Package 'GMPro'

January 20, 2025

Type Package

Title Graph Matching with Degree Profiles

Version 0.1.0

Author Yaofang Hu [aut, cre],
Wanjie Wang [aut],
Yi Yu [aut]

Maintainer Yaofang Hu <yaofang.hu@u.nus.edu>

Description Functions for graph matching via nodes' degree profiles are provided in this package. The models we can handle include Erdos-Renyi random graphs and stochastic block models(SBM). More details are in the reference paper: Yaofang Hu, Wanjie Wang and Yi Yu (2020) <arXiv:2006.03284>.

Imports combinat, stats, transport, igraph

License GPL-2

Encoding UTF-8

LazyData true

URL https://arxiv.org/abs/2006.03284

RoxygenNote 7.1.0

Suggests testthat

NeedsCompilation no

Repository CRAN

Date/Publication 2020-06-25 14:00:02 UTC

Contents

`DPdistance` *calculate degree profile distances between 2 graphs.*

Description

This function constructs empirical distributions of degree profiles for each vertex and then calculate distances between each pair of vertices, one from graph *A* and the other from graph *B*. The default distance used is the 1-Wasserstein distance.

Usage

```
DPdistance(A, B, fun = NULL)
```

Arguments

`A, B`	Two 0/1 adjacency matrices.
`fun`	Optional function that computes distance between two distributions.

Value

A distance matrix. Rows represent nodes in graph *A* and columns represent nodes in graph *B*. Its *(i, j)* element is the distance between $i \in A$ and $i \in B$.

Examples

```
set.seed(2020)
n = 10; q = 1/2; s = 1; p = 1
Parent = matrix(rbinom(n*n, 1, q), nrow = n, ncol = n)
Parent[lower.tri(Parent)] = t(Parent)[lower.tri(Parent)]
diag(Parent) <- 0
### Generate graph A
dA = matrix(rbinom(n*n, 1, s), nrow = n, ncol=n);
dA[lower.tri(dA)] = t(dA)[lower.tri(dA)]
A1 = Parent*dA
tmp = rbinom(n, 1, p)
n.A = length(which(tmp == 1))
indA = sample(1:n, n.A, replace = FALSE)
A = A1[indA, indA]
### Generate graph B
dB = matrix(rbinom(n*n, 1, s), nrow = n, ncol=n);
dB[lower.tri(dB)] = t(dB)[lower.tri(dB)]
B1 = Parent*dB
tmp = rbinom(n, 1, p)
n.B = length(which(tmp == 1))
indB = sample(1:n, n.B, replace = FALSE)
B = B1[indB, indB]
DPdistance(A, B)
```

DPedge *The edge-exploited version of DPmatching.*

Description

This functions is based on *DPmatching*. Instead of allowing each vertex in *A* to connect to one and only one vertex in *B*, here by introducing parameter d, this function allows for d edges for each vertex in *A*. More details are in *DPmatching*.

Usage

```
DPedge(A = NULL, B = NULL, d, W = NULL)
```

Arguments

A, B	Two symmetric 0/1 addjacency matrices.
d	A positive integer, indicating the number of candidate matching.
W	A distance matrix between A and B. This argumnet can be null. If W is null, A and B cannot be null.

Value

Dist	The distance matrix between two graphs.
Z	An indicator matrix. Entry $Z_{i,j} = 1$ indicates a matching between node i in graph A and node j in graph B, 0 otherwise.

Examples

```
set.seed(2020)
n = 10; q = 1/2; s = 1; p = 1
Parent = matrix(rbinom(n*n, 1, q), nrow = n, ncol = n)
Parent[lower.tri(Parent)] = t(Parent)[lower.tri(Parent)]
diag(Parent) <- 0
### Generate graph A
dA = matrix(rbinom(n*n, 1, s), nrow = n, ncol=n);
dA[lower.tri(dA)] = t(dA)[lower.tri(dA)]
A1 = Parent*dA
tmp = rbinom(n, 1, p)
n.A = length(which(tmp == 1))
indA = sample(1:n, n.A, replace = FALSE)
A = A1[indA, indA]
### Generate graph B
dB = matrix(rbinom(n*n, 1, s), nrow = n, ncol=n);
dB[lower.tri(dB)] = t(dB)[lower.tri(dB)]
B1 = Parent*dB
tmp = rbinom(n, 1, p)
n.B = length(which(tmp == 1))
indB = sample(1:n, n.B, replace = FALSE)
```

```
B = B1[indB, indB]
DPmatching(A, B)
W = DPdistance(A, B)
DPedge(A, B, d = 5)
```

DPmatching *calculate degree profile distances between two graphs and match nodes.*

Description

This function constructs empirical distributions of degree profiles for each vertex and then calculate distances between each pair of vertices, one from graph A and the other from graph B. The default used is the 1-Wasserstein distance. This function also matches vertices in A with vertices in B via the distance matrix between A and B. The distance matrix can be null and *DPmatching* will calculate it. A and B cannot be null when the distance matrix is null.

Usage

```
DPmatching(A, B, W = NULL)
```

Arguments

A, B	Two 0/1 adjacency matrices.
W	A distance matrix between A and B, which can be null. If null, this function will calculate it. More details in *DPdistance*.

Value

Dist	The distance matrix between two graphs.
match	A vector containing matching results.

Examples

```
set.seed(2020)
n = 10; q = 1/2; s = 1; p = 1
Parent = matrix(rbinom(n*n, 1, q), nrow = n, ncol = n)
Parent[lower.tri(Parent)] = t(Parent)[lower.tri(Parent)]
diag(Parent) <- 0
### Generate graph A
dA = matrix(rbinom(n*n, 1, s), nrow = n, ncol=n);
dA[lower.tri(dA)] = t(dA)[lower.tri(dA)]
A1 = Parent*dA
tmp = rbinom(n, 1, p)
n.A = length(which(tmp == 1))
indA = sample(1:n, n.A, replace = FALSE)
A = A1[indA, indA]
### Generate graph B
dB = matrix(rbinom(n*n, 1, s), nrow = n, ncol=n);
```

```
dB[lower.tri(dB)] = t(dB)[lower.tri(dB)]
B1 = Parent*dB
tmp = rbinom(n, 1, p)
n.B = length(which(tmp == 1))
indB = sample(1:n, n.B, replace = FALSE)
B = B1[indB, indB]
DPmatching(A, B)
W = DPdistance(A, B)
DPmatching(A, B, W)
```

DP_SBM *Degree profile graph matching with community detection.*

Description

Given two community-structured networks, this function first applies a spectral clustering method *SCORE* to detect perceivable communities and then applies *DPmatching* or *EEpost* to match different communities. More details are in *SCORE*, *DPmatching* and *EEpost*.

Usage

```
DP_SBM(
  A,
  B,
  K,
  fun = c("DPmatching", "EEpost"),
  rep = NULL,
  tau = NULL,
  d = NULL
)
```

Arguments

A, B	Two 0/1 addjacency matrices.
K	A positive integer, the number of communities in A and B.
fun	A graph matching algorithm. Choices include *DPmatching* and *EEpost*.
rep	A parameter if choosing *EEpost* as the initial graph matching algorithm.
tau	Optional parameter if choosing *EEpost* as the initial graph matching algorithm. The default value is $rep/10$.
d	Optional parameter if choosing *EEpost* as the initial graph matching algorithm. The default value is 1.

Details

The graphs to be matched are expected to have community structures. The result is the collection of all possible permutations on {1, . . . ,K}.

Value

A list of matching results for all possible permutations on {1, . . . ,K}.

Examples

```
### Here we use graphs under stochastic block model(SBM).
set.seed(2020)
K = 2; n = 30; s = 1;
P  = matrix(c(1/2, 1/4, 1/4, 1/2), byrow = TRUE, nrow = K)
### define community label matrix Pi
distribution = c(1, 2);
l = sample(distribution, n, replace=TRUE, prob = c(1/2, 1/2))
Pi = matrix(0, n, 2) # label matrix
for (i in 1:n){
  Pi[i, l[i]] = 1
  }
### define the expectation of the parent graph's adjacency matrix
Omega = Pi %*% P %*% t(Pi)
### construct the parent graph G
G = matrix(runif(n*n, 0, 1), nrow = n)
G = G - Omega
temp = G
G[which(temp >0)] = 0
G[which(temp <=0)] = 1
diag(G) = 0
G[lower.tri(G)] = t(G)[lower.tri(G)];
### Sample Graphs Generation
### generate graph A from G
dA = matrix(rbinom(n*n, 1, s), nrow = n, ncol=n)
dA[lower.tri(dA)] = t(dA)[lower.tri(dA)]
A1 = G*dA
indA = sample(1:n, n, replace = FALSE)
labelA = l[indA]
A = A1[indA, indA]
### similarly, generate graph B from G
dB = matrix(rbinom(n*n, 1, s), nrow = n, ncol=n)
dB[lower.tri(dB)] = t(dB)[lower.tri(dB)]
B1 = G*dB
indB = sample(1:n, n, replace = FALSE)
labelB = l[indB]
B = B1[indB, indB]
DP_SBM(A = A, B = B, K = 2, fun = "EEpost", rep = 10, d = 3)
```

EEpost *Post-processing step for edge-exploited graph matching.*

Description

Funtions *DPmatching* or *DPedge* can produce a preliminary graph matching result. This function, *EEPost* works on refining the result iteratively. In addition, *EEpost* is able to provide a convergence indicator vector *FLAG* for each matching as a reference for the certainty about the matching since in practice,it has been observed that the true matches usually reach the convergence and stay the same after a few iterations, while the false matches may keep changing in the iterations.

Usage

```
EEpost(W = NULL, A, B, rep, tau = NULL, d = NULL, matching = NULL)
```

Arguments

`W`	A distance matrix.
`A`, `B`	Two 0/1 adjacency matrices.
`rep`	A positive integer, indicating the number of iterations.
`tau`	A positive threshold. The default value is $rep/10$.
`d`	A positive integer, indicating the number of candidate matching. The default value is 1.
`matching`	A preliminary matching result for *EEpost*. If `matching` is null, *EEpost* will apply *DPedge* accordingly to generate the initial matching.

Details

Similar to function *EEpre*, *EEpost* uses maximum bipartite matching to maximize the number of common neighbours for the matched vertices with the knowledge of a preliminary matching result by defining the similarity between $i \in A$ and $j \in B$ as the number of common neighbours between i and j according to the preliminary matching. Then, given a matching result Π_t, post processing step is to seek a refinement Π_{t+1} satisfying $\Pi_{t+1} \in \text{argmax} \langle \Pi, A\Pi_t B \rangle$, where Π is a permutation matrix of dimension (n_A, n_B).

Value

`Dist`	The distance matrix between two graphs.
`match`	A vector containing matching results.
`FLAG`	An indicator vector indicating whether the matching result is converged. 0 for No and 1 for Yes.
`converged.match`	Converged match result. `NA` indicates the matching result for a certain node is not v=convergent.
`converged.size`	The number of converged nodes.

Examples

```
set.seed(2020)
n = 10;p = 1; q = 1/2; s = 1
Parent = matrix(rbinom(n*n, 1, q), nrow = n, ncol = n)
Parent[lower.tri(Parent)] = t(Parent)[lower.tri(Parent)]
diag(Parent) <- 0
### Generate graph A
dA = matrix(rbinom(n*n, 1, s), nrow = n, ncol=n)
dA[lower.tri(dA)] = t(dA)[lower.tri(dA)]
A1 = Parent*dA;
tmp = rbinom(n, 1, p)
n.A = length(which(tmp == 1))
indA = sample(1:n, n.A, replace = FALSE)
A = A1[indA, indA]
### Generate graph B
dB = matrix(rbinom(n*n, 1, s), nrow = n, ncol=n)
dB[lower.tri(dB)] = t(dB)[lower.tri(dB)]
B1 = Parent*dB
tmp = rbinom(n, 1, p)
n.B = length(which(tmp == 1))
indB = sample(1:n, n.B, replace = FALSE)
B = B1[indB, indB]
matching1= DPmatching(A, B)$Dist
EEpost(A = A, B = B, rep = 10, d = 5)
EEpost(A = A, B = B, rep = 10, d = 5, matching = matching1)
```

EEpre *Edge exploited degree profile graph matching with preprocessing.*

Description

This function uses seeds to compute edge-exploited matching results. Seeds are nodes with high degrees. *EEpre* uses seeds to extend the matching of seeds to the matching of all nodes.

Usage

```
EEpre(A, B, d, seed = NULL, AB_dist = NULL)
```

Arguments

A, B	Two 0/1 addjacency matrices.
d	A positive integer, indicating the number of candicate matching.
seed	A matrix indicating pair of seeds. seed can be null.
AB_dist	A nonnegative distance matrix, which can be null. If AB_dist is null, *EEpre* will apply *DPdistance* to find it.

Details

The high degree vertices have many neighbours and enjoy ample information for a successful matching. Thereforem, this function employ these high degree vertices to match other nodes. If the information of seeds is unavailable, *EEpre* will conduct a grid search grid search to find the optimal collection of seeds. These vertices are expected to have high degress and their distances are supposed to be the smallest among the pairs in consideration.

Value

`Dist`	The distance matrix between two graphs
`Z`	An indicator matrix. Entry $Z_{i,j} = 1$ indicates a matching between node $i \in A$ and node $j \in B$, 0 otherwise.

Examples

```
set.seed(2020)
n = 10;p = 1; q = 1/2; s = 1
Parent = matrix(rbinom(n*n, 1, q), nrow = n, ncol = n)
Parent[lower.tri(Parent)] = t(Parent)[lower.tri(Parent)]
diag(Parent) <- 0
### Generate graph A
dA = matrix(rbinom(n*n, 1, s), nrow = n, ncol=n)
dA[lower.tri(dA)] = t(dA)[lower.tri(dA)]
A1 = Parent*dA;
tmp = rbinom(n, 1, p)
n.A = length(which(tmp == 1))
indA = sample(1:n, n.A, replace = FALSE)
A = A1[indA, indA]
### Generate graph B
dB = matrix(rbinom(n*n, 1, s), nrow = n, ncol=n)
dB[lower.tri(dB)] = t(dB)[lower.tri(dB)]
B1 = Parent*dB
tmp = rbinom(n, 1, p)
n.B = length(which(tmp == 1))
indB = sample(1:n, n.B, replace = FALSE)
B = B1[indB, indB]
EEpre(A = A, B = B, d = 5)
```

`EE_SBM` *Edge exploited degree profile graph matching with community detection.*

Description

Given two community-structured networks, this function first applies a spectral clustering method *SCORE* to detect perceivable communities and then applies a certain graph matching method to match different communities.

Usage

```
EE_SBM(
  A,
  B,
  K,
  fun = c("DPmatching", "EEpost"),
  rep = NULL,
  tau = NULL,
  d = NULL
)
```

Arguments

A, B	Two 0/1 addjacency matrices.
K	A positive integer, indicating the number of communities in A and B.
fun	A graph matching algorithm. Choices include *DPmatching* and EEpost.
rep	Optional parameter if *EEpost* is the initial graph matching algorithm.
tau	Optional parameter if *EEpost* is the initial graph matching algorithm. The default value is $rep/10$.
d	Optional parameter if *EEpost* is the initial graph matching algorithm. The default value is 1.

Details

EE_SBM can be regarded as a post processing version of *DP_SBM* using *EEpost*.

Value

match	A vector containing matching results.
FLAG	An indicator vector indicating whether the matching result is converged, 0 for No and 1 for Yes.

Examples

```
### Here we use graphs under stochastic block model(SBM).
set.seed(2020)
K = 2; n = 30; s = 1;
P  = matrix(c(1/2, 1/4, 1/4, 1/2), byrow = TRUE, nrow = K)
### define community label matrix Pi
distribution = c(1, 2);
l = sample(distribution, n, replace=TRUE, prob = c(1/2, 1/2))
Pi = matrix(0, n, 2) # label matrix
for (i in 1:n){
  Pi[i, l[i]] = 1
  }
### define the expectation of the parent graph's adjacency matrix
Omega = Pi %*% P %*% t(Pi)
### construct the parent graph G
```

```
G = matrix(runif(n*n, 0, 1), nrow = n)
G = G - Omega
temp = G
G[which(temp >0)] = 0
G[which(temp <=0)] = 1
diag(G) = 0
G[lower.tri(G)] = t(G)[lower.tri(G)];
### Sample Graphs Generation
### generate graph A from G
dA = matrix(rbinom(n*n, 1, s), nrow = n, ncol=n)
dA[lower.tri(dA)] = t(dA)[lower.tri(dA)]
A1 = G*dA
indA = sample(1:n, n, replace = FALSE)
labelA = l[indA]
A = A1[indA, indA]
### similarly, generate graph B from G
dB = matrix(rbinom(n*n, 1, s), nrow = n, ncol=n)
dB[lower.tri(dB)] = t(dB)[lower.tri(dB)]
B1 = G*dB
indB = sample(1:n, n, replace = FALSE)
labelB = l[indB]
B = B1[indB, indB]
EE_SBM(A = A, B = B, K = 2, fun = "EEpost", rep = 10, d = 3)
```

SCORE *Spectral Clustering On Ratios-of-Eigenvectors.*

Description

Using ratios-of-eigenvectors to detect underlying communities.

Usage

```
SCORE(G, K, itermax = NULL, startn = NULL)
```

Arguments

G	A 0/1 adjacency matrix.
K	A positive integer, indictaing the number of underlying communities in graph G.
itermax	k-means parameter, indicating the maximum number of iterations allowed. The default value is 100.
startn	k-means parameter. If centers is a number, how many random sets should be chosen? The default value is 10.

Details

SCORE is fully established in *Fast community detection by SCORE* of Jin (2015). *SCORE* uses the entry-wise ratios between the first leading eigenvector and each of the other leading eigenvectors for clustering.

Value

A label vector.

References

Jin, J. (2015) *Fast community detection by score*, *The Annals of Statistics 43 (1), 57–89*
https://projecteuclid.org/euclid.aos/1416322036

Examples

```
set.seed(2020)
n = 10; K = 2
P  = matrix(c(1/2, 1/4, 1/4, 1/2), byrow = TRUE, nrow = K)
distribution = c(1, 2)
l = sample(distribution, n, replace=TRUE, prob = c(1/2, 1/2))
Pi = matrix(0, n, 2)
for (i in 1:n){
  Pi[i, l[i]] = 1
  }
### define the expectation of the parent graph's adjacency matrix
Omega = Pi %*% P %*% t(Pi)
### construct the parent graph G
G = matrix(runif(n*n, 0, 1), nrow = n)
G = G - Omega
temp = G
G[which(temp >0)] = 0
G[which(temp <=0)] = 1
diag(G) = 0
G[lower.tri(G)] = t(G)[lower.tri(G)]
SCORE(G, 2)
```

Index