Package 'dream'

July 22, 2025

Title Dynamic Relational Event Analysis and Modeling

Version 0.1.0

Maintainer Kevin A. Carson <kacarson@arizona.edu>

Description A set of tools for relational and event analysis, including two- and one-mode network brokerage and structural measures, and helper functions optimized for relational event analysis with large datasets, including creating relational risk sets, computing network statistics, estimating relational event models, and simulating relational event sequences. For more information on relational event models, see Butts (2008) <doi:10.1111/j.1467-9531.2008.00203.x>, Lerner and Lomi (2020) <doi:10.1017/nws.2019.57>, Bianchi et al. (2024) <doi:10.1146/annurev-statistics-040722-060248>, and Butts et al. (2023) <doi:10.1017/nws.2023.9>. In terms of the structural measures in this package, see Leal (2025) <doi:10.1177/00491241251322517>, Burchard and Cornwell (2018) <doi:10.1016/j.socnet.2018.04.001>, and Fujimoto et al. (2018) <doi:10.1017/nws.2018.11>. This package was developed with support from the National Science Foundation's (NSF) Human Networks and Data Science Program (HNDS) under award number 2241536 (PI: Diego F. Leal). Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.2

Depends R (>= 3.5.0)

Imports collapse, data.table, fastmatch, dqrng, foreach, parallel, doParallel, Rcpp

LinkingTo Rcpp

NeedsCompilation yes

Author Kevin A. Carson [aut, cre] (ORCID: <https://orcid.org/0009-0001-5762-2586>), Diego F. Leal [aut] (ORCID: <https://orcid.org/0000-0001-9973-4626>)

Repository CRAN

Date/Publication 2025-07-19 09:10:16 UTC

Contents

`computeBCConstraint` *Compute Burchard and Cornwell's (2018) Two-Mode Constraint*

Description

This function calculates the values for two-mode network constraint for weighted and unweighted two-mode networks based on Burchard and Cornwell (2018).

Usage

```
computeBCConstraint(net, isolates = NA, returnCIJmat = FALSE, weighted = FALSE)
```

Arguments

`net`	A two-mode adjacency matrix or affiliation matrix.
`isolates`	What value should isolates be given? Preset to be NA.
`returnCIJmat`	TRUE/FALSE. TRUE indicates that the full constraint matrix, that is, the network constraint from an alter j on node i, will be returned to the user. FALSE indicates that the total constraint will be returned. Set to FALSE by default.
`weighted`	TRUE/FALSE. TRUE indicates the statistic will be based on the weighted formula (see the details section). FALSE indicates the statistic will be based on the original non-weighted formula. Set to FALSE by default.

Details

Following Burchard and Cornwell (2018), the formula for two-mode constraint is:

$$c_{ij} = \left(\frac{|\zeta(j) \cap \zeta(i)|}{|\zeta^{(i*)}|} \right)^2$$

where:

- c_{ij} is the constraint of ego *i* with respect to actor *j*.
- $|\zeta(j) \cap \zeta(i)|$ is the number of opposite-class contacts that *i* and *j* both share.
- The denominator, $|\zeta^{(i*)}|$, represents the total number of opposite-class contacts of ego *i* excluding pendants (level 2 groups that only have one member).

The total constraint for ego *i* is given by:

$$C_i = \sum_{j \in \sigma(i)} c_{ij}$$

The function returns the aggregate constraint for each actor; however, the user can specify the function to return the constraint matrix by setting *returnCIJmat* to TRUE.

The function can also compute constraint for weighted two-mode networks by setting *weighted* to TRUE. The formula for two-mode weighted constraint is:

$$c_{ij} = \left(\frac{|\zeta(j) \cap \zeta(i)|}{|\zeta^{(i*)}|} \right)^2 \times w_t$$

where w_t is the average of the tie weights that *i* and *j* send to their shared opposite-class contacts.

Value

The vector of two-mode constraint scores for level 1 actors in a two-mode network.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Burchard, Jake and Benjamin Cornwell. 2018. "Structural Holes and Bridging in Two-Mode Networks." *Social Networks* 55:11-20.

Examples

```
# For this example, we recreate Figure 2 in Burchard and Cornwell (2018: 13)
BCNet <- matrix(
 c(1,1,0,0,
   1,0,1,0,
   1,0,0,1,
   0,1,1,1),
 nrow = 4, ncol = 4, byrow = TRUE)
colnames(BCNet) <- c("1", "2", "3", "4")
rownames(BCNet) <- c("i", "j", "k", "m")
#library(sna) #To plot the two mode network, we use the sna R package
#gplot(BCNet, usearrows = FALSE,
#      gmode = "twomode", displaylabels = TRUE)
computeBCConstraint(BCNet)

#For this example, we recreate Figure 9 in Burchard and Cornwell (2018:18) for
#weighted two mode networks.
BCweighted <- matrix(c(1,2,1, 1,0,0,
                       0,2,1,0,0,1),
                     nrow = 4, ncol = 3,
                     byrow = TRUE)
rownames(BCweighted) <- c("i", "j", "k", "l")
computeBCConstraint(BCweighted, weighted = TRUE)
```

`computeBCES` *Compute Burchard and Cornwell's (2018) Two-Mode Effective Size*

Description

This function calculates the values for two-mode effective size for weighted and unweighted two-mode networks based on Burchard and Cornwell (2018).

Usage

```
computeBCES(
  net,
  inParallel = FALSE,
  nCores = NULL,
  isolates = NA,
  weighted = FALSE
)
```

Arguments

net	A two-mode adjacency matrix or affiliation matrix
inParallel	TRUE/FALSE. TRUE indicates that parallel processing will be used to compute the statistic with the *foreach* package. FALSE indicates that parallel processing will not be used. Set to FALSE by default.
nCores	If inParallel = TRUE, the number of computing cores for parallel processing. If this value is not specified, then the function internally provides it by dividing the number of available cores in half.
isolates	What value should isolates be given? Preset to be NA.
weighted	TRUE/FALSE. TRUE indicates the statistic will be based on the weighted formula (see the details section). FALSE indicates the statistic will be based on the original non-weighted formula. Set to FALSE by default.

Details

The formula for two-mode effective size is:

$$ES_i = |\sigma(i)| - \sum_{j \in \sigma(i)} r_{ij}$$

where:

- ES_i is the effective size of ego *i*.
- $|\sigma(i)|$ is the number of same-class contacts of ego *i*.
- $\sum_{j \in \sigma(i)} r_{ij}$ is the summation of the redundancy for each alter *j* in the two-mode ego network of *i*.

This function allows the user to compute the scores in parallel through the *foreach* and *doParallel* R packages. If the matrix is weighted, the user should specify *weighted = TRUE*. If the matrix is weighted, following Burchard and Cornwell (2018), the formula for two-mode weighted redundancy is:

$$r_{ij} = \frac{|\sigma(j) \cap \sigma(i)|}{|\sigma(i)| \times w_t}$$

where w_t is the average of the tie weights that *i* and *j* send to their shared opposite class contacts.

Value

The vector of two-mode effective size values for level 1 actors in a two-mode network.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Burchard, Jake and Benjamin Cornwell. 2018. "Structural Holes and Bridging in Two-Mode Networks." *Social Networks* 55:11-20.

Examples

```
# For this example, we recreate Figure 2 in Burchard and Cornwell (2018: 13)
BCNet <- matrix(
 c(1,1,0,0,
   1,0,1,0,
   1,0,0,1,
   0,1,1,1),
 nrow = 4, ncol = 4, byrow = TRUE)
colnames(BCNet) <- c("1", "2", "3", "4")
rownames(BCNet) <- c("i", "j", "k", "m")
#library(sna) #To plot the two mode network, we use the sna R package
#gplot(BCNet, usearrows = FALSE,
#      gmode = "twomode", displaylabels = TRUE)
computeBCES(BCNet)

#In this example, we recreate Figure 9 in Burchard and Cornwell (2018:18)
#for weighted two mode networks.
BCweighted <- matrix(c(1,2,1, 1,0,0,
                       0,2,1,0,0,1),
                       nrow = 4, ncol = 3,
                       byrow = TRUE)
rownames(BCweighted) <- c("i", "j", "k", "l")
computeBCES(BCweighted, weighted = TRUE)
```

`computeBCRedund` *Compute Burchard and Cornwell's (2018) Two-Mode Redundancy*

Description

This function calculates the values for two mode redundancy for weighted and unweighted two-mode networks based on Burchard and Cornwell (2018).

Usage

```
computeBCRedund(
  net,
  inParallel = FALSE,
  nCores = NULL,
  isolates = NA,
```

```
  weighted = FALSE
)
```

Arguments

net	A two-mode adjacency matrix or affiliation matrix.
inParallel	TRUE/FALSE. TRUE indicates that parallel processing will be used to compute the statistic with the *foreach* package. FALSE indicates that parallel processing will not be used. Set to FALSE by default.
nCores	If inParallel = TRUE, the number of computing cores for parallel processing. If this value is not specified, then the function internally provides it by dividing the number of available cores in half.
isolates	What value should isolates be given? Preset to be NA.
weighted	TRUE/FALSE. TRUE indicates the statistic will be based on the weighted formula (see the details section). FALSE indicates the statistic will be based on the original non-weighted formula. Set to FALSE by default.

Details

The formula for two-mode redundancy is:

$$r_{ij} = \frac{|\sigma(j) \cap \sigma(i)|}{|\sigma(i)|}$$

where:

- r_{ij} is the redundancy of ego *i* with respect to actor *j*.
- $|\sigma(j) \cap \sigma(i)|$ is the number of same-class contacts (e.g., medical doctors in a hospital) that *i* and *j* both share.
- $|\sigma(i)|$ is the number of same-class contacts of ego *i*.

The two-mode redundancy is ego-bound, that is, the redundancy is only based on the two-mode ego network of *i*. Put differently, r_{ij} only considers the perspective of the ego. This function allows the user to compute the scores in parallel through the *foreach* and *doParallel* R packages. If the matrix is weighted, the user should specify *weighted = TRUE*. Following Burchard and Cornwell (2018), the formula for two-mode weighted redundancy is:

$$r_{ij} = \frac{|\sigma(j) \cap \sigma(i)|}{|\sigma(i)| \times w_t}$$

where w_t is the average of the tie weights that *i* and *j* send to their shared opposite class contacts.

Value

An *n x n* matrix with level 1 redundancy scores for actors in a two-mode network.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Burchard, Jake and Benjamin Cornwell. 2018. "Structural Holes and bridging in two-mode networks." *Social Networks* 55:11-20.

Examples

```
# For this example, we recreate Figure 2 in Burchard and Cornwell (2018: 13)
BCNet <- matrix(
 c(1,1,0,0,
   1,0,1,0,
   1,0,0,1,
   0,1,1,1),
 nrow = 4, ncol = 4, byrow = TRUE)
colnames(BCNet) <- c("1", "2", "3", "4")
rownames(BCNet) <- c("i", "j", "k", "m")
#library(sna) #To plot the two mode network, we use the sna R package
#gplot(BCNet, usearrows = FALSE,
#      gmode = "twomode", displaylabels = TRUE)
#this values replicate those reported by Burchard and Cornwell (2018: 14)
computeBCRedund(BCNet)


#For this example, we recreate Figure 9 in Burchard and Cornwell (2018:18)
#for weighted two mode networks.
BCweighted <- matrix(c(1,2,1, 1,0,0,
                       0,2,1,0,0,1),
                       nrow = 4, ncol = 3,
                       byrow = TRUE)
rownames(BCweighted) <- c("i", "j", "k", "l")
computeBCRedund(BCweighted, weighted = TRUE)
```

`computeBurtsConstraint` *Compute Burt's (1992) Constraint for Ego Networks from a Sociomatrix*

Description

This function computes Burt's (1992) one-mode ego constraint based upon a sociomatrix.

Usage

```
computeBurtsConstraint(
  net,
  inParallel = FALSE,
  nCores = NULL,
  isolates = NA,
```

```
  pendants = 1
)
```

Arguments

net	A one-mode sociomatrix with network ties.
inParallel	TRUE/FALSE. TRUE indicates that parallel processing will be used to compute the statistic with the *foreach* package. FALSE indicates that parallel processing will not be used. Set to FALSE by default.
nCores	If inParallel = TRUE, the number of computing cores for parallel processing. If this value is not specified, then the function internally provides it by dividing the number of available cores in half.
isolates	What value should isolates be given? Set to NA by default.
pendants	What value should be given to pendant vertices? Set to 1 by default.

Details

The formula for Burt's (1992) one-mode ego constraint is:

$$c_{ij} = \left(p_{ij} + \sum_{q} p_{iq} p_{qj} \right)^2 \quad ; \; q \neq i \neq j$$

where:

- p_{iq} is formulated as: $p_{iq} = \frac{z_{iq} + z_{qi}}{\sum_j (z_{ij} + z_{ji})} \quad ; \; i \neq j$

Finally, the aggregate constraint of an ego *i* is:

$$C_i = \sum_j c_{ij}$$

While this function internally locates isolates (i.e., nodes who have no ties) and pendants (i.e., nodes who only have one tie), the user should specify what values for constraint are returned for them via the *isolates* and *pendants* options.

Lastly, this function allows users to compute the values in parallel via the *foreach*, *doParallel*, and *parallel* R packages.

Value

The vector of ego network constraint values.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Burt, Ronald. 1992. *Structural Holes: The Social Structure of Competition*. Harvard University Press.

Examples

```
# For this example, we recreate the ego network provided in Burt (1992: 56):
BurtEgoNet <- matrix(c(
  0,1,0,0,1,1,1,
 1,0,0,1,0,0,1,
 0,0,0,0,0,0,1,
 0,1,0,0,0,0,1,
 1,0,0,0,0,0,1,
 1,0,0,0,0,0,1,
 1,1,1,1,1,1,0),
 nrow = 7, ncol = 7)
colnames(BurtEgoNet) <- rownames(BurtEgoNet) <- c("A", "B", "C", "D", "E",
                                                  "F", "ego")
#the constraint value for the ego replicates that provided in Burt (1992: 56)
computeBurtsConstraint(BurtEgoNet)
```

`computeBurtsES` *Compute Burt's (1992) Effective Size for Ego Networks from a Sociomatrix*

Description

This function computes Burt's (1992) one-mode ego effective size based upon a sociomatrix (see details).

Usage

```
computeBurtsES(
  net,
  inParallel = FALSE,
  nCores = NULL,
  isolates = NA,
  pendants = 1
)
```

Arguments

`net`	The one-mode sociomatrix with network ties.
`inParallel`	TRUE/FALSE. TRUE indicates that parallel processing will be used to compute the statistic with the *foreach* package. FALSE indicates that parallel processing will not be used. Set to FALSE by default.
`nCores`	If inParallel = TRUE, the number of computing cores for parallel processing. If this value is not specified, then the function internally provides it by dividing the number of available cores in half.
`isolates`	The numerical value that represents what value will isolates be given. Set to NA by default.

`pendants` The numerical value that represents what value will pendant vertices be given. Set to 1 by default.

Details

The formula for Burt's (1992; see also Borgatti 1997) one-mode ego effective size is:

$$E_i = \sum_j 1 - \sum_q p_{iq} m_{jq}; q \neq i \neq j$$

where E_i is the ego effective size for an ego *i*. p_{iq} is formulated as:

$$\frac{(z_{iq} + z_{qi})}{\sum_j (z_{ij} + z_{ji})}; i \neq j$$

and m_{jq} is:

$$m_{jq} = \frac{(z_{jq} + z_{qj})}{max(z_{jk} + z_{kj})}$$

While this function internally locates isolates (i.e., nodes who have no ties) and pendants (i.e., nodes who only have one tie), the user should specify what values for constraint are returned for them via the *isolates* and *pendants* options.

Value

The vector of ego network effective size values.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Burt, Ronald. 1992. *Structural Holes: The Social Structure of Competition*. Harvard University Press.

Borgatti, Stephen. 1997. "Structural Holes: Unpacking Burt's Redundancy Measures." *Connections* 20(1): 35-38.

Examples

```
# For this example, we recreate the ego network provided in Borgatti (1997):
BorgattiEgoNet <- matrix(
 c(0,1,0,0,0,0,0,0,1,
   1,0,0,0,0,0,0,0,1,
   0,0,0,1,0,0,0,0,1,
   0,0,1,0,0,0,0,0,1,
   0,0,0,0,0,1,0,0,1,
  0,0,0,0,1,0,0,0,1,
  0,0,0,0,0,0,0,1,1,
   0,0,0,0,0,0,1,0,1,
   1,1,1,1,1,1,1,1,0),
```

```
 nrow = 9, ncol = 9, byrow = TRUE)
colnames(BorgattiEgoNet) <- rownames(BorgattiEgoNet) <- c("A", "B", "C",
                                                          "D", "E", "F",
                                                         "G", "H", "ego")
#the effective size value for the ego replicates that provided in Borgatti (1997)
computeBurtsES(BorgattiEgoNet)

# For this example, we recreate the ego network provided in Burt (1992: 56):
BurtEgoNet <- matrix(c(
  0,1,0,0,1,1,1,
 1,0,0,1,0,0,1,
 0,0,0,0,0,0,1,
 0,1,0,0,0,0,1,
 1,0,0,0,0,0,1,
 1,0,0,0,0,0,1,
 1,1,1,1,1,1,0),
 nrow = 7, ncol = 7)
colnames(BurtEgoNet) <- rownames(BurtEgoNet) <- c("A", "B", "C", "D", "E",
                                                  "F", "ego")
#the effective size value for the ego replicates that provided in Burt (1992: 56)
computeBurtsES(BurtEgoNet)
```

`computeFourCycles` *Compute the Four-Cycles Network Statistic for Event Dyads in a Relational Event Sequence*

Description

The function computes the four-cycles network sufficient statistic for a two-mode relational sequence with the exponential weighting function (Lerner and Lomi 2020). In essence, the four-cycles measure captures the tendency for clustering to occur in the network of past events, whereby an event is more likely to occur between a sender node *a* and receiver node *b* given that *a* has interacted with other receivers in past events who have received events from other senders that interacted with *b* (e.g., Duxbury and Haynie 2021, Lerner and Lomi 2020). The function allows users to use two different weighting functions, reduce computational runtime, employ a sliding windows framework for large relational sequences, and specify a dyadic cutoff for relational relevancy.

Usage

```
computeFourCycles(
  observed_time,
  observed_sender,
  observed_receiver,
  processed_time,
  processed_sender,
  processed_receiver,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  counts = FALSE,
```

```
  halflife = 2,
  dyadic_weight = 0,
  window_size = NA,
  Lerneretal_2013 = FALSE,
  priorStats = FALSE,
  sender_OutDeg = NULL,
  receiver_InDeg = NULL
)
```

Arguments

observed_time	The vector of event times from the pre-processing event sequence.
observed_sender	The vector of event senders from the pre-processing event sequence.
observed_receiver	The vector of event receivers from the pre-processing event sequence
processed_time	The vector of event times from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
processed_sender	The vector of event senders from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
processed_receiver	The vector of event receivers from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
sliding_windows	TRUE/FALSE. TRUE indicates that the sliding windows computational approach will be used to compute the network statistic, while FALSE indicates the approach will not be used. Set to FALSE by default. It's important to note that the sliding windows framework should only be used when the pre-processed event sequence is 'big', such as the 360 million pre-processed event sequence used in Lerner and Lomi (2020), as it aims to reduce the computational burden of sorting 'big' datasets. In general, most pre-processed event sequences will not need to use the sliding windows approach. There is not a strict cutoff for 'big' dataset. This definition depends on both the size of the observed event sequence and the post-processing sampling dataset. For instance, according to our internal tests, when the event sequence is relatively large (i.e., 100,000 observed events) with probability of sampling from the observed event sequence set to 0.05 and using 10 controls per sampled event, the sliding windows framework for computing repetition is about 11% faster than the non-sliding windows framework. Yet, in a smaller dataset (i.e., 10,000 observed events) the sliding windows framework is about 25% slower than the non-sliding framework with the same conditions as before.
processed_seqIDs	If sliding_windows is set to TRUE, the vector of event sequence IDs from the post-processing event sequence. The event sequence IDs represents the index for when the event occurred in the observed event sequence (e.g., the 5th event in the sequence will have a value of 5 in this vector).

`counts` TRUE/FALSE. TRUE indicates that the counts of past events should be computed (see the details section). FALSE indicates that the temporal exponential weighting function should be used to downweigh past events (see the details section). Set to FALSE by default.

`halflife` A numerical value that is the halflife value to be used in the exponential weighting function (see the details section). Preset to 2 (should be updated by user).

`dyadic_weight` A numerical value that is the dyadic cutoff weight that represents the numerical cutoff value for temporal relevancy based on the exponential weighting function. For example, a numerical value of 0.01, indicates that an exponential weight less than 0.01 will become 0 and will not be included in the sum of the past event weights (see the details section). Set to 0 by default.

`window_size` If sliding_windows is set to TRUE, the sizes of the windows that are used for the sliding windows computational framework. If NA, the function internally divides the dataset into ten slices (may not be optimal).

`Lerneretal_2013` TRUE/FALSE. TRUE indicates that the Lerner et al. (2013) exponential weighting function will be used (see the details section). FALSE indicates that the Lerner and Lomi (2020) exponential weighting function will be used (see the details section). Set to FALSE by default.

`priorStats` TRUE/FALSE. Set to FALSE by default. TRUE indicates that the user has previously computed the sender outdegree and target indegree network statistics. Set to FALSE by default. The four-cycles network statistics is computationally burdensome. If priorStats =TRUE, the function speeds things up by setting the statistic for an event dyad to 0 if either a) the current event sender was not a sender in a previous event or b) the current event receiver was not a receiver in a past event, then the four-cycles statistics for that event dyad will be 0.

`sender_OutDeg` If priorStats = TRUE, the vector of previously computed sender outdegree scores.

`receiver_InDeg` If priorStats = TRUE, the vector of previously computed receiver indegree scores.

Details

The function calculates the four-cycles network statistic for two-mode relational event models based on the exponential weighting function used in either Lerner and Lomi (2020) or Lerner et al. (2013).

Following Lerner and Lomi (2020), the exponential weighting function in relational event models is:

$$w(s,r,t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}}$$

Following Lerner et al. (2013), the exponential weighting function in relational event models is:

$$w(s,r,t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}}$$

In both of the above equations, *s* is the current event sender, *r* is the current event receiver (target), *t* is the current event time, *t'* is the past event times that meet the weight subset (in this case, all events that have the same sender and receiver), and $T_{1/2}$ is the halflife parameter.

The formula for four-cycles for event e_i is:

$$fourcycles_{e_i} = \sqrt[3]{\sum_{s' and r'} w(s', r, t) \cdot w(s, r', t) \cdot w(s', r', t)}$$

That is, the four-cycle measure captures all the past event structures in which the current event pair, sender *s* and target *r* close a four-cycle. In particular, it finds all events in which: a past sender *s'* had a relational event with target *r*, a past target *r'* had a relational event with current sender *s*, and finally, a relational event occurred between sender *s'* and target *r'*.

Four-cycles are computationally expensive, especially for large relational event sequences (see Lerner and Lomi 2020 for a discussion on this), therefore this function allows the user to input previously computed target indegree and sender outdegree scores to reduce the runtime. Relational events where either the event target or event sender were not involved in any prior relational events (i.e., a target indegree or sender outdegree score of 0) will close no-four cycles. This function exploits this feature.

Moreover, researchers interested in modeling temporal relevancy (see Quintane, Mood, Dunn, and Falzone 2022; Lerner and Lomi 2020) can specify the dyadic weight cutoff, that is, the minimum value for which the weight is considered relationally relevant. Users who do not know the specific dyadic cutoff value to use, can use the `computeRemDyadCut` function.

Following Lerner and Lomi (2020), if the counts of the past events are requested, the formula for four-cycles formation for event e_i is:

$$fourcycles_{e_i} = \sum_{i=1}^{|S'|} \sum_{j=1}^{|R'|} \min \left[d(s'_i, r, t),\ d(s, r'_j, t),\ d(s'_i, r'_j, t) \right]$$

where, $d()$ is the number of past events that meet the specific set operations, $d(s'_i, r, t)$ is the number of past events where the current event receiver received a tie from another sender s'_i, $d(s, r'_j, t)$ is the number of past events where the current event sender sent a tie to a another receiver r'_j, and $d(s'_i, r'_j, t)$ is the number of past events where the sender s'_i sent a tie to the receiver r'_j. Moreover, the counting equation can leverage relational relevancy, by specifying the halflife parameter, exponential weighting function, and the dyadic cut off weight values (see the above sections for help with this). If the user is not interested in modeling relational relevancy, then those value should be left at their default values.

Value

The vector of four-cycle statistics for the two-mode relational event sequence.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Duxbury, Scott and Dana Haynie. 2021. "Shining a Light on the Shadows: Endogenous Trade Structure and the Growth of an Online Illegal Market." *American Journal of Sociology* 127(3): 787-827.

Quintane, Eric, Martin Wood, John Dunn, and Lucia Falzon. 2022. "Temporal Brokering: A Measure of Brokerage as a Behavioral Process." *Organizational Research Methods* 25(3): 459-489.

Lerner, Jürgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: How to fit a relational event model to 360 million dyadic events." *Network Science* 8(1): 97-135.

Lerner, Jürgen, Margit Bussman, Tom A.B. Snijders, and Ulrik Brandes. 2013. "Modeling Frequency and Type of Interaction in Event Networks." *The Corvinus Journal of Sociology and Social Policy* 4(1): 3-32.

Examples

```
data("WikiEvent2018.first100k")
WikiEvent2018 <- WikiEvent2018.first100k[1:1000,] #the first one thousand events
WikiEvent2018$time <- as.numeric(WikiEvent2018$time) #making the variable numeric
### Creating the EventSet By Employing Case-Control Sampling With M = 5 and
### Sampling from the Observed Event Sequence with P = 0.01
EventSet <- processTMEventSeq(
 data = WikiEvent2018, # The Event Dataset
 time = WikiEvent2018$time, # The Time Variable
 eventID = WikiEvent2018$eventID, # The Event Sequence Variable
 sender = WikiEvent2018$user, # The Sender Variable
 receiver = WikiEvent2018$article, # The Receiver Variable
 p_samplingobserved = 0.01, # The Probability of Selection
 n_controls = 8, # The Number of Controls to Sample from the Full Risk Set
 seed = 9999) # The Seed for Replication

#### Estimating the Four-Cycle Statistic Without the Sliding Windows Framework
EventSet$fourcycle <- computeFourCycles(
   observed_time = WikiEvent2018$time,
   observed_sender = WikiEvent2018$user,
   observed_receiver = WikiEvent2018$article,
   processed_time = EventSet$time,
   processed_sender = EventSet$sender,
   processed_receiver = EventSet$receiver,
   halflife = 2.592e+09, #halflife parameter
   dyadic_weight = 0,
   Lerneretal_2013 = FALSE)

#### Estimating the Four-Cycle Statistic With the Sliding Windows Framework
EventSet$cycle4SW <- computeFourCycles(
   observed_time = WikiEvent2018$time,
   observed_sender = WikiEvent2018$user,
   observed_receiver = WikiEvent2018$article,
   processed_time = EventSet$time,
   processed_sender = EventSet$sender,
   processed_receiver = EventSet$receiver,
   processed_seqIDs = EventSet$sequenceID,
   halflife = 2.592e+09, #halflife parameter
   dyadic_weight = 0,
   sliding_window = TRUE,
   Lerneretal_2013 = FALSE)
```

```
#The results with and without the sliding windows are the same (see correlation
#below). Using the sliding windows method is recommended when the data are
#big' so that memory allotment is more efficient.
cor(EventSet$fourcycle, EventSet$cycle4SW)

#### Estimating the Four-Cycle Statistic  with the Counts of Events Returned
EventSet$cycle4C <- computeFourCycles(
   observed_time = WikiEvent2018$time,
   observed_sender = WikiEvent2018$user,
   observed_receiver = WikiEvent2018$article,
   processed_time = EventSet$time,
   processed_sender = EventSet$sender,
   processed_receiver = EventSet$receiver,
   processed_seqIDs = EventSet$sequenceID,
   halflife = 2.592e+09, #halflife parameter
   dyadic_weight = 0,
   sliding_window = FALSE,
   counts = TRUE,
   Lerneretal_2013 = FALSE)

cbind(EventSet$fourcycle,
     EventSet$cycle4SW,
     EventSet$cycle4C)
```

`computeHomFourCycles` *Compute Fujimoto, Snijders, and Valente's (2018) Homophilous Four-Cycles for Two-Mode Networks*

Description

This function computes the number of homophilous four-cycles in a two-mode network as proposed by Fujimoto, Snijders, and Valente (2018: 380). See Fujimoto, Snijders, and Valente (2018) for more details about this measure.

Usage

```
computeHomFourCycles(net, mem)
```

Arguments

`net`	The two-mode adjacency matrix.
`mem`	The vector of membership values that the homophilous four-cycles will be based on.

Details

Following Fujimoto, Snijders, and Valente (2018: 380), the number of homophilous four-cycles for actor *i* is:

$$\sum_{j} \sum_{a \neq b} y_{ia} y_{ib} y_{ja} y_{jb} I v_i = v_j$$

where *y* is the two-mode adjacency matrix, *v* is the vector of membership scores (e.g., sports/club membership), *a* and *b* represent the level two groups, and $Iv_i = v_j$ is the indicator function that is 1 if the values are the same and 0 if not.

Value

The vector of counts of homophilous four-cycles for the two-mode network.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Fujimoto, Kayo, Tom A.B. Snijders, and Thomas W. Valente. 2018. "Multivariate dynamics of one-mode and two-mode networks: Explaining similarity in sports participation among friends." *Network Science* 6(3): 370-395.

Examples

```
# For this example, we use the Davis Southern Women's Dataset.
data("southern.women")
#creating a random binary membership vector
set.seed(9999)
membership <- sample(0:1, nrow(southern.women), replace = TRUE)
#the homophilous four-cycle values
computeHomFourCycles(southern.women, mem = membership)
```

`computeISP` *Compute Butts' (2008) Incoming Shared Partners Network Statistic for Event Dyads in a Relational Event Sequence*

Description

This function calculates the incoming shared partners (ISP) network sufficient statistic for a relational event sequence (see Lerner and Lomi 2020; Butts 2008). In essence, the incoming shared partners measure captures the tendency of triadic closure to occur in the network of past events, in which the past triadic closure is based upon the incoming shared partners structure (see Butts 2008 for an empirical example). This measure allows for ISP scores to be computed only for the sampled events, while creating the weights based on the full event sequence (see Lerner and Lomi 2020; Vu et al. 2015). The function allows users to use two different weighting functions, reduce computational runtime, employ a sliding windows framework for large relational sequences, and specify a dyadic cutoff for relational relevancy.

Usage

```
computeISP(
  observed_time,
  observed_sender,
  observed_receiver,
  processed_time,
  processed_sender,
  processed_receiver,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  counts = FALSE,
  halflife = 2,
  dyadic_weight = 0,
  window_size = NA,
  Lerneretal_2013 = FALSE
)
```

Arguments

observed_time	The vector of event times from the pre-processing event sequence.
observed_sender	The vector of event senders from the pre-processing event sequence.
observed_receiver	The vector of event receivers from the pre-processing event sequence.
processed_time	The vector of event times from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
processed_sender	The vector of event senders from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
processed_receiver	The vector of event receivers from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
sliding_windows	TRUE/FALSE. TRUE indicates that the sliding windows computational approach will be used to compute the network statistic, while FALSE indicates the approach will not be used. Set to FALSE by default. It's important to note that the sliding windows framework should only be used when the pre-processed event sequence is 'big', such as the 360 million pre-processed event sequence used in Lerner and Lomi (2020), as it aims to reduce the computational burden of sorting 'big' datasets. In general, most pre-processed event sequences will not need to use the sliding windows approach. There is not a strict cutoff for 'big' dataset. This definition depends on both the size of the observed event sequence and the post-processing sampling dataset. For instance, according to our internal tests, when the event sequence is relatively large (i.e., 100,000 observed events) with probability of sampling from the observed event sequence set to 0.05 and using 10 controls per sampled event, the sliding windows framework for computing repetition is about 11% faster than the non-sliding windows framework. Yet, in

a smaller dataset (i.e., 10,000 observed events) the sliding windows framework is about 25% slower than the non-sliding framework with the same conditions as before.

`processed_seqIDs`
: If sliding_windows is set to TRUE, the vector of event sequence IDs from the post-processing event sequence. The event sequence IDs represents the index for when the event occurred in the observed event sequence (e.g., the 5th event in the sequence will have a value of 5 in this vector).

`counts`
: TRUE/FALSE. TRUE indicates that the counts of past events should be computed (see the details section). FALSE indicates that the temporal exponential weighting function should be used to downweigh past events (see the details section). Set to FALSE by default.

`halflife`
: A numerical value that is the halflife value to be used in the exponential weighting function (see the details section). Preset to 2 (should be updated by user).

`dyadic_weight`
: A numerical value that is the dyadic cutoff weight that represents the numerical cutoff value for temporal relevancy based on the exponential weighting function. For example, a numerical value of 0.01, indicates that an exponential weight less than 0.01 will become 0 and will not be included in the sum of the past event weights (see the details section). Set to 0 by default.

`window_size`
: If sliding_windows is set to TRUE, the sizes of the windows that are used for the sliding windows computational framework. If NA, the function internally divides the dataset into ten slices (may not be optimal).

`Lerneretal_2013`
: TRUE/FALSE. TRUE indicates that the Lerner et al. (2013) exponential weighting function will be used (see the details section). FALSE indicates that the Lerner and Lomi (2020) exponential weighting function will be used (see the details section). Set to FALSE by default

Details

This function calculates incoming shared partners scores for relational event sequences based on the exponential weighting function used in either Lerner and Lomi (2020) or Lerner et al. (2013).

Following Lerner and Lomi (2020), the exponential weighting function in relational event models is:

$$w(s,r,t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}}$$

Following Lerner et al. (2013), the exponential weighting function in relational event models is:

$$w(s,r,t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}}$$

In both of the above equations, *s* is the current event sender, *r* is the current event receiver (target), *t* is the current event time, *t'* is the past event times that meet the weight subset, and $T_{1/2}$ is the halflife parameter.

The general formula for incoming shared partners for event e_i is:

$$ISP_{e_i} = \sqrt{\sum_h w(h,s,t) \cdot w(h,r,t)}$$

That is, as discussed in Butts (2008), incoming shared partners finds all past events where the current sender and target were themselves the target in a relational event from the same *h* node

Moreover, researchers interested in modeling temporal relevancy (see Quintane, Mood, Dunn, and Falzone 2022; Lerner and Lomi 2020) can specify the dyadic weight cutoff, that is, the minimum value for which the weight is considered relationally relevant. Users who do not know the specific dyadic cutoff value to use, can use the `computeRemDyadCut` function.

Following Butts (2008), if the counts of the past events are requested, the formula for incoming shared partners for event e_i is:

$$ISP_{e_i} = \sum_{i=1}^{|H|} \min\left[d(h,s,t), d(h,r,t)\right]$$

Where, $d()$ is the number of past events that meet the specific set operations, $d(h,s,t)$ is the number of past events where the current event sender received a tie from a third actor, *h*, and $d(h,r,t)$ is the number of past events where the current event receiver received a tie from a third actor, *h*. The sum loops through all unique actors that have formed past incoming shared partners structures with the current event sender and receiver. Moreover, the counting equation can leverage relational relevancy by specifying the halflife parameter, exponential weighting function, and the dyadic cut off weight values. If the user is not interested in modeling relational relevancy, then those value should be left at their defaults.

Value

The vector of incoming shared partner statistics for the relational event sequence.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A Relational Event Framework for Social Action." *Sociological Methodology* 38(1): 155-200.

Quintane, Eric, Martin Wood, John Dunn, and Lucia Falzon. 2022. "Temporal Brokering: A Measure of Brokerage as a Behavioral Process." *Organizational Research Methods* 25(3): 459-489.

Lerner, Jürgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: How to fit a relational event model to 360 million dyadic events." *Network Science* 8(1): 97-135.

Lerner, Jürgen, Margit Bussman, Tom A.B. Snijders, and Ulrik Brandes. 2013. " Modeling Frequency and Type of Interaction in Event Networks." *The Corvinus Journal of Sociology and Social Policy* 4(1): 3-32.

Vu, Duy, Philippa Pattison, and Garry Robins. 2015. "Relational event models for social learning in MOOCs." *Social Networks* 43: 121-135.

Examples

```
events <- data.frame(time = 1:18,
                                eventID = 1:18,
                                sender = c("A", "B", "C",
                                           "A", "D", "E",
                                           "F", "B", "A",
                                           "F", "D", "B",
                                           "G", "B", "D",
                                          "H", "A", "D"),
                               target = c("B", "C", "D",
                                          "E", "A", "F",
                                          "D", "A", "C",
                                          "G", "B", "C",
                                          "H", "J", "A",
                                          "F", "C", "B"))

eventSet <- processOMEventSeq(data = events,
                      time = events$time,
                      eventID = events$eventID,
                      sender = events$sender,
                      receiver = events$target,
                      p_samplingobserved = 1.00,
                      n_controls = 1,
                      seed = 9999)

# Computing Incoming Shared Partners Statistic without the sliding windows framework
eventSet$ISP <- computeISP(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   dyadic_weight = 0,
   Lerneretal_2013 = FALSE)

# Computing Incoming Shared Partners Statistic with the sliding windows framework
eventSet$ISP_SW <- computeISP(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   processed_seqIDs = eventSet$sequenceID,
   dyadic_weight = 0,
   sliding_window = TRUE,
   Lerneretal_2013 = FALSE)

#The results with and without the sliding windows are the same (see correlation
```

```
#below). Using the sliding windows method is recommended when the data are
#big' so that memory allotment is more efficient.
cor(eventSet$ISP , eventSet$ISP_SW)

# Computing Incoming Shared Partners Statistics with the counts of events being returned
eventSet$ISPC <- computeISP(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   dyadic_weight = 0,
   sliding_window = FALSE,
   counts = TRUE,
   Lerneretal_2013 = FALSE)

cbind(eventSet$ISP,
     eventSet$ISP_SW,
     eventSet$ISPC)
```

`computeITP` *Compute Butts' (2008) Incoming Two Paths Network Statistic for Event Dyads in a Relational Event Sequence*

Description

The function computes the incoming two path (ITP) network sufficient statistic for a relational event sequence (see Lerner and Lomi 2020; Butts 2008). In essence, the incoming two paths measure captures the tendency of triadic closure to occur in the network of past events, in which the past triadic closure is based upon the incoming two paths structure (see Butts 2008 for an empirical example). This measure allows for ITP scores to be only computed for the sampled events, while creating the weights based on the full event sequence (see Lerner and Lomi 2020; Vu et al. 2015). The function allows users to use two different weighting functions, reduce computational runtime, employ a sliding windows framework for large relational sequences, and specify a dyadic cutoff for relational relevancy.

Usage

```
computeITP(
  observed_time,
  observed_sender,
  observed_receiver,
  processed_time,
  processed_sender,
  processed_receiver,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
```

```
  counts = FALSE,
  halflife = 2,
  dyadic_weight = 0,
  window_size = NA,
  Lerneretal_2013 = FALSE
)
```

Arguments

observed_time	The vector of event times from the pre-processing event sequence.
observed_sender	The vector of event senders from the pre-processing event sequence.
observed_receiver	The vector of event receivers from the pre-processing event sequence
processed_time	The vector of event times from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
processed_sender	The vector of event senders from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
processed_receiver	The vector of event receivers from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
sliding_windows	TRUE/FALSE. TRUE indicates that the sliding windows computational approach will be used to compute the network statistic, while FALSE indicates the approach will not be used. Set to FALSE by default. It's important to note that the sliding windows framework should only be used when the pre-processed event sequence is 'big', such as the 360 million pre-processed event sequence used in Lerner and Lomi (2020), as it aims to reduce the computational burden of sorting 'big' datasets. In general, most pre-processed event sequences will not need to use the sliding windows approach. There is not a strict cutoff for 'big' dataset. This definition depends on both the size of the observed event sequence and the post-processing sampling dataset. For instance, according to our internal tests, when the event sequence is relatively large (i.e., 100,000 observed events) with probability of sampling from the observed event sequence set to 0.05 and using 10 controls per sampled event, the sliding windows framework for computing repetition is about 11% faster than the non-sliding windows framework. Yet, in a smaller dataset (i.e., 10,000 observed events) the sliding windows framework is about 25% slower than the non-sliding framework with the same conditions as before.
processed_seqIDs	If sliding_windows is set to TRUE, the vector of event sequence IDs from the post-processing event sequence. The event sequence IDs represents the index for when the event occurred in the observed event sequence (e.g., the 5th event in the sequence will have a value of 5 in this vector).
counts	TRUE/FALSE. TRUE indicates that the counts of past events should be computed (see the details section). FALSE indicates that the temporal exponential

weighting function should be used to downweigh past events (see the details section). Set to FALSE by default.

`halflife` A numerical value that is the halflife value to be used in the exponential weighting function (see the details section). Preset to 2 (should be updated by user).

`dyadic_weight` A numerical value that is the dyadic cutoff weight that represents the numerical cutoff value for temporal relevancy based on the exponential weighting function. For example, a numerical value of 0.01, indicates that an exponential weight less than 0.01 will become 0 and will not be included in the sum of the past event weights (see the details section). Set to 0 by default.

`window_size` If sliding_windows is set to TRUE, the sizes of the windows that are used for the sliding windows computational framework. If NA, the function internally divides the dataset into ten slices (may not be optimal).

`Lerneretal_2013`
TRUE/FALSE. TRUE indicates that the Lerner et al. (2013) exponential weighting function will be used (see the details section). FALSE indicates that the Lerner and Lomi (2020) exponential weighting function will be used (see the details section). Set to FALSE by default

Details

The function calculates incoming two paths scores for relational event sequences based on the exponential weighting function used in either Lerner and Lomi (2020) or Lerner et al. (2013).

Following Lerner and Lomi (2020), the exponential weighting function in relational event models is:

$$w(s, r, t) = e^{-(t-t') \cdot \frac{ln(2)}{T_{1/2}}}$$

Following Lerner et al. (2013), the exponential weighting function in relational event models is:

$$w(s, r, t) = e^{-(t-t') \cdot \frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}}$$

In both of the above equations, *s* is the current event sender, *r* is the current event receiver (target), *t* is the current event time, *t'* is the past event times that meet the weight subset, and $T_{1/2}$ is the halflife parameter.

The general formula for incoming two paths for event e_i is:

$$ITP_{e_i} = \sqrt{\sum_h w(r, h, t) \cdot w(h, s, t)}$$

That is, as discussed in Butts (2008), incoming two paths finds all past events where the current sender was the receiver in a relational event where the sender was a node h and the current target was the sender in a past relational event where the target was the same node h.

Moreover, researchers interested in modeling temporal relevancy (see Quintane, Mood, Dunn, and Falzone 2022; Lerner and Lomi 2020) can specify the dyadic weight cutoff, that is, the minimum value for which the weight is considered relationally relevant. Users who do not know the specific dyadic cutoff value to use, can use the `computeRemDyadCut` function.

Following Butts (2008), if the counts of the past events are requested, the formula for incoming two paths for event e_i is:

$$ITP_{e_i} = \sum_{i=1}^{|H|} \min\left[d(r,h,t), d(h,s,t\right]$$

Where, $d()$ is the number of past events that meet the specific set operations. $d(r,h,t)$ is the number of past events where the current event receiver sent a tie to a third actor, *h*, and $d(h,s,t$ is the number of past events where the third actor *h* sent a tie to the current event sender. The sum loops through all unique actors that have formed past incoming two path structures with the current event sender and receiver. Moreover, the counting equation can be used in tandem with relational relevancy, by specifying the halflife parameter, exponential weighting function, and the dyadic cut off weight values. If the user is not interested in modeling relational relevancy, then those value should be left at their baseline values.

Value

The vector of incoming two path statistics for the relational event sequence.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A Relational Event Framework for Social Action." *Sociological Methodology* 38(1): 155-200.

Quintane, Eric, Martin Wood, John Dunn, and Lucia Falzon. 2022. "Temporal Brokering: A Measure of Brokerage as a Behavioral Process." *Organizational Research Methods* 25(3): 459-489.

Lerner, Jürgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: How to fit a relational event model to 360 million dyadic events." *Network Science* 8(1): 97-135.

Lerner, Jürgen, Margit Bussman, Tom A.B. Snijders, and Ulrik Brandes. 2013. " Modeling Frequency and Type of Interaction in Event Networks." *The Corvinus Journal of Sociology and Social Policy* 4(1): 3-32.

Vu, Duy, Philippa Pattison, and Garry Robins. 2015. "Relational event models for social learning in MOOCs." *Social Networks* 43: 121-135.

Examples

```
events <- data.frame(time = 1:18,
                                eventID = 1:18,
                                sender = c("A", "B", "C",
                                           "A", "D", "E",
                                           "F", "B", "A",
                                           "F", "D", "B",
                                           "G", "B", "D",
                                          "H", "A", "D"),
                               target = c("B", "C", "D",
                                          "E", "A", "F",
```

```
                                          "D", "A", "C",
                                          "G", "B", "C",
                                          "H", "J", "A",
                                          "F", "C", "B"))

eventSet <- processOMEventSeq(data = events,
                      time = events$time,
                      eventID = events$eventID,
                      sender = events$sender,
                      receiver = events$target,
                      p_samplingobserved = 1.00,
                      n_controls = 1,
                      seed = 9999)

# Computing Incoming Two Paths Statistics without the sliding windows framework
eventSet$ITP <- computeITP(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   dyadic_weight = 0,
   Lerneretal_2013 = FALSE)

# Computing Incoming Two Paths Statistics with the sliding windows framework
eventSet$ITP_SW <- computeITP(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   processed_seqIDs = eventSet$sequenceID,
   dyadic_weight = 0,
   sliding_window = TRUE,
   Lerneretal_2013 = FALSE)

#The results with and without the sliding windows are the same (see correlation
#below). Using the sliding windows method is recommended when the data are
#big' so that memory allotment is more efficient.
cor(eventSet$ITP, eventSet$ITP_SW)

# Computing Incoming Shared Partners Statistics with the counts of events being returned
eventSet$ITPC <- computeITP(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   processed_receiver = eventSet$receiver,
```

```
   halflife = 2, #halflife parameter
   dyadic_weight = 0,
   sliding_window = FALSE,
   counts = TRUE,
   Lerneretal_2013 = FALSE)

cbind(eventSet$ITP,
     eventSet$ITP_SW,
     eventSet$ITPC)
```

`computeLealBrokerage` *Compute Potential for Cultural Brokerage (PIB) Based on Leal (2025)*

Description

Following Leal (2025), this function calculates node's Potential for Intercultural Brokerage (PIB) in a one-mode network. For example, users can examine PIB across gender. The option count determines what is returned by the function. If count is true, then the count of culturally dissimilar pairs brokered by ego is included (i.e., ego's total count of brokered open triangles where the alters at the two endpoints of said open triangles are culturally dissimilar from one another). If count is false, the proportion of ego's brokered open triangles where the endpoints are culturally dissimilar out of all of ego's brokered open triangles (regardless of the cultural identity of the alters) is returned. The formula for computing interpersonal brokerage is presented in the details section.

Usage

```
computeLealBrokerage(
  net,
  g.mem,
  symmetric = TRUE,
  triad.type = NULL,
  count = TRUE,
  isolate = NA
)
```

Arguments

`net`	The one-mode adjacency matrix.
`g.mem`	The vector of membership values that the brokerage scores will be based on.
`symmetric`	TRUE/FALSE. TRUE indicates that network matrix will be treated as symmetric. FALSE indicates that the network matrix will be treated as asymmetric. Set to TRUE by default.
`triad.type`	The string value (or vector) that indicates what specific triadic (star) structures the potential for cultural brokerage will be computed for. Possible values are "ANY", "OTS", "ITS", "MTS" (see the details section). The function defaults to "ANY".

count	TRUE/FALSE. TRUE indicates that the number of culturally brokered open triangles will be returned. FALSE indicates that the proportion of culturally brokered open triangles to all open triangles will be returned (see the details section). Set to TRUE by default.
isolate	If count = FALSE, the numerical value that will be given to isolates. This value is set to NA by default, as 0/0 is undefined. The user can specify this value!

Details

Following Leal (2025), the formula for interpersonal brokerage is:

$$\text{PIB}_i = \sum_{j<k} \frac{S_{jik}}{S_{jk}} m_{jk}, \quad S_{jik} \neq 0 \text{ and } i \neq j \neq k$$

where:

- $S_{jik} = 1$ if there is an (un)directed two-path connecting actors *j* and *k* through actor *i*; 0 otherwise.
- $m_{jk} = 1$ if actors *j* and *k* are on different sides of a symbolic boundary; 0 otherwise.
- Following Gould (1989), S_{jik} represents the total number of two-paths between actors *j* and *k*.

If the network is non-symmetric (i.e., the user specified symmetric = FALSE), then the function can compute the cultural brokerage scores for different star structures. The possible values are: "ANY", which computes the scores for all structures, where a tie exists between *i* and *j*, *j* and *k*, and one does not exist between *i* and *k*. "OTS" computes the values for outgoing two-stars (i<-j->k or the 021D triad according to the M.A.N. notation; see Wasserman and Faust 1994), where j is the broker. "ITS" computes the values for incoming two-stars (i->j<-k or the 021U triad according to the M.A.N. notation; see Wasserman and Faust 1994), where j is the broker. "MTS" computes PIB for mixed triadic structures (i<-j<-k or i->j->k or the 021C triad according to the M.A.N. notation; see Wasserman and Faust 1994). If not specified, the function defaults to the "ANY" category. This function can also compute all of the formations at once.

Value

The vector of interpersonal cultural brokerage values for the one-mode network.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Gould, Roger. 1989. "Power and Social Structure in Community Elites." *Social Forces* 68(2): 531-552.

Leal, Diego F. 2025. "Locating Cultural Holes Brokers in Diffusion Dynamics Across Bright Symbolic Boundaries." *Sociological Methods & Research* doi:10.1177/00491241251322517

Wasserman, Stanley and Katherine Faust. 1994. *Social Network Analysis: Methods and Applications*. Cambridge: Cambridge University Press.

Examples

```
# For this example, we recreate Figure 3 in Leal (2025)
LealNet <- matrix( c(
 0,1,0,0,0,0,0,
 1,0,1,1,0,0,0,
 0,1,0,0,1,1,0,
 0,1,0,0,1,0,0,
 0,0,1,1,0,0,0,
 0,0,1,0,0,0,1,
 0,0,0,0,0,1,0),
 nrow = 7, ncol = 7, byrow = TRUE)

colnames(LealNet) <- rownames(LealNet) <- c("A", "B", "C","D",
                                            "E", "F", "G")
categorical_variable <- c(0,0,1,0,0,0,0)
#These values are exactly the same as reported by Leal (2025)
computeLealBrokerage(LealNet,
   symmetric = TRUE,
   g.mem = categorical_variable)
```

`computeNPaths` *Compute the Number of Paths of Length K in a One-Mode Network*

Description

This function calculates the number of paths of length *k* between any two vertices in an unweighted one-mode network.

Usage

```
computeNPaths(net, k)
```

Arguments

`net`	An unweighted one-mode network adjacency matrix.
`k`	A numerical value that corresponds to the length of the paths to be computed.

Details

A nice result from graph theory is that the number of paths of length *k* between vertices *i* and *j* can be found by:

$$A_{ij}^{k}$$

This function is similar to the functions provided in *igraph* that provide the path between two vertices. The main difference is that this function provides the counts of paths between all vertices in the network. In addition, this function assumes that there are no self-loops (i.e., the diagonal of the matrix is 0).

Value

An n x n matrix of counts of paths.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

Examples

```
# For this example, we generate a random one-mode graph with the sna package.
#creating the random network with 10 actors
set.seed(9999)
rnet <- matrix(sample(c(0,1), 10*10, replace = TRUE, prob = c(0.8,0.2)),
               nrow = 10, ncol = 10, byrow = TRUE)
diag(rnet) <- 0 #setting self ties to 0
#counting the paths of length 2
computeNPaths(rnet, k = 2)
#counting the paths of length 5
computeNPaths(rnet, k = 5)
```

---

computeOSP *Compute Butts' (2008) Outgoing Shared Partners Network Statistic for Event Dyads in a Relational Event Sequence*

Description

The function computes the outgoing shared partners (OSP) network sufficient statistic for a relational event sequence (see Lerner and Lomi 2020; Butts 2008). In essence, the outgoing shared partners measure captures the tendency of triadic closure to occur in the network of past events, in which the past triadic closure is based upon the outgoing shared partners structure (see Butts 2008 for an empirical example). This measure allows for OSP scores to be only computed for the sampled events, while creating the weights based on the full event sequence (see Lerner and Lomi 2020; Vu et al. 2015). The function allows users to use two different weighting functions, reduce computational runtime, employ a sliding windows framework for large relational sequences, and specify a dyadic cutoff for relational relevancy.

Usage

```
computeOSP(
  observed_time,
  observed_sender,
  observed_receiver,
  processed_time,
  processed_sender,
  processed_receiver,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  counts = FALSE,
```

```
  halflife = 2,
  dyadic_weight = 0,
  window_size = NA,
  Lerneretal_2013 = FALSE
)
```

Arguments

observed_time	The vector of event times from the pre-processing event sequence.
observed_sender	The vector of event senders from the pre-processing event sequence.
observed_receiver	The vector of event receivers from the pre-processing event sequence
processed_time	The vector of event times from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
processed_sender	The vector of event senders from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
processed_receiver	The vector of event receivers from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
sliding_windows	TRUE/FALSE. TRUE indicates that the sliding windows computational approach will be used to compute the network statistic, while FALSE indicates the approach will not be used. Set to FALSE by default. It's important to note that the sliding windows framework should only be used when the pre-processed event sequence is 'big', such as the 360 million pre-processed event sequence used in Lerner and Lomi (2020), as it aims to reduce the computational burden of sorting 'big' datasets. In general, most pre-processed event sequences will not need to use the sliding windows approach. There is not a strict cutoff for 'big' dataset. This definition depends on both the size of the observed event sequence and the post-processing sampling dataset. For instance, according to our internal tests, when the event sequence is relatively large (i.e., 100,000 observed events) with probability of sampling from the observed event sequence set to 0.05 and using 10 controls per sampled event, the sliding windows framework for computing repetition is about 11% faster than the non-sliding windows framework. Yet, in a smaller dataset (i.e., 10,000 observed events) the sliding windows framework is about 25% slower than the non-sliding framework with the same conditions as before.
processed_seqIDs	If sliding_windows is set to TRUE, the vector of event sequence IDs from the post-processing event sequence. The event sequence IDs represents the index for when the event occurred in the observed event sequence (e.g., the 5th event in the sequence will have a value of 5 in this vector).
counts	TRUE/FALSE. TRUE indicates that the counts of past events should be computed (see the details section). FALSE indicates that the temporal exponential weighting function should be used to downweigh past events (see the details section). Set to FALSE by default.

`halflife` A numerical value that is the halflife value to be used in the exponential weighting function (see the details section). Preset to 2 (should be updated by user).

`dyadic_weight` A numerical value that is the dyadic cutoff weight that represents the numerical cutoff value for temporal relevancy based on the exponential weighting function. For example, a numerical value of 0.01, indicates that an exponential weight less than 0.01 will become 0 and will not be included in the sum of the past event weights (see the details section). Set to 0 by default.

`window_size` If sliding_windows is set to TRUE, the sizes of the windows that are used for the sliding windows computational framework. If NA, the function internally divides the dataset into ten slices (may not be optimal).

`Lerneretal_2013`
TRUE/FALSE. TRUE indicates that the Lerner et al. (2013) exponential weighting function will be used (see the details section). FALSE indicates that the Lerner and Lomi (2020) exponential weighting function will be used (see the details section). Set to FALSE by default

Details

The function calculates the outgoing shared partners statistics for relational event sequences based on the exponential weighting function used in either Lerner and Lomi (2020) or Lerner et al. (2013).

Following Lerner and Lomi (2020), the exponential weighting function in relational event models is:

$$w(s,r,t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}}$$

Following Lerner et al. (2013), the exponential weighting function in relational event models is:

$$w(s,r,t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}}$$

In both of the above equations, *s* is the current event sender, *r* is the current event receiver (target), *t* is the current event time, *t'* is the past event times that meet the weight subset, and $T_{1/2}$ is the halflife parameter.

The general formula for outgoing shared partners for event e_i is:

$$OSP_{e_i} = \sqrt{\sum_h w(s,h,t) \cdot w(r,h,t)}$$

That is, as discussed in Butts (2008), outgoing shared partners finds all past events where the current sender and target sent a relational tie (i.e., were a sender in a relational event) to the same *h* node.

Moreover, researchers interested in modeling temporal relevancy (see Quintane, Mood, Dunn, and Falzone 2022; Lerner and Lomi 2020) can specify the dyadic weight cutoff, that is, the minimum value for which the weight is considered relationally relevant. Users who do not know the specific dyadic cutoff value to use, can use the `computeRemDyadCut` function.

Following Butts (2008), if the counts of the past events are requested, the formula for outgoing shared partners for event e_i is:

$$OSPe_i = \sum_{i=1}^{|H|} \min\left[d(s,h,t), d(s,h,t)\right]$$

Where, $d()$ is the number of past events that meet the specific set operations. $d(s, h, t)$ is the number of past events where the current event sender sent a tie to a third actor, h, and $d(r, h, t)$ is the number of past events where the current event receiver sent a tie to a third actor, h. The sum loops through all unique actors that have formed past outgoing shared partners structures with the current event sender and receiver. Moreover, the counting equation can be used in tandem with relational relevancy, by specifying the halflife parameter, exponential weighting function, and the dyadic cut off weight values. If the user is not interested in modeling relational relevancy, then those value should be left at their defaults.

Value

The vector of outgoing shared partner statistics for the relational event sequence.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A Relational Event Framework for Social Action." *Sociological Methodology* 38(1): 155-200.

Quintane, Eric, Martin Wood, John Dunn, and Lucia Falzon. 2022. "Temporal Brokering: A Measure of Brokerage as a Behavioral Process." *Organizational Research Methods* 25(3): 459-489.

Lerner, Jürgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: How to fit a relational event model to 360 million dyadic events." *Network Science* 8(1): 97-135.

Lerner, Jürgen, Margit Bussman, Tom A.B. Snijders, and Ulrik Brandes. 2013. " Modeling Frequency and Type of Interaction in Event Networks." *The Corvinus Journal of Sociology and Social Policy* 4(1): 3-32.

Vu, Duy, Philippa Pattison, and Garry Robins. 2015. "Relational event models for social learning in MOOCs." *Social Networks* 43: 121-135.

Examples

```
events <- data.frame(time = 1:18,
                                 eventID = 1:18,
                                 sender = c("A", "B", "C",
                                            "A", "D", "E",
                                            "F", "B", "A",
                                            "F", "D", "B",
                                            "G", "B", "D",
                                           "H", "A", "D"),
                                target = c("B", "C", "D",
                                           "E", "A", "F",
                                           "D", "A", "C",
                                           "G", "B", "C",
                                           "H", "J", "A",
                                           "F", "C", "B"))
```

```
eventSet <- processOMEventSeq(data = events,
                      time = events$time,
                      eventID = events$eventID,
                      sender = events$sender,
                      receiver = events$target,
                      p_samplingobserved = 1.00,
                      n_controls = 1,
                      seed = 9999)

# Computing Outgoing Shared Partners Statistics without the sliding windows framework
eventSet$OSP <- computeOSP(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   dyadic_weight = 0,
   Lerneretal_2013 = FALSE)

# Computing Outgoing Shared Partners Statistics with the sliding windows framework
eventSet$OSP_SW <- computeOSP(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   processed_seqIDs = eventSet$sequenceID,
   dyadic_weight = 0,
   sliding_window = TRUE,
   Lerneretal_2013 = FALSE)

#The results with and without the sliding windows are the same (see correlation
#below). Using the sliding windows method is recommended when the data are
#big' so that memory allotment is more efficient.
cor(eventSet$OSP , eventSet$OSP_SW)

# Computing  Outgoing Shared Partners Statistics with the counts of events being returned
eventSet$OSP_C <- computeOSP(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   dyadic_weight = 0,
   sliding_window = FALSE,
   counts = TRUE,
   Lerneretal_2013 = FALSE)
```

```
cbind(eventSet$OSP,
     eventSet$OSP_SW,
     eventSet$OSP_C)
```

`computeOTP` *Compute Butts' (2008) Outgoing Two Paths Network Statistic for Event Dyads in a Relational Event Sequence*

Description

The function computes the outgoing two paths (OTP) network sufficient statistic for a relational event sequence (see Lerner and Lomi 2020; Butts 2008). In essence, the outgoing two paths measure captures the tendency of triadic closure to occur in the network of past events, in which the past triadic closure is based upon the outgoing two paths structure (see Butts 2008 for an empirical example). This measure allows for OTP scores to be only computed for the sampled events, while creating the weights based on the full event sequence (see Lerner and Lomi 2020; Vu et al. 2015). The function allows users to use two different weighting functions, reduce computational runtime, employ a sliding windows framework for large relational sequences, and specify a dyadic cutoff for relational relevancy.

Usage

```
computeOTP(
  observed_time,
  observed_sender,
  observed_receiver,
  processed_time,
  processed_sender,
  processed_receiver,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  counts = FALSE,
  halflife = 2,
  dyadic_weight = 0,
  window_size = NA,
  Lerneretal_2013 = FALSE
)
```

Arguments

`observed_time` The vector of event times from the pre-processing event sequence.

`observed_sender`
: The vector of event senders from the pre-processing event sequence.

`observed_receiver`
: The vector of event receivers from the pre-processing event sequence

`processed_time` The vector of event times from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).

`processed_sender`
The vector of event senders from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).

`processed_receiver`
The vector of event receivers from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).

`sliding_windows`
TRUE/FALSE. TRUE indicates that the sliding windows computational approach will be used to compute the network statistic, while FALSE indicates the approach will not be used. Set to FALSE by default. It's important to note that the sliding windows framework should only be used when the pre-processed event sequence is 'big', such as the 360 million pre-processed event sequence used in Lerner and Lomi (2020), as it aims to reduce the computational burden of sorting 'big' datasets. In general, most pre-processed event sequences will not need to use the sliding windows approach. There is not a strict cutoff for 'big' dataset. This definition depends on both the size of the observed event sequence and the post-processing sampling dataset. For instance, according to our internal tests, when the event sequence is relatively large (i.e., 100,000 observed events) with probability of sampling from the observed event sequence set to 0.05 and using 10 controls per sampled event, the sliding windows framework for computing repetition is about 11% faster than the non-sliding windows framework. Yet, in a smaller dataset (i.e., 10,000 observed events) the sliding windows framework is about 25% slower than the non-sliding framework with the same conditions as before.

`processed_seqIDs`
If sliding_windows is set to TRUE, the vector of event sequence IDs from the post-processing event sequence. The event sequence IDs represents the index for when the event occurred in the observed event sequence (e.g., the 5th event in the sequence will have a value of 5 in this vector).

`counts` TRUE/FALSE. TRUE indicates that the counts of past events should be computed (see the details section). FALSE indicates that the temporal exponential weighting function should be used to downweigh past events (see the details section). Set to FALSE by default.

`halflife` A numerical value that is the halflife value to be used in the exponential weighting function (see the details section). Preset to 2 (should be updated by user).

`dyadic_weight` A numerical value that is the dyadic cutoff weight that represents the numerical cutoff value for temporal relevancy based on the exponential weighting function. For example, a numerical value of 0.01, indicates that an exponential weight less than 0.01 will become 0 and will not be included in the sum of the past event weights (see the details section). Set to 0 by default.

`window_size` If sliding_windows is set to TRUE, the sizes of the windows that are used for the sliding windows computational framework. If NA, the function internally divides the dataset into ten slices (may not be optimal).

`Lerneretal_2013`
TRUE/FALSE. TRUE indicates that the Lerner et al. (2013) exponential weighting function will be used (see the details section). FALSE indicates that the

Lerner and Lomi (2020) exponential weighting function will be used (see the details section). Set to FALSE by default

Details

The function calculates the outgoing two paths statistic for relational event sequences based on the exponential weighting function used in either Lerner and Lomi (2020) or Lerner et al. (2013).

Following Lerner and Lomi (2020), the exponential weighting function in relational event models is:

$$w(s, r, t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}}$$

Following Lerner et al. (2013), the exponential weighting function in relational event models is:

$$w(s, r, t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}}$$

In both of the above equations, *s* is the current event sender, *r* is the current event receiver (target), *t* is the current event time, *t'* is the past event times that meet the weight subset, and $T_{1/2}$ is the halflife parameter.

The general formula for outgoing two paths for event e_i is:

$$OTP_{e_i} = \sqrt{\sum_h w(s, h, t) \cdot w(h, r, t)}$$

That is, as discussed in Butts (2008), outgoing two paths finds all past events where the current sender sends a relational tie to node *h* and the current target receives a relational tie from the same *h* node.

Moreover, researchers interested in modeling temporal relevancy (see Quintane, Mood, Dunn, and Falzone 2022; Lerner and Lomi 2020) can specify the dyadic weight cutoff, that is, the minimum value for which the weight is considered relationally relevant. Users who do not know the specific dyadic cutoff value to use, can use the `computeRemDyadCut` function.

Following Butts (2008), if the counts of the past events are requested, the formula for outgoing two paths for event e_i is:

$$OTP_{e_i} = \sum_{i=1}^{|H|} \min\left[d(s, h, t), d(h, r, t)\right]$$

Where, $d()$ is the number of past events that meet the specific set operations. $d(s, h, t)$ is the number of past events where the current event sender sent a tie to a third actor, *h*, and $d(h, r, t)$ is the number of past events where the third actor *h* sent a tie to the current event receiver. The sum loops through all unique actors that have formed past outgoing two path structures with the current event sender and receiver. Moreover, the counting equation can be used in tandem with relational relevancy, by specifying the halflife parameter, exponential weighting function, and the dyadic cut off weight values. If the user is not interested in modeling relational relevancy, then those values should be left at their defaults.

Value

The vector of outgoing two path statistics for the relational event sequence.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A Relational Event Framework for Social Action." *Sociological Methodology* 38(1): 155-200.

Quintane, Eric, Martin Wood, John Dunn, and Lucia Falzon. 2022. "Temporal Brokering: A Measure of Brokerage as a Behavioral Process." *Organizational Research Methods* 25(3): 459-489.

Lerner, Jürgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: How to fit a relational event model to 360 million dyadic events." *Network Science* 8(1): 97-135.

Lerner, Jürgen, Margit Bussman, Tom A.B. Snijders, and Ulrik Brandes. 2013. " Modeling Frequency and Type of Interaction in Event Networks." *The Corvinus Journal of Sociology and Social Policy* 4(1): 3-32.

Vu, Duy, Philippa Pattison, and Garry Robins. 2015. "Relational event models for social learning in MOOCs." *Social Networks* 43: 121-135.

Examples

```
events <- data.frame(time = 1:18,
                                eventID = 1:18,
                                sender = c("A", "B", "C",
                                           "A", "D", "E",
                                           "F", "B", "A",
                                           "F", "D", "B",
                                           "G", "B", "D",
                                          "H", "A", "D"),
                               target = c("B", "C", "D",
                                          "E", "A", "F",
                                          "D", "A", "C",
                                          "G", "B", "C",
                                          "H", "J", "A",
                                          "F", "C", "B"))

eventSet <- processOMEventSeq(data = events,
                      time = events$time,
                      eventID = events$eventID,
                      sender = events$sender,
                      receiver = events$target,
                      p_samplingobserved = 1.00,
                      n_controls = 1,
                      seed = 9999)

# Computing Outgoing Two Paths Statistics without the sliding windows framework
eventSet$OTP <- computeOTP(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
   processed_time = eventSet$time,
```

```
   processed_sender = eventSet$sender,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   dyadic_weight = 0,
   Lerneretal_2013 = FALSE)

# Computing Outgoing Two Paths Statistics with the sliding windows framework
eventSet$OTP_SW <- computeOTP(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   processed_seqIDs = eventSet$sequenceID,
   dyadic_weight = 0,
   sliding_window = TRUE,
   Lerneretal_2013 = FALSE)

#The results with and without the sliding windows are the same (see correlation
#below). Using the sliding windows method is recommended when the data are
#big' so that memory allotment is more efficient.
cor(eventSet$OTP , eventSet$OTP_SW)

# Computing Outgoing Two Paths Statistics with the counts of events being returned
eventSet$OTPC <- computeOTP(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   dyadic_weight = 0,
   sliding_window = FALSE,
   counts = TRUE,
   Lerneretal_2013 = FALSE)

cbind(eventSet$OTP,
     eventSet$OTP_SW,
     eventSet$OTPC)
```

`computePersistence` *Compute Butts' (2008) Persistence Network Statistic for Event Dyads in a Relational Event Sequence*

Description

This function computes the persistence network sufficient statistic for a relational event sequence (see Butts 2008). Persistence measures the proportion of past ties sent from the event sender that

went to the current event receiver. Furthermore, this measure allows for persistence scores to be only computed for the sampled events, while creating the weights based on the full event sequence. Moreover, the function allows users to specify relational relevancy for the statistic and employ a sliding windows framework for large relational sequences.

Usage

```
computePersistence(
  observed_time,
  observed_sender,
  observed_receiver,
  processed_time,
  processed_sender,
  processed_receiver,
  sender = TRUE,
  dependency = FALSE,
  relationalTimeSpan = NULL,
  nopastEvents = NA,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  window_size = NA
)
```

Arguments

observed_time	The vector of event times from the pre-processing event sequence.
observed_sender	The vector of event senders from the pre-processing event sequence.
observed_receiver	The vector of event receivers from the pre-processing event sequence
processed_time	The vector of event times from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
processed_sender	The vector of event senders from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
processed_receiver	The vector of event receivers from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
sender	TRUE/FALSE. TRUE indicates that the persistence statistic will be computed in reference to the sender's past relational history (see details section). FALSE indicates that the persistence statistic will be computed in reference to the target's past relational history (see details section). Set to TRUE by default.
dependency	TRUE/FALSE. TRUE indicates that temporal relevancy will be modeled (see the details section). FALSE indicates that temporal relevancy will not be modeled, that is, all past events are relevant (see the details section). Set to FALSE by default.

`relationalTimeSpan`
: If dependency = TRUE, a numerical value that corresponds to the temporal span for relational relevancy, which must be the same measurement unit as the observed_time and processed_time objects. When dependency = TRUE, the relevant events are events that have occurred between current event time, *t*, and *t-relationalTimeSpan*. For example, if the time measurement is the number of days since the first event and the value for relationalTimeSpan is set to 10, then only those events which occurred in the past 10 days are included in the computation of the statistic.

`nopastEvents`
: The numerical value that specifies what value should be given to events in which the sender has sent not past ties (i's neighborhood when sender = TRUE) or has not received any past ties (j's neighborhood when sender = FALSE). Set to NA by default.

`sliding_windows`
: TRUE/FALSE. TRUE indicates that the sliding windows computational approach will be used to compute the network statistic, while FALSE indicates the approach will not be used. Set to FALSE by default. It's important to note that the sliding windows framework should only be used when the pre-processed event sequence is 'big', such as the 360 million pre-processed event sequence used in Lerner and Lomi (2020), as it aims to reduce the computational burden of sorting 'big' datasets. In general, most pre-processed event sequences will not need to use the sliding windows approach. There is not a strict cutoff for 'big' dataset. This definition depends on both the size of the observed event sequence and the post-processing sampling dataset. For instance, according to our internal tests, when the event sequence is relatively large (i.e., 100,000 observed events) with probability of sampling from the observed event sequence set to 0.05 and using 10 controls per sampled event, the sliding windows framework for computing repetition is about 11% faster than the non-sliding windows framework. Yet, in a smaller dataset (i.e., 10,000 observed events) the sliding windows framework is about 25% slower than the non-sliding framework with the same conditions as before.

`processed_seqIDs`
: If sliding_windows is set to TRUE, the vector of event sequence IDs from the post-processing event sequence. The event sequence IDs represents the index for when the event occurred in the observed event sequence (e.g., the 5th event in the sequence will have a value of 5 in this vector).

`window_size`
: If sliding_windows is set to TRUE, the sizes of the windows that are used for the sliding windows computational framework. If NA, the function internally divides the dataset into ten slices (may not be optimal).

Details

The function calculates the persistence network sufficient statistic for a relational event sequence based on Butts (2008).

The formula for persistence for event e_i with reference to the sender's past relational history is:

$$Persistence_{e_i} = \frac{d(s(e_i), r(e_i), A_t)}{d(s(e_i), A_t)}$$

where $d(s(e_i), r(e_i), A_t)$ is the number of past events where the current event sender sent a tie to the current event receiver, and $d(s(e_i), A_t)$ is the number of past events where the current sender sent a tie.

The formula for persistence for event e_i with reference to the target's past relational history is:

$$Persistence_{e_i} = \frac{d(s(e_i), r(e_i), A_t)}{d(r(e_i), A_t)}$$

where $d(s(e_i), r(e_i), A_t)$ is the number of past events where the current event sender sent a tie to the current event receiver, and $d(r(e_i), A_t)$ is the number of past events where the current receiver recieved a tie.

Moreover, researchers interested in modeling temporal relevancy (see Quintane, Mood, Dunn, and Falzone 2022) can specify the relational time span, that is, length of time for which events are considered relationally relevant. This should be specified via the option *relationalTimeSpan* with *dependency* set to TRUE.

Value

The vector of persistence network statistics for the relational event sequence.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A relational event framework for social action." *Sociological Methodology* 38(1): 155-200.

Quintane, Eric, Martin Wood, John Dunn, and Lucia Falzon. 2022. "Temporal Brokering: A Measure of Brokerage as a Behavioral Process." *Organizational Research Methods* 25(3): 459-489.

Examples

```
# A Dummy One-Mode Event Dataset
events <- data.frame(time = 1:18,
                                eventID = 1:18,
                                sender = c("A", "B", "C",
                                           "A", "D", "E",
                                           "F", "B", "A",
                                           "F", "D", "B",
                                           "G", "B", "D",
                                           "H", "A", "D"),
                                target = c("B", "C", "D",
                                           "E", "A", "F",
                                           "D", "A", "C",
                                           "G", "B", "C",
                                           "H", "J", "A",
                                           "F", "C", "B"))
```

```
# Creating the Post-Processing Event Dataset with Null Events
eventSet <- processOMEventSeq(data = events,
                          time = events$time,
                          eventID = events$eventID,
                          sender = events$sender,
                          receiver = events$target,
                          p_samplingobserved = 1.00,
                          n_controls = 6,
                          seed = 9999)

#Compute Persistence with respect to the sender's past relational history without
#the sliding windows framework and no temporal dependency
eventSet$persist <- computePersistence(observed_time = events$time,
                                       observed_receiver = events$target,
                                       observed_sender = events$sender,
                                       processed_time = eventSet$time,
                                       processed_receiver = eventSet$receiver,
                                       processed_sender = eventSet$sender,
                                       sender = TRUE,
                                       nopastEvents = 0)

#Compute Persistence with respect to the sender's past relational history with
#the sliding windows framework and no temporal dependency
eventSet$persistSW <- computePersistence(observed_time = events$time,
                                       observed_receiver = events$target,
                                       observed_sender = events$sender,
                                       processed_time = eventSet$time,
                                       processed_receiver = eventSet$receiver,
                                       processed_sender = eventSet$sender,
                                       sender = TRUE,
                                       sliding_windows = TRUE,
                                       processed_seqIDs = eventSet$sequenceID,
                                       nopastEvents = 0)

#The results with and without the sliding windows are the same (see correlation
#below). Using the sliding windows method is recommended when the data are
#big' so that memory allotment is more efficient.
cor(eventSet$persist,eventSet$persistSW)


#Compute Persistence with respect to the sender's past relational history without
#the sliding windows framework and temporal dependency
eventSet$persistDep <- computePersistence(observed_time = events$time,
                                       observed_receiver = events$target,
                                       observed_sender = events$sender,
                                       processed_time = eventSet$time,
                                       processed_receiver = eventSet$receiver,
                                       processed_sender = eventSet$sender,
                                       sender = TRUE,
                                       dependency = TRUE,
                                       relationalTimeSpan = 5, #the past 5 events
                                       nopastEvents = 0)
```

```
#Compute Persistence with respect to the receiver's past relational history without
#the sliding windows framework and no temporal dependency
eventSet$persistT <- computePersistence(observed_time = events$time,
                                        observed_receiver = events$target,
                                        observed_sender = events$sender,
                                        processed_time = eventSet$time,
                                        processed_receiver = eventSet$receiver,
                                        processed_sender = eventSet$sender,
                                        sender = FALSE,
                                        nopastEvents = 0)

#Compute Persistence with respect to the receiver's past relational history with
#the sliding windows framework and no temporal dependency
eventSet$persistSWT <- computePersistence(observed_time = events$time,
                                        observed_receiver = events$target,
                                        observed_sender = events$sender,
                                        processed_time = eventSet$time,
                                        processed_receiver = eventSet$receiver,
                                        processed_sender = eventSet$sender,
                                        sender = FALSE,
                                        sliding_windows = TRUE,
                                        processed_seqIDs = eventSet$sequenceID,
                                        nopastEvents = 0)

#The results with and without the sliding windows are the same (see correlation
#below). Using the sliding windows method is recommended when the data are
#big' so that memory allotment is more efficient.
cor(eventSet$persistT,eventSet$persistSWT)


#Compute Persistence with respect to the receiver's past relational history without
#the sliding windows framework and temporal dependency
eventSet$persistDepT <- computePersistence(observed_time = events$time,
                                        observed_receiver = events$target,
                                        observed_sender = events$sender,
                                        processed_time = eventSet$time,
                                        processed_receiver = eventSet$receiver,
                                        processed_sender = eventSet$sender,
                                        sender = FALSE,
                                        dependency = TRUE,
                                        relationalTimeSpan = 5, #the past 5 events
                                        nopastEvents = 0)
```

---

computePrefAttach *Compute Butts' (2008) Preferential Attachment Network Statistic for Event Dyads in a Relational Event Sequence*

Description

The function computes the preferential attachment network sufficient statistic for a relational event sequence (see Butts 2008). Preferential attachment measures the tendency towards a positive feedback loop in which actors involved in more past events are more likely to be involved in future events (see Butts 2008 for an empirical example and discussion).This measure allows for preferential attachment scores to be only computed for the sampled events, while creating the statistics based on the full event sequence. Moreover, the function allows users to specify relational relevancy for the statistic and employ a sliding windows framework for large relational sequences.

Usage

```
computePrefAttach(
  observed_time,
  observed_sender,
  observed_receiver,
  processed_time,
  processed_sender,
  processed_receiver,
  dependency = FALSE,
  relationalTimeSpan = NULL,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  window_size = NA
)
```

Arguments

observed_time	The vector of event times from the pre-processing event sequence.
observed_sender	The vector of event senders from the pre-processing event sequence.
observed_receiver	The vector of event receivers from the pre-processing event sequence
processed_time	The vector of event times from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
processed_sender	The vector of event senders from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
processed_receiver	The vector of event receivers from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
dependency	TRUE/FALSE. TRUE indicates that temporal relevancy will be modeled (see the details section). FALSE indicates that temporal relevancy will not be modeled, that is, all past events are relevant (see the details section). Set to FALSE by default.
relationalTimeSpan	If dependency = TRUE, a numerical value that corresponds to the temporal span for relational relevancy, which must be the same measurement unit as the ob-

served_time and processed_time objects. When dependency = TRUE, the relevant events are events that have occurred between current event time, *t*, and *t - relationalTimeSpan*. For example, if the time measurement is the number of days since the first event and the value for relationalTimeSpan is set to 10, then only those events which occurred in the past 10 days are included in the computation of the statistic.

`sliding_windows`
: TRUE/FALSE. TRUE indicates that the sliding windows computational approach will be used to compute the network statistic, while FALSE indicates the approach will not be used. Set to FALSE by default. It's important to note that the sliding windows framework should only be used when the pre-processed event sequence is 'big', such as the 360 million pre-processed event sequence used in Lerner and Lomi (2020), as it aims to reduce the computational burden of sorting 'big' datasets. In general, most pre-processed event sequences will not need to use the sliding windows approach. There is not a strict cutoff for 'big' dataset. This definition depends on both the size of the observed event sequence and the post-processing sampling dataset. For instance, according to our internal tests, when the event sequence is relatively large (i.e., 100,000 observed events) with probability of sampling from the observed event sequence set to 0.05 and using 10 controls per sampled event, the sliding windows framework for computing repetition is about 11% faster than the non-sliding windows framework. Yet, in a smaller dataset (i.e., 10,000 observed events) the sliding windows framework is about 25% slower than the non-sliding framework with the same conditions as before.

`processed_seqIDs`
: If sliding_windows is set to TRUE, the vector of event sequence IDs from the post-processing event sequence. The event sequence IDs represents the index for when the event occurred in the observed event sequence (e.g., the 5th event in the sequence will have a value of 5 in this vector).

`window_size`
: If sliding_windows is set to TRUE, the sizes of the windows that are used for the sliding windows computational framework. If NA, the function internally divides the dataset into ten slices (may not be optimal).

Details

The function calculates preferential attachment for a relational event sequence based on Butts (2008).

Following Butts (2008), the formula for preferential attachment for event e_i is:

$$PA_{e_i} = \frac{d^+(r(e_i), A_t) + d^-(r(e_i), A_t)}{\sum_{i=1}^{|S|}(d^+(i, A_t) + d^-(i, A_t))}$$

where $d^+(r(e_i), A_t)$ is the past outdegree of the receiver for e_i, $d^-(r(e_i), A_t)$ is the past indegree of the receiver for e_i, $\sum_{i=1}^{|S|}(d^+(i, A_t) + d^-(i, A_t))$ is the sum of the past outdegree and indegree for all past event senders in the relational history.

Moreover, researchers interested in modeling temporal relevancy (see Quintane, Mood, Dunn, and Falzone 2022) can specify the relational time span, that is, length of time for which events are considered relationally relevant. This should be specified via the option *relationalTimeSpan* with *dependency* set to TRUE.

Value

The vector of event preferential attachment statistics for the relational event sequence.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A relational event framework for social action." *Sociological Methodology* 38(1): 155-200.

Quintane, Eric, Martin Wood, John Dunn, and Lucia Falzon. 2022. "Temporal Brokering: A Measure of Brokerage as a Behavioral Process." *Organizational Research Methods* 25(3): 459-489.

Examples

```
# A Dummy One-Mode Event Dataset
events <- data.frame(time = 1:18,
                                eventID = 1:18,
                                sender = c("A", "B", "C",
                                           "A", "D", "E",
                                           "F", "B", "A",
                                           "F", "D", "B",
                                           "G", "B", "D",
                                           "H", "A", "D"),
                                target = c("B", "C", "D",
                                           "E", "A", "F",
                                           "D", "A", "C",
                                           "G", "B", "C",
                                           "H", "J", "A",
                                           "F", "C", "B"))

# Creating the Post-Processing Event Dataset with Null Events
eventSet <- processOMEventSeq(data = events,
                          time = events$time,
                          eventID = events$eventID,
                          sender = events$sender,
                          receiver = events$target,
                          p_samplingobserved = 1.00,
                          n_controls = 6,
                          seed = 9999)

# Compute Preferential Attachment Statistic without Sliding Windows Framework and
# No Temporal Dependency
eventSet$pref <- computePrefAttach(observed_time = events$time,
                                   observed_receiver = events$target,
                                   observed_sender = events$sender,
                                   processed_time = eventSet$time,
                                   processed_receiver = eventSet$receiver,
                                   processed_sender = eventSet$sender,
```

```
                                    dependency = FALSE)

# Compute Preferential Attachment Statistic with Sliding Windows Framework and
# No Temporal Dependency
eventSet$prefSW <- computePrefAttach(observed_time = events$time,
                                     observed_receiver = events$target,
                                     observed_sender = events$sender,
                                     processed_time = eventSet$time,
                                     processed_receiver = eventSet$receiver,
                                     processed_sender = eventSet$sender,
                                     dependency = FALSE,
                                     sliding_windows = TRUE,
                                     processed_seqIDs = eventSet$sequenceID)

#The results with and without the sliding windows are the same (see correlation
#below). Using the sliding windows method is recommended when the data are
#big' so that memory allotment is more efficient.
cor(eventSet$pref,eventSet$prefSW) #the correlation of the values


# Compute Preferential Attachment Statistic without Sliding Windows Framework and
# Temporal Dependency
eventSet$prefdep <- computePrefAttach(observed_time = events$time,
                                      observed_receiver = events$target,
                                      observed_sender = events$sender,
                                      processed_time = eventSet$time,
                                      processed_receiver = eventSet$receiver,
                                      processed_sender = eventSet$sender,
                                      dependency = TRUE,
                                      relationalTimeSpan = 10)

# Compute Preferential Attachment Statistic with Sliding Windows Framework and
# Temporal Dependency
eventSet$pref1dep <- computePrefAttach(observed_time = events$time,
                                       observed_receiver = events$target,
                                       observed_sender = events$sender,
                                       processed_time = eventSet$time,
                                       processed_receiver = eventSet$receiver,
                                       processed_sender = eventSet$sender,
                                       dependency = TRUE,
                                       relationalTimeSpan = 10,
                                      sliding_windows = TRUE,
                                       processed_seqIDs = eventSet$sequenceID)

#The results with and without the sliding windows are the same (see correlation
#below). Using the sliding windows method is recommended when the data are
#big' so that memory allotment is more efficient.
cor(eventSet$prefdep,eventSet$pref1dep) #the correlation of the values
```

`computeReceiverIndegree` *Compute the Indegree Network Statistic for Event Receivers in a Relational Event Sequence*

Description

The function computes the indegree network sufficient statistic for event receivers in a relational event sequence (see Lerner and Lomi 2020; Butts 2008). This measure allows for the indegree scores to be computed only for the sampled events, while creating the weights based on the full event sequence (see Lerner and Lomi 2020; Vu et al. 2015). The function allows users to use two different weighting functions, reduce computational runtime, employ a sliding windows framework for large relational sequences, and specify a dyadic cutoff for relational relevancy.

Usage

```
computeReceiverIndegree(
  observed_time,
  observed_receiver,
  processed_time,
  processed_receiver,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  counts = FALSE,
  halflife = 2,
  dyadic_weight = 0,
  window_size = NA,
  Lerneretal_2013 = FALSE
)
```

Arguments

`observed_time` The vector of event times from the pre-processing event sequence.

`observed_receiver` The vector of event receivers from the pre-processing event sequence

`processed_time` The vector of event times from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).

`processed_receiver` The vector of event receivers from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).

`sliding_windows` TRUE/FALSE. TRUE indicates that the sliding windows computational approach will be used to compute the network statistic, while FALSE indicates the approach will not be used. Set to FALSE by default. It's important to note that the sliding windows framework should only be used when the pre-processed event sequence is 'big', such as the 360 million pre-processed event sequence used in Lerner and Lomi (2020), as it aims to reduce the computational burden of sorting 'big' datasets. In general, most pre-processed event sequences will not need

to use the sliding windows approach. There is not a strict cutoff for 'big' dataset. This definition depends on both the size of the observed event sequence and the post-processing sampling dataset. For instance, according to our internal tests, when the event sequence is relatively large (i.e., 100,000 observed events) with probability of sampling from the observed event sequence set to 0.05 and using 10 controls per sampled event, the sliding windows framework for computing repetition is about 11% faster than the non-sliding windows framework. Yet, in a smaller dataset (i.e., 10,000 observed events) the sliding windows framework is about 25% slower than the non-sliding framework with the same conditions as before.

`processed_seqIDs`
: If sliding_windows is set to TRUE, the vector of event sequence IDs from the post-processing event sequence. The event sequence IDs represents the index for when the event occurred in the observed event sequence (e.g., the 5th event in the sequence will have a value of 5 in this vector).

`counts`
: TRUE/FALSE. TRUE indicates that the counts of past events should be computed (see the details section). FALSE indicates that the temporal exponential weighting function should be used to downweigh past events (see the details section). Set to FALSE by default.

`halflife`
: A numerical value that is the halflife value to be used in the exponential weighting function (see the details section). Preset to 2 (should be updated by user).

`dyadic_weight`
: A numerical value that is the dyadic cutoff weight that represents the numerical cutoff value for temporal relevancy based on the exponential weighting function. For example, a numerical value of 0.01, indicates that an exponential weight less than 0.01 will become 0 and will not be included in the sum of the past event weights (see the details section). Set to 0 by default.

`window_size`
: If sliding_windows is set to TRUE, the sizes of the windows that are used for the sliding windows computational framework. If NA, the function internally divides the dataset into ten slices (may not be optimal).

`Lerneretal_2013`
: TRUE/FALSE. TRUE indicates that the Lerner et al. (2013) exponential weighting function will be used (see the details section). FALSE indicates that the Lerner and Lomi (2020) exponential weighting function will be used (see the details section). Set to FALSE by default

Details

The function calculates receiver indegree scores for relational event sequences based on the exponential weighting function used in either Lerner and Lomi (2020) or Lerner et al. (2013).

Following Lerner and Lomi (2020), the exponential weighting function in relational event models is:

$$w(s, r, t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}}$$

Following Lerner et al. (2013), the exponential weighting function in relational event models is:

$$w(s, r, t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}}$$

In both of the above equations, *s* is the current event sender, *r* is the current event receiver (target), *t* is the current event time, *t'* is the past event times that meet the weight subset, and $T_{1/2}$ is the halflife parameter.

The formula for receiver indegree for event e_i is:

$$recieverindegree_{e_i} = w(s', r, t)$$

That is, all past events in which the event receiver is the current receiver.

Moreover, researchers interested in modeling temporal relevancy (see Quintane, Mood, Dunn, and Falzone 2022; Lerner and Lomi 2020) can specify the dyadic weight cutoff, that is, the minimum value for which the weight is considered relationally relevant. Users who do not know the specific dyadic cutoff value to use, can use the `computeRemDyadCut` function.

Following Butts (2008), if the counts of the past events are requested, the formula for receiver indegree for event e_i is:

$$repetition_{e_i} = d(r' = r, t')$$

where, $d()$ is the number of past events where the past event receiver, *r'*, is the current event receiver (target). Moreover, the counting equation can be used in tandem with relational relevancy, by specifying the halflife parameter, exponential weighting function, and the dyadic cut off weight values. If the user is not interested in modeling relational relevancy, then those value should be left at their defaults.

Value

The vector of receiver indegree statistics for the relational event sequence.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A Relational Event Framework for Social Action." *Sociological Methodology* 38(1): 155-200.

Quintane, Eric, Martin Wood, John Dunn, and Lucia Falzon. 2022. "Temporal Brokering: A Measure of Brokerage as a Behavioral Process." *Organizational Research Methods* 25(3): 459-489.

Lerner, Jürgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: How to fit a relational event model to 360 million dyadic events." *Network Science* 8(1): 97-135.

Lerner, Jürgen, Margit Bussman, Tom A.B. Snijders, and Ulrik Brandes. 2013. " Modeling Frequency and Type of Interaction in Event Networks." *The Corvinus Journal of Sociology and Social Policy* 4(1): 3-32.

Vu, Duy, Philippa Pattison, and Garry Robins. 2015. "Relational event models for social learning in MOOCs." *Social Networks* 43: 121-135.

Examples

```
events <- data.frame(time = 1:18,
                                eventID = 1:18,
                                sender = c("A", "B", "C",
                                           "A", "D", "E",
                                           "F", "B", "A",
                                           "F", "D", "B",
                                           "G", "B", "D",
                                          "H", "A", "D"),
                               target = c("B", "C", "D",
                                          "E", "A", "F",
                                          "D", "A", "C",
                                          "G", "B", "C",
                                          "H", "J", "A",
                                          "F", "C", "B"))

eventSet <- processOMEventSeq(data = events,
                      time = events$time,
                      eventID = events$eventID,
                      sender = events$sender,
                      receiver = events$target,
                      p_samplingobserved = 1.00,
                      n_controls = 1,
                      seed = 9999)

# Computing Target Indegree Statistics without the sliding windows framework
eventSet$target_indegree <- computeReceiverIndegree(
   observed_time = events$time,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   dyadic_weight = 0,
   Lerneretal_2013 = FALSE)

# Computing Target Indegree Statistics with the sliding windows framework
eventSet$target_indegreeSW <- computeReceiverIndegree(
   observed_time = events$time,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   processed_seqIDs = eventSet$sequenceID,
   dyadic_weight = 0,
   sliding_window = TRUE,
   Lerneretal_2013 = FALSE)

#The results with and without the sliding windows are the same (see correlation
#below). Using the sliding windows method is recommended when the data are
#big' so that memory allotment is more efficient.
cor(eventSet$target_indegree , eventSet$target_indegreeSW )
```

```
# Computing Target Indegree Statistics with the counts of events being returned
eventSet$target_indegreeC <- computeReceiverIndegree(
   observed_time = events$time,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   processed_seqIDs = eventSet$sequenceID,
   dyadic_weight = 0,
   sliding_window = TRUE,
   Lerneretal_2013 = FALSE,
   counts = TRUE)

cbind(eventSet$target_indegree,
     eventSet$target_indegreeSW,
     eventSet$target_indegreeC)
```

`computeReceiverOutdegree` *Compute the Outdegree Network Statistic for Event Receivers in a Relational Event Sequence*

Description

The function computes the receiver outdegree network sufficient statistic for a relational event sequence (see Lerner and Lomi 2020; Butts 2008). This measure allows for outdegree scores to be only computed for the sampled events, while creating the weights based on the full event sequence (see Lerner and Lomi 2020; Vu et al. 2015). The function allows users to use two different weighting functions, reduce computational runtime, employ a sliding windows framework for large relational sequences, and specify a dyadic cutoff for relational relevancy.

Usage

```
computeReceiverOutdegree(
  observed_time,
  observed_sender,
  observed_receiver,
  processed_time,
  processed_sender,
  processed_receiver,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  counts = FALSE,
  halflife = 2,
  dyadic_weight = 0,
  window_size = NA,
  Lerneretal_2013 = FALSE
)
```

Arguments

observed_time	The vector of event times from the pre-processing event sequence.
observed_sender	The vector of event senders from the pre-processing event sequence.
observed_receiver	The vector of event receivers from the pre-processing event sequence
processed_time	The vector of event times from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
processed_sender	The vector of event senders from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
processed_receiver	The vector of event receivers from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
sliding_windows	TRUE/FALSE. TRUE indicates that the sliding windows computational approach will be used to compute the network statistic, while FALSE indicates the approach will not be used. Set to FALSE by default. It's important to note that the sliding windows framework should only be used when the pre-processed event sequence is 'big', such as the 360 million pre-processed event sequence used in Lerner and Lomi (2020), as it aims to reduce the computational burden of sorting 'big' datasets. In general, most pre-processed event sequences will not need to use the sliding windows approach. There is not a strict cutoff for 'big' dataset. This definition depends on both the size of the observed event sequence and the post-processing sampling dataset. For instance, according to our internal tests, when the event sequence is relatively large (i.e., 100,000 observed events) with probability of sampling from the observed event sequence set to 0.05 and using 10 controls per sampled event, the sliding windows framework for computing repetition is about 11% faster than the non-sliding windows framework. Yet, in a smaller dataset (i.e., 10,000 observed events) the sliding windows framework is about 25% slower than the non-sliding framework with the same conditions as before.
processed_seqIDs	If sliding_windows is set to TRUE, the vector of event sequence IDs from the post-processing event sequence. The event sequence IDs represents the index for when the event occurred in the observed event sequence (e.g., the 5th event in the sequence will have a value of 5 in this vector).
counts	TRUE/FALSE. TRUE indicates that the counts of past events should be computed (see the details section). FALSE indicates that the temporal exponential weighting function should be used to downweigh past events (see the details section). Set to FALSE by default.
halflife	A numerical value that is the halflife value to be used in the exponential weighting function (see the details section). Preset to 2 (should be updated by user).
dyadic_weight	A numerical value that is the dyadic cutoff weight that represents the numerical cutoff value for temporal relevancy based on the exponential weighting function. For example, a numerical value of 0.01, indicates that an exponential weight less

than 0.01 will become 0 and will not be included in the sum of the past event weights (see the details section). Set to 0 by default.

`window_size` If sliding_windows is set to TRUE, the sizes of the windows that are used for the sliding windows computational framework. If NA, the function internally divides the dataset into ten slices (may not be optimal).

`Lerneretal_2013`

TRUE/FALSE. TRUE indicates that the Lerner et al. (2013) exponential weighting function will be used (see the details section). FALSE indicates that the Lerner and Lomi (2020) exponential weighting function will be used (see the details section). Set to FALSE by default

Details

The function calculates reciever outdegree scores for relational event sequences based on the exponential weighting function used in either Lerner and Lomi (2020) or Lerner et al. (2013).

Following Lerner and Lomi (2020), the exponential weighting function in relational event models is:

$$w(s, r, t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}}$$

Following Lerner et al. (2013), the exponential weighting function in relational event models is:

$$w(s, r, t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}}$$

In both of the above equations, *s* is the current event sender, *r* is the current event receiver (target), *t* is the current event time, *t'* is the past event times that meet the weight subset, and $T_{1/2}$ is the halflife parameter.

The formula for receiver outdegree for event e_i is:

$$receiveroutdegree_{e_i} = w(r', r, t)$$

That is, all past events in which the past receiver is the current sender and the event receiver can be any past actor.

Moreover, researchers interested in modeling temporal relevancy (see Quintane, Mood, Dunn, and Falzone 2022; Lerner and Lomi 2020) can specify the dyadic weight cutoff, that is, the minimum value for which the weight is considered relationally relevant. Users who do not know the specific dyadic cutoff value to use, can use the `computeRemDyadCut` function.

Following Butts (2008), if the counts of the past events are requested, the formula for receiver outdegree for event e_i is:

$$receiveroutdegreee_i = d(s' = r, t')$$

Where, $d()$ is the number of past events where the event sender, *s'*, is the current event receiver, *r'*. Moreover, the counting equation can be used in tandem with relational relevancy, by specifying the halflife parameter, exponential weighting function, and the dyadic cut off weight values. If the user is not interested in modeling relational relevancy, then those value should be left at their baseline values.

Value

The vector of receiver outdegree statistics for the relational event sequence.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A Relational Event Framework for Social Action." *Sociological Methodology* 38(1): 155-200.

Quintane, Eric, Martin Wood, John Dunn, and Lucia Falzon. 2022. "Temporal Brokering: A Measure of Brokerage as a Behavioral Process." *Organizational Research Methods* 25(3): 459-489.

Lerner, Jürgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: How to fit a relational event model to 360 million dyadic events." *Network Science* 8(1): 97-135.

Lerner, Jürgen, Margit Bussman, Tom A.B. Snijders, and Ulrik Brandes. 2013. " Modeling Frequency and Type of Interaction in Event Networks." *The Corvinus Journal of Sociology and Social Policy* 4(1): 3-32.

Vu, Duy, Philippa Pattison, and Garry Robins. 2015. "Relational event models for social learning in MOOCs." *Social Networks* 43: 121-135.

Examples

```
events <- data.frame(time = 1:18,
                                 eventID = 1:18,
                                 sender = c("A", "B", "C",
                                            "A", "D", "E",
                                            "F", "B", "A",
                                            "F", "D", "B",
                                            "G", "B", "D",
                                           "H", "A", "D"),
                                target = c("B", "C", "D",
                                            "E", "A", "F",
                                            "D", "A", "C",
                                            "G", "B", "C",
                                            "H", "J", "A",
                                            "F", "C", "B"))

eventSet <- processOMEventSeq(data = events,
                      time = events$time,
                      eventID = events$eventID,
                      sender = events$sender,
                      receiver = events$target,
                      p_samplingobserved = 1.00,
                      n_controls = 1,
                      seed = 9999)

# Computing Target Outdegree Statistics without the sliding windows framework
```

```
eventSet$target_outdegree <- computeReceiverOutdegree(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   dyadic_weight = 0,
   Lerneretal_2013 = FALSE)

# Computing Target Outdegree Statistics with the sliding windows framework
eventSet$target_outdegreeSW <- computeReceiverOutdegree(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   processed_seqIDs = eventSet$sequenceID,
   dyadic_weight = 0,
   sliding_window = TRUE,
   Lerneretal_2013 = FALSE)

#The results with and without the sliding windows are the same (see correlation
#below). Using the sliding windows method is recommended when the data are
#big' so that memory allotment is more efficient.
cor(eventSet$target_outdegreeSW , eventSet$target_outdegree)

# Computing  Target Outdegree Statistic with the counts of events being returned
eventSet$target_outdegreeC <- computeReceiverOutdegree(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   dyadic_weight = 0,
   sliding_window = FALSE,
   counts = TRUE,
   Lerneretal_2013 = FALSE)

cbind(eventSet$target_outdegree,
     eventSet$target_outdegreeSW,
     eventSet$target_outdegreeC)
```

`computeRecency` *Compute Butts' (2008) Recency Network Statistic for Event Dyads in a Relational Event Sequence*

Description

This function computes the recency network sufficient statistic for a relational event sequence (see Butts 2008; Vu et al. 2015; Meijerink-Bosman et al. 2022). The recency statistic captures the tendency in which more recent events (i.e., an exchange between two medical doctors) are more likely to reoccur in comparison to events that happened in the distant past (see Butts 2008 for a discussion). This measure allows for recency scores to be only computed for the sampled events, while creating the statistics based on the full event sequence. Moreover, the function allows users to specify relational relevancy for the statistic and employ a sliding windows framework for large relational sequences.

Usage

```
computeRecency(
  observed_time,
  observed_sender,
  observed_receiver,
  processed_time,
  processed_sender,
  processed_receiver,
  type = c("raw.diff", "inv.diff.plus1", "rank.ordered.count"),
  i_neighborhood = TRUE,
  dependency = FALSE,
  relationalTimeSpan = NULL,
  nopastEvents = NA,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  window_size = NA
)
```

Arguments

observed_time	The vector of event times from the pre-processing event sequence.
observed_sender	The vector of event senders from the pre-processing event sequence.
observed_receiver	The vector of event receivers from the pre-processing event sequence
processed_time	The vector of event times from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
processed_sender	The vector of event senders from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
processed_receiver	The vector of event receivers from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
type	A string value that specifies which recency formula will be used to compute the statistics. The options are "raw.diff", "inv.diff.plus1", "rank.ordered.count" (see the details section).

`i_neighborhood` TRUE/FALSE. TRUE indicates that the recency statistic will be computed in reference to the sender's past relational history (see details section). FALSE indicates that the persistence statistic will be computed in reference to the target's past relational history (see details section). Set to TRUE by default.

`dependency` TRUE/FALSE. TRUE indicates that temporal relevancy will be modeled (see the details section). FALSE indicates that temporal relevancy will not be modeled, that is, all past events are relevant (see the details section). Set to FALSE by default.

`relationalTimeSpan`
If dependency = TRUE, a numerical value that corresponds to the temporal span for relational relevancy, which must be the same measurement unit as the observed_time and processed_time objects. When dependency = TRUE, the relevant events are events that have occurred between current event time, *t*, and *t - relationalTimeSpan*. For example, if the time measurement is the number of days since the first event and the value for relationalTimeSpan is set to 10, then only those events which occurred in the past 10 days are included in the computation of the statistic.

`nopastEvents` The numerical value that specifies what value should be given to events in which the sender has sent not past ties (i's neighborhood when i_neighborhood = TRUE) or has not received any past ties (j's neighborhood when i_neighborhood = FALSE). Set to NA by default.

`sliding_windows`
TRUE/FALSE. TRUE indicates that the sliding windows computational approach will be used to compute the network statistic, while FALSE indicates the approach will not be used. Set to FALSE by default. It's important to note that the sliding windows framework should only be used when the pre-processed event sequence is 'big', such as the 360 million pre-processed event sequence used in Lerner and Lomi (2020), as it aims to reduce the computational burden of sorting 'big' datasets. In general, most pre-processed event sequences will not need to use the sliding windows approach. There is not a strict cutoff for 'big' dataset. This definition depends on both the size of the observed event sequence and the post-processing sampling dataset. For instance, according to our internal tests, when the event sequence is relatively large (i.e., 100,000 observed events) with probability of sampling from the observed event sequence set to 0.05 and using 10 controls per sampled event, the sliding windows framework for computing repetition is about 11% faster than the non-sliding windows framework. Yet, in a smaller dataset (i.e., 10,000 observed events) the sliding windows framework is about 25% slower than the non-sliding framework with the same conditions as before.

`processed_seqIDs`
If sliding_windows is set to TRUE, the vector of event sequence IDs from the post-processing event sequence. The event sequence IDs represents the index for when the event occurred in the observed event sequence (e.g., the 5th event in the sequence will have a value of 5 in this vector).

`window_size` If sliding_windows is set to TRUE, the sizes of the windows that are used for the sliding windows computational framework. If NA, the function internally divides the dataset into ten slices (may not be optimal).

Details

This function calculates the recency network sufficient statistic for a relational event based on Butts (2008), Vu et al. (2015), or Meijerink-Bosman et al. (2022). Depending on the type and neighborhood requested, different formulas will be used.

In the below equations, when *i_neighborhood* is TRUE:

$$t^* = max(t \in \{(s', r', t') \in E : s' = s \wedge r' = r \wedge t' < t\})$$

When *i_neighborhood* is FALSE, the following formula is used:

$$t^* = max(t \in \{(s', r', t') \in E : s' = r \wedge r' = s \wedge t' < t\})$$

The formula for recency for event e_i with type set to "raw.diff" and *i_neighborhood* is TRUE (Vu et al. 2015):

$$recency_{e_i} = t_{e_i} - t^*$$

where t^*, is the most recent time in which the past event has the same receiver and sender as the current event. If there are no past events within the current dyad, then the value defaults to the *nopastEvents* argument.

The formula for recency for event e_i with type set to "raw.diff" and *i_neighborhood* is FALSE (Vu et al. 2015):

$$recency_{e_i} = t_{e_i} - t^*$$

where t^*, is the most recent time in which the past event's sender is the current event receiver and the past event receiver is the current event sender. If there are no past events within the current dyad, then the value defaults to the *nopastEvents* argument.

The formula for recency for event e_i with type set to "inv.diff.plus1" and *i_neighborhood* is TRUE (Meijerink-Bosman et al. 2022):

$$recency_{e_i} = \frac{1}{t_{e_i} - t^* + 1}$$

where t^*, is the most recent time in which the past event has the same receiver and sender as the current event. If there are no past events within the current dyad, then the value defaults to the *nopastEvents* argument.

The formula for recency for event e_i with type set to "inv.diff.plus1" and *i_neighborhood* is FALSE (Meijerink-Bosman et al. 2022):

$$recency_{e_i} = \frac{1}{t_{e_i} - t^* + 1}$$

where t^*, is the most recent time in which the past event's sender is the current event receiver and the past event receiver is the current event sender. If there are no past events within the current dyad, then the value defaults to the *nopastEvents* argument.

The formula for recency for event e_i with type set to "rank.ordered.count" and *i_neighborhood* is TRUE (Butts 2008):

$$recency_{e_i} = \rho(s(e_i), r(e_i), A_t)^{-1}$$

where $\rho(s(e_i), r(e_i), A_t)$, is the current event receiver's rank amongst the current sender's recent relational events. That is, as Butts (2008: 174) argues, "$\rho(s(e_i), r(e_i), A_t)$ is j's recency rank among

i's in-neighborhood. Thus, if j is the last person to have called i, then $\rho(s(e_i), r(e_i), A_t)^{-1} = 1$. This falls to 1/2 if j is the second most recent person to call i, 1/3 if j is the third most recent person, etc." Moreover, if j is not in i's neighborhood, the value defaults to infinity. If there are no past events with the current sender, then the value defaults to the *nopastEvents* argument.

The formula for recency for event e_i with type set to "rank.ordered.count" and *i_neighborhood* is FALSE (Butts 2008):

$$recency_{e_i} = \rho(r(e_i), s(e_i), A_t)^{-1}$$

where $\rho(r(e_i), s(e_i), A_t)$, is the current event sender's rank amongst the current receiver's recent relational events. That is, this measure is the same as above where the dyadic pair is flipped for the past relational events. Moreover, if j is not in i's neighborhood, the value defaults to infinity. If there are no past events with the current sender, then the value defaults to the *nopastEvents* argument.

Finally, researchers interested in modeling temporal relevancy (see Quintane, Mood, Dunn, and Falzone 2022) can specify the relational time span, that is, length of time for which events are considered relationally relevant. This should be specified via the option *relationalTimeSpan* with *dependency* set to TRUE.

Value

The vector of recency network statistics for the relational event sequence.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A relational event framework for social action." *Sociological Methodology* 38(1): 155-200.

Meijerink-Bosman, Marlyne, Roger Leenders, and Joris Mulder. 2022. "Dynamic relational event modeling: Testing, exploring, and applying." *PLOS One* 17(8): e0272309.

Quintane, Eric, Martin Wood, John Dunn, and Lucia Falzon. 2022. "Temporal Brokering: A Measure of Brokerage as a Behavioral Process." *Organizational Research Methods* 25(3): 459-489.

Vu, Duy, Philippa Pattison, and Garry Robbins. 2015. "Relational event models for social learning in MOOCs." *Social Networks* 43: 121-135.

Examples

```
# A Dummy One-Mode Event Dataset
events <- data.frame(time = 1:18,
                                eventID = 1:18,
                                sender = c("A", "B", "C",
                                           "A", "D", "E",
                                           "F", "B", "A",
                                           "F", "D", "B",
                                           "G", "B", "D",
                                           "H", "A", "D"),
                                target = c("B", "C", "D",
```

```
                                          "E", "A", "F",
                                          "D", "A", "C",
                                          "G", "B", "C",
                                          "H", "J", "A",
                                          "F", "C", "B"))

# Creating the Post-Processing Event Dataset with Null Events
eventSet <- processOMEventSeq(data = events,
                           time = events$time,
                           eventID = events$eventID,
                           sender = events$sender,
                           receiver = events$target,
                           p_samplingobserved = 1.00,
                          n_controls = 6,
                          seed = 9999)

# Compute Recency Statistic without Sliding Windows Framework and
# No Temporal Dependency
eventSet$recency_rawdiff <- computeRecency(
 observed_time = events$time,
 observed_receiver = events$target,
 observed_sender = events$sender,
 processed_time = eventSet$time,
 processed_receiver = eventSet$receiver,
 processed_sender = eventSet$sender,
 type = "raw.diff",
 dependency = FALSE,
 i_neighborhood = TRUE,
 nopastEvents = 0)

# Compute Recency Statistic without Sliding Windows Framework and
# No Temporal Dependency
eventSet$recency_inv <- computeRecency(
 observed_time = events$time,
 observed_receiver = events$target,
 observed_sender = events$sender,
 processed_time = eventSet$time,
 processed_receiver = eventSet$receiver,
 processed_sender = eventSet$sender,
 type = "inv.diff.plus1",
 dependency = FALSE,
 i_neighborhood = TRUE,
 nopastEvents = 0)


# Compute Recency Statistic without Sliding Windows Framework and
# No Temporal Dependency
eventSet$recency_rank <- computeRecency(
 observed_time = events$time,
 observed_receiver = events$target,
 observed_sender = events$sender,
 processed_time = eventSet$time,
 processed_receiver = eventSet$receiver,
```

```
 processed_sender = eventSet$sender,
 type = "rank.ordered.count",
 dependency = FALSE,
 i_neighborhood = TRUE,
 nopastEvents = 0)

# Compute Recency Statistic with Sliding Windows Framework and No Temporal Dependency
eventSet$recency_rawdiffSW <- computeRecency(
 observed_time = events$time,
 observed_receiver = events$target,
 observed_sender = events$sender,
 processed_time = eventSet$time,
 processed_receiver = eventSet$receiver,
 processed_sender = eventSet$sender,
 type = "raw.diff",
 dependency = FALSE,
 i_neighborhood = TRUE,
 sliding_windows = TRUE,
 processed_seqIDs = eventSet$sequenceID,
 nopastEvents = 0)


# Compute Recency Statistic with Sliding Windows Framework and No Temporal Dependency
eventSet$recency_invSW <- computeRecency(
 observed_time = events$time,
 observed_receiver = events$target,
 observed_sender = events$sender,
 processed_time = eventSet$time,
 processed_receiver = eventSet$receiver,
 processed_sender = eventSet$sender,
 type = "inv.diff.plus1",
 dependency = FALSE,
 i_neighborhood = TRUE,
 sliding_windows = TRUE,
 processed_seqIDs = eventSet$sequenceID,
 nopastEvents = 0)


# Compute Recency Statistic with Sliding Windows Framework and No Temporal Dependency
eventSet$recency_rankSW <- computeRecency(
 observed_time = events$time,
 observed_receiver = events$target,
 observed_sender = events$sender,
 processed_time = eventSet$time,
 processed_receiver = eventSet$receiver,
 processed_sender = eventSet$sender,
 type = "rank.ordered.count",
 dependency = FALSE,
 i_neighborhood = TRUE,
 sliding_windows = TRUE,
 processed_seqIDs = eventSet$sequenceID,
 nopastEvents = 0)
```

`computeReciprocity` *Compute the Reciprocity Network Statistic for Event Dyads in a Relational Event Sequence*

Description

This function calculates the reciprocity network sufficient statistic for a relational event sequence (see Lerner and Lomi 2020; Butts 2008). The reciprocity statistic captures the tendency in which a sender *a* sends a tie to receiver *b* given that *b* sent a tie to *a* in the past (i.e., an exchange between two medical doctors). This measure allows for reciprocity scores to be only computed for the sampled events, while creating the weights based on the full event sequence (see Lerner and Lomi 2020; Vu et al. 2015). The function allows users to use two different weighting functions, reduce computational runtime, employ a sliding windows framework for large relational sequences, and specify a dyadic cutoff for relational relevancy.

Usage

```
computeReciprocity(
  observed_time,
  observed_sender,
  observed_receiver,
  processed_time,
  processed_sender,
  processed_receiver,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  counts = FALSE,
  halflife = 2,
  dyadic_weight = 0,
  window_size = NA,
  Lerneretal_2013 = FALSE
)
```

Arguments

`observed_time` The vector of event times from the pre-processing event sequence.

`observed_sender` The vector of event senders from the pre-processing event sequence.

`observed_receiver` The vector of event receivers from the pre-processing event sequence

`processed_time` The vector of event times from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).

`processed_sender` The vector of event senders from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).

`processed_receiver`
: The vector of event receivers from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).

`sliding_windows`
: TRUE/FALSE. TRUE indicates that the sliding windows computational approach will be used to compute the network statistic, while FALSE indicates the approach will not be used. Set to FALSE by default. It's important to note that the sliding windows framework should only be used when the pre-processed event sequence is 'big', such as the 360 million pre-processed event sequence used in Lerner and Lomi (2020), as it aims to reduce the computational burden of sorting 'big' datasets. In general, most pre-processed event sequences will not need to use the sliding windows approach. There is not a strict cutoff for 'big' dataset. This definition depends on both the size of the observed event sequence and the post-processing sampling dataset. For instance, according to our internal tests, when the event sequence is relatively large (i.e., 100,000 observed events) with probability of sampling from the observed event sequence set to 0.05 and using 10 controls per sampled event, the sliding windows framework for computing repetition is about 11% faster than the non-sliding windows framework. Yet, in a smaller dataset (i.e., 10,000 observed events) the sliding windows framework is about 25% slower than the non-sliding framework with the same conditions as before.

`processed_seqIDs`
: If sliding_windows is set to TRUE, the vector of event sequence IDs from the post-processing event sequence. The event sequence IDs represents the index for when the event occurred in the observed event sequence (e.g., the 5th event in the sequence will have a value of 5 in this vector).

`counts`
: TRUE/FALSE. TRUE indicates that the counts of past events should be computed (see the details section). FALSE indicates that the temporal exponential weighting function should be used to downweigh past events (see the details section). Set to FALSE by default.

`halflife`
: A numerical value that is the halflife value to be used in the exponential weighting function (see the details section). Preset to 2 (should be updated by user).

`dyadic_weight`
: A numerical value that is the dyadic cutoff weight that represents the numerical cutoff value for temporal relevancy based on the exponential weighting function. For example, a numerical value of 0.01, indicates that an exponential weight less than 0.01 will become 0 and will not be included in the sum of the past event weights (see the details section). Set to 0 by default.

`window_size`
: If sliding_windows is set to TRUE, the sizes of the windows that are used for the sliding windows computational framework. If NA, the function internally divides the dataset into ten slices (may not be optimal).

`Lerneretal_2013`
: TRUE/FALSE. TRUE indicates that the Lerner et al. (2013) exponential weighting function will be used (see the details section). FALSE indicates that the Lerner and Lomi (2020) exponential weighting function will be used (see the details section). Set to FALSE by default

Details

This function calculates reciprocity scores for relational event models based on the exponential weighting function used in either Lerner and Lomi (2020) or Lerner et al. (2013).

Following Lerner and Lomi (2020), the exponential weighting function in relational event models is:

$$w(s, r, t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}}$$

Following Lerner et al. (2013), the exponential weighting function in relational event models is:

$$w(s, r, t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}}$$

In both of the above equations, *s* is the current event sender, *r* is the current event receiver (target), *t* is the current event time, *t'* is the past event times that meet the weight subset, and $T_{1/2}$ is the halflife parameter.

The formula for reciprocity for event e_i is:

$$reciprocity_{e_i} = w(r, s, t)$$

That is, all past events in which the past sender is the current receiver and the past receiver is the current sender.

Moreover, researchers interested in modeling temporal relevancy (see Quintane, Mood, Dunn, and Falzone 2022; Lerner and Lomi 2020) can specify the dyadic weight cutoff, that is, the minimum value for which the weight is considered relationally relevant. Users who do not know the specific dyadic cutoff value to use, can use the `computeRemDyadCut` function.

Following Butts (2008), if the counts of the past events are requested, the formula for reciprocity for event e_i is:

$$reciprocity_{e_i} = d(r = s', s = r', t')$$

Where, $d()$ is the number of past events where the event sender, *s'*, is the current event receiver, *r*, and the event receiver (target), *r'*, is the current event sender, *s*. Moreover, the counting equation can be used in tandem with relational relevancy, by specifying the halflife parameter, exponential weighting function, and the dyadic cut off weight values. If the user is not interested in modeling relational relevancy, then those value should be left at their baseline values.

Value

The vector of reciprocity statistics for the relational event sequence.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A Relational Event Framework for Social Action." *Sociological Methodology* 38(1): 155-200.

Quintane, Eric, Martin Wood, John Dunn, and Lucia Falzon. 2022. "Temporal Brokering: A Measure of Brokerage as a Behavioral Process." *Organizational Research Methods* 25(3): 459-489.

Lerner, Jürgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: How to fit a relational event model to 360 million dyadic events." *Network Science* 8(1): 97-135.

Lerner, Jürgen, Margit Bussman, Tom A.B. Snijders, and Ulrik Brandes. 2013. " Modeling Frequency and Type of Interaction in Event Networks." *The Corvinus Journal of Sociology and Social Policy* 4(1): 3-32.

Vu, Duy, Philippa Pattison, and Garry Robins. 2015. "Relational event models for social learning in MOOCs." *Social Networks* 43: 121-135.

Examples

```
events <- data.frame(time = 1:18, eventID = 1:18,
                                 sender = c("A", "B", "C",
                                            "A", "D", "E",
                                            "F", "B", "A",
                                            "F", "D", "B",
                                            "G", "B", "D",
                                           "H", "A", "D"),
                                target = c("B", "C", "D",
                                           "E", "A", "F",
                                           "D", "A", "C",
                                           "G", "B", "C",
                                           "H", "J", "A",
                                           "F", "C", "B"))

eventSet <- processOMEventSeq(data = events,
                      time = events$time,
                      eventID = events$eventID,
                      sender = events$sender,
                      receiver = events$target,
                      p_samplingobserved = 1.00,
                      n_controls = 1,
                      seed = 9999)

# Computing Reciprocity Statistics without the sliding windows framework
eventSet$recip <- computeReciprocity(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   dyadic_weight = 0,
   Lerneretal_2013 = FALSE)
```

```
# Computing Reciprocity Statistics with the sliding windows framework
eventSet$recipSW <- computeReciprocity(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   processed_seqIDs = eventSet$sequenceID,
   dyadic_weight = 0,
   sliding_window = TRUE,
   Lerneretal_2013 = FALSE)

#The results with and without the sliding windows are the same (see correlation
#below). Using the sliding windows method is recommended when the data are
#big' so that memory allotment is more efficient.
cor(eventSet$recipSW , eventSet$recip)

# Computing Reciprocity Statistics with the counts of events being returned
eventSet$recipC <- computeReciprocity(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   dyadic_weight = 0,
   sliding_window = FALSE,
   counts = TRUE,
   Lerneretal_2013 = FALSE)

cbind(eventSet$recip,
     eventSet$recipSW,
     eventSet$recipC)
```

`computeRemDyadCut` *A Helper Function to Assist Researchers in Finding Dyadic Weight Cutoff Values*

Description

A user-helper function to assist researchers in finding the dyadic cutoff value to compute sufficient statistics for relational event models based upon temporal dependency.

Usage

```
computeRemDyadCut(halflife, relationalWidth, Lerneretal_2013 = FALSE)
```

Arguments

`halflife`	The user specified halflife value for the weighting function.
`relationalWidth`	The numerical value that corresponds to the time range for which the user specifies for temporal relevancy.
`Lerneretal_2013`	TRUE/FALSE. TRUE indicates that the Lerner et al. (2013) exponential weighting function will be used (see the details section). FALSE indicates that the Lerner and Lomi (2020) exponential weighting function will be used (see the details section). Set to FALSE by default

Details

This function is specifically designed as a user-helper function to assist researchers in finding the dyadic cutoff value for creating sufficient statistics based upon temporal dependency. In other words, this function estimates a dyadic cutoff value for relational relevance, that is, the minimum dyadic weight for past events to be potentially relevant (i.e., to possibly have an impact) on the current event. All non-relevant events (i.e., events too distant in the past from the current event to be considered relevant, that is, those below the cutoff value) will have a weight of 0. This cutoff value is based upon two user-specified values: the events' halflife (i..e, `halflife`) and the relationally relevant event or time span (i.e., `relationalWidth`). Ideally, both the values for `halflife` and `relationalWidth` would be based on the researcher's command of the relevant substantive literature. Importantly, `halflife` and `relationalWidth` must be in the same units of measurement (e.g., days). If not, the function will not return the correct answer.

For example, let's say that the user defines the `halflife` to be 15 days (i.e., two weeks) and the relationally relevant event or time span (i.e., `relationalWidth`) to be 30 days (i.e., events that occurred more than 1 month in the past are not considered relationally relevant for the current event). The user would then specify `halflife` = 15 and `relationalWidth` = 30.

Following Lerner and Lomi (2020), the exponential weighting function in relational event models is:

$$w(s, r, t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}}$$

Following Lerner et al. (2013), the exponential weighting function in relational event models is:

$$w(s, r, t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}}$$

In both of the above equations, *s* is the current event sender, *r* is the current event receiver (target), *t* is the current event time, *t'* is the past event times that meet the weight subset, and $T_{1/2}$ is the halflife parameter. The task of this function is to find the weight, $w(s, r, t)$, that corresponds to the time difference provided by the user.

Value

The dyadic weight cutoff based on user specified values.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Lerner, Jürgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: How to fit a relational event model to 360 million dyadic events." *Network Science* 8(1): 97-135.

Lerner, Jürgen, Margit Bussman, Tom A.B. Snijders, and Ulrik Brandes. 2013. " Modeling Frequency and Type of Interaction in Event Networks." *The Corvinus Journal of Sociology and Social Policy* 4(1): 3-32.

Examples

```
#To replicate the example in the details section:
# with the Lerner et al. 2013 weighting function
computeRemDyadCut(halflife = 15,
                  relationalWidth = 30,
                  Lerneretal_2013 = TRUE)

# without the Lerner et al. 2013 weighting function
computeRemDyadCut(halflife = 15,
                  relationalWidth = 30,
                  Lerneretal_2013 = FALSE)

# A result to test the function (should come out to 0.50)
computeRemDyadCut(halflife = 30,
                  relationalWidth = 30,
                  Lerneretal_2013 = FALSE)


# Replicating Lerner and Lomi (2020):
#"We set T1/2 to 30 days so that an event counts as (close to) one in the very next instant of time,
#it counts as 1/2 one month later, it counts as 1/4 two months after the event, and so on. To reduce
#the memory consumption needed to store the network of past events, we set a dyadic weight to
#zero if its value drops below 0.01. If a single event occurred in some dyad this would happen after
#6.64×T1/2, that is after more than half a year." (Lerner and Lomi 2020: 104).

# Based upon Lerner and Lomi (2020: 104), the result should be around 0.01. Since the
# time values in Lerner and Lomi (2020) are in milliseconds, we have to change
# all measurements into milliseconds
computeRemDyadCut(halflife = (30*24*60*60*1000), #30 days in milliseconds
                relationalWidth = (6.64*30*24*60*60*1000), #Based upon the paper
                #using the Lerner and Lomi (2020) weighting function
                Lerneretal_2013 = FALSE)
```

`computeRepetition` *Compute Butts' (2008) Repetition Network Statistic for Event Dyads in a Relational Event Sequence*

Description

This function computes the repetition network sufficient statistic for a relational event sequence (see Lerner and Lomi 2020; Butts 2008). Repetition measures the increased tendency for events between S and R to occur given that S and R have interacted in the past. Furthermore, this measure allows for repetition scores to be only computed for the sampled events, while creating the weights based on the full event sequence (see Lerner and Lomi 2020; Vu et al. 2015). The function allows users to use two different weighting functions, reduce computational runtime, employ a sliding windows framework for large relational sequences, and specify a dyadic cutoff for relational relevancy.

Usage

```
computeRepetition(
  observed_time,
  observed_sender,
  observed_receiver,
  processed_time,
  processed_sender,
  processed_receiver,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  halflife = 2,
  counts = FALSE,
  dyadic_weight = 0,
  window_size = NA,
  Lerneretal_2013 = FALSE
)
```

Arguments

observed_time	The vector of event times from the pre-processing event sequence.
observed_sender	The vector of event senders from the pre-processing event sequence.
observed_receiver	The vector of event receivers from the pre-processing event sequence
processed_time	The vector of event times from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
processed_sender	The vector of event senders from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
processed_receiver	The vector of event receivers from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
sliding_windows	TRUE/FALSE. TRUE indicates that the sliding windows computational approach will be used to compute the network statistic, while FALSE indicates the approach will not be used. Set to FALSE by default. It's important to note that the sliding windows framework should only be used when the pre-processed event

sequence is 'big', such as the 360 million pre-processed event sequence used in Lerner and Lomi (2020), as it aims to reduce the computational burden of sorting 'big' datasets. In general, most pre-processed event sequences will not need to use the sliding windows approach. There is not a strict cutoff for 'big' dataset. This definition depends on both the size of the observed event sequence and the post-processing sampling dataset. For instance, according to our internal tests, when the event sequence is relatively large (i.e., 100,000 observed events) with probability of sampling from the observed event sequence set to 0.05 and using 10 controls per sampled event, the sliding windows framework for computing repetition is about 11% faster than the non-sliding windows framework. Yet, in a smaller dataset (i.e., 10,000 observed events) the sliding windows framework is about 25% slower than the non-sliding framework with the same conditions as before.

`processed_seqIDs`
: If sliding_windows is set to TRUE, the vector of event sequence IDs from the post-processing event sequence. The event sequence IDs represents the index for when the event occurred in the observed event sequence (e.g., the 5th event in the sequence will have a value of 5 in this vector).

`halflife`
: A numerical value that is the halflife value to be used in the exponential weighting function (see the details section). Preset to 2 (should be updated by user).

`counts`
: TRUE/FALSE. TRUE indicates that the counts of past events should be computed (see the details section). FALSE indicates that the temporal exponential weighting function should be used to downweigh past events (see the details section). Set to FALSE by default.

`dyadic_weight`
: A numerical value that is the dyadic cutoff weight that represents the numerical cutoff value for temporal relevancy based on the exponential weighting function. For example, a numerical value of 0.01, indicates that an exponential weight less than 0.01 will become 0 and will not be included in the sum of the past event weights (see the details section). Set to 0 by default.

`window_size`
: If sliding_windows is set to TRUE, the sizes of the windows that are used for the sliding windows computational framework. If NA, the function internally divides the dataset into ten slices (may not be optimal).

`Lerneretal_2013`
: TRUE/FALSE. TRUE indicates that the Lerner et al. (2013) exponential weighting function will be used (see the details section). FALSE indicates that the Lerner and Lomi (2020) exponential weighting function will be used (see the details section). Set to FALSE by default

Details

This function calculates the repetition scores for relational event models based on the exponential weighting function used in either Lerner and Lomi (2020) or Lerner et al. (2013).

Following Lerner and Lomi (2020), the exponential weighting function in relational event models is:

$$w(s, r, t) = e^{-(t-t') \cdot \frac{ln(2)}{T_{1/2}}}$$

Following Lerner et al. (2013), the exponential weighting function in relational event models is:

$$w(s,r,t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}}$$

In both of the above equations, *s* is the current event sender, *r* is the current event receiver (target), *t* is the current event time, *t'* is the past event times that meet the weight subset (in this case, all events that have the same sender and receiver), and $T_{1/2}$ is the halflife parameter.

The formula for repetition for event e_i is:

$$repetition_{e_i} = w(s,r,t)$$

Moreover, researchers interested in modeling temporal relevancy (see Quintane, Mood, Dunn, and Falzone 2022; Lerner and Lomi 2020) can specify the dyadic weight cutoff, that is, the minimum value for which the weight is considered relationally relevant. Users who do not know the specific dyadic cutoff value to use, can use the `computeRemDyadCut` function.

Following Butts (2008), if the counts of the past events are requested, the formula for repetition for event e_i is:

$$repetition_{e_i} = d(s = s', r = r', t')$$

Where, $d()$ is the number of past events where the event sender, *s'*, is the current event sender, *s*, the event receiver (target), *r'*, is the current event receiver, *r*. Moreover, the counting equation can be used in tandem with relational relevancy, by specifying the halflife parameter, exponential weighting function, and the dyadic cut off weight values. If the user is not interested in modeling relational relevancy, then those value should be left at their baseline values.

Value

The vector of repetition statistics for the relational event sequence.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A Relational Event Framework for Social Action." *Sociological Methodology* 38(1): 155-200.

Quintane, Eric, Martin Wood, John Dunn, and Lucia Falzon. 2022. "Temporal Brokering: A Measure of Brokerage as a Behavioral Process." *Organizational Research Methods* 25(3): 459-489.

Lerner, Jürgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: How to fit a relational event model to 360 million dyadic events." *Network Science* 8(1): 97-135.

Lerner, Jürgen, Margit Bussman, Tom A.B. Snijders, and Ulrik Brandes. 2013. " Modeling Frequency and Type of Interaction in Event Networks." *The Corvinus Journal of Sociology and Social Policy* 4(1): 3-32.

Vu, Duy, Philippa Pattison, and Garry Robins. 2015. "Relational event models for social learning in MOOCs." *Social Networks* 43: 121-135.

Examples

```
data("WikiEvent2018.first100k")
WikiEvent2018 <- WikiEvent2018.first100k[1:10000,] #the first ten thousand events
WikiEvent2018$time <- as.numeric(WikiEvent2018$time) #making the variable numeric
### Creating the EventSet By Employing Case-Control Sampling With M = 5 and
### Sampling from the Observed Event Sequence with P = 0.01
EventSet <- processTMEventSeq(
 data = WikiEvent2018, # The Event Dataset
 time = WikiEvent2018$time, # The Time Variable
 eventID = WikiEvent2018$eventID, # The Event Sequence Variable
 sender = WikiEvent2018$user, # The Sender Variable
 receiver = WikiEvent2018$article, # The Receiver Variable
 p_samplingobserved = 0.01, # The Probability of Selection
 n_controls = 5, # The Number of Controls to Sample from the Full Risk Set
 seed = 9999) # The Seed for Replication
#### Estimating Repetition Scores Without the Sliding Windows Framework
EventSet$rep <- computeRepetition(
   observed_time = WikiEvent2018$time,
   observed_sender = WikiEvent2018$user,
   observed_receiver = WikiEvent2018$article,
   processed_time = EventSet$time,
   processed_sender = EventSet$sender,
   processed_receiver = EventSet$receiver,
   halflife = 2.592e+09, #halflife parameter
   dyadic_weight = 0,
   Lerneretal_2013 = FALSE)

EventSet$sw_rep <- computeRepetition(
   observed_time = WikiEvent2018$time,
   observed_sender = WikiEvent2018$user,
   observed_receiver = WikiEvent2018$article,
   processed_time = EventSet$time,
   processed_sender = EventSet$sender,
   processed_receiver = EventSet$receiver,
   processed_seqIDs = EventSet$sequenceID,
   halflife = 2.592e+09, #halflife parameter
   dyadic_weight = 0,
   sliding_window = TRUE,
   Lerneretal_2013 = FALSE)

#The results with and without the sliding windows are the same (see correlation
#below). Using the sliding windows method is recommended when the data are
#big' so that memory allotment is more efficient.
cor(EventSet$sw_rep, EventSet$rep)

#### Estimating Repetition Scores with the Counts of Events Returned
EventSet$repC <- computeRepetition(
   observed_time = WikiEvent2018$time,
   observed_sender = WikiEvent2018$user,
   observed_receiver = WikiEvent2018$article,
   processed_time = EventSet$time,
   processed_sender = EventSet$sender,
```

```
    processed_receiver = EventSet$receiver,
    halflife = 2.592e+09, #halflife parameter
    dyadic_weight = 0,
    Lerneretal_2013 = FALSE,
    counts = TRUE)

cbind(EventSet$rep,
      EventSet$sw_rep,
      EventSet$repC)
```

`computeSenderIndegree` *Compute the Indegree Network Statistic for Event Senders in a Relational Event Sequence*

Description

The function computes the indegree network sufficient statistic for event senders in a relational event sequence (see Lerner and Lomi 2020; Butts 2008). This measure allows for indegree scores to be only computed for the sampled events, while creating the weights based on the full event sequence (see Lerner and Lomi 2020; Vu et al. 2015). The function allows users to use two different weighting functions, reduce computational runtime, employ a sliding windows framework for large relational sequences, and specify a dyadic cutoff for relational relevancy.

Usage

```
computeSenderIndegree(
  observed_time,
  observed_sender,
  observed_receiver,
  processed_time,
  processed_sender,
  processed_receiver,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  counts = FALSE,
  halflife = 2,
  dyadic_weight = 0,
  window_size = NA,
  Lerneretal_2013 = FALSE
)
```

Arguments

`observed_time`	The vector of event times from the pre-processing event sequence.
`observed_sender`	The vector of event senders from the pre-processing event sequence.

observed_receiver
: The vector of event receivers from the pre-processing event sequence

processed_time
: The vector of event times from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).

processed_sender
: The vector of event senders from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).

processed_receiver
: The vector of event receivers from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).

sliding_windows
: TRUE/FALSE. TRUE indicates that the sliding windows computational approach will be used to compute the network statistic, while FALSE indicates the approach will not be used. Set to FALSE by default. It's important to note that the sliding windows framework should only be used when the pre-processed event sequence is 'big', such as the 360 million pre-processed event sequence used in Lerner and Lomi (2020), as it aims to reduce the computational burden of sorting 'big' datasets. In general, most pre-processed event sequences will not need to use the sliding windows approach. There is not a strict cutoff for 'big' dataset. This definition depends on both the size of the observed event sequence and the post-processing sampling dataset. For instance, according to our internal tests, when the event sequence is relatively large (i.e., 100,000 observed events) with probability of sampling from the observed event sequence set to 0.05 and using 10 controls per sampled event, the sliding windows framework for computing repetition is about 11% faster than the non-sliding windows framework. Yet, in a smaller dataset (i.e., 10,000 observed events) the sliding windows framework is about 25% slower than the non-sliding framework with the same conditions as before.

processed_seqIDs
: If sliding_windows is set to TRUE, the vector of event sequence IDs from the post-processing event sequence. The event sequence IDs represents the index for when the event occurred in the observed event sequence (e.g., the 5th event in the sequence will have a value of 5 in this vector).

counts
: TRUE/FALSE. TRUE indicates that the counts of past events should be computed (see the details section). FALSE indicates that the temporal exponential weighting function should be used to downweigh past events (see the details section). Set to FALSE by default.

halflife
: A numerical value that is the halflife value to be used in the exponential weighting function (see the details section). Preset to 2 (should be updated by user).

dyadic_weight
: A numerical value that is the dyadic cutoff weight that represents the numerical cutoff value for temporal relevancy based on the exponential weighting function. For example, a numerical value of 0.01, indicates that an exponential weight less than 0.01 will become 0 and will not be included in the sum of the past event weights (see the details section). Set to 0 by default.

window_size
: If sliding_windows is set to TRUE, the sizes of the windows that are used for the sliding windows computational framework. If NA, the function internally divides the dataset into ten slices (may not be optimal).

`Lerneretal_2013`
: TRUE/FALSE. TRUE indicates that the Lerner et al. (2013) exponential weighting function will be used (see the details section). FALSE indicates that the Lerner and Lomi (2020) exponential weighting function will be used (see the details section). Set to FALSE by default

Details

The function calculates sender indegree scores for relational event sequences based on the exponential weighting function used in either Lerner and Lomi (2020) or Lerner et al. (2013).

Following Lerner and Lomi (2020), the exponential weighting function in relational event models is:

$$w(s,r,t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}}$$

Following Lerner et al. (2013), the exponential weighting function in relational event models is:

$$w(s,r,t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}}$$

In both of the above equations, *s* is the current event sender, *r* is the current event receiver (target), *t* is the current event time, *t'* is the past event times that meet the weight subset, and $T_{1/2}$ is the halflife parameter.

The formula for sender indegree for event e_i is:

$$senderindegree_{e_i} = w(s', s, t)$$

That is, all past events in which the event receiver is the current sender.

Moreover, researchers interested in modeling temporal relevancy (see Quintane, Mood, Dunn, and Falzone 2022; Lerner and Lomi 2020) can specify the dyadic weight cutoff, that is, the minimum value for which the weight is considered relationally relevant. Users who do not know the specific dyadic cutoff value to use, can use the `computeRemDyadCut` function.

Following Butts (2008), if the counts of the past events are requested, the formula for sender indegree for event e_i is:

$$senderindegree_{e_i} = d(r' = s, t')$$

Where, $d()$ is the number of past events where the event receiver, *r'*, is the current event sender *s* . Moreover, the counting equation can be used in tandem with relational relevancy, by specifying the halflife parameter, exponential weighting function, and the dyadic cut off weight values. If the user is not interested in modeling relational relevancy, then those value should be left at their defaults.

Value

The vector of sender indegree statistics for the relational event sequence.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A Relational Event Framework for Social Action." *Sociological Methodology* 38(1): 155-200.

Quintane, Eric, Martin Wood, John Dunn, and Lucia Falzon. 2022. "Temporal Brokering: A Measure of Brokerage as a Behavioral Process." *Organizational Research Methods* 25(3): 459-489.

Lerner, Jürgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: How to fit a relational event model to 360 million dyadic events." *Network Science* 8(1): 97-135.

Lerner, Jürgen, Margit Bussman, Tom A.B. Snijders, and Ulrik Brandes. 2013. " Modeling Frequency and Type of Interaction in Event Networks." *The Corvinus Journal of Sociology and Social Policy* 4(1): 3-32.

Vu, Duy, Philippa Pattison, and Garry Robins. 2015. "Relational event models for social learning in MOOCs." *Social Networks* 43: 121-135.

Examples

```
events <- data.frame(time = 1:18,
                                eventID = 1:18,
                                sender = c("A", "B", "C",
                                           "A", "D", "E",
                                           "F", "B", "A",
                                           "F", "D", "B",
                                           "G", "B", "D",
                                          "H", "A", "D"),
                               target = c("B", "C", "D",
                                          "E", "A", "F",
                                          "D", "A", "C",
                                          "G", "B", "C",
                                          "H", "J", "A",
                                          "F", "C", "B"))

eventSet <- processOMEventSeq(data = events,
                      time = events$time,
                      eventID = events$eventID,
                      sender = events$sender,
                      receiver = events$target,
                      p_samplingobserved = 1.00,
                      n_controls = 1,
                      seed = 9999)

# Computing Sender Indegree Statistics without the sliding windows framework
eventSet$sender.indegree <- computeSenderIndegree(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   dyadic_weight = 0,
```

```
   Lerneretal_2013 = FALSE)

# Computing Sender Indegree Statistics with the sliding windows framework
eventSet$sender.indegree.SW <- computeSenderIndegree(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   processed_seqIDs = eventSet$sequenceID,
   dyadic_weight = 0,
   sliding_window = TRUE,
   Lerneretal_2013 = FALSE)

#The results with and without the sliding windows are the same (see correlation
#below). Using the sliding windows method is recommended when the data are
#big' so that memory allotment is more efficient.
cor(eventSet$sender.indegree.SW,eventSet$sender.indegree)

# Computing Sender Indegree Statistics with the counts of events being returned
eventSet$sender.indegreeC <- computeSenderIndegree(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   dyadic_weight = 0,
   Lerneretal_2013 = FALSE,
   counts = TRUE)

cbind(eventSet$sender.indegree.SW,
     eventSet$sender.indegree,
     eventSet$sender.indegreeC)
```

`computeSenderOutdegree` *Compute the Outdegree Network Statistic for Event Senders in a Relational Event Sequence*

Description

The function computes the sender outdegree network sufficient statistic for a relational event sequence (see Lerner and Lomi 2020; Butts 2008). This measure allows for outdegree scores to be only computed for the sampled events, while creating the weights based on the full event sequence (see Lerner and Lomi 2020; Vu et al. 2015). The function allows users to use two different weighting functions, reduce computational runtime, employ a sliding windows framework for large relational sequences, and specify a dyadic cutoff for relational relevancy.

Usage

```
computeSenderOutdegree(
  observed_time,
  observed_sender,
  processed_time,
  processed_sender,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  counts = FALSE,
  halflife = 2,
  dyadic_weight = 0,
  window_size = NA,
  Lerneretal_2013 = FALSE
)
```

Arguments

observed_time	The vector of event times from the pre-processing event sequence.
observed_sender	The vector of event senders from the pre-processing event sequence.
processed_time	The vector of event times from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
processed_sender	The vector of event senders from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
sliding_windows	TRUE/FALSE. TRUE indicates that the sliding windows computational approach will be used to compute the network statistic, while FALSE indicates the approach will not be used. Set to FALSE by default. It's important to note that the sliding windows framework should only be used when the pre-processed event sequence is 'big', such as the 360 million pre-processed event sequence used in Lerner and Lomi (2020), as it aims to reduce the computational burden of sorting 'big' datasets. In general, most pre-processed event sequences will not need to use the sliding windows approach. There is not a strict cutoff for 'big' dataset. This definition depends on both the size of the observed event sequence and the post-processing sampling dataset. For instance, according to our internal tests, when the event sequence is relatively large (i.e., 100,000 observed events) with probability of sampling from the observed event sequence set to 0.05 and using 10 controls per sampled event, the sliding windows framework for computing repetition is about 11% faster than the non-sliding windows framework. Yet, in a smaller dataset (i.e., 10,000 observed events) the sliding windows framework is about 25% slower than the non-sliding framework with the same conditions as before.
processed_seqIDs	If sliding_windows is set to TRUE, the vector of event sequence IDs from the post-processing event sequence. The event sequence IDs represents the index for when the event occurred in the observed event sequence (e.g., the 5th event in the sequence will have a value of 5 in this vector).

`counts` TRUE/FALSE. TRUE indicates that the counts of past events should be computed (see the details section). FALSE indicates that the temporal exponential weighting function should be used to downweigh past events (see the details section). Set to FALSE by default.

`halflife` A numerical value that is the halflife value to be used in the exponential weighting function (see the details section). Preset to 2 (should be updated by user).

`dyadic_weight` A numerical value that is the dyadic cutoff weight that represents the numerical cutoff value for temporal relevancy based on the exponential weighting function. For example, a numerical value of 0.01, indicates that an exponential weight less than 0.01 will become 0 and will not be included in the sum of the past event weights (see the details section). Set to 0 by default.

`window_size` If sliding_windows is set to TRUE, the sizes of the windows that are used for the sliding windows computational framework. If NA, the function internally divides the dataset into ten slices (may not be optimal).

`Lerneretal_2013` TRUE/FALSE. TRUE indicates that the Lerner et al. (2013) exponential weighting function will be used (see the details section). FALSE indicates that the Lerner and Lomi (2020) exponential weighting function will be used (see the details section). Set to FALSE by default

Details

The function calculates sender outdegree scores for relational event sequences based on the exponential weighting function used in either Lerner and Lomi (2020) or Lerner et al. (2013).

Following Lerner and Lomi (2020), the exponential weighting function in relational event models is:

$$w(s,r,t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}}$$

Following Lerner et al. (2013), the exponential weighting function in relational event models is:

$$w(s,r,t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}}$$

In both of the above equations, *s* is the current event sender, *r* is the current event receiver (target), *t* is the current event time, *t'* is the past event times that meet the weight subset, and $T_{1/2}$ is the halflife parameter.

The formula for sender outdegree for event e_i is:

$$senderoutdegree_{e_i} = w(s,r',t)$$

That is, all past events in which the past sender is the current sender and the event target can be any past user.

Moreover, researchers interested in modeling temporal relevancy (see Quintane, Mood, Dunn, and Falzone 2022; Lerner and Lomi 2020) can specify the dyadic weight cutoff, that is, the minimum value for which the weight is considered relationally relevant. Users who do not know the specific dyadic cutoff value to use, can use the `computeRemDyadCut` function.

Following Butts (2008), if the counts of the past events are requested, the formula for sender outdegree for event e_i is:

$$senderoutdegree_{e_i} = d(s = s', t')$$

Where, $d()$ is the number of past events where the sender *s'* is the current event sender, *s*. Moreover, the counting equation can be used in tandem with relational relevancy, by specifying the halflife parameter, exponential weighting function, and the dyadic cutoff weight values. If the user is not interested in modeling relational relevancy, then those value should be left at their defaults.

Value

The vector of sender outdegree statistics for the relational event sequence.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A Relational Event Framework for Social Action." *Sociological Methodology* 38(1): 155-200.

Quintane, Eric, Martin Wood, John Dunn, and Lucia Falzon. 2022. "Temporal Brokering: A Measure of Brokerage as a Behavioral Process." *Organizational Research Methods* 25(3): 459-489.

Lerner, Jürgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: How to fit a relational event model to 360 million dyadic events." *Network Science* 8(1): 97-135.

Lerner, Jürgen, Margit Bussman, Tom A.B. Snijders, and Ulrik Brandes. 2013. " Modeling Frequency and Type of Interaction in Event Networks." *The Corvinus Journal of Sociology and Social Policy* 4(1): 3-32.

Vu, Duy, Philippa Pattison, and Garry Robins. 2015. "Relational event models for social learning in MOOCs." *Social Networks* 43: 121-135.

Examples

```
events <- data.frame(time = 1:18,
                                eventID = 1:18,
                                sender = c("A", "B", "C",
                                           "A", "D", "E",
                                           "F", "B", "A",
                                           "F", "D", "B",
                                           "G", "B", "D",
                                          "H", "A", "D"),
                               target = c("B", "C", "D",
                                          "E", "A", "F",
                                          "D", "A", "C",
                                          "G", "B", "C",
                                          "H", "J", "A",
                                          "F", "C", "B"))

eventSet <- processOMEventSeq(data = events,
```

```
                        time = events$time,
                        eventID = events$eventID,
                        sender = events$sender,
                        receiver = events$target,
                        p_samplingobserved = 1.00,
                        n_controls = 1,
                        seed = 9999)

# Computing Sender Outdegree Statistics without the sliding windows framework
eventSet$sender_outdegree <- computeSenderOutdegree(
   observed_time = events$time,
   observed_sender = events$sender,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   halflife = 2, #halflife parameter
   dyadic_weight = 0,
   Lerneretal_2013 = FALSE)

# Computing Sender Outdegree Statistics with the sliding windows framework
eventSet$sender_outdegreeSW <- computeSenderOutdegree(
   observed_time = events$time,
   observed_sender = events$sender,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   halflife = 2, #halflife parameter
   processed_seqIDs = eventSet$sequenceID,
   dyadic_weight = 0,
   sliding_window = TRUE,
   Lerneretal_2013 = FALSE)

#The results with and without the sliding windows are the same (see correlation
#below). Using the sliding windows method is recommended when the data are
#big' so that memory allotment is more efficient.
cor(eventSet$sender_outdegreeSW , eventSet$sender_outdegree)

# Computing  Sender Outdegree Statistic with the counts of events being returned
eventSet$sender_outdegreeC <- computeSenderOutdegree(
   observed_time = events$time,
   observed_sender = events$sender,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   halflife = 2, #halflife parameter
   dyadic_weight = 0,
   sliding_window = FALSE,
   counts = TRUE,
   Lerneretal_2013 = FALSE)

cbind(eventSet$sender_outdegree,
     eventSet$sender_outdegreeSW,
     eventSet$sender_outdegreeC)
```

computeTMDegree *Compute Degree Centrality Values for Two-Mode Networks*

Description

This function computes the degree centrality values for two-mode networks following Knoke and Yang (2020). The computed degree centrality is based on the specified level. That is, in an affiliation matrix, the density can be computed on the symmetric *g x g* co-membership matrix of level 1 actors (e.g., medical doctors) or on the symmetric *h x h* shared actors matrix for level 2 groups (e.g., hospitals) based on their shared members.

Usage

```
computeTMDegree(net, level1 = TRUE)
```

Arguments

net	A two-mode adjacency matrix
level1	TRUE/FALSE. TRUE indicates that the degree centrality will be computed for level 1 nodes. FALSE indicates that the degree centrality will be computed for level 2 nodes. Set to TRUE by default.

Details

Following Knoke and Yang (2020), the computation of degree for two-mode affiliation networks is level specific. A two-mode affiliation matrix *X* with dimensions *g x h*, where *g* is the number of level 1 nodes (e.g., medical doctors) and *h* is the number of level 2 nodes (i.e., hospitals). If the function is defined on the level 1 nodes, the degree centrality of an actor *i* is computed as:

$$X^G = XX^T$$

$$C_D^G(g_i) = \sum_{i=1}^{g} x_{ij}^g \quad (i \neq j)$$

In contrast, if it is defined on the level 2 nodes, the degree centrality of an actor *i* is computed as:

$$X^H = X^T X$$

$$C_D^H(h_i) = \sum_{i=1}^{h} x_{ij}^h \quad (i \neq j)$$

Value

The vector of two-mode level-specific degree centrality values.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Knoke, David and Song Yang. 2020. *Social Network Analysis*. Sage: Quantitative Applications in the Social Sciences (154)

Examples

```
#Replicating the biparitate graph presented in Knoke and Yang (2020: 109)
knoke_yang_PC <- matrix(c(1,1,0,0, 1,1,0,0,
                          1,1,1,0, 0,0,1,1,
                          0,0,1,1), byrow = TRUE,
                          nrow = 5, ncol = 4)
colnames(knoke_yang_PC) <- c("Rubio-R","McConnell-R", "Reid-D", "Sanders-D")
rownames(knoke_yang_PC) <- c("UPS", "MS", "HD", "SEU", "ANA")
computeTMDegree(knoke_yang_PC, level1 = TRUE) #this value matches the book
computeTMDegree(knoke_yang_PC, level1 = FALSE) #this value matches the book
```

---

computeTMDens *Compute Level-Specific Graph Density for Two-Mode Networks*

Description

This function computes the density of a two-mode network following Wasserman and Faust (1994) and Knoke and Yang (2020). The density is computed based on the specified level. That is, in an affiliation matrix, density can be computed on the symmetric *g x g* matrix of co-membership for the level 1 actors or on the symmetric *h x h* matrix of shared actors for level 2 groups.

Usage

```
computeTMDens(net, binary = FALSE, level1 = TRUE)
```

Arguments

net	A two-mode adjacency matrix.
binary	TRUE/FALSE. TRUE indicates that the transposed matrices will be binarized (see Wasserman and Faust 1995: 316). FALSE indicates that the transposed matrices will not be binarized. Set to FALSE by default.
level1	TRUE/FALSE. TRUE indicates that the graph density will be computed for level 1 nodes. FALSE indicates that the graph density will be computed for level 2 nodes. Set to FALSE by default.

Details

Following Wasserman and Faust (1994) and Knoke and Yang (2020), the computation of density for two-mode networks is level specific. A two-mode matrix X with dimensions g x h, where g is the number of level 1 nodes (e.g., medical doctors) and h is the number of level 2 nodes (i.e., hospitals). If the function is defined on the level 1 nodes, the density is computed as:

$$X^g = XX^T$$

$$D^g = \frac{\sum_{i=1}^{g}\sum_{j=1}^{g} x_{ij}^g}{g(g-1)}$$

In contrast, if it is defined on the level 2 nodes, the density is:

$$X^h = X^T X$$

$$D^h = \frac{\sum_{i=1}^{h}\sum_{j=1}^{h} x_{ij}^h}{h(h-1)}$$

Moreover, as discussed in Wasserman and Faust (1994: 316), the density can be based on the dichotomous relations instead of the shared membership values. This can be specified by *binary = TRUE*.

Value

The level-specific network density for the two-mode graph.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Wasserman, Stanley and Katherine Faust. 1994. *Social Network Analysis: Methods and Applications*. Cambridge University Press.

Knoke, David and Song Yang. 2020. *Social Network Analysis*. Sage: Quantitative Applications in the Social Sciences (154).

Examples

```
#Replicating the biparitate graph presented in Knoke and Yang (2020: 109)
knoke_yang_PC <- matrix(c(1,1,0,0, 1,1,0,0,
                          1,1,1,0, 0,0,1,1,
                          0,0,1,1), byrow = TRUE,
                          nrow = 5, ncol = 4)
colnames(knoke_yang_PC) <- c("Rubio-R","McConnell-R", "Reid-D", "Sanders-D")
rownames(knoke_yang_PC) <- c("UPS", "MS", "HD", "SEU", "ANA")
#compute two-mode density for level 1
#note: this value does not match that of Knoke and Yang (which we believe
#is a typo in that book), but does match that of Wasserman and
#Faust (1995: 317) for the ceo dataset.
computeTMDens(knoke_yang_PC, level1 = TRUE)
#compute two-mode density for level 2.
#note: this value matches that of the book
computeTMDens(knoke_yang_PC, level1 = FALSE)
```

`computeTMEgoDis` *Compute Fujimoto, Snijders, and Valente's (2018) Ego Homophily Distance for Two-Mode Networks*

Description

This function computes the ego homophily distance in two-mode networks as proposed by Fujimoto, Snijders, and Valente (2018: 380). See Fujimoto, Snijders, and Valente (2018) for more details about this measure.

Usage

```
computeTMEgoDis(net, mem, standardize = FALSE)
```

Arguments

`net`	The two-mode adjacency matrix.
`mem`	The vector of membership values that the homophilous four cycles will be based on.
`standardize`	TRUE/FALSE. TRUE indicates that the sores will be standardized by the number of level 2 nodes the level 1 node is connected to. FALSE indicates that the scores will not be standardized. Set to FALSE by default.

Details

The formula for ego homophily distance in two-mode networks is:

$$Ego2Dist_i = \sum_a y_{ia} 1 - |v_i - p_i a|$$

where:

- $\sum_a$ sums across all level 2 nodes in the network
- y_{ia} is the 1 if node i is tied to node a and 0 else.
- v_i is the value of the respondent. Within the function this is predefined to be 1 if there are multiple categories.
- $p_i a$ is the proportion of same-category actors that are tied to node a not including the ego itself.
- $|v_i - p_i a|$ is equal to 1 if all the level 1 nodes that are tied to the level 2 node share the same categorical membership and 0 if all level 1 nodes are a different category.

If the ego is a level 2 isolate or a level 2 pendant, that is, only one level 1 node (e.g., patient) is connected to that specific level 2 node (e.g., medical doctor), then they are given a value of 0. In particular, the contribution to the ego distance for a pendant is 0. The ego distance value can be standardized by the number of groups which would provide the average ego distance as a proportion between 0 and 1.

Value

The vector of two-mode ego homophily distance.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Fujimoto, Kayo, Tom A.B. Snijders, and Thomas W. Valente. 2018. "Multivariate dynamics of one-mode and two-mode networks: Explaining similarity in sports participation among friends." *Network Science* 6(3): 370-395.

Examples

```
# For this example, we use the Davis Southern Women's Dataset.
data("southern.women")
#creating a random binary membership vector
set.seed(9999)
membership <- sample(0:1, nrow(southern.women), replace = TRUE)
#the ego 2 mode distance non-standardized
computeTMEgoDis(southern.women, mem = membership)
#the ego 2 mode distance standardized
computeTMEgoDis(southern.women, mem = membership, standardize = TRUE)
```

---

computeTriads — *Compute the Triadic Closure Network Statistic for Event Dyads in a Relational Event Sequence*

Description

This function computes the triadic closure network sufficient statistic for a relational event sequence (see Lerner and Lomi 2020; Butts 2008). This measure allows for triadic scores to be only computed for the sampled events, while creating the weights based on the full event sequence (see Lerner and Lomi 2020; Vu et al. 2015). The function allows users to use two different weighting functions, reduce computational runtime, employ a sliding windows framework for large relational sequences, and specify a dyadic cutoff for relational relevancy.

Usage

```
computeTriads(
  observed_time,
  observed_sender,
  observed_receiver,
  processed_time,
  processed_sender,
  processed_receiver,
```

```
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  counts = FALSE,
  halflife = 2,
  dyadic_weight = 0,
  window_size = NA,
  Lerneretal_2013 = FALSE
)
```

Arguments

observed_time	The vector of event times from the pre-processing event sequence.
observed_sender	The vector of event senders from the pre-processing event sequence.
observed_receiver	The vector of event receivers from the pre-processing event sequence
processed_time	The vector of event times from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
processed_sender	The vector of event senders from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
processed_receiver	The vector of event receivers from the post-processing event sequence (i.e., the event sequence that contains the observed and null events).
sliding_windows	TRUE/FALSE. TRUE indicates that the sliding windows computational approach will be used to compute the network statistic, while FALSE indicates the approach will not be used. Set to FALSE by default. It's important to note that the sliding windows framework should only be used when the pre-processed event sequence is 'big', such as the 360 million pre-processed event sequence used in Lerner and Lomi (2020), as it aims to reduce the computational burden of sorting 'big' datasets. In general, most pre-processed event sequences will not need to use the sliding windows approach. There is not a strict cutoff for 'big' dataset. This definition depends on both the size of the observed event sequence and the post-processing sampling dataset. For instance, according to our internal tests, when the event sequence is relatively large (i.e., 100,000 observed events) with probability of sampling from the observed event sequence set to 0.05 and using 10 controls per sampled event, the sliding windows framework for computing repetition is about 11% faster than the non-sliding windows framework. Yet, in a smaller dataset (i.e., 10,000 observed events) the sliding windows framework is about 25% slower than the non-sliding framework with the same conditions as before.
processed_seqIDs	If sliding_windows is set to TRUE, the vector of event sequence IDs from the post-processing event sequence. The event sequence IDs represents the index for when the event occurred in the observed event sequence (e.g., the 5th event in the sequence will have a value of 5 in this vector).

`counts` TRUE/FALSE. TRUE indicates that the counts of past events should be computed (see the details section). FALSE indicates that the temporal exponential weighting function should be used to downweigh past events (see the details section). Set to FALSE by default.

`halflife` A numerical value that is the halflife value to be used in the exponential weighting function (see the details section). Preset to 2 (should be updated by user).

`dyadic_weight` A numerical value that is the dyadic cutoff weight that represents the numerical cutoff value for temporal relevancy based on the exponential weighting function. For example, a numerical value of 0.01, indicates that an exponential weight less than 0.01 will become 0 and will not be included in the sum of the past event weights (see the details section). Set to 0 by default.

`window_size` If sliding_windows is set to TRUE, the sizes of the windows that are used for the sliding windows computational framework. If NA, the function internally divides the dataset into ten slices (may not be optimal).

`Lerneretal_2013` TRUE/FALSE. TRUE indicates that the Lerner et al. (2013) exponential weighting function will be used (see the details section). FALSE indicates that the Lerner and Lomi (2020) exponential weighting function will be used (see the details section). Set to FALSE by default

Details

This function calculates triadic scores for relational event sequences based on the exponential weighting function used in either Lerner and Lomi (2020) or Lerner et al. (2013).

Following Lerner and Lomi (2020), the exponential weighting function in relational event models is:

$$w(s, r, t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}}$$

Following Lerner et al. (2013), the exponential weighting function in relational event models is:

$$w(s, r, t) = e^{-(t-t')\cdot\frac{ln(2)}{T_{1/2}}} \cdot \frac{ln(2)}{T_{1/2}}$$

In both of the above equations, *s* is the current event sender, *r* is the current event receiver (target), *t* is the current event time, *t'* is the past event times that meet the weight subset, and $T_{1/2}$ is the halflife parameter.

The general formula for triadic structures for event e_i is:

$$triadic_{e_i} = \sqrt{\sum_k w(s, r', t) \cdot w(s', r, t)}$$

That is, this function combines all triadic structures discussed in Butts (2008) into a single summation such that the computed scores include incoming shared partners, outgoing shared partners, incoming two paths, and outgoing two paths.

Moreover, researchers interested in modeling temporal relevancy (see Quintane, Mood, Dunn, and Falzone 2022; Lerner and Lomi 2020) can specify the dyadic weight cutoff, that is, the minimum

value for which the weight is considered relationally relevant. Users who do not know the specific dyadic cutoff value to use, can use the computeRemDyadCut function.

Following Butts (2008), if the counts of the past events are requested, the formula for triadic structures for event e_i is:

$$TS_{e_i} = \sum_{i=1}^{|H|} \min\left[d(s, r', t), d(s', r, t)\right]$$

where, $d()$ is the number of past events that meet the specific set operations. Notably, this function combines all triadic structures discussed in Butts (2008) into a single summation, such that the computed scores include incoming shared partners, outgoing shared partners, incoming two paths, and outgoing two paths. The sum loops through all unique actors that have formed past sent or received ties from the current event sender and receiver. Moreover, the counting equation can be used in tandem with relational relevancy, by specifying the halflife parameter, exponential weighting function, and the dyadic cut off weight values. If the user is not interested in modeling relational relevancy, then those value should be left at their baseline values.

Value

The vector of triadic closure network statistics for the relational event sequence.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A Relational Event Framework for Social Action." *Sociological Methodology* 38(1): 155-200.

Quintane, Eric, Martin Wood, John Dunn, and Lucia Falzon. 2022. "Temporal Brokering: A Measure of Brokerage as a Behavioral Process." *Organizational Research Methods* 25(3): 459-489.

Lerner, Jürgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: How to fit a relational event model to 360 million dyadic events." *Network Science* 8(1): 97-135.

Lerner, Jürgen, Margit Bussman, Tom A.B. Snijders, and Ulrik Brandes. 2013. " Modeling Frequency and Type of Interaction in Event Networks." *The Corvinus Journal of Sociology and Social Policy* 4(1): 3-32.

Vu, Duy, Philippa Pattison, and Garry Robins. 2015. "Relational event models for social learning in MOOCs." *Social Networks* 43: 121-135.

Examples

```
events <- data.frame(time = 1:18,
                                eventID = 1:18,
                                sender = c("A", "B", "C",
                                           "A", "D", "E",
                                           "F", "B", "A",
                                           "F", "D", "B",
                                           "G", "B", "D",
```

```
                                        "H", "A", "D"),
                                target = c("B", "C", "D",
                                           "E", "A", "F",
                                           "D", "A", "C",
                                           "G", "B", "C",
                                           "H", "J", "A",
                                           "F", "C", "B"))

eventSet <- processOMEventSeq(data = events,
                      time = events$time,
                      eventID = events$eventID,
                      sender = events$sender,
                      receiver = events$target,
                      p_samplingobserved = 1.00,
                      n_controls = 1,
                      seed = 9999)

# Computing Triadic Statistics without the sliding windows framework
eventSet$triadic <- computeTriads(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   dyadic_weight = 0,
   Lerneretal_2013 = FALSE)

# Computing Triadic Statistics with the sliding windows framework
eventSet$triadicSW <- computeTriads(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
   processed_time = eventSet$time,
   processed_sender = eventSet$sender,
   processed_receiver = eventSet$receiver,
   halflife = 2, #halflife parameter
   processed_seqIDs = eventSet$sequenceID,
   dyadic_weight = 0,
   sliding_window = TRUE,
   Lerneretal_2013 = FALSE)

#The results with and without the sliding windows are the same (see correlation
#below). Using the sliding windows method is recommended when the data are
#big' so that memory allotment is more efficient.
cor(eventSet$triadic , eventSet$triadicSW)

# Computing Triadic Statistics with the counts of events being returned
eventSet$triadicC <- computeTriads(
   observed_time = events$time,
   observed_sender = events$sender,
   observed_receiver = events$target,
```

```
    processed_time = eventSet$time,
    processed_sender = eventSet$sender,
    processed_receiver = eventSet$receiver,
    halflife = 2, #halflife parameter
    dyadic_weight = 0,
    sliding_window = FALSE,
    counts = TRUE,
    Lerneretal_2013 = FALSE)

cbind(eventSet$triadic,
      eventSet$triadicSW,
      eventSet$triadicC)
```

`dream` *dream: A Package for Dynamic Relational Event Analysis and Modeling*

Description

The dream package provides users with helpful functions for relational and event analysis. In particular, dream provides users with helper functions for large relational event analysis, such as recently proposed sampling procedures for creating relational risk sets. Alongside the set of functions for relational event analysis, this package includes functions for the structural analysis of one- and two-mode networks, such as network constraint and effective size measures. This package was developed with support from the National Science Foundation's (NSF) Human Networks and Data Science Program (HNDS) under award number 2241536 (PI: Diego F. Leal). Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

dream functions

The functions in dream can be grouped into four useful categories:

- Create Dynamic Risk Sets for (Large) Relational Event Models
  - Functions: `processOMEventSeq` and `processTMEventSeq`.
- Compute Network Statistics for (Large) Relational Event Models
  - Functions: `computeISP`, `computeITP`, `computeOSP`, `computeOTP`, `computeFourCycles`, `computeFourCycles`, `computePersistence`, `computePrefAttach`, `computeReceiverIndegree`, `computeReceiverOutdegree`, `computeRecency`, `computeReciprocity`, `computeRemDyadCut`, `computeRepetition`, `computeSenderIndegree`, `computeSenderOutdegree`, and `computeTriads`.
- Estimate and Simulate (Large) Relational Event Models
  - Functions: `estimateREM` and `simulateRESeq`.
- Compute One- and Two-Mode Network Structural Measures
  - Functions: `computeBCConstraint`, `computeBCES`, `computeBCRedund`, `computeBurtsConstraint`, `computeBurtsES`, `computeHomFourCycles`, `computeLealBrokerage`, `computeNPaths`, `computeTMDegree`, `computeTMDens`, and `computeTMEgoDis`.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

---

`estimateREM` *Fit a Relational Event Model (REM) to Event Sequence Data*

Description

This function estimates the ordinal timing relational event model by maximizing the likelihood function given by Butts (2008) via maximum likelihood estimation. A nice outcome is that the ordinal timing relational event model is equivalent to the conditional logistic regression (see Greene 2003; for R functions, see `clogit`). In addition, based on this outcome and the structure of the data, this function can estimate the Cox proportional hazards model (see Box-Steffensmeier and Jones 2004; for R functions, see `coxph`) given that the likelihood functions are equivalent. An important assumption this model makes is that only one event occurs at each time point. If this is unfeasible for the user's specific dataset, we encourage the user to see the `clogit` function for the Breslow approximation technique (Box-Steffensmeier and Jones 2004). Future versions of the package will include options for interval timing relational event models and tied event data (e.g., multiple events at one time point).

Usage

```
estimateREM(
  formula,
  event.cluster,
  data,
  ordinal = TRUE,
  multiple.events = FALSE,
  newton.rhapson = TRUE,
  optim.method = "BFGS",
  optim.control = list(),
  tolerance = 1e-09,
  maxit = 20,
  starting.beta = NULL,
  ...
)
```

Arguments

`formula`	A formula object with the dependent variable on the left hand side of ~ and the covariates on the right hand side. This is the same argument found in `lm` and `glm`.
`event.cluster`	An integer or factor vector that groups each observed event with its corresponding control (null) events. This vector defines the strata in the event sequence, ensuring that each stratum contains one observed event and its associated null alternatives. It is used to structure the likelihood by stratifying events based on their occurrence in time.

`data`	The data.frame that contains the variable included in the formula argument.
`ordinal`	TRUE/FALSE. Currently, this function supports only the estimation of ordinal timing relational event models (see Butts 2008). Future versions of the package will include estimation options for interval timing relational event models. At this time, this argument is preset to TRUE and should not be modified by the user.
`multiple.events`	TRUE/FALSE. Currently, this function assumes that only one event occurs per event cluster (i.e., time point). Future versions of the package will include estimation options for multiple events per time point, commonly referred to as tied events, via the Breslow approximation technique (see Box-Steffensmeier and Jones 2004). At this moment, this argument is preset to FALSE and should not be modified by the user.
`newton.rhapson`	TRUE/FALSE. TRUE indicates an internal Newton-Rhapson iteration procedure with line searching is used to find the set of maximum likelihood estimates. FALSE indicates that the log likelihood function will be optimized via the `optim` function. The function defaults to TRUE.
`optim.method`	If newton.rhapson is FALSE, what optim method should be used in conjunction with the `optim` function. Defaults to "BFGS". See the `optim` function for the set of options.
`optim.control`	If newton.rhapson is FALSE, a list of control to be used in the `optim` function. See the `optim` function for the set of controls.
`tolerance`	If newton.rhapson is TRUE, the stopping criterion for the absolute difference in the log likelihoods for each Newton-Rhapson iteration. The optimization procedure stops when the absolute change in the log likelihoods is less than `tolerance` (see Greene 2003).
`maxit`	If newton.rhapson is TRUE, the maximum number of iterations for the Newton-Rhapson optimization procedure (see Greene 2003).
`starting.beta`	A numeric vector that represents the starting parameter estimates for the Newton-Rhapson optimization procedure. This may be a beneficial argument if the optimization procedure fails, since the Newton-Rhapson optimization procedure is sensitive to starting values. Preset to NULL.
`...`	Additional arguments.

Details

This function maximizes the ordinal timing relational event model likelihood function provided in the seminal REM paper by Butts (2008). The likelihood function is:

$$L(E|\beta) = \prod_{i=1}^{|E|} \frac{\lambda_{e_i}}{\sum_{e' \in RS_{e_i}} \lambda_{e'}}$$

where, following Butts (2008) and Duxbury (2020), E is the relational event sequence, λ_{e_i} is the hazard rate for event *i*, which is formulated to be equal to $exp(\beta^T z(x, Y))$, that is, the linear combination of user-specific covariates, $z(x, Y)$, and associated REM parameters, β. Following Duxbury (2020), $z(x, Y)$ is a mapping function that represents the endogenous network statistics computed

on the network of past events,x, and exogenous covariates, Y. The user provides these covariates via the `formula` argument.

This function provides two numerical optimization techniques to find the maximum likelihood estimates for the associated parameters. First, this function allows the user to use the `optim` function to find the associated parameters based on the above likelihood function. Secondly, and by default, this function employs a Newton-Rhapson iteration algorithm with line-searching to find the unknown parameters (see Greene 2003 for a discussion of this algorithm). If desired, the user can provide the initial searching values for both algorithms with the `starting.beta` argument.

It's important to note that the modeling concerns of the conditional logistic regression apply to the ordinal timing relational event model, such as no within-sequence fixed effects, that is, a variable that does not vary within event cluster (i.e., a variable that is the same for both the null and observed events). The function internally checks for this and provides the user with a warning if any requested effects has no total within-event variance. Moreover, any observed events that have no associated control events are removed from the analysis as they provide no information to the log likelihood (see Greene 2003). The function removes these events from the sequence prior to estimation.

Value

An object of class "dream" as a list containing the following components:

optimization.method The optimzation method used to find the parameters..

converged TRUE/FALSE. TRUE indicates that the REM converged.

loglikelihood.null The log likelihood of the null model (i.e., the model where the parameters are assumed to be 0).

loglikelihood.full The log likelihood of the estimated model.

chi.stat The chi-statistic of the likelihood ratio test.

loglikelihood.test The p-value of the likelihood ratio test.

df.null The degrees of freedom of the null model.

df.full The degrees of freedom of the full model.

parameters The MLE parameter estimates.

hessian The estimated hessian matrix.

gradient The estimated gradient vector.

se.parameter The standard errors of the MLE parameter estimates.

covariance.mat The estimated variance-covariance matrix.

z.values The z-scores for the MLE parameter estimates.

p.values The p-values for the MLE parameter estimates.

AIC The AIC of the estimated REM.

BIC The BIC of the estimated REM.

n.events The number of observed events in the relational event sequence.

null.events The number of control events in the relational event sequence.

newton.iterations The number of Newton-Rhapson iterations.

search.algo A data.frame object that contains the Newton-Rhapson searching algorithm results.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Box-Steffensmeier, Janet and Bradford S. Jones. 2004. *Event History Modeling: A Guide for Social Scientists*. Cambridge University Press.

Butts, Carter T. 2008. "A Relational Event Framework for Social Action." *Sociological Methodology* 38(1): 155-200.

Duxbury, Scott. 2020. *Longitudinal Network Models*. Sage University Press. Quantitative Applications in the Social Sciences: 192.

Greene, William H. 2003. *Econometric Analysis*. Fifth Edition. Prentice Hall Press.

Examples

```
#Creating a psuedo one-mode relational event sequence with ordinal timing
relational.seq <- simulateRESeq(n_actors = 8,
                                n_events = 50,
                                inertia = TRUE,
                                inertia_p = 0.10,
                                sender_outdegree = TRUE,
                                sender_outdegree_p = 0.05)

#Creating a post-processing event sequence for the above relational sequence
post.processing <- processOMEventSeq(data = relational.seq,
                                     time = relational.seq$eventID,
                                     eventID = relational.seq$eventID,
                                     sender = relational.seq$sender,
                                     receiver = relational.seq$target,
                                     n_controls = 5)

#Computing the sender-outdegree statistic for the above post-processing
#one-mode relational event sequence
post.processing$sender.outdegree <- computeSenderOutdegree(
                                    observed_time = relational.seq$eventID,
                                    observed_sender = relational.seq$sender,
                                    processed_time = post.processing$time,
                                    processed_sender = post.processing$sender,
                                    counts = TRUE)

#Computing the inertia/repetition statistic for the above post-processing
#one-mode relational event sequence
post.processing$inertia <- computeRepetition(
                           observed_time = relational.seq$eventID,
                           observed_sender = relational.seq$sender,
                           observed_receiver = relational.seq$target,
                           processed_time = post.processing$time,
                           processed_sender = post.processing$sender,
                           processed_receiver = post.processing$receiver,
                           counts = TRUE)
```

```
#Fitting a (ordinal) relational event model to the above one-mode relational
#event sequence
rem <- estimateREM(observed~sender.outdegree+inertia,
                   event.cluster = post.processing$time,
                   data=post.processing)
summary(rem) #summary of the relational event model

#Fitting a (ordinal) relational event model to the above one-mode relational
#event sequence via the optim function
rem1 <- estimateREM(observed~sender.outdegree+inertia,
                   event.cluster = post.processing$time,
                   data=post.processing,
                   newton.rhapson=FALSE) #use the optim function
summary(rem1) #summary of the relational event model
```

`print.dream` *Print Method for Summary of dream Model*

Description

Print Method for Summary of dream Model

Usage

```
## S3 method for class 'dream'
print(x, digits = 4, ...)
```

Arguments

`x`	An object of class "summary.dream".
`digits`	The number of digits to print after the decimal point.
`...`	Additional arguments (currently unused).

Value

No return value. Prints out the main results of a 'dream' object.

print.summary.dream *Print Method for dream Model*

Description

Print Method for dream Model

Usage

```
## S3 method for class 'summary.dream'
print(x, digits = 4, ...)
```

Arguments

x	An object of class "dream".
digits	The number of digits to print after the decimal point.
...	Additional arguments (currently unused).

Value

No return value. Prints out the main results of a 'dream' summary object.

processOMEventSeq *Process and Create Risk Sets for a One-Mode Relational Event Sequence*

Description

This function creates a one-mode post-sampling eventset with options for case-control sampling (Vu et al. 2015), sampling from the observed event sequence (Lerner and Lomi 2020), and time- or event-dependent risk sets. Case-control sampling samples an arbitrary m number of controls from the risk set for any event (Vu et al. 2015). Lerner and Lomi (2020) proposed sampling from the observed event sequence where observed events are sampled with probability p. The time- and event-dependent risk sets generate risk sets where the potential null events are based upon a specified past relational time window, such as events that have occurred in the past year. Importantly, this function creates risk sets based upon the assumption that only actors active in past events are in relevant for the creation of the risk set. Users interested in generating risk sets that assume all actors active at any time point within the event sequence are in the risk set at every time point should consult the `createRemDataset` and `remify` functions. Future versions of this package will incorporate this option into the function.

Usage

```
processOMEventSeq(
  data,
  time,
  eventID,
  sender,
  receiver,
  p_samplingobserved = 1,
  n_controls,
  time_dependent = FALSE,
  timeDV = NULL,
  timeDif = NULL,
  seed = 9999
)
```

Arguments

`data`	The full relational event sequence dataset.
`time`	The vector of event time values from the observed event sequence.
`eventID`	The vector of event IDs from the observed event sequence (typically a numerical event sequence that goes from 1 to *n*).
`sender`	The vector of event senders from the observed event sequence.
`receiver`	The vector of event receivers from the observed event sequence.
`p_samplingobserved`	The numerical value for the probability of selection for sampling from the observed event sequence. Set to 1 by default indicating that all observed events from the event sequence will be included in the post-processing event sequence.
`n_controls`	The numerical value for the number of null event controls for each (sampled) observed event.
`time_dependent`	TRUE/FALSE. TRUE indicates that a time- or event-dependent dynamic risk set will be created in which only actors involved in a user-specified relationally relevant (time or event) span (i.e., the 'stretch' of relational relevancy, such as one month for a time-dependent risk set or 100 events for an event-dependent risk set) are included in the potential risk set. FALSE indicates the complete set of actors involved in past events will be included in the risk set (see the details section). Set to FALSE by default.
`timeDV`	If time_dependent = TRUE, the vector of event time values that corresponds to the creation of the time- *or* event-dependent dynamic risk set (see the details section). *This may or may not be the same vector provided to the time argument*. The *timeDV* vector can be the same vector provided to the *time* argument, in which the relational time span will be based on the event timing within the dataset. In contrast, the *timeDV* vector can also be the vector of numerical event IDs which correspond to the number sequence of events. Moreover, the *timeDV* can also be another measurement that is not the *time* argument or a numerical event ID sequence, such as the number of days, months, years, etc. since the first event.

`timeDif` If time_dependent = TRUE, the numerical value that represents the time or event span for the creation of the risk set (see the details section). This argument must be in the same measurement unit as the `timeDV` argument. For instance, in an event-dependent dynamic risk set, if `timeDV` is the number of events since the first event (i.e., a numerical event ID sequence) and only those actors involved in the past, say, 100 events, are considered relationally relevant for the creation of the null events for the current observed event, then `timeDIF` should be set to 100. In the time-dependent dynamic risk set case, let's say that only those actors involved in events that occurred in the past month are considered relationally relevant for the risk set. Let's also assume that the `timeDV` vector is measured in the number of days since the first event. Then `timeDif` should be set to 30 in this particular case.

`seed` The random number seed for user replication.

Details

This function processes observed events from the set E, where each event e_i is defined as:

$$e_i \in E = (s_i, r_i, t_i, G[E;t])$$

where:

- s_i is the sender of the event.
- r_i is the receiver of the event.
- t_i represents the time of the event.
- $G[E;t] = \{e_1, e_2, \ldots, e_{t'} \mid t' < t\}$ is the network of past events, that is, all events that occurred prior to the current event, e_i.

Following Butts (2008) and Butts and Marcum (2017), we define the risk (support) set of all possible events at time t, A_t, as the full Cartesian product of prior senders and receivers in the set $G[E;t]$ that could have occurred at time t. Formally:

$$A_t = \{(s,r) \mid s \in G[E;t] \text{ X } r \in G[E;t]\}$$

where $G[E;t]$ is the set of events up to time t.

Case-control sampling maintains the full set of observed events, that is, all events in E, and samples an arbitrary number m of non-events from the support set A_t (Vu et al. 2015; Lerner and Lomi 2020). This process generates a new support set, SA_t, for any relational event e_i contained in E given a network of past events $G[E;t]$. SA_t is formally defined as:

$$SA_t \subseteq \{(s,r) \mid s \in G[E;t] \text{ X } r \in G[E;t]\}$$

and in the process of sampling from the observed events, n number of observed events are sampled from the set E with known probability $0 < p \leq 1$. More formally, sampling from the observed set generates a new set $SE \subseteq E$.

A time *or* event-dependent dynamic risk set can be created where the set of potential events, that is, all events in the risk set, At, is based only on the set of actors active in a specified event or time span from the current event (e.g., such as within the past month or within the past 100 events). In other words, the specified event or time span can be based on either: a) a specified time span based

upon the actual timing of the past events (e.g., years, months, days or even milliseconds as in the case of Lerner and Lomi 2020), or b) a specified number of events based on the ordering of the past events (e.g., such as all actors involved in the past 100 events). Thus, if time- or event-dependent dynamic risk sets are desired, the user should set time_dependent to TRUE, and then specify the accompanying time vector, `timeDV`, defined as the number of time units (e.g., days) or the number of events since the first event. Moreover, the user should also specify the cutoff threshold with the `timeDif` value that corresponds directly to the measurement unit of `timeDV` (e.g., days). For example, let's say you wanted to create a time-dependent dynamic risk set that only includes actors active within the past month, then you should create a vector of values `timeDV`, which for each event represents the number of days since the first event, and then specify `timeDif` to 30. Similarly, let's say you wanted to create an event-dependent dynamic risk set that only includes actors involved in the past 100 events, then you should create a vector of values `timeDV`, that is, the counts of events since the first event (e.g., 1:n), and then specify `timeDif` to 100.

Value

A post-processing data table with the following columns:

- `sender` - The event senders of the sampled and observed events.
- `receiver` - The event targets (receivers) of the sampled and observed events.
- `time` - The event time for the sampled and observed events.
- `sequenceID` - The numerical event sequence ID for the sampled and observed events.
- `observed` - Boolean indicating if the event is a sampled event or observed event. (1 = observed; 0 = sampled)

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A Relational Event Framework for Social Action." *Sociological Methodology* 38(1): 155-200.

Butts, Carter T. and Christopher Steven Marcum. 2017. "A Relational Event Approach to Modeling Behavioral Dynamics." In A. Pilny & M. S. Poole (Eds.), *Group processes: Data-driven computational approaches*. Springer International Publishing.

Lerner, Jürgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: How to fit a relational event model to 360 million dyadic events." *Network Science* 8(1): 97–135.

Vu, Duy, Philippa Pattison, and Garry Robins. 2015. "Relational event models for social learning in MOOCs." *Social Networks* 43: 121-135.

Examples

```
# A random one-mode relational event sequence
set.seed(9999)
events <- data.frame(time = sort(rexp(1:18)),
                                  eventID = 1:18,
```

```
                                  sender = c("A", "B", "C",
                                             "A", "D", "E",
                                             "F", "B", "A",
                                             "F", "D", "B",
                                             "G", "B", "D",
                                             "H", "A", "D"),
                                target = c("B", "C", "D",
                                           "E", "A", "F",
                                           "D", "A", "C",
                                           "G", "B", "C",
                                           "H", "J", "A",
                                           "F", "C", "B"))

# Creating a one-mode relational risk set with p = 1.00 (all true events)
# and 5 controls
eventSet <- processOMEventSeq(data = events,
                      time = events$time,
                      eventID = events$eventID,
                      sender = events$sender,
                      receiver = events$target,
                      p_samplingobserved = 1.00,
                      n_controls = 5,
                      seed = 9999)

# Creating a event-dependent one-mode relational risk set with p = 1.00 (all
# true events) and 3 controls based upon the past 5 events prior to the current event.
events$timeseq <- 1:nrow(events)
eventSetT <- processOMEventSeq(data = events,
                       time = events$time,
                       eventID = events$eventID,
                       sender = events$sender,
                       receiver = events$target,
                       p_samplingobserved = 1.00,
                       time_dependent = TRUE,
                       timeDV = events$timeseq,
                       timeDif = 5,
                       n_controls = 3,
                       seed = 9999)

# Creating a time-dependent one-mode relational risk set with p = 1.00 (all
# true events) and 3 controls based upon the past 0.40 time units.
eventSetT <- processOMEventSeq(data = events,
                       time = events$time,
                       eventID = events$eventID,
                       sender = events$sender,
                       receiver = events$target,
                       p_samplingobserved = 1.00,
                       time_dependent = TRUE,
                       timeDV = events$time, #the original time variable
                       timeDif = 0.40, #time difference of 0.40 units
                       n_controls = 3,
                       seed = 9999)
```

processTMEventSeq *Process and Create Risk Sets for a Two-Mode Relational Event Sequence*

Description

This function creates a two-mode post-sampling eventset with options for case-control sampling (Vu et al. 2015), sampling from the observed event sequence (Lerner and Lomi 2020), and time- or event-dependent risk sets. Case-control sampling samples an arbitrary m number of controls from the risk set for any event (Vu et al. 2015). Lerner and Lomi (2020) proposed sampling from the observed event sequence where observed events are sampled with probability p. The time- and event-dependent risk sets generate risk sets where the potential null events are based upon a specified past relational time window, such as events that have occurred in the past month. Users interested in generating risk sets that assume all actors active at any time point within the event sequence are in the risk set at every time point should consult the `createRemDataset` and `remify` functions. Future versions of this package will incorporate this option into the function.

Usage

```
processTMEventSeq(
  data,
  time,
  eventID,
  sender,
  receiver,
  p_samplingobserved = 1,
  n_controls,
  time_dependent = FALSE,
  timeDV = NULL,
  timeDif = NULL,
  seed = 9999
)
```

Arguments

`data`	The full relational event sequence dataset.
`time`	The vector of event time values from the observed event sequence.
`eventID`	The vector of event IDs from the observed event sequence (typically a numerical event sequence that goes from 1 to n).
`sender`	The vector of event senders from the observed event sequence.
`receiver`	The vector of event receivers from the observed event sequence.
`p_samplingobserved`	The numerical value for the probability of selection for sampling from the observed event sequence. Set to 1 by default indicating that all observed events from the event sequence will be included in the post-processing event sequence.

`n_controls` The numerical value for the number of null event controls for each (sampled) observed event.

`time_dependent` TRUE/FALSE. TRUE indicates that a time- or event-dependent dynamic risk set will be created in which only actors involved in a user-specified relationally relevant (time or event) span (i.e., the 'stretch' of relational relevancy, such as one month for a time-dependent risk set or 100 events for an event-dependent risk set) are included in the potential risk set. FALSE indicates the complete set of actors involved in past events will be included in the risk set (see the details section). Set to FALSE by default.

`timeDV` If time_dependent = TRUE, the vector of event time values that corresponds to the creation of the time- *or* event-dependent dynamic risk set (see the details section). *This may or may not be the same vector provided to the time argument*. The *timeDV* vector can be the same vector provided to the *time* argument, in which the relational time span will be based on the event timing within the dataset. In contrast, the *timeDV* vector can also be the vector of numerical event IDs which correspond to the number sequence of events. Moreover, the *timeDV* can also be another measurement that is not the *time* argument or a numerical event ID sequence, such as the number of days, months, years, etc. since the first event.

`timeDif` If time_dependent = TRUE, the numerical value that represents the time or event span for the creation of the risk set (see the details section). This argument must be in the same measurement unit as the `timeDV` argument. For instance, in an event-dependent dynamic risk set, if `timeDV` is the number of events since the first event (i.e., a numerical event ID sequence) and only those actors involved in the past, say, 100 events, are considered relationally relevant for the creation of the null events for the current observed event, then `timeDIF` should be set to 100. In the time-dependent dynamic risk set case, let's say that only those actors involved in events that occurred in the past month are considered relationally relevant for the risk set. Let's also assume that the `timeDV` vector is measured in the number of days since the first event. Then `timeDif` should be set to 30 in this particular case.

`seed` The random number seed for user replication.

Details

This function processes observed events from the set E, where each event e_i is defined as:

$$e_i \in E = (s_i, r_i, t_i, G[E; t])$$

where:

- s_i is the sender of the event.
- r_i is the receiver of the event.
- t_i represents the time of the event.
- $G[E; t] = \{e_1, e_2, \ldots, e_{t'} \mid t' < t\}$ is the network of past events, that is, all events that occurred prior to the current event, e_i.

Following Butts (2008) and Butts and Marcum (2017), we define the risk (support) set of all possible events at time t, A_t, as the cross product of two disjoint sets, namely, prior senders and receivers, in the set $G[E;t]$ that could have occurred at time t. Formally:

$$A_t = \{(s,r) \mid s \in G[E;t] \text{ X } r \in G[E;t]\}$$

where $G[E;t]$ is the set of events up to time t.

Case-control sampling maintains the full set of observed events, that is, all events in E, and samples an arbitrary number m of non-events from the support set A_t (Vu et al. 2015; Lerner and Lomi 2020). This process generates a new support set, SA_t, for any relational event e_i contained in E given a network of past events $G[E;t]$. SA_t is formally defined as:

$$SA_t \subseteq \{(s,r) \mid s \in G[E;t] \text{ X } r \in G[E;t]\}$$

and in the process of sampling from the observed events, n number of observed events are sampled from the set E with known probability $0 < p \leq 1$. More formally, sampling from the observed set generates a new set $SE \subseteq E$.

A time *or* event-dependent dynamic risk set can be created where the set of potential events, that is, all events in the risk set, At, is based only on the set of actors active in a specified event or time span from the current event (e.g., such as within the past month or within the past 100 events). In other words, the specified event or time span can be based on either: a) a specified time span based upon the actual timing of the past events (e.g., years, months, days or even milliseconds as in the case of Lerner and Lomi 2020), or b) a specified number of events based on the ordering of the past events (e.g., such as all actors involved in the past 100 events). Thus, if time- or event-dependent dynamic risk sets are desired, the user should set time_dependent to TRUE, and then specify the accompanying time vector, `timeDV`, defined as the number of time units (e.g., days) or the number of events since the first event. Moreover, the user should also specify the cutoff threshold with the `timeDif` value that corresponds directly to the measurement unit of `timeDV` (e.g., days). For example, let's say you wanted to create a time-dependent dynamic risk set that only includes actors active within the past month, then you should create a vector of values `timeDV`, which for each event represents the number of days since the first event, and then specify `timeDif` to 30. Similarly, let's say you wanted to create an event-dependent dynamic risk set that only includes actors involved in the past 100 events, then you should create a vector of values `timeDV`, that is, the counts of events since the first event (e.g., 1:n), and then specify `timeDif` to 100.

Value

A post-processing data table with the following columns:

- `sender` - The event senders of the sampled and observed events.
- `receiver` - The event targets (receivers) of the sampled and observed events.
- `time` - The event time for the sampled and observed events.
- `sequenceID` - The numerical event sequence ID for the sampled and observed events.
- `observed` - Boolean indicating if the event is a sampled event or observed event. (1 = observed; 0 = sampled)

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Butts, Carter T. 2008. "A Relational Event Framework for Social Action." *Sociological Methodology* 38(1): 155-200.

Butts, Carter T. and Christopher Steven Marcum. 2017. "A Relational Event Approach to Modeling Behavioral Dynamics." In A. Pilny & M. S. Poole (Eds.), *Group processes: Data-driven computational approaches*. Springer International Publishing.

Lerner, Jürgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: How to fit a relational event model to 360 million dyadic events." *Network Science* 8(1): 97–135.

Vu, Duy, Philippa Pattison, and Garry Robins. 2015. "Relational event models for social learning in MOOCs." *Social Networks* 43: 121-135.

Examples

```
data("WikiEvent2018.first100k")
WikiEvent2018.first100k$time <- as.numeric(WikiEvent2018.first100k$time)
### Creating the EventSet By Employing Case-Control Sampling With M = 10 and
### Sampling from the Observed Event Sequence with P = 0.01
EventSet <- processTMEventSeq(
  data = WikiEvent2018.first100k, # The Event Dataset
  time = WikiEvent2018.first100k$time, # The Time Variable
  eventID = WikiEvent2018.first100k$eventID, # The Event Sequence Variable
  sender = WikiEvent2018.first100k$user, # The Sender Variable
  receiver = WikiEvent2018.first100k$article, # The Receiver Variable
  p_samplingobserved = 0.01, # The Probability of Selection
  n_controls = 10, # The Number of Controls to Sample from the Full Risk Set
  seed = 9999) # The Seed for Replication


### Creating A New EventSet with more observed events and less control events
### Sampling from the Observed Event Sequence with P = 0.02
### Employing Case-Control Sampling With M = 2
EventSet1 <- processTMEventSeq(
  data = WikiEvent2018.first100k, # The Event Dataset
  time = WikiEvent2018.first100k$time, # The Time Variable
  eventID = WikiEvent2018.first100k$eventID, # The Event Sequence Variable
  sender = WikiEvent2018.first100k$user, # The Sender Variable
  receiver = WikiEvent2018.first100k$article, # The Receiver Variable
  p_samplingobserved = 0.02, # The Probability of Selection
  n_controls = 2, # The Number of Controls to Sample from the Full Risk Set
  seed = 9999) # The Seed for Replication

### Creating An Event-Dependent EventSet with P = 0.001 and m = 5 with
### where only actors involved in the past 20 events are involved in the
### creation of the risk set.
event_dependent <- processTMEventSeq(
 data = WikiEvent2018.first100k,
 time = WikiEvent2018.first100k$time,
 sender = WikiEvent2018.first100k$user,
 receiver = WikiEvent2018.first100k$article,
```

```
 eventID = WikiEvent2018.first100k$eventID,
 p_samplingobserved = 0.001,
 n_controls = 5,
 time_dependent = TRUE,
 timeDV = 1:nrow(WikiEvent2018.first100k),
 timeDif = 20, #20 past events
 seed = 9999)
### Creating An Time-Dependent EventSet with P = 0.001 and m = 5 with
### where only actors involved in the past 30 days are involved in the
### creation of the risk set.
timeSinceStart <- WikiEvent2018.first100k$time-WikiEvent2018.first100k$time[1]
timeDifMonth <- 30*24*60*60*1000
timedependent <- processTMEventSeq(
 data = WikiEvent2018.first100k,
 time = WikiEvent2018.first100k$time,
 sender = WikiEvent2018.first100k$user,
 receiver = WikiEvent2018.first100k$article,
 eventID = WikiEvent2018.first100k$eventID,
 p_samplingobserved = 0.001,
 n_controls = 5,
 time_dependent = TRUE,
 timeDV = timeSinceStart,
 timeDif = timeDifMonth,
 seed = 9999)
```

`remExpWeights` *Helper Function to Compute Minimum Effective Time and Exponential Weights for REM Statistics*

Description

A helper function for computing exponential decay weights and the corresponding minimum effective time used to calculate network statistics in relational event models within the **dream** package. This implementation follows the formulations of Lerner et al. (2013) and Lerner & Lomi (2020). Although primarily designed for internal use (e.g., within `computeReciprocity`), it may also be of interest to users working directly with REM statistics (e.g., creating new statistics).

Usage

```
remExpWeights(
  current,
  past = NULL,
  halflife,
  dyadic_weight,
  Lerneretal_2013 = FALSE,
  exp.weights = TRUE
)
```

Arguments

`current`	The current relational event time.
`past`	The numeric vector of past event times (for exponential weighting only).
`halflife`	The halflife parameter for exponential weighting.
`dyadic_weight`	The dyadic (event) weight cutoff for relational relevancy.
`Lerneretal_2013`	TRUE/FALSE. If TRUE, the function uses the Lerner et al. (2013) exponential weighting function. If FALSE, the function uses the Lerner and Lomi (2020) exponential weighting function.
`exp.weights`	TRUE/FALSE. If TRUE, the function computes the exponential weights for past relational events. If FALSE, the function computes the minimum effective time for a relational event (that is, the minimum past time that would result in a 0 value for an exponential weight).

Details

- **Exponential Weighting Function**:
  - Lerner & Lomi (2020): $w(u,a,t) = \sum \exp(-(t-t') * (\log(2)/T_{1/2}))$
  - Lerner et al. (2013): $w(u,a,t) = \sum \exp(-(t-t') * (\log(2)/T_{1/2})) * (\log(2)/T_{1/2})$
- **Minimum Effective Time (MEF)**:
  - Lerner & Lomi (2020): $MEF = t + \log(w)/(\log(2)/T_{1/2})$
  - Lerner et al. (2013): $MEF = t + [T_{1/2} * \log((w * T_{1/2})/\log(2))]/\log(2)$

Value

When exp.weights = TRUE, the numeric vector of exponential decay weights. When exp.weights = FALSE, the scalar for the minimum event cut-off time.

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Lerner, Jürgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: How to fit a relational event model to 360 million dyadic events." *Network Science* 8(1): 97-135.

Lerner, Jürgen, Margit Bussman, Tom A.B. Snijders, and Ulrik Brandes. 2013. " Modeling Frequency and Type of Interaction in Event Networks." *The Corvinus Journal of Sociology and Social Policy* 4(1): 3-32.

simulateRESeq *Simulate a Random One-Mode Relational Event Sequence*

Description

The function allows users to simulate a random one-mode relational event sequence between *n* actors for *k* events. Importantly, this function follows the methods discussed in Butts (2008), Amati, Lomi, and Snijders (2024), and Scheter and Quintane (2021). See the details for more information on this algorithm. Critically, this function can be used to simulate a random event sequence, to assess the goodness of fit for ordinal timing relational event models (see Amati, Lomi, and Snijders 2024), and simulate random outcomes for relational outcome models.

Usage

```
simulateRESeq(
  n_actors,
  n_events,
  inertia = FALSE,
  inertia_p = 0,
  recip = FALSE,
  recip_p = 0,
  sender_outdegree = FALSE,
  sender_outdegree_p = 0,
  sender_indegree = FALSE,
  sender_indegree_p = 0,
  target_outdegree = FALSE,
  target_outdegree_p = 0,
  target_indegree = FALSE,
  target_indegree_p = 0,
  assort = FALSE,
  assort_p = 0,
  trans_trips = FALSE,
  trans_trips_p = 0,
  three_cycles = FALSE,
  three_cycles_p = 0,
  starting_events = NULL,
  returnStats = FALSE
)
```

Arguments

n_actors	The number of potential actors in the event sequence.
n_events	The number of simulated events for the relational event sequence.
inertia	TRUE/FALSE. True indicates the effect will be included (see the details section). FALSE indicates the effect will not be included.

`inertia_p` If *inertia* = TRUE, the numerical value that corresponds to the parameter weight for the inertia statistic.

`recip` TRUE/FALSE. True indicates the effect will be included (see the details section). FALSE indicates the effect will not be included.

`recip_p` If *recip* = TRUE, the numerical value that corresponds to the parameter weight for the reciprocity statistic.

`sender_outdegree` TRUE/FALSE. True indicates the effect will be included (see the details section). FALSE indicates the effect will not be included.

`sender_outdegree_p` If *sender_outdegree* = TRUE, the numerical value that corresponds to the parameter weight for the outdegree statistic.

`sender_indegree` TRUE/FALSE. True indicates the effect will be included (see the details section). FALSE indicates the effect will not be included.

`sender_indegree_p` If *sender_indegree* = TRUE, the numerical value that corresponds to the parameter weight for the indegree statistic.

`target_outdegree` TRUE/FALSE. True indicates the effect will be included (see the details section). FALSE indicates the effect will not be included.

`target_outdegree_p` If *target_outdegree* = TRUE, the numerical value that corresponds to the parameter weight for the outdegree statistic.

`target_indegree` TRUE/FALSE. True indicates the effect will be included (see the details section). FALSE indicates the effect will not be included.

`target_indegree_p` If *target_indegree* = TRUE, the numerical value that corresponds to the parameter weight for the indegree statistic.

`assort` Boolean. TRUE/FALSE. True indicates the effect will be included (see the details section). FALSE indicates the effect will not be included.

`assort_p` If *assort* = TRUE, the numerical value that corresponds to the parameter weight for the assortativity statistic.

`trans_trips` TRUE/FALSE. True indicates the effect will be included (see the details section). FALSE indicates the effect will not be included.

`trans_trips_p` If *trans_trips* = TRUE, the numerical value that corresponds to the parameter weight for the transitive triplets statistic.

`three_cycles` TRUE/FALSE. True indicates the effect will be included (see the details section). FALSE indicates the effect will not be included.

`three_cycles_p` If *three_cycles* = TRUE, the numerical value that corresponds to the parameter weight for the three cycles statistic.

`starting_events` A *n* x 2 dataframe with *n* starting events and 2 columns. The first column should be the sender and the second should be the target.

`returnStats` TRUE/FALSE. TRUE indicates that the requested network statistics will be returned alongside the simulated relational event sequence. FALSE indicates that only the simulated relational event sequence will be returned. Set to FALSE by default.

Details

Following the authors listed in the descriptions section, the probability of selecting a new event for *t+1* based on the past relational history, H_t, from $0 < t < t + 1$ is given by:

$$p(e_t) = \frac{\lambda ij(t;\theta)}{\sum_{(u,v)\in R_t} \lambda_{uv}(t;\theta)}$$

where *(i,j,t)* is the triplet that corresponds to the dyadic pair with sender *i* and target *j* at time *t* contained in the full risk set, R_t, based on the past relational history. $\lambda_{ij}(t;\theta)$ is formulated as:

$$\lambda_{ij}(t;\theta) = e^{\sum_p \theta_p X_{ijp}(H_t)}$$

where θ_p corresponds to the specific parameter weight given by the user, and X_{ijp} represents the value of the specific statistic based on the current past relational history H_t.

Following Scheter and Quintane (2021) and Amati, Lomi, and Snijders (2024), the algorithm for simulating the random relational sequence for *k* events is:

- 1. Initialize the full risk set, R_t, which is the full Cartesian plot of actors.
- 2. Randomly sample the first event e_1 and add that event into the relational history, H_t.
- 3. Until *i* = *k*, compute the sufficient statistics for each event in the risk set, sample a new event e_i based on the probability function specified above, and add that element into the relational history.
- 4. End when *i* > *k*.

Currently, the function supports 6 statistics for one-mode networks. These are:

- Inertia: n_{ijt}
- Reciprocity: n_{jit}
- Target Indegree: $\sum_k n_{kjt}$
- Target Outdegree: $\sum_k n_{jkt}$
- Sender Outdegree: $\sum_k n_{ikt}$
- Sender Indegree: $\sum_k n_{kit}$
- Assortativity: $\sum_k n_{kit} \cdot \sum_k n_{ikt}$
- Transitive Triplets: $\sum_k n_{ikt} \cdot n_{kjt}$
- Three Cycles: $\sum_k n_{jkt} \cdot n_{kit}$

Where *n* represents the counts of past events, *i* is the event sender, and *j* is the event target. See Scheter and Quintane (2021) and Butts (2008) for a further discussion of these statistics.

Users are allowed to insert a starting event sequence to base the simulation on. A few things are worth nothing. The starting event sequence should be a matrix with *n* rows indicating the number of

starting events and 2 columns, with the first representing the event senders and the second column representing the event targets. Internally, the number of actors is ignored, as the number of possible actors in the risk set is based only on the actors present in the starting event sequence. Finally, the sender and target actor IDs should be numerical values.

Value

A data frame that contains the simulated relational event sequence with the sufficient statistics (if requested).

Author(s)

Kevin A. Carson kacarson@arizona.edu, Diego F. Leal dflc@arizona.edu

References

Amati, Viviana, Alessandro Lomi, and Tom A.B. Snijders. 2024. "A goodness of fit framework for relational event models." *Journal of the Royal Statistical Society Series A: Statistics in Society* 187(4): 967-988.

Butts, Carter T. "A Relational Framework for Social Action." *Sociological Methodology* 38: 155-200.

Schecter, Aaron and Eric Quintane. 2021 "The Power, Accuracy, and Precision of the Relational Event Model." *Organizational Research Methods* 24(4): 802-829.

Examples

```
#Creating a random relational sequence with 5 actors and 25 events
rem1<- simulateRESeq(n_actors = 25,
                     n_events = 1000,
                     inertia = TRUE,
                     inertia_p = 0.12,
                     recip = TRUE,
                     recip_p = 0.08,
                     sender_outdegree = TRUE,
                     sender_outdegree_p = 0.09,
                     target_indegree = TRUE,
                     target_indegree_p = 0.05,
                     assort = TRUE,
                     assort_p = -0.01,
                     trans_trips = TRUE,
                     trans_trips_p = 0.09,
                     three_cycles = TRUE,
                     three_cycles_p = 0.04,
                     starting_events = NULL,
                     returnStats = TRUE)
rem1

#Creating a random relational sequence with 100 actors and 1000 events with
#only inertia and reciprocity
rem2 <- simulateRESeq(n_actors = 100,
                     n_events = 1000,
```

```
                      inertia = TRUE,
                      inertia_p = 0.12,
                      recip = TRUE,
                      recip_p = 0.08,
                      returnStats = TRUE)
rem2

#Creating a random relational sequence based on the starting sequence with
#only inertia and reciprocity
rem3 <- simulateRESeq(n_actors = 100, #does not matter can be any value, this is
                                    #overridden by the starting event sequence
                    n_events = 100,
                    inertia = TRUE,
                    inertia_p = 0.12,
                    recip = TRUE,
                    recip_p = 0.08,
                    #a random starting event sequence
                    starting_events = matrix(c(1:10, 10:1),
                    nrow = 10, ncol = 2,  byrow = FALSE),
                    returnStats = TRUE)
rem3
```

`southern.women` *Davis Southern Women's Dataset*

Description

Davis Southern Women's Dataset

Usage

```
data(southern.women)
```

Format

`southern.women`**:**

Two-Mode Affliation Matrix from Davis et al.(1941) Southern Women study. 18 women x 14 events. Dataset is taken from the networkdata R package (Almquist)

Source

Almquist ZW (2014). *networkdata: Lin Freeman's Network Data Collection*. R package version 0.01, https://github.com/Z-co/networkdata.

Brieger, Ronald. 1974. "Duality of Persons and Groups." *Social Forces* 53(2): 181-190.

Davis, Allison, Burleigh B. Gardner, and Mary R. Gardner. 1941. *Deep South: A Social Anthropological Study of Caste and Class*. University of Chicago Press.

summary.dream *Summary Method for dream Objects*

Description

Summarizes the results of an ordinal timing relational event model.

Usage

```
## S3 method for class 'dream'
summary(object, digits = 4, ...)
```

Arguments

object	An object of class "dream".
digits	The number of digits to print after the decimal point.
...	Additional arguments (currently unused).

Value

A list of summary statistics for the relational event model including parameter estimates, (null) likelihoods, and tests of significance for likelihood ratios and estimated parameters.

WikiEvent2018.first100k *Wikipedia Edit Event Sequence 2018*

Description

The first 100,000 events of the Wikipedia edit event sequence, where an event is described as a Wikipedia user editing a Wikipedia article. The user column represents the unique event senders, the article column represents the unique event recievers (targets), and the time variable is in milliseconds.

Usage

```
data(WikiEvent2018.first100k)
```

Format

`WikiEvent2018.first100k`**:**

The first 100,000 events of the Wikipedia edit event sequence, where an event is described as a Wikipedia user editing a Wikipedia article. The user column represents the unique event senders, the article column represents the unique event recievers (targets), and the time variable is in milliseconds.

user the column that represents the unique event senders

article the article column represents the unique event recievers

time the event time variable in milliseconds

eventID the numerical id for each event in the event sequence

Source

https://zenodo.org/records/1626323

Lerner, Jurgen and Alessandro Lomi. 2020. "Reliability of relational event model estimates under sampling: how to fit a relational event model to 360 million dyadic events." *Network Science* 8(1):97-135. (DOI: https://doi.org/10.1017/nws.2019.57)

Index