Package 'redr'

June 12, 2025

Title Relational and Event Dynamics in R

Version 0.0.0.9000

Author Kevin Carson [aut, cre], Diego Leal [aut]

Maintainer Kevin Carson <kacarson@arizona.edu>

Description
A series of tools for relational and event analysis, including two- and one-mode network brokerage and structural measures, and helper functions for large relational event analysis, including creating relational risk sets, computing sufficient statistics, and simulating relational event sequences.

License MIT + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

Depends R (>= 2.10)

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

Contents

`minimum_effective_time` *Compute the Minimum Effective Time for Exponential Weighting in Relational Event Models*

Description

Compute the Minimum Effective Time for Exponential Weighting in Relational Event Models

Usage

```
minimum_effective_time(
  eventtime,
  dyadicweight,
  halflife,
  Lerneretal_2013 = FALSE
)
```

Arguments

`eventtime`	The current event time.
`dyadicweight`	The dyadic (event) weight cutoff for relational relevancy.
`halflife`	The halflife parameter for recency weighting.
`Lerneretal_2013`	Boolean indicating if the weighting function of Lerner et al. 2013 should be used.

Value

The minimum cut-off time for relational relevancy

om.constraint *Compute Burt's (1992) Constraint for Ego Networks*

Description

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

Usage

```
om.constraint(
  net,
  inParallel = FALSE,
  nCores = NULL,
  isolates = NA,
  pendants = 1
)
```

Arguments

net	A sociomatrix with network nominations
inParallel	Boolean. TRUE indicates that the values should be computed in parallel (see the foreach package). FALSE does not use parallel processing.
nCores	The number of computing cores for parallel processing. If this value is not specified then the function internally provides it.
isolates	What value should isolates be given? Preset to be NA.
pendants	What value should be given to pendant vertices? Preset to 1

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 outgoing edges) and pendants (i.e., nodes who only have one edge), 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)
om.constraint(BurtEgoNet)
```

`om.effective` *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

```
om.effective(
  net,
  inParallel = FALSE,
  nCores = NULL,
  isolates = NA,
  pendants = 1
)
```

Arguments

net	A sociomatrix with network nominations
inParallel	Boolean. TRUE indicates that the values should be computed in parallel (see the foreach package). FALSE does not use parallel processing.
nCores	The number of computing cores for parallel processing. If this value is not specified then the function internally provides it.
isolates	What value should isolates be given? Preset to be NA.
pendants	What value should be given to pendant vertices? Preset to 1

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 outgoing edges) and pendants (i.e., nodes who only have one edge), 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,
   0,0,0,0,0,0,0,1,0,1,
   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)
om.effective(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)
om.effective(BurtEgoNet)
```

---

om.npaths *Compute the Number of Paths of Length K*

Description

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

Usage

```
om.npaths(net, k)
```

Arguments

net	A one-mode network adjacency matrix
k	A scalar that corresponds to the length of the paths

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
om.npaths(rnet, k = 2)
#counting the paths of length 5
om.npaths(rnet, k = 5)
```

---

om.pi.brokerage *Compute Interpersonal Cultural Brokerage Based on Leal (2025)*

Description

Following Leal (2025), this function calculates the values for interpersonal cultural brokerage for a one-mode network. For example, users can examine the potential for cultural brokerage across gender. The option *count* determines what is returned by the function. If count is true, then the counts of opportunities for cultural brokerage is included, whereas, if count is false, the proportion of culturally brokered open triangles is returned, defined as the number of culturally brokered open triangles one is brokering divided by the total number of open triangles the actor is brokering. In this case, the proportion indicates how much of the brokerage opportunites is based on out-group brokerage vs. in-group brokerage. The formula for computing interpersonal brokerage is presented in the details section.

Usage

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

Arguments

net	A one-mode adjacency matrix
g.mem	A vector of membership values
symmetric	Boolean indicating if the network matrix should be treated as symmetric

`triad.type`	Vector or scalar indicating if the the potential for cultural brokerage should only be computed for specific triadic (star) structures. If this is not specified, it defaults to "ANY". Possible values: c("ANY", "OTS", "ITS", "MTS"). See details.
`count`	Boolean indicating if the number of culturally brokered open triangles should be returned, or if the proportion of culturally brokered open triangles to all open triangles should be returned. (see the description)
`isolate`	If count = FALSE, then what value should isolates be given? The function defaults to make the value NA, since 0/0 is undefined, however, the user can specify this value!

Details

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 undirected 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 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 according to the M.A.N. notation; see Wasserman and Faust 1994), where j is the broker. "MTS" computes the mixed formations (i<-j<-k or i->j->k or the 021C 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 compute all of the formations at once.

Value

The vector of interpersonal cultural brokerage values.

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* https://doi.org/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)
om.pi.brokerage(LealNet,
   symmetric = TRUE,
   g.mem = categorical_variable)
```

`rem.riskset.om` *Create Dynamic One-Mode Risk Sets for Relational Event Sequences*

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 month.

Usage

```
rem.riskset.om(
  data,
  time,
  eventID,
  sender,
  receiver,
  p_samplingobserved,
  n_controls,
  time_dependent = FALSE,
  timeDV = NULL,
  timeDif = NULL,
  seed = 9999
)
```

Arguments

`data`	The full event data sequence.
`time`	The vector of 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`	A scalar for the probability of selection for case control sampling.
`n_controls`	A scalar for the number of null event controls for each (sampled) observed event.
`time_dependent`	Boolean. Should a time- or event-dependent dynamic risk set be created? TRUE = yes, FALSE = no.
`timeDV`	A vector of time values that corresponds to the sliding windows risk set (see details).
`timeDif`	A scalar value for the time difference (see details).
`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-dependent risk set can be created where the set of potential events, that is all events in the risk set, A_t, is based only on the set of actors active in a specified time window (e.g., such as the past

year). The specified time window can be based upon either a) a specified time window, in which the window is based upon the actual timing of the past events (e.g., such as all events that occurred in the past month) or b) a specified event number window, in which the window is based upon the event ordering of the past events (e.g., such as all actors involved in the past 10 events). If this type of risk set is desired, the user should set time_dependent to *TRUE*, and then specify the time vector *timeDV*. For instance, *timeDV* can be the number of minutes, hours, days, weeks, months, or events since the first event. Moreover, the user should specify the cutoff threshold with the *timeDif* value that corresponds directly to the measurement unit of *timeDV* (e.g., number of hours). For example, let's say you wanted to create a temporally dependent risk set that only includes events active within the past year, then you could create a vector of values *timeDV*, that is, the number of days for each event since the first time point, and then specify *timeDif* to 365. However, let's say you wanted to create a event dependent risk set that only includes the past 100 events, then you could 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 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
events <- data.table::data.table(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 a one-mode relational risk set with p = 1.00 (all true events) and 5 controls
eventSet <- rem.riskset.om(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$timeseq <- 1:nrow(events)
eventSetT <- rem.riskset.om(data = events,
                       time = events$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)
```

`rem.riskset.tm` *Create Dynamic Two-Mode Risk Sets for Relational Event Sequences*

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.

Usage

```
rem.riskset.tm(
  data,
  time,
  eventID,
  sender,
```

```
  receiver,
  p_samplingobserved,
  n_controls,
  time_dependent = FALSE,
  timeDV = NULL,
  timeDif = NULL,
  seed = 9999
)
```

Arguments

data	The full event data sequence.
time	The vector of 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	A scalar for the probability of selection for case control sampling.
n_controls	A scalar for the number of null event controls for each (sampled) observed event.
time_dependent	Boolean. Should a time- or event-dependent dynamic risk set be created? TRUE = yes, FALSE = no.
timeDV	A vector of time values that corresponds to the sliding windows risk set (see details).
timeDif	A scalar value for the time difference (see details).
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-dependent risk set can be created where the set of potential events, that is all events in the risk set, A_t, is based only on the set of actors active in a specified time window (e.g., such as the past year). The specified time window can be based upon either a) a specified time window, in which the window is based upon the actual timing of the past events (e.g., such as all events that occurred in the past month) or b) a specified event number window, in which the window is based upon the event ordering of the past events (e.g., such as all actors involved in the past 10 events). If this type of risk set is desired, the user should set time_dependent to *TRUE*, and then specify the time vector *timeDV*. For instance, *timeDV* can be the number of minutes, hours, days, weeks, months, or events since the first event. Moreover, the user should specify the cutoff threshold with the *timeDif* value that corresponds directly to the measurement unit of *timeDV* (e.g., number of hours). For example, let's say you wanted to create a temporally dependent risk set that only includes events active within the past year, then you could create a vector of values *timeDV*, that is, the number of days for each event since the first time point, and then specify *timeDif* to 365. However, let's say you wanted to create a event dependent risk set that only includes the past 100 events, then you could 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 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 <- rem.riskset.tm(
  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 = 0.10
### Employing Case-Control Sampling With M = 2
EventSet1 <- rem.riskset.tm(
  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
```

---

rem.simulate *Simulate A One-Mode Random 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

```
rem.simulate(
  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 requested.
inertia	Boolean. Should inertia be computed? True indicates the effect should be included.
inertia_p	If inertia is included, the scalar value that corresponds to the parameter weight for the inertia statistic.
recip	Boolean. Should reciprocity be computed? True indicates the effect should be included.
recip_p	If recip is included, the scalar value that corresponds to the parameter weight for the reciprocity statistic.
sender_outdegree	Boolean. Should outdegree be computed? True indicates the effect should be included.
sender_outdegree_p	If sender outdegree is included, the scalar value that corresponds to the parameter weight for the outdegree statistic.
sender_indegree	Boolean. Should indegree be computed? True indicates the effect should be included.
sender_indegree_p	If sender indegree is included, the scalar value that corresponds to the parameter weight for the indegree statistic.
target_outdegree	Boolean. Should outdegree be computed? True indicates the effect should be included.
target_outdegree_p	If target outdegree is included, the scalar value that corresponds to the parameter weight for the outdegree statistic.
target_indegree	Boolean. Should indegree be computed? True indicates the effect should be included.

`target_indegree_p`
: If target indegree is included, the scalar value that corresponds to the parameter weight for the indegree statistic.

`assort`
: Boolean. Should assortativity be computed? True indicates the effect should be included.

`assort_p`
: If assortativity is included, the scalar value that corresponds to the parameter weight for the assortativity statistic.

`trans_trips`
: Boolean. Should transitive triplets be computed? True indicates the effect should be included.

`trans_trips_p`
: If transitive triplets is included, the scalar value that corresponds to the parameter weight for the transitive triplets statistic.

`three_cycles`
: Boolean. Should three cycles be computed? True indicates the effect should be included.

`three_cycles_p`
: If three cycles is included, the scalar 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`
: Boolean. If true the sufficient statistics for each requested effect will be returned, if false, only the event sequence will be returned.

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<- rem.simulate(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 <- rem.simulate(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 <- rem.simulate(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
```

`rem.stat.cycle4` *Compute The Four-Cycles Statistic for Two-Mode Relational Event Sequences*

Description

The function computes the four-cycles network sufficient statistic for a two-mode relational sequence with the exponential weighting function (Lerner and Lomi 2021). 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

```
rem.stat.cycle4(
  observed_time,
  observed_sender,
  observed_receiver,
  processed_time,
  processed_sender,
  processed_receiver,
  sliding_windows = FALSE,
```

```
  processed_seqIDs = NULL,
  counts = FALSE,
  halflife,
  dyadic_weight,
  window_size = NA,
  Lerneretal_2013 = FALSE,
  priorStats = FALSE,
  sender_OutDeg = NULL,
  receiver_InDeg = NULL
)
```

Arguments

observed_time	A vector of values that contains the time values from the full event dataset (i.e., the real events).
observed_sender	A vector of contains the event senders from the full event dataset (i.e., the real events).
observed_receiver	A vector of contains the event receivers from the full event dataset (i.e., the real events).
processed_time	A vector of values that contains the time values from the sampled event dataset (i.e., the dataset with real and null events).
processed_sender	A vector of values that contains the sender values from the sampled event dataset (i.e., the dataset with real and null events).
processed_receiver	A vector of values that contains the receiver values from the sampled event dataset (i.e., the dataset with real and null events).
sliding_windows	Boolean indicating if the sliding windows framework should be used.
processed_seqIDs	A vector of values that contains the event sequence IDs from the sampled event dataset for the sliding windows method. TRUE = yes; FALSE = no.
counts	Boolean indicating if the raw counts of events should be returned or the exponential weighting function should be used (TRUE = counts; FALSE = exponential weighting).
halflife	The halflife value for the weighting function.
dyadic_weight	The dyadic cutoff weight for events that no longer matter.
window_size	The sizes of the windows that ared used for the sliding windows computational framework, if NA, the function internally divides the dataset into ten slices.
Lerneretal_2013	Boolean indicating if the weighting function of Lerner et al. 2013 should be used (see the detials section).
priorStats	Boolean indicating if the user already pre-computed receiver indegree and sender outdegree measures. TRUE = yes; FALSE = no.
sender_OutDeg	The vector that contains the previously computed sender outdegree scores.
receiver_InDeg	The vector that contains the 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:

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

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 occured 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 of this), therefore this function allows the user to input previously computed target indegree and sender outdegree scores to reduce the runtime. In particular, 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 `rem.stat.dyadCut` 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 be used in tandem with 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

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[1:10000,] #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 <- rem.riskset.tm(
 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 the Four-Cycle Statistic Without the Sliding Windows Framework
EventSet$fourcycle <- rem.stat.cycle4(
   observed_time = WikiEvent2018$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$cycle4SW <- rem.stat.cycle4(
   observed_time = WikiEvent2018$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$cycle4C <- rem.stat.cycle4(
   observed_time = WikiEvent2018$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)
```

`rem.stat.dyadCut` *Helper Function to Assist Researchers 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

```
rem.stat.dyadCut(halflife, timeValue, timeWidth, Lerneretal_2013 = FALSE)
```

Arguments

`halflife`	The user specified halflife value for the weighting function.
`timeValue`	The value that corresponds to a time value within an event dataset.
`timeWidth`	The value that corresponds to the time range for which the user specifies for temporal relevancy.
`Lerneretal_2013`	Boolean indicating if the weighting function of Lerner et al. 2013 should be used.

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 the dyadic cutoff value, that is, the minimum dyadic weight value that would be relevant, based upon the user inputted half-life, time value, and time range. The time value and time range must be in the same unit measurement. If not, *the function will not return the correct answer*. For example, let's say that the user pre-defines the half-life to be 30 days, the event time vector is in the number of days since the first event, and the user wants to find the dyadic cut off value that corresponds to a 60-day range (that is, events that occurred more than 60 days in the past are not relationally relevant). The user would then specify: half-life = 30, timeValue = 100 (this is arbitrary), timeWidth = 60.

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
rem.stat.dyadCut(halflife = 30,
                 timeValue = 100,
                 timeWidth = 60,
                 Lerneretal_2013 = TRUE)

# without the Lerner et al. 2013 weighting function
rem.stat.dyadCut(halflife = 30,
```

```
                  timeValue = 100,
                  timeWidth = 60,
                  Lerneretal_2013 = FALSE)

# A result to test the function (should come out to 0.50)
rem.stat.dyadCut(halflife = 30,
                  timeValue = 100,
                  timeWidth = 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 are in milliseconds, we have to change all measurements into milliseconds
rem.stat.dyadCut(halflife = (30*24*60*60*1000), #30 days in milliseconds
                #the first value in the Lerner and Lomi (2020) WikiEvent 2018 dataset
                timeValue = 979686793000,
                timeWidth = (6.64*30*24*60*60*1000), #Based upon the paper
                #using the Lerner and Lomi (2020) weighting function
                Lerneretal_2013 = FALSE)
```

`rem.stat.indegreeR` *Compute Indegree Statistic for Event Receivers in Relational Event Sequences*

Description

The function computes the indegree network sufficient statistic for event receivers in a relational event sequence (see Lerner and Lomi 2021; 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 2021; 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

```
rem.stat.indegreeR(
  observed_time,
  observed_receiver,
  processed_time,
  processed_receiver,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  counts = FALSE,
  halflife,
```

```
  dyadic_weight,
  window_size = NA,
  Lerneretal_2013 = FALSE
)
```

Arguments

observed_time	A vector of values that contains the time values from the full event dataset (i.e., the real events).
observed_receiver	A vector of contains the event receivers from the full event dataset (i.e., the real events).
processed_time	A vector of values that contains the time values from the sampled event dataset (i.e., the dataset with real and null events).
processed_receiver	A vector of values that contains the receiver values from the sampled event dataset (i.e., the dataset with real and null events).
sliding_windows	Boolean indicating if the sliding windows framework should be used.
processed_seqIDs	A vector of values that contains the event sequence IDs from the sampled event dataset for the sliding windows method. TRUE = yes; FALSE = no.
counts	Boolean indicating if the raw counts of events should be returned or the exponential weighting function should be used (TRUE = counts; FALSE = exponential weighting).
halflife	The halflife value for the weighting function.
dyadic_weight	The dyadic cutoff weight for events that no longer matter.
window_size	The sizes of the windows that ared used for the sliding windows computational framework, if NA, the function internally divides the dataset into ten slices.
Lerneretal_2013	Boolean indicating if the weighting function of Lerner et al. 2013 should be used (see the detials section).

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 reciever 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 reciever.

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 `rem.stat.dyadCut` function.

Following Butts (2008), if the counts of the past events are requested, the formula for target 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.table::data.table(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 <- rem.riskset.om(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$target_indegree <- rem.stat.indegreeR(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   processed_time = eventSet$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$target_indegreeSW <- rem.stat.indegreeR(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   processed_time = eventSet$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$target_indegreeC <- rem.stat.indegreeR(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   processed_time = eventSet$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)
```

`rem.stat.indegreeS` *Compute Indegree Statistic for Event Senders in Relational Event Sequences*

Description

The function computes the indegree network sufficient statistic for event senders in a relational event sequence (see Lerner and Lomi 2021; 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 2021; 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

```
rem.stat.indegreeS(
  observed_time,
  observed_sender,
  observed_receiver,
  processed_time,
  processed_sender,
  processed_receiver,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  counts = FALSE,
  halflife,
  dyadic_weight,
  window_size = NA,
  Lerneretal_2013 = FALSE
)
```

Arguments

`observed_time`	A vector of values that contains the time values from the full event dataset (i.e., the real events).
`observed_sender`	A vector of contains the event senders from the full event dataset (i.e., the real events).
`observed_receiver`	A vector of contains the event receivers from the full event dataset (i.e., the real events).
`processed_time`	A vector of values that contains the time values from the sampled event dataset (i.e., the dataset with real and null events).
`processed_sender`	A vector of values that contains the sender values from the sampled event dataset (i.e., the dataset with real and null events).
`processed_receiver`	A vector of values that contains the receiver values from the sampled event dataset (i.e., the dataset with real and null events).

`sliding_windows`
: Boolean indicating if the sliding windows framework should be used.

`processed_seqIDs`
: A vector of values that contains the event sequence IDs from the sampled event dataset for the sliding windows method. TRUE = yes; FALSE = no.

`counts`
: Boolean indicating if the raw counts of events should be returned or the exponential weighting function should be used (TRUE = counts; FALSE = exponential weighting).

`halflife`
: The halflife value for the weighting function.

`dyadic_weight`
: The dyadic cutoff weight for events that no longer matter.

`window_size`
: The sizes of the windows that ared used for the sliding windows computational framework, if NA, the function internally divides the dataset into ten slices.

`Lerneretal_2013`
: Boolean indicating if the weighting function of Lerner et al. 2013 should be used (see the detials section).

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 `rem.stat.dyadCut` 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.table::data.table(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 <- rem.riskset.om(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$sender.indegree <- rem.stat.indegreeS(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   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$sender.indegree.SW <- rem.stat.indegreeS(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   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$sender.indegreeC <- rem.stat.indegreeS(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   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)
```

`rem.stat.ISP` *Compute Incoming Shared Partner Statistic for Relational Event Sequences*

Description

This function calculates the incoming shared partners (ISP) network sufficient statistic for a relational event sequence (see Lerner and Lomi 2021; Butts 2008). 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 2021; 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

```
rem.stat.ISP(
  observed_time,
  observed_sender,
  observed_receiver,
  processed_time,
  processed_sender,
  processed_receiver,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  counts = FALSE,
  halflife,
  dyadic_weight,
  window_size = NA,
  Lerneretal_2013 = FALSE
)
```

Arguments

observed_time	A vector of values that contains the time values from the full event dataset (i.e., the real events).
observed_sender	A vector of contains the event senders from the full event dataset (i.e., the real events).
observed_receiver	A vector of contains the event receivers from the full event dataset (i.e., the real events).
processed_time	A vector of values that contains the time values from the sampled event dataset (i.e., the dataset with real and null events).
processed_sender	A vector of values that contains the sender values from the sampled event dataset (i.e., the dataset with real and null events).
processed_receiver	A vector of values that contains the receiver values from the sampled event dataset (i.e., the dataset with real and null events).
sliding_windows	Boolean indicating if the sliding windows framework should be used.
processed_seqIDs	A vector of values that contains the event sequence IDs from the sampled event dataset for the sliding windows method. TRUE = yes; FALSE = no.
counts	Boolean indicating if the raw counts of events should be returned or the exponential weighting function should be used (TRUE = counts; FALSE = exponential weighting).
halflife	The halflife value for the weighting function.
dyadic_weight	The dyadic cutoff weight for events that no longer matter.
window_size	The sizes of the windows that ared used for the sliding windows computational framework, if NA, the function internally divides the dataset into ten slices.
Lerneretal_2013	Boolean indicating if the weighting function of Lerner et al. 2013 should be used (see the detials section).

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* endpoint.

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 `rem.stat.dyadCut` function.

Following Butts (2008), if the counts of the past events are requested, the formula for outgoing 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 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 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.table::data.table(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 <- rem.riskset.om(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$ISP <- rem.stat.ISP(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   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 <- rem.stat.ISP(
```

```
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   processed_time = eventSet$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 Reciprocity Statistics with the counts of events being returned
eventSet$ISPC <- rem.stat.ISP(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   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)
```

`rem.stat.ITP` *Compute Incoming Two Path Statistic for Relational Event Sequences*

Description

The function computes the incoming two path (ITP) network sufficient statistic for a relational event sequence (see Lerner and Lomi 2021; Butts 2008). 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 2021; 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

```
rem.stat.ITP(
  observed_time,
  observed_sender,
  observed_receiver,
  processed_time,
```

```
  processed_sender,
  processed_receiver,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  counts = FALSE,
  halflife,
  dyadic_weight,
  window_size = NA,
  Lerneretal_2013 = FALSE
)
```

Arguments

observed_time	A vector of values that contains the time values from the full event dataset (i.e., the real events).
observed_sender	A vector of contains the event senders from the full event dataset (i.e., the real events).
observed_receiver	A vector of contains the event receivers from the full event dataset (i.e., the real events).
processed_time	A vector of values that contains the time values from the sampled event dataset (i.e., the dataset with real and null events).
processed_sender	A vector of values that contains the sender values from the sampled event dataset (i.e., the dataset with real and null events).
processed_receiver	A vector of values that contains the receiver values from the sampled event dataset (i.e., the dataset with real and null events).
sliding_windows	Boolean indicating if the sliding windows framework should be used.
processed_seqIDs	A vector of values that contains the event sequence IDs from the sampled event dataset for the sliding windows method. TRUE = yes; FALSE = no.
counts	Boolean indicating if the raw counts of events should be returned or the exponential weighting function should be used (TRUE = counts; FALSE = exponential weighting).
halflife	The halflife value for the weighting function.
dyadic_weight	The dyadic cutoff weight for events that no longer matter.
window_size	The sizes of the windows that ared used for the sliding windows computational framework, if NA, the function internally divides the dataset into ten slices.
Lerneretal_2013	Boolean indicating if the weighting function of Lerner et al. 2013 should be used (see the detials section).

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. One way to estimate this value is to find what time frame you consider relationally relevant, such as a year, and then add that value to the time unit that is within your event dataset (days, seconds, hours) and then input those values into the above weighting equations and, finally, use that weight value.

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.table::data.table(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 <- rem.riskset.om(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$ITP <- rem.stat.ITP(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   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 <- rem.stat.ITP(
```

```
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   processed_time = eventSet$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 Reciprocity Statistics with the counts of events being returned
eventSet$ITPC <- rem.stat.ITP(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   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)
```

`rem.stat.OSP` *Compute Outgoing Shared Partner Statistic for Relational Event Sequences*

Description

The function computes the outgoing shared partners (OSP) network sufficient statistic for a relational event sequence (see Lerner and Lomi 2021; Butts 2008). 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 2021; 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

```
rem.stat.OSP(
  observed_time,
  observed_sender,
  observed_receiver,
```

```
  processed_time,
  processed_sender,
  processed_receiver,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  counts = FALSE,
  halflife,
  dyadic_weight,
  window_size = NA,
  Lerneretal_2013 = FALSE
)
```

Arguments

observed_time	A vector of values that contains the time values from the full event dataset (i.e., the real events).
observed_sender	A vector of contains the event senders from the full event dataset (i.e., the real events).
observed_receiver	A vector of contains the event receivers from the full event dataset (i.e., the real events).
processed_time	A vector of values that contains the time values from the sampled event dataset (i.e., the dataset with real and null events).
processed_sender	A vector of values that contains the sender values from the sampled event dataset (i.e., the dataset with real and null events).
processed_receiver	A vector of values that contains the receiver values from the sampled event dataset (i.e., the dataset with real and null events).
sliding_windows	Boolean indicating if the sliding windows framework should be used.
processed_seqIDs	A vector of values that contains the event sequence IDs from the sampled event dataset for the sliding windows method. TRUE = yes; FALSE = no.
counts	Boolean indicating if the raw counts of events should be returned or the exponential weighting function should be used (TRUE = counts; FALSE = exponential weighting).
halflife	The halflife value for the weighting function.
dyadic_weight	The dyadic cutoff weight for events that no longer matter.
window_size	The sizes of the windows that ared used for the sliding windows computational framework, if NA, the function internally divides the dataset into ten slices.
Lerneretal_2013	Boolean indicating if the weighting function of Lerner et al. 2013 should be used (see the detials section).

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* endpoint.

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 `rem.stat.dyadCut` 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.table::data.table(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 <- rem.riskset.om(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$OSP <- rem.stat.OSP(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   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$OSP_SW <- rem.stat.OSP(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   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$OSP_C <- rem.stat.OSP(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   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)
```

`rem.stat.OTP` *Compute Outgoing Two Path Statistic for Relational Event Sequences*

Description

The function computes the outgoing two paths (OTP) network sufficient statistic for a relational event sequence (see Lerner and Lomi 2021; Butts 2008). 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 2021; 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

```
rem.stat.OTP(
  observed_time,
  observed_sender,
  observed_receiver,
  processed_time,
  processed_sender,
  processed_receiver,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  counts = FALSE,
  halflife,
```

```
  dyadic_weight,
  window_size = NA,
  Lerneretal_2013 = FALSE
)
```

Arguments

observed_time	A vector of values that contains the time values from the full event dataset (i.e., the real events).
observed_sender	A vector of contains the event senders from the full event dataset (i.e., the real events).
observed_receiver	A vector of contains the event receivers from the full event dataset (i.e., the real events).
processed_time	A vector of values that contains the time values from the sampled event dataset (i.e., the dataset with real and null events).
processed_sender	A vector of values that contains the sender values from the sampled event dataset (i.e., the dataset with real and null events).
processed_receiver	A vector of values that contains the receiver values from the sampled event dataset (i.e., the dataset with real and null events).
sliding_windows	Boolean indicating if the sliding windows framework should be used.
processed_seqIDs	A vector of values that contains the event sequence IDs from the sampled event dataset for the sliding windows method. TRUE = yes; FALSE = no.
counts	Boolean indicating if the raw counts of events should be returned or the exponential weighting function should be used (TRUE = counts; FALSE = exponential weighting).
halflife	The halflife value for the weighting function.
dyadic_weight	The dyadic cutoff weight for events that no longer matter.
window_size	The sizes of the windows that ared used for the sliding windows computational framework, if NA, the function internally divides the dataset into ten slices.
Lerneretal_2013	Boolean indicating if the weighting function of Lerner et al. 2013 should be used (see the detials section).

Details

The function calculates the outgoing two paths 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 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 recieves a relational tie from the same *h* endpoint.

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 `rem.stat.dyadCut` 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 value 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.table::data.table(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 <- rem.riskset.om(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$OTP <- rem.stat.OTP(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   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$OTP_SW <- rem.stat.OTP(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   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 Reciprocity Statistics with the counts of events being returned
eventSet$OTPC <- rem.stat.OTP(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   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)
```

`rem.stat.outdegreeR` *Compute Outdegree Statistic for Event Receivers in Relational Event Sequences*

Description

The function computes the reciever outdegree network sufficient statistic for a relational event sequence (see Lerner and Lomi 2021; 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 2021; 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

```
rem.stat.outdegreeR(
  observed_time,
  observed_sender,
  observed_receiver,
  processed_time,
  processed_sender,
  processed_receiver,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  counts = FALSE,
  halflife,
  dyadic_weight,
  window_size = NA,
  Lerneretal_2013 = FALSE
)
```

Arguments

`observed_time`	A vector of values that contains the time values from the full event dataset (i.e., the real events).

`observed_sender`
: A vector of contains the event senders from the full event dataset (i.e., the real events).

`observed_receiver`
: A vector of contains the event receivers from the full event dataset (i.e., the real events).

`processed_time`
: A vector of values that contains the time values from the sampled event dataset (i.e., the dataset with real and null events).

`processed_sender`
: A vector of values that contains the sender values from the sampled event dataset (i.e., the dataset with real and null events).

`processed_receiver`
: A vector of values that contains the receiver values from the sampled event dataset (i.e., the dataset with real and null events).

`sliding_windows`
: Boolean indicating if the sliding windows framework should be used.

`processed_seqIDs`
: A vector of values that contains the event sequence IDs from the sampled event dataset for the sliding windows method. TRUE = yes; FALSE = no.

`counts`
: Boolean indicating if the raw counts of events should be returned or the exponential weighting function should be used (TRUE = counts; FALSE = exponential weighting).

`halflife`
: The halflife value for the weighting function.

`dyadic_weight`
: The dyadic cutoff weight for events that no longer matter.

`window_size`
: The sizes of the windows that ared used for the sliding windows computational framework, if NA, the function internally divides the dataset into ten slices.

`Lerneretal_2013`
: Boolean indicating if the weighting function of Lerner et al. 2013 should be used (see the details section).

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 `rem.stat.dyadCut` 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.table::data.table(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 <- rem.riskset.om(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$target_outdegree <- rem.stat.outdegreeR(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   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$target_outdegreeSW <- rem.stat.outdegreeR(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   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$target_outdegreeC <- rem.stat.outdegreeR(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   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)
```

`rem.stat.outdegreeS` *Compute Outdegree Statistic for Event Senders in Relational Event Sequences*

Description

The function computes the sender outdegree network sufficient statistic for a relational event sequence (see Lerner and Lomi 2021; 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 2021; 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

```
rem.stat.outdegreeS(
  observed_time,
  observed_sender,
  processed_time,
  processed_sender,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  counts = FALSE,
  halflife,
  dyadic_weight,
  window_size = NA,
  Lerneretal_2013 = FALSE
)
```

Arguments

`observed_time`	A vector of values that contains the time values from the full event dataset (i.e., the real events).
`observed_sender`	A vector of contains the event senders from the full event dataset (i.e., the real events).
`processed_time`	A vector of values that contains the time values from the sampled event dataset (i.e., the dataset with real and null events).
`processed_sender`	A vector of values that contains the sender values from the sampled event dataset (i.e., the dataset with real and null events).
`sliding_windows`	Boolean indicating if the sliding windows framework should be used.
`processed_seqIDs`	A vector of values that contains the event sequence IDs from the sampled event dataset for the sliding windows method. TRUE = yes; FALSE = no.

counts	Boolean indicating if the raw counts of events should be returned or the exponential weighting function should be used (TRUE = counts; FALSE = exponential weighting).
halflife	The halflife value for the weighting function.
dyadic_weight	The dyadic cutoff weight for events that no longer matter.
window_size	The sizes of the windows that ared used for the sliding windows computational framework, if NA, the function internally divides the dataset into ten slices.
Lerneretal_2013	Boolean indicating if the weighting function of Lerner et al. 2013 should be used (see the details section).

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 `rem.stat.dyadCut` 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.table::data.table(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 <- rem.riskset.om(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$sender_outdegree <- rem.stat.outdegreeS(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
   processed_time = eventSet$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$sender_outdegreeSW <- rem.stat.outdegreeS(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
```

```
   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$sender_outdegreeC <- rem.stat.outdegreeS(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
   processed_time = eventSet$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)
```

`rem.stat.pref.attach` *Compute the Preferential Attachment Statistic for Relational Event Sequences*

Description

The function computes the preferential attachment network sufficient statistic for a relational event sequence (see Butts 2008). 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

```
rem.stat.pref.attach(
  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	A vector of values that contains the time values from the full event dataset (i.e., the real events).
observed_sender	A vector of contains the event senders from the full event dataset (i.e., the real events).
observed_receiver	A vector of contains the event receivers from the full event dataset (i.e., the real events).
processed_time	A vector of values that contains the time values from the sampled event dataset (i.e., the dataset with real and null events).
processed_sender	A vector of values that contains the sender values from the sampled event dataset (i.e., the dataset with real and null events).
processed_receiver	A vector of values that contains the receiver values from the sampled event dataset (i.e., the dataset with real and null events).
dependency	Boolean indicating if relational relevancy should be considered when computing the network statistic.
relationalTimeSpan	A scalar that indicates the temporal span for relational relevancy (should be specified if dependency is set to TRUE). For instance, if event time is in number of events since the first starting event and the researcher wants the temporal time span to be 10 events, this this value should be specified to 10.
sliding_windows	Boolean indicating if the sliding windows framework should be used (TRUE = yes; FALSE = no).
processed_seqIDs	A vector of values that contains the event sequence IDs from the sampled event dataset for the sliding windows method.
window_size	The sizes of the windows that ared used for the sliding windows computational framework, if NA, the function internally divides the dataset into ten slices.

Details

The function calculates preferential attachment for a relational event sequence based on 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.table::data.table(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 <- rem.riskset.om(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$pref <- rem.stat.pref.attach(observed_time = events$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$prefSW <- rem.stat.pref.attach(observed_time = events$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$prefdep <- rem.stat.pref.attach(observed_time = events$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$pref1dep <- rem.stat.pref.attach(observed_time = events$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
```

`rem.stat.presistence` *Compute Persistence Statistic for Relational Event Sequences*

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

```
rem.stat.persistence(
  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	A vector of values that contains the time values from the full event dataset (i.e., the real events).
observed_sender	A vector of contains the event senders from the full event dataset (i.e., the real events).
observed_receiver	A vector of contains the event receivers from the full event dataset (i.e., the real events).
processed_time	A vector of values that contains the time values from the sampled event dataset (i.e., the dataset with real and null events).
processed_sender	A vector of values that contains the sender values from the sampled event dataset (i.e., the dataset with real and null events).
processed_receiver	A vector of values that contains the receiver values from the sampled event dataset (i.e., the dataset with real and null events).
sender	Boolean indicating if the presistence scores should be computed in reference to the sender's past relational history or the target's past relational history. TRUE = sender; FALSE = target.
dependency	Boolean indicating if relational relevancy should be considered when computing the network statistic.

`relationalTimeSpan`
: A scalar that indicates the temporal span for relational relevancy (should be specified if dependency is set to TRUE). For instance, if event time is in number of events since the first starting event and the researcher wants the temporal time span to be 10 events, this this value should be specified to 10.

`nopastEvents`
: What value should be given to event's where the sender has no sent any past ties (i's neighborhood) or has not received any past ties (j's neighborhood). Preset to NA.

`sliding_windows`
: Boolean indicating if the sliding windows framework should be used.

`processed_seqIDs`
: A vector of values that contains the event sequence IDs from the sampled event dataset for the sliding windows method.

`window_size`
: The sizes of the windows that ared used for the sliding windows computational framework, if NA, the function internally divides the dataset into ten slices.

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.table::data.table(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 <- rem.riskset.om(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$persist <- rem.stat.persistence(observed_time = events$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$persistSW <- rem.stat.persistence(observed_time = events$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$persistDep <- rem.stat.persistence(observed_time = events$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$persistT <- rem.stat.persistence(observed_time = events$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$persistSWT <- rem.stat.persistence(observed_time = events$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$persistDepT <- rem.stat.persistence(observed_time = events$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)
```

`rem.stat.recency` *Compute the Recency Statistic for Relational Event Sequences*

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). 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

```
rem.stat.recency(
  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`	A vector of values that contains the time values from the full event dataset (i.e., the real events).
`observed_sender`	A vector of contains the event senders from the full event dataset (i.e., the real events).
`observed_receiver`	A vector of contains the event receivers from the full event dataset (i.e., the real events).
`processed_time`	A vector of values that contains the time values from the sampled event dataset (i.e., the dataset with real and null events).

`processed_sender`
: A vector of values that contains the sender values from the sampled event dataset (i.e., the dataset with real and null events).

`processed_receiver`
: A vector of values that contains the receiver values from the sampled event dataset (i.e., the dataset with real and null events).

`type`
: Which measurement formula should be used for computing the recency statistic? The options are "raw.diff", "inv.diff.plus1", "rank.ordered.count". See the details section.

`i_neighborhood`
: Boolean indicating if recency should be computed on the sender's (i) neighborhood or the targets (j) neighborhood. See the details section.

`dependency`
: Boolean indicating if relational relevancy should be considered when computing the network statistic.

`relationalTimeSpan`
: A scalar that indicates the temporal span for relational relevancy (should be specified if dependency is set to TRUE). For instance, if event time is in number of events since the first starting event and the researcher wants the temporal time span to be 10 events, this this value should be specified to 10.

`nopastEvents`
: What value should be given to event's where the sender has no sent any past ties (i's neighborhood) or has not received any past ties (j's neighborhood). Preset to NA.

`sliding_windows`
: Boolean indicating if the sliding windows framework should be used.

`processed_seqIDs`
: A vector of values that contains the event sequence IDs from the sampled event dataset for the sliding windows method

`window_size`
: The sizes of the windows that ared used for the sliding windows computational framework, if NA, the function internally divides the dataset into ten slices.

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 period 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 period 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 period 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 period 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 reciever'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.table::data.table(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 <- rem.riskset.om(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$recency_rawdiff <- rem.stat.recency(
 observed_time = events$time, # variable (column) name that contains the time variable
 observed_receiver = events$target, # variable (column) name that contains the receiver variable
 observed_sender = events$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$recency_inv <- rem.stat.recency(
 observed_time = events$time, # variable (column) name that contains the time variable
 observed_receiver = events$target, # variable (column) name that contains the receiver variable
 observed_sender = events$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$recency_rank <- rem.stat.recency(
 observed_time = events$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$recency_rawdiffSW <- rem.stat.recency(
 observed_time = events$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$recency_invSW <- rem.stat.recency(
 observed_time = events$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$recency_rankSW <- rem.stat.recency(
 observed_time = events$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)
```

`rem.stat.recip` *Compute Reciprocity Statistic for Relational Event Sequences*

Description

TThe function calculates the reciprocity network sufficient statistic for a relational event sequence (see Lerner and Lomi 2021; Butts 2008). 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 2021; 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

```
rem.stat.recip(
  observed_time,
  observed_sender,
  observed_receiver,
  processed_time,
  processed_sender,
  processed_receiver,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  counts = FALSE,
  halflife,
  dyadic_weight,
  window_size = NA,
  Lerneretal_2013 = FALSE
)
```

Arguments

`observed_time`	A vector of values that contains the time values from the full event dataset (i.e., the real events).

`observed_sender`
: A vector of contains the event senders from the full event dataset (i.e., the real events).

`observed_receiver`
: A vector of contains the event receivers from the full event dataset (i.e., the real events).

`processed_time`
: A vector of values that contains the time values from the sampled event dataset (i.e., the dataset with real and null events).

`processed_sender`
: A vector of values that contains the sender values from the sampled event dataset (i.e., the dataset with real and null events).

`processed_receiver`
: A vector of values that contains the receiver values from the sampled event dataset (i.e., the dataset with real and null events).

`sliding_windows`
: Boolean indicating if the sliding windows framework should be used.

`processed_seqIDs`
: A vector of values that contains the event sequence IDs from the sampled event dataset for the sliding windows method. TRUE = yes; FALSE = no.

`counts`
: Boolean indicating if the raw counts of events should be returned or the exponential weighting function should be used (TRUE = counts; FALSE = exponential weighting).

`halflife`
: The halflife value for the weighting function.

`dyadic_weight`
: The dyadic cutoff weight for events that no longer matter.

`window_size`
: The sizes of the windows that ared used for the sliding windows computational framework, if NA, the function internally divides the dataset into ten slices.

`Lerneretal_2013`
: Boolean indicating if the weighting function of Lerner et al. 2013 should be used (see the detials section).

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 `rem.stat.dyadCut` 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.table::data.table(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 <- rem.riskset.om(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$recip <- rem.stat.recip(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   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$recipSW <- rem.stat.recip(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   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$recipC <- rem.stat.recip(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   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)
```

`rem.stat.repetition` *Compute Repetition Statistic for Relational Event Sequences*

Description

This function computes the repitition network sufficient statistic for a relational event sequence (see Lerner and Lomi 2021; Butts 2008). Repetition measures the increased tendency for events between S and R to occur in the future 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 2021; 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

```
rem.stat.repetition(
  observed_time,
  observed_sender,
  observed_receiver,
  processed_time,
  processed_sender,
  processed_receiver,
  sliding_windows = FALSE,
  processed_seqIDs = NULL,
  halflife,
  counts = FALSE,
  dyadic_weight,
  window_size = NA,
  Lerneretal_2013 = FALSE
)
```

Arguments

`observed_time` A vector of values that contains the time values from the full event dataset (i.e., the real events).

`observed_sender` A vector of contains the event senders from the full event dataset (i.e., the real events).

`observed_receiver` A vector of contains the event receivers from the full event dataset (i.e., the real events).

`processed_time` A vector of values that contains the time values from the sampled event dataset (i.e., the dataset with real and null events).

`processed_sender` A vector of values that contains the sender values from the sampled event dataset (i.e., the dataset with real and null events).

`processed_receiver`
: A vector of values that contains the receiver values from the sampled event dataset (i.e., the dataset with real and null events).

`sliding_windows`
: Boolean indicating if the sliding windows framework should be used.

`processed_seqIDs`
: A vector of values that contains the event sequence IDs from the sampled event dataset for the sliding windows method. TRUE = yes; FALSE = no.

`halflife`
: The halflife value for the weighting function.

`counts`
: Boolean indicating if the raw counts of events should be returned or the exponential weighting function should be used (TRUE = counts; FALSE = exponential weighting).

`dyadic_weight`
: The dyadic cutoff weight for events that no longer matter.

`window_size`
: The sizes of the windows that ared used for the sliding windows computational framework, if NA, the function internally divides the dataset into ten slices.

`Lerneretal_2013`
: Boolean indicating if the weighting function of Lerner et al. 2013 should be used (see the details section).

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 `rem.stat.dyadCut` 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 <- rem.riskset.tm(
 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$rep <- rem.stat.repetition(
   observed_time = WikiEvent2018$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$sw_rep <- rem.stat.repetition(
   observed_time = WikiEvent2018$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$repC <- rem.stat.repetition(
   observed_time = WikiEvent2018$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)
```

`rem.stat.triad` *Compute Triad Closure Statistics for Relational Event Sequences*

Description

This function computes the triadic closure network sufficient statistic for a relational event sequence (see Lerner and Lomi 2021; 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 2021; 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

```
rem.stat.triad(
  observed_time,
  observed_sender,
  observed_receiver,
  processed_time,
  processed_sender,
  processed_receiver,
```

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

Arguments

observed_time	A vector of values that contains the time values from the full event dataset (i.e., the real events).
observed_sender	A vector of contains the event senders from the full event dataset (i.e., the real events).
observed_receiver	A vector of contains the event receivers from the full event dataset (i.e., the real events).
processed_time	A vector of values that contains the time values from the sampled event dataset (i.e., the dataset with real and null events).
processed_sender	A vector of values that contains the sender values from the sampled event dataset (i.e., the dataset with real and null events).
processed_receiver	A vector of values that contains the receiver values from the sampled event dataset (i.e., the dataset with real and null events).
sliding_windows	Boolean indicating if the sliding windows framework should be used.
processed_seqIDs	A vector of values that contains the event sequence IDs from the sampled event dataset for the sliding windows method. TRUE = yes; FALSE = no.
counts	Boolean indicating if the raw counts of events should be returned or the exponential weighting function should be used (TRUE = counts; FALSE = exponential weighting).
halflife	The halflife value for the weighting function.
dyadic_weight	The dyadic cutoff weight for events that no longer matter.
window_size	The sizes of the windows that ared used for the sliding windows computational framework, if NA, the function internally divides the dataset into ten slices.
Lerneretal_2013	Boolean indicating if the weighting function of Lerner et al. 2013 should be used (see the details section).

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 `rem.stat.dyadCut` 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 recieved ties from the current event sender and recevier. 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.table::data.table(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 <- rem.riskset.om(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$triadic <- rem.stat.triad(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   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$triadicSW <- rem.stat.triad(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   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$triadicC <- rem.stat.triad(
   observed_time = events$time, # variable (column) name that contains the time variable
  observed_sender = events$sender, # variable (column) name that contains the sender variable
  observed_receiver = events$target, # variable (column) name that contains the receiver variable
   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)
```

`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.

`tm.constraint` *Compute Burchard and Cornwell (2018) Two-Mode Constraint*

Description

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

Usage

```
tm.constraint(net, isolates = NA, returnCIJmat = FALSE, weighted = FALSE)
```

Arguments

`net`	A two-mode adjacency matrix/affiliation matrix
`isolates`	What value should isolates be given? Preset to be NA.
`returnCIJmat`	Should the total constraint be returned or the full constraint matrix? TRUE indicates the full matrix
`weighted`	Boolean. TRUE indicates the matrix is weighted a should use the weighted correction. FALSE assumes the matrix is not weighted.

Details

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)
tm.constraint(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")
tm.constraint(BCweighted, weighted = TRUE)
```

`tm.degree` *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 or on the symmetric *h x h* shared actors matrix for level 2 groups.

Usage

```
tm.degree(net, level1 = TRUE)
```

Arguments

net	A two-mode adjacency matrix
level1	TRUE indicating if the degree should be computed on level 1 nodes, FALSE computes centraility for the level 2 nodes.

Details

Following Knoke and Yang (2020), the computation of degree 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., students) and *h* is the number of level 2 nodes (i.e., school clubs and sports). If the function is defined on the level 1 nodes, the degree centrality is an actor *i* computed as:

$$X^G = XX^T$$

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

In constrast, if it is defined on the level 2 nodes, the degree centrality is an actor *i* 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 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: Quanitative 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")
tm.degree(knoke_yang_PC, level1 = TRUE) #note: this value matches that of the book
tm.degree(knoke_yang_PC, level1 = FALSE) #note: this value matches that of the book
```

`tm.density` *Compute 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

```
tm.density(net, binary = FALSE, level1 = TRUE)
```

Arguments

`net`	A two-mode adjacency matrix
`binary`	Boolean: TRUE indicating if the transposed matrices should be binarized (see Wasserman and Faust 1995: 316)
`level1`	TRUE indicating if the density should be computed on level 1 nodes, FALSE computes graph density for the level 2 nodes.

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., students) and *h* is the number of level 2 nodes (i.e., school clubs and sports). 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 constrast, 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 specifed by *binary = TRUE*.

Value

Two-mode density

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: Quanitative 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")
tm.density(knoke_yang_PC, level1 = TRUE) #note: this value does not match that of the book,
                                  #but does match that of Wasserman and Faust (1995: 317)
                                   #for the ceo dataset.
tm.density(knoke_yang_PC, level1 = FALSE) #note: this value matches that of the book
```

---

tm.effective *Compute Burchard and Cornwell (2018) Two-Mode Effective Size*

Description

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

Usage

```
tm.effective(
  net,
  inParallel = FALSE,
  nCores = NULL,
  isolates = NA,
  weighted = FALSE
)
```

Arguments

net	A two-mode adjacency matrix/affiliation matrix
inParallel	Boolean. TRUE indicates that the values should be computed in parallel (see the foreach package). FALSE does not use parallel processing.
nCores	The number of computing cores for parallel processing. If this value is not specified then the function internally provides it.
isolates	What value should isolates be given? Preset to be NA.
weighted	Boolean. TRUE indicates the matrix is weighted a should use the weighted correction. FALSE assumes the matrix is not weighted.

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)
tm.effective(BCNet)

#For this example, we recreate Figure 9 in Burchard and Cornwell (2018:18) f
#or 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")
tm.effective(BCweighted, weighted = TRUE)
```

`tm.homo4cycles` *Compute Homophlious Four Cycles in 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

```
tm.homo4cycles(net, mem)
```

Arguments

`net`	A two-mode adjacency matrix
`mem`	A vector of membership scores

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 two-mode homophilous four cycles.

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
tm.homo4cycles(southern.women, mem = membership)
```

`tm.homoDis` *Compute Ego Homophily Distance in 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

```
tm.homoDis(net, mem, standardize = FALSE)
```

Arguments

`net`	A two-mode adjacency matrix
`mem`	A vector of membership scores
`standardize`	Boolean. Should the scores be standardized by the number of level 2 nodes the level 1 node has. See details. TRUE = yes; FAlSE = no.

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 if they are a pendant, that is, a level 1 actor who is the only level 1 nodes that is connected to that a specific level 2 node, 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
tm.homoDis(southern.women, mem = membership)
#the ego 2 mode distance standardized
tm.homoDis(southern.women, mem = membership, standardize = TRUE)
```

---

tm.redundancy *Compute Burchard and Cornwell (2018) Two-Mode Redundancy*

Description

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

Usage

```
tm.redundancy(
  net,
  inParallel = FALSE,
  nCores = NULL,
  isolates = NA,
  weighted = FALSE
)
```

Arguments

net	A two-mode adjacency matrix/affiliation matrix
inParallel	Boolean. TRUE indicates that the values should be computed in parallel (see the foreach package). FALSE does not use parallel processing.
nCores	The number of computing cores for parallel processing. If this value is not specified then the function internally provides it.

`isolates`	What value should isolates be given? Preset to be NA.
`weighted`	Boolean. TRUE indicates the matrix is weighted a should use the weighted correction. FALSE assumes the matrix is not weighted.

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., employees in an organization) 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)
```

```
tm.redundancy(BCNet) #this values replicate those reported by Burchard and Cornwell (2018: 14)


#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")
tm.redundancy(BCweighted, weighted = TRUE)
```

`weight` *Compute the Exponential Weights in Relational Event Models*

Description

Compute the Exponential Weights in Relational Event Models

Usage

```
weight(current, past, halflife, dyadic_weight, Lerneretal_2013 = FALSE)
```

Arguments

`current`	The current event time.
`past`	The vector of past event times.
`halflife`	The halflife parameter for recency weighting.
`dyadic_weight`	The dyadic (event) weight cutoff for relational relevancy.
`Lerneretal_2013`	Boolean indicating if the weighting function of Lerner et al. 2013 should be used.

Value

The event exponential weights

`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