March 2013

Author Alex Cornish and Florian Markowetz

Maintainer Alex Cornish <a.cornish12@imperial.ac.uk>

Imports msm

Depends R (>= 2.14), igraph, snow

Suggests RUnit, BiocGenerics, org.Sc.sgd.db

Description This package provides methods for measuring the strength of association between a network and a phenotype. It does this by measuring clustering of the phenotype across the network. Vertices can also be individually ranked by their strength of association with high-weight vertices.

License Artistic-2.0

biocViews NetworkAnalysis, NetworkEnrichment, Clustering

LazyLoad yes

R topics documented:

`CheckAttributes` *Check that a graph is associated with the correct vertex and edge attributes and can be passed to the* `Knet` *and* `Knode` *function.*

Description

In order for the `Knet` and `Knode` functions to operate correctly when applied to a graph, the graph must be associated with vertex and edge attributes that represent the weight of the vertices and the distances along the edges. This function ensures that an igraph object contains these attributes and converts them to the correct format if required.

Usage

```
CheckAttributes(g, vertex.attr = "pheno", edge.attr = "distance")
```

Arguments

`g`	An igraph object, the graph to work on.
`vertex.attr`	A character vector, containing the one or more names of the vertex attributes under which vertex weights to be tested are stored.
`edge.attr`	A string, containing the name of the edge attribute to be used as distances along the edges. If no edge distances are associated with the graph, then each edge is assigned a distance of 1 and these distances saved under this attribute name.

Details

In order for the `Knet` and `Knode` functions to work correctly, one or more vertex attributes representing vertex weights and one edge attribute representing the distances along edges need to be present within the igraph object. `CheckAttributes` checks that these attributes are present and in a suitable format. If no edge distances are associated with the graph, then each edge is assigned a distance of 1 by the function.

This function is included within the `Knet` and `Knode` functions and does not need to be run separately.

If vertex weights are not present under any of the attribute names in `vertex.attr`, an error message is returned.

If edge distances are not present under the attribute name `edge.attr`, then each edge is assigned a distance of 1 under this attribute name.

If the vertex weights or edge distances are not numeric, then they are converted to numerals. If they cannot be converted, an error message is returned.

If any of the vertex weights or edge distances are negative, an error message is returned.

Value

An igraph object, the input graph with any required modifications.

Author(s)

Alex Cornish <a.cornish12@imperial.ac.uk>

See Also

Knet, Knode

Examples

```
# Create a graph, assign vertex weights and use the CheckAttributes function
# to check that the vertex weights are in the correct format and assign edge
# distances
g1 <- CreateGraph(10, gen.vertex.weights=FALSE)
g1 <- set.vertex.attribute(g1, name="weights", value=runif(vcount(g1)))
g1 <- CheckAttributes(g1, vertex.attr="weights", edge.attr="distances")
get.vertex.attribute(g1, name="weights")
get.edge.attribute(g1, name="distances")
```

CreateGraph *Generate a simple undirected graph, with or without vertex weights and edge distances.*

Description

Generate a simple and undirected graph containing a specified number of vertices, with or without vertex weights and edge distances. Vertex weights added to the graph can be binary or continuous, randomly distributed or grouped into one or more clusters of high-weight vertices (hits).

Usage

```
CreateGraph(n=100, type="barabasi", m=2, p.or.m=(2*n-2*floor(sqrt(n))), vertex.weights=NULL, edge.dis
```

Arguments

n	An integer value, the number of vertices to be included.
type	A string, the algorithm to be used to build the graph. Can either be barabasi, erdos.renyi or grid.
m	An integer value, the number of edges to be added in each step of the barabasi algorithm, if used.
p.or.m	An integer value, the number of edges to be included within the graph, if the erdos.renyi algorithm is used.

`vertex.weights` A numeric vector, containing vertex weights to be added to the graph. The length of the vector must equal `n`. If the graph contains `nhits` hits, then the `nhits`-highest vertex weights are randomly assigned to the hit vertices. The remaining weights are randomly assigned to the remaining vertices (the misses). The vertex weights are added under a vertex attribute `pheno`.

`edge.distances` A numeric vector, containing edge distances to be added to the graph. The length of the vector must equal the number of edges in the graph. The edge distances are randomly assigned to the edges under an edge attribute `distance`.

`gen.vertex.weights`
A logical constant, denoting whether vertex weights should be generated. If `TRUE` and `vertex.weights=NULL`, then vertex weights for hits and misses are generated and added under the vertex attribute `pheno`.

`nclusters` An integer value, the number of clusters that the hits are grouped into.

`lambda` A numeric value, the strength of the hit clustering. If `lambda=0`, then the hits are randomly distributed. The greater the value of `lambda`, the greater the strength of hit clustering. Hits are only added if `gen.vertex.weights=TRUE` and/or `vertex.weights!=NULL`.

`nlinks` An integer value, the number of linking edges to be added between clusters.

`nhits` An integer value, the number of hits to be added to the graph. Hits are only added if `gen.vertex.weights=TRUE` and/or `vertex.weights!=NULL`.

`binary.pheno` A logical value, denoting whether generated vertex weights should be binary or continuous. If `TRUE`, then hit and miss vertices are assigned weights of 1 and 0 under the vertex attribute `pheno`. If `FALSE`, then weights are generated using two truncated normal distributions with parameters `mean.hit`, `sd.hit`, `mean.miss` and `sd.miss`. These continuous weights also range between 0 and 1.

`dist.method` A string, the method used to calculate the distance between each vertex and the start vertex. Can either be `shortest.paths`, `diffusion` or `mfpt`.

`mean.hit` A numeric value, the mean parameter in the truncated normal distribution used to generate the vertex weight of the hits.

`sd.hit` A numeric value, the standard deviation parameter in the truncated normal distribution used to generate the vertex weight of the hits.

`mean.miss` A numeric value, the mean parameter in the truncated normal distribution used to generate the vertex weight of the misses.

`sd.miss` A numeric value, the standard deviation parameter in the truncated normal distribution used to generate the vertex weight of the misses.

Details

This function creates simple, undirected graphs, with or without associated vertex weights and edge distances.

1 of 3 different algorithms can be used to build the graph. The `barabasi` algorithm builds a scale-free graph through preferencial attachment of vertices. The `erdos.renyi` algorithm builds a random graph with a fixed number of edges. The `grid` algorithm builds a graph with a grid-like structure. The graph produced is always simple, meaning that it contains no loops and no multiple edges.

Vertex weights can be added to the graph as a vertex attribute under the name `pheno`. Larger vertex weights indicate that the vertex is more strongly associated with a certain phenotype or function. The number of vertex weights input must equal the number of vertices present within the graph. Larger vertex weights are randomly assigned to the hit vertices and smaller vertex weights are randomly assigned to the misses.

Edge distances can be added to the graph as an edge attribute under the name `distance`. Smaller edge distances indicate that two vertices are more strongly connected. The number of edge distances input must equal the number of edges present in the graph. The edge distances are randomly assigned to the different edges.

If `vertex.weights!=NULL` and/or `gen.vertex.weights=TRUE`, then hits are added across the graph. `lambda` shapes the probability distribution under which the hits are added. If `lambda=0`, then hits are added to each vertex with equal probability. If `lambda>0`, then the hits become clustered around one or more random vertices on the graph. The greater the value of `lambda`, the greater the strength of the clustering. The probability of vertex i being a hit is proportional to

$$P[i] \sim lambda * exp^{-lambda * d[start,i]}$$

where $d[start, i]$ is the distance between the start vertex and vertex i, according to the shortest paths distance measure.

If `binary.pheno=TRUE`, then hits and misses are assigned weights of 1 and 0 respectively. Otherwise, vertex weights ranging between 0 and 1 are generated using 2 truncated normal probability distributions - one distribution for the hits and another for the misses. The shape of these distributions are set using the `mean.hit`, `sd.hit`, `mean.miss` and `sd.miss` parameters.

Multiple clusters of hits can be added to the graph. These clusters are created by adding hits to multiple graphs. These graphs are then connected through the addition of `nlinks` linking edges.

Value

A igraph object. If vertex weights are added, then whether the vertex is a hit or a miss is specified under the vertex attribute `hits`, the cluster from which the hit is from under the vertex attribute `hits.cluster` and the vertex weight under the vertex attribute `pheno`. A vertex attribute named `color` is also added to allow for the identification of hits when the graph is plotted using `plot`.

Author(s)

Alex Cornish <a.cornish12@imperial.ac.uk>

References

Barabasi, A.L. and Albert, R. (1999). *Emergence of scaling in random networks*. Science 286: 509-512.

Erdos, P. and Renyi, A. (1959). *On random graphs*. Publicationes Mathematicae 6: 290-297.

See Also

`CreateGrid`, `barabasi.game`, `erdos.renyi.game`

Examples

```
# Create a 30-vertex graph with a scale-free structure and no vertex weights
g1 <- CreateGraph(n=30, type="barabasi", gen.vertex.weights=FALSE)
plot(g1, layout=layout.fruchterman.reingold)

# Create a 30-vertex graph with a random structure and 2 clusters of high-weight vertices
g2 <- CreateGraph(n=30, type="erdos.renyi", gen.vertex.weights=TRUE, nhits=8, nclusters=2,
lambda=10, binary.pheno=TRUE)
plot(g2, layout=layout.fruchterman.reingold)

# Create a 36-vertex grid-shaped graph with 6 randomly-distributed high-weight vertices
g3 <- CreateGraph(n=36, type="grid", gen.vertex.weights=TRUE, nhits=6, lambda=0,
binary.pheno=FALSE)
plot(g3, layout=layout.fruchterman.reingold)
```

CreateGrid *Generate a grid-like graph.*

Description

Generate a graph with a grid-like arrangement of edges.

Usage

```
CreateGrid(n = 100)
```

Arguments

n	An integer value, the number of vertices to be included.

Details

This is a simple algorithm that creates a grid-like graph. Vertices are arranged in the largest square lattice possible. Vertices not included within this square are added as an additional row. Vertices are connected by edges to their closest neighbours.

Value

An igraph object.

Author(s)

Alex Cornish <a.cornish12@imperial.ac.uk>

See Also

CreateGraph

Examples

```
# Generate and plot a grid-like graph containing 100 vertices.
g <- CreateGrid(n = 100)
plot(g, layout=layout.fruchterman.reingold)
```

`data-treated.dataframe`

Edges from the genetic interaction network created by Bandyopadhyay et al. (2010) using yeast treated with the DNA-damaging agent methyl methanesulfonate (MMS)

Description

It has been demonstrated that some genetic interactions (GIs) are condition-dependant. This data frame contains a filtered set of the interactions identified in yeast when exposed to DNA-damaging conditions. The data frame contains 415 genes linked by 4428 positive and negative GIs. GIs with a score between -1.75 and 1.75 have been removed.

The data frame contains 3 columns, two containing the binding partners and one containing the GI scores.

References

Bandyopadhyay, S., Mehta, M., Kuo, D., Sung, M. K., Chuang, R., Jaehnig, E. J., Bodenmiller, B., Licon, K., Copeland, W., Shales, M., Fiedler, D., Dutkowski, J., Guenole, A., Attikum, H., Shokat, K. M., Kolodner, R. D., Huh, W. K., Aebersold, R., Keogh, M. C., Krogan, N. and Ideker, T. (2010) *Rewiring of genetic networks in response to DNA damage* Science, 330: 1385-1389.

Examples

```
# Load the data frame, create the network and display the number of vertices and edges
# contained within.
data(treated.dataframe)
g.treated <- graph.data.frame(treated.dataframe, directed=FALSE)
vcount(g.treated)
ecount(g.treated)
```

`data-untreated.dataframe`

Edges from the genetic interaction network created by Bandyopadhyay et al. (2010) using yeast under normal laboratory conditions.

Description

It has been demonstrated that some genetic interactions (GIs) are condition-dependant. This data frame contains a filtered set of the interactions identified in yeast when exposed to normal laboratory conditions. The data frame contains 411 genes linked by 3197 positive and negative GIs. GIs with a score between -1.75 and 1.75 have been removed.

The data frame contains 3 columns, two containing the binding partners and one containing the GI scores.

References

Bandyopadhyay, S., Mehta, M., Kuo, D., Sung, M. K., Chuang, R., Jaehnig, E. J., Bodenmiller, B., Licon, K., Copeland, W., Shales, M., Fiedler, D., Dutkowski, J., Guenole, A., Attikum, H., Shokat, K. M., Kolodner, R. D., Huh, W. K., Aebersold, R., Keogh, M. C., Krogan, N. and Ideker, T. (2010) *Rewiring of genetic networks in response to DNA damage* Science, 330: 1385-1389.

Examples

```
# Load the data frame, create the network and display the number of vertices and edges
# contained within.
data(untreated.dataframe)
g.untreated <- graph.data.frame(untreated.dataframe, directed=FALSE)
vcount(g.untreated)
ecount(g.untreated)
```

DistGraph *Compute the vertex pair distance matrix of a graph.*

Description

Compute the distances between pairs of vertices in a graph, using a shortest path, diffusion kernel, or mean first-passage time-based measure.

Usage

```
DistGraph(g, v = V(g), edge.attr = NULL,
dist.method = c("shortest.paths", "diffusion", "mfpt"), correct.inf = TRUE, correct.factor=1)
```

Arguments

g	An igraph object, the graph to work on.
v	An igraph object or numeric vector, the vertices from which each distance is calculated.
edge.attr	A string, containing the name of the edge attribute to be used as distances along the edges. If left equal to NULL, then each edge is assumed to have a distance of 1.
dist.method	A string, the method used to calculate the distance between each vertex pair. Can either be shortest.paths, diffusion or mfpt.

correct.inf	A logical value. If TRUE, then infinite vertex pair distances are replaced with distances equal to the maximum distance measured across the network, multiplied by correct.factor. If FALSE, then these distances are returned as Inf.
correct.factor	A numeric value, the factor by which the maximum measured distance is multiplied.

Details

This function calculates a distance matrix for a graph. Different methods can be used to calculate the distance between each pair of vertices. By specifying a set of vertices, a smaller distance matrix containing only the rows of the input vertices can be returned.

Value

A numeric matrix, containing the distances between vertex pairs.

Author(s)

Alex Cornish <a.cornish12@imperial.ac.uk>

References

Kondor, R.I. and Lafferty, J. (2002). *Diffusion Kernels on Graph and Other Discrete Structures*. Proc. Intl. Conf. Machine Learning.

White, S. and Smyth, P. (2003). *Algorithms for Estimating Relative Importance in Networks*. Technical Report UCI-ICS 04-25.

See Also

GraphDiffusion, GraphMFPT, shortest.paths

Examples

```
# Create a graph and calculate the distance matrix using the shortest paths measure
g1 <- CreateGraph(n=6, type="barabasi")
DistGraph(g1, dist.method="shortest.paths")
plot(g1, layout=layout.fruchterman.reingold)

# Create a graph, assign edge distances and calculate the distance matrix using the
# diffusion kernel-based measure
g2 <- CreateGraph(n=6, type="erdos.renyi")
g2 <- set.edge.attribute(g2, name="distance", value=runif(ecount(g2)))
DistGraph(g2, dist.method="diffusion", edge.attr="distance")
plot(g2, layout=layout.fruchterman.reingold)
```

GraphDiffusion *Compute the distances between vertex pairs using a diffusion kernel-based method.*

Description

Using a diffusion kernel-based algorithm, compute the distance between vertex pairs in an undirected graph, with or without edge distances. This algorithm provides an alternative to the `shortest.paths` and `mfpt` measures of vertex pair distance.

Usage

```
GraphDiffusion(g, v=V(g), edge.attr=NULL, beta=1, correct.factor=1, correct.neg=TRUE)
```

Arguments

`g`	An igraph object, the graph to work on.
`v`	An igraph object or numeric vector, the vertices from which each distance is calculated.
`edge.attr`	A string, the names of the edge attribute to be used as distances along the edges. If left equal to `NULL`, then each edge is assumed to have a distance of 1.
`beta`	A numeric value, the probability that the lazy random walk will take each of the edges emanating from a vertex.
`correct.factor`	A numeric value, the factor by which the maximum measured distance is multiplied.
`correct.neg`	A logical value. If `TRUE`, then negative edge distances are set to 0.

Details

Diffusion across a graph follows a process similar to a random walk. This provides a method of measuring the distance between vertex pairs that does not simply take into account a single path (like the `shortest.paths` algorithm) but instead incorporates multiple paths. This function uses a diffusion kernel-based approach to calculate distances. The algorithm implemented is detailed in the referenced paper.

The distance from vertex a to vertex a is always 0.

Value

A numeric matrix, containing the diffusion kernel-derived vertex pair distance between each vertex in `v` and every vertex in `g`.

Author(s)

Alex Cornish <a.cornish12@imperial.ac.uk>

References

Kondor, R.I. and Lafferty, J. (2002). *Diffusion Kernels on Graph and Other Discrete Structures*. Proc. Intl. Conf. Machine Learning.

See Also

GraphMFPT, shortest.paths

Examples

```
# Create a graph and calculate the diffusion kernel-derived vertex pair distance matrix
g <- CreateGraph(n=6, type="barabasi")
GraphDiffusion(g)
plot(g, layout=layout.fruchterman.reingold)
```

GraphMFPT *Compute the distances between vertex pairs using a mean first-passage time-based method.*

Description

Using the mean first-passage time algorithm, compute the distance between vertex pairs in an undirected graph, with or without edge distances. This algorithm provides an alternative to the shortest.paths and diffusion measures of vertex pair distance.

Usage

```
GraphMFPT(g, v = V(g), edge.attr = NULL, average.distances = TRUE)
```

Arguments

g	An igraph object, the graph to work on.
v	An igraph object or numeric vector, the vertices from which each distance is calculated.
edge.attr	A string, the name of the edge attribute to be used as distances along the edges. If left equal to NULL, then each edge is assumed to have a distance of 1.
average.distances	A logical value. If TRUE, then the distance from vertex A to B and the distance from vertex B to A are averaged to give a single distance. Otherwise, two different distances may be returned.

Details

The mean first-passage time from vertex a to vertex b is defined as the expected number of steps taken on a random walk from vertex a until the first arrival at vertex b. This provides a method of measuring the distance between pairs of vertices that does not simply take into account the distance along the shortest path, but rather incorporates how well the two vertices are connected across multiple paths.

The mean first-passage time from vertex a to vertex b is not necessarily the same as the mean first-passage time from vertex b to vertex a. If a symmetric distance matrix is required, reciprocal distances can be averaged to give a single value for each vertex pair.

If a vertex pair is unconnected, then the distance between the vertices is `Inf`.

The distance from vertex a to vertex a is always 0.

Value

A numeric matrix, containing the mean first-passage time-derived vertex pair distance between each vertex in `v` and every vertex in `g`.

Author(s)

Alex Cornish <a.cornish12@imperial.ac.uk>

References

White, S. and Smyth, P. (2003). *Algorithms for Estimating Relative Importance in Networks*. Technical Report UCI-ICS 04-25.

See Also

`GraphDiffusion`, `shortest.paths`

Examples

```
# Create a graph and calculate the mean first-passage time-based vertex pair distance matrix
g <- CreateGraph(n=6, type="erdos.renyi")
GraphMFPT(g)
plot(g, layout=layout.fruchterman.reingold)
```

`Knet` *Measure the strength of association between a phenotype and a network by computing the strength of hit clustering on the network.*

Description

Compute the strength of clustering of high-weight vertices (hits) on a graph using a modified version of Ripley's K-statistic. This method can be used to measure the strength of association between a phenotype or function and a network.

Usage

```
Knet(g, nperm = 100, dist.method = "shortest.paths", vertex.attr = "pheno",
edge.attr = "distance", correct.factor=1, nsteps = 1000,
prob = c(0, 0.05, 0.5, 0.95, 1), parallel = NULL)
```

Arguments

`g`	An igraph object, the graph to work on.
`nperm`	An integer value, the number of permutations to be completed. In each permutation, the vertex weights are randomly redistributed across the graph and the Knet function recalculated. The permuted Knet-function results are then compared to the observed Knet function result in order to derive a p-value for the significance of the hit clustering.
`dist.method`	A string, the method used to calculate the distance between each hit and every other vertex in the graph. Can either be `shortest.paths`, `diffusion` or `mfpt`.
`vertex.attr`	A character vector, containing the name of the vertex attributes under which the vertex weights to be tested are stored. The vector can contain one or more elements. If more than one set of vertex weights are tested, then results for each set of weights are returned as a list. Vertex weights should be greater or equal that zero or equal to NA if the weight is missing. Vertices with missing weights are still included within the graph. However, their weights do not contribute to the final Knet statistics.
`edge.attr`	A string, containing the name of the edge attribute to be used as distances along the edges. If an edge attribute with this name is not found, then each edge is assumed to have a distance of 1.
`correct.factor`	A numeric value, the value by which the maximum measured vertex pair distance is multiplied when used to replace infinite distances. Infinite vertex pair distances can arrise when either the `shortest.paths` or `mfpt` distance measures are used and not all vertices within the graph are connected.
`nsteps`	An integer value, the number of bins into which the vertex pairs are put according to distance before the Knet function is calculated. Greater values of `nsteps` result in greater accuracy and greater run times.
`prob`	A numeric vector, containing the quantiles to be calculated for the `Knet` permutations.
`parallel`	A numeric value or `NULL`. If parallel computing is possible, `parallel` can be used to split permutations over multiple cores. The `snow` package is used to manage the parallel computing. If `parallel=NULL` or parallel computing is not possible, then only one core is used. If a positive integer is input and parallel computing is possible, then the permutations are split over up to this many cores.

Details

The SANTA method uses the 'guilt-by-association' principle to measure the strength of association between a network and a phenotype. It does this by measuring the strength of clustering of the phenotype scores across the network. The stronger the clustering, the greater the association between the network and the phenotype.

The SANTA method applies Ripley's K-function, a well-established approach to spatial statistics that measures the strength of clustering of points on a plane, and extends it in a number of ways. First, a Knet function is defined by adapting the approach for graphs using vertex pair distance measures. Second, vertex weights are incorporated into Knet and the importance of vertices made relative to their own associated weight. Third, the mean vertex weight is subtracted from each individual vertex weight when calculating the Knet function. This means that the Knet function measures the degree of vertex weight clustering relative to a random distribution of vertex weights. The Knet function is defined as

$$K^{net}[s] = \frac{2}{p^2} \sum_i p_i \sum_j (p_j - \bar{p}) I(d^g[i,j] <= s)$$

where p_i is the weight of vertex i, $\bar{p}$ is the mean vertex weight across all vertices, and $I(dg[i,j] <= s)$ is an identity function, equaling 1 if vertex i and vertex j are within distance s and 0 otherwise.

In order to derive a p-value and quantify the significance of the observed distribution of weights, the observed Knet-curve is compared to Knet-curves obtained using the same graph but randomly permuted vertex weights. The area under the Knet-curve (AUK) is calculated for the observed graph and each of the permuted graphs and a z-score derived. From this, a p-value can be produced. This p-value indicates the probability an observed AUK at least this high is seen given the null hypothesis that the vertex weights are randomly distributed.

Ripley's K-function has previously been applied to geographical networks (such as road networks) in order to identify the clustering of objects along these networks (Okabe and Yamada 2001). However, key differences between the previous implementation and the implementation of the K-function used in this package allows for the function to be applied to numerous biological networks.

Value

If one vertex attribute is input, `Knet` is run on the single set of vertex weights and a list containing the statistics below is returned. If more than one vertex attribute is input, then `Knet` is run on each set of vertex weights and a list containing an element for each vertex attribute is returned. Each element contains a sub-list containing the statistics below for the relavent vertex attribute.

`K.obs`	The Knet-function curve for the observed vertex weights.
`AUK.obs`	The area under the Knet-function curve (AUK) for the observed vertex weights.
`K.perm`	The Knet-function curve for each permutation of vertex weights. Equals `NA` if no permutations are completed.
`AUK.perm`	The area under the Knet-function curve (AUK) for each permutation of vertex weights. Equals `NA` if no permutations are completed.
`K.quan`	The quantiles for the permuted Knet-function curves. Equals `NA` if no permutations are completed.
`nodeK`	The Knode-function curve for each of the vertices using the observed set of vertex weights.
`nodeAUK`	The area under the Knode-function curve (AUK) for each of the vertices using the observed set of vertex weights.
`pval`	The p-value, calculated from a z-score derived from the observed and permuted AUKs. Equals `NA` if no permutations are completed.

Author(s)

Alex Cornish <a.cornish12@imperial.ac.uk> and Florian Markowetz

References

Paper in preparation.

Okabe, A. and Yamada, I. (2001). *The K-function method on a network and its computational implementation* Geographical Analysis. 33(3): 271-290.

See Also

Knode

Examples

```
# Apply Knet to a graph with hit clustering
g.clustered <- CreateGraph(n=50, type="barabasi", gen.vertex.weights=TRUE,
lambda=10, nhits=10, binary.pheno=FALSE)
res.clustered <- Knet(g.clustered, nperm=100)
res.clustered$pval
plot(res.clustered)

# Apply Knet to a graph without hit clustering
g.unclustered <- CreateGraph(n=50, type="barabasi", gen.vertex.weights=TRUE,
lambda=0, nhits=10, binary.pheno=FALSE)
res.unclustered <- Knet(g.unclustered, nperm=100)
res.unclustered$pval
plot(res.unclustered)
```

Knode *Rank vertices by their strength of association with high-weight vertices.*

Description

Rank vertices by their strength of association with high-weight vertices using a modified version of Ripley's K-statistic. Vertex weights can either be binary or positive and continuous.

Usage

```
Knode(g, dist.method="shortest.paths", vertex.attr="pheno", edge.attr="distance",
correct.factor=1, nsteps=1000, only.Knode=TRUE, vertex.weight=TRUE, cluster.id=FALSE,
vertex.degree=TRUE, boncich.power=FALSE, markov.centr=FALSE)
```

Arguments

g	An igraph object, the graph to work on.
dist.method	A string, the method used to calculate the distance between each vertex. Can either be shortest.paths, diffusion or mfpt.
vertex.attr	A character vector, containing the name of the vertex attributes under which the vertex weights to be tested are stored. The vector can contain one or more elements. If more than one set of vertex weights are tested, then results for each set of weights are returned in a list. Vertex weights should be greater than or equal to 0, or equal to NA if the weight is missing. The Knode statistic is still counted for nodes with missing weights. However, the weight of these nodes is excluded in the calculation of other node's Knode statistics.
edge.attr	A string, containing the name of the edge attribute to be used as distances along the edges. If an edge attribute with this name is not found, then each edge is assumed to have a distance of 1.
correct.factor	A numeric value, the value by which the maximum measured vertex pair is multiplied when used to replace infinite distances. Infinite vertex pair distances can arrise when either the shortest.paths or mfpt distance measures are used and not all vertices within the graph are connected.
nsteps	An integer value, the number of bins into which the vertex pairs are put according to distance before the Knode function is calculated. Greater values of nsteps result in greater accuracy and greater run times.
only.Knode	A logical value. If TRUE, only the Knode AUK for each vertex is calculated. If FALSE, then other centrality scores are also calculated. These scores can include cluster ID, vertex degree, vertex betweenness score, Boncich power centrality, Burt's constraint, eigenvector centrality, Google PageRank, Kleinberg's hub and authority scores and the Markov centrality score. The calculation of some of these scores can be switched on or off using the parameters below.
vertex.weight	A logical value. If vertex.weight==TRUE and only.Knode==FALSE, then the weight of each vertex is returned.
cluster.id	A logical value. If cluster.id==TRUE and only.Knode==FALSE and the graph contains a hit.cluster vertex attribute, then the cluster ID for each hit (each vertex with a non-zero weight) is returned. The hit.cluster attribute is added by the CreateGraph function.
vertex.degree	A logical value. If vertex.degree==TRUE and only.Knode==FALSE, then the degree of each vertex is returned. The degree of a vertex is the number of adjacent edges.
boncich.power	A logical value. If boncich.power==TRUE and only.Knode==FALSE, then the Boncick power centrality of each vertex is returned.
markov.centr	A logical value. If markov.centr==TRUE and only.Knode==FALSE, then the Markov centrality score of each vertex is returned.

Details

Using the inner sum of the Knet equation, it becomes possible to prioritise vertices by how well they are connected, or associated, with high-weight vertices. The inner sum of the Knet equation is

$$K_i^{node}[s] = \frac{2}{p}\sum_j (p_j - \bar{p}) I(d^g(i,j) <= s)$$

where p_j is the weight of vertex j, $\bar{p}$ is the mean vertex weight across all vertices, and $I(dg[i,j] <= s)$ is an identity function, equaling 1 if vertex i and vertex j are within distance s and 0 otherwise.

The Knode function can also be used to return a number of other centrality measures, including cluster ID, vertex degree, vertex betweenness score, Boncich power centrality, Burt's constraint, eigenvector centrality, Google PageRank, Kleinberg's hub and authority scores and the Markov centrality score. If the name of each vertex is stored within a vertex attribute called name, then these names are applied to the rows of the returned data frame. Otherwise, the rows are named with the vertex number.

Value

A sorted data frame containing the Knode AUK for each vertex and any other centrality scores.

If one vertex attribute is input, then the Knode AUK and other centrality scores are calculated and a single sorted data frame containing these scores is returned. If more than one vertex attribute is input, then a list of data frames, one for each set of vertex weights, is returned.

Author(s)

Alex Cornish <a.cornish12@imperial.ac.uk> and Florian Markowetz

References

Bonacich, P. (1987). *Power and Centrality: A Family of Measures*. American Journal of Sociology. 92: 1170-1182.

Brin, S. and Page, L. (1998). *The Anatomy of a Large-Scale Hypertextual Web Search Engine*. Proceedings of the 7th World-Wide Web Conference, Bisbane, Australia.

Burt, R.S. (2004). *Structural holes and good ideas*. American Journal of Sociology. 110: 349-399.

Kleinberg, J. (1997). *Authoritative sources in hyperlinked environment*. Proc. 9th. ACM-SIAM Symposium on Discrete Algorithms.

Ulrik Brandes. (2001). *A Faster Algorithm for Betweenness Centrality*. Journal of Mathematical Sociology. 25 (2): 163-177.

White, S. and Smyth, P. (2003). *Algorithms for Estimating Relative Importance in Networks*. Technical Report: UCI-ICS 04-25.

See Also

Knet

Examples

```
# Create a graph with a single cluster of high-weight vertices. Rank all vertices
# by their stength of association with the high-weight vertices.
g1 <- CreateGraph(n=15, gen.vertex.weights=TRUE, lambda=10, nhits=3, binary.pheno=TRUE)
Knode(g1, only.Knode=FALSE)
plot(g1)

# Create a graph with two clusters of high-weight vertices. Rank all vertices by
# their strength of association with the high-weight vertices.
g2 <- CreateGraph(n=15, gen.vertex.weights=TRUE, lambda=10, nhits=6, nclusters=2,
binary.pheno=FALSE)
Knode(g2, only.Knode=FALSE, cluster.id=TRUE)
plot(g2)
```

MarkovCentrality *Compute the Markov centrality score for each vertex in a graph.*

Description

The Markov centrality score uses the concept of a random walk through the graph to calculate the centrality of each vertex. The method uses the mean first-passage time from every vertex to every other vertex to produce a score for each vertex. These scores can be used as a ranking of centrality within the graph.

Usage

```
MarkovCentrality(g, edge.attr = NULL)
```

Arguments

g	An igraph object, the graph to work on.
edge.attr	A string, the name of the edge attribute to be used as distances along the edges. If left equal to NULL, then each edge is assumed to have a distance of 1.

Details

The mean first-passage time can be used as a measure of how closely each vertex is connected to every other vertex in a graph. The mean first-passage time from vertex a to vertex b is the mean number of steps a random walk emanating from vertex a takes to reach vertex b for the first time. Random walks are more likely to reach well-connected vertices quicker and therefore this method can be used to measure distance.

In order to calculate the Markov centrality of each vertex in a graph, the inverse of the mean distance between each vertex and every other vertex is taken. Vertices with smaller average distances to all other vertices have higher Markov centrality scores, indicating that they occupy a more central position within the graph. These values can be used to rank the vertices.

Value

A numeric vector, containing the Markov centrality score of each vertex.

Author(s)

Alex Cornish <a.cornish12@imperial.ac.uk>

References

White, S. and Smyth, P. (2003). *Algorithms for Estimating Relative Importance in Networks*. Technical Report UCI-ICS 04-25.

See Also

GraphMFPT

Examples

```
# Create a graph and compute the Markov centrality score for each vertex
g <- CreateGraph(n=6, type="barabasi")
MarkovCentrality(g)
plot(g, layout=layout.fruchterman.reingold)
```

---

plot.Knet *Plot the results of the Knet function.*

Description

Plot the observed Knet curve against the quantiles of the permuted Knet curves and the observed AUK against the permuted AUKs.

Usage

```
## S3 method for class 'Knet'
plot(x, sequential = FALSE, ...)
```

Arguments

x	The results from the Knet function. If no permutations are contained within these results, then only the observed Knet curve and observed AUK are plotted.
sequential	A logical value. If TRUE, then the plots are sequential. Otherwise, the two plots are plotted alongside each other.
...	Additional arguements to be passed to plot.

Details

If the high-weight vertices are clustered, then the observed Knet curve and AUK will be high relative to the permuted Knet curves and AUKs. The greater the degree of clustering, the greater the difference between the observed and permuted statistics. If the degree of clustering is low, then the observed and permuted curves and AUKs will likely overlap.

The first plot displays the the observed curve in red and the quantiles of the permuted curves in yellow. The quantile boundaries are displayed as grey lines. These boundaries are specified in the Knet function. The second plot displays the observed AUK as a red line and the distribution of permuted AUKs in grey.

Value

Two plots. The first showing the observed Knet curves against the quantiles of the permuted Knet curves. The second showing the observed AUK against the distribution of permuted AUKs.

Author(s)

Alex Cornish <a.cornish12@imperial.ac.uk> and Florian Markowetz

References

Paper in preparation.

See Also

Knet

Examples

```
# Plot results with hit clustering
g.clustered <- CreateGraph(n=100, type="barabasi", gen.vertex.weights=TRUE, lambda=10,
nhits=10, binary.pheno=FALSE)
res.clustered <- Knet(g.clustered, nperm=10)
res.clustered$pval
plot(res.clustered)

# Plot results without hit clustering
g.unclustered <- CreateGraph(n=100, type="barabasi", gen.vertex.weights=TRUE, lambda=0,
nhits=10, binary.pheno=FALSE)
res.unclustered <- Knet(g.unclustered, nperm=10)
res.unclustered$pval
plot(res.unclustered)
```

SpreadHits *Spread hits across a graph.*

Description

Spread hits across a graph using an exponential probability distribution related to the distance of each vertex from the start vertex.

Usage

```
SpreadHits(g, h = 10, lambda = 1, dist.method = "shortest.paths",
edge.attr = NULL, start.vertex = NULL, hit.color = "red", D = NULL)
```

Arguments

g	An igraph object, the graph to work on.
h	An integer value, the number of hits to be added to the graph.
lambda	A numeric value, the strength of hit clustering. If lambda=0, then the hits are randomly distributed. If lambda>0, then the hits are clustered. The greater the value of lambda, the greater the strength of the hit clustering.
dist.method	A string, the method used to calculate the distance between each vertex and the start vertex. Can either be shortest.paths, diffusion or mfpt.
edge.attr	A string, containing the name of the edge attribute to be used as distances along the edges. If left equal to NULL, then each edge is assumed to have a distance of 1.
start.vertex	An igraph object, containing the start vertex.
hit.color	A string, the colour that hits will be if the graph is plotted using plot().
D	A pre-calculated numeric distance matrix.

Details

SpreadHits can be used to add hits to a graph without hits, or replace hits on a graph with hits. The probability of a vertex being labelled as a hit depends on the distance it is from the starting vertex. The value of lambda denotes the shape of the probability distribution used to spread the hits. The greater the value of lambda, the greater the strength of hit clustering. The probability that vertex i is labelled a hit is proportional to:

$$P[i] \sim lambda * exp^{-lambda * d[start,i]}$$

where $d[start, i]$ is the distance between the start vertex and vertex i.

Hits are denoted as 1 under the vertex attributes hits and pheno, while misses are denoted as 0. A color can also be chosen to highlight the hits when the graph is plotted. Misses are automatically coloured grey.

Value

A modified version of the input igraph object. Whether a vertex is a hit or miss is given under the vertex attributes `hits` and `pheno`.

Author(s)

Alex Cornish <a.cornish12@imperial.ac.uk>

See Also

`CreateGraph`

Examples

```
# Create a graph and add 5 clustered hits
g1 <- CreateGraph(n=30, gen.vertex.weights=FALSE)
g1 <- SpreadHits(g1, h=5, lambda=10)
plot(g1, layout=layout.fruchterman.reingold)

# Create a graph and add 5 unclustered hits
g2 <- CreateGraph(n=30, gen.vertex.weights=FALSE)
g2 <- SpreadHits(g2, h=5, lambda=0)
plot(g2, layout=layout.fruchterman.reingold)
```

Index