Package 'DynComm'

July 14, 2020

Type Package

Title Dynamic Network Communities Detection and Generation

Version 2020.1.4

Author Rui Portocarrero Sarmento, Luís Lemos, Mário Cordeiro, Giulio Rossetti

Maintainer Rui P. Sarmento <email@ruisarmento.com>

Description Used for evolving network analysis regarding
community detection. Implements several algorithms that calculate communities
for graphs whose nodes and edges change over time.
Edges, which can have new nodes, can be added or deleted. Changes in the
communities are calculated without recalculating communities for the entire
graph.
REFERENCE: M. Cordeiro et al. (2016) <DOI:10.1007/s13278-016-0325-1>
G. Rossetti et al. (2017) <DOI:10.1007/s10994-016-5582-8>
G. Rossetti (2017) <DOI:10.1093/comnet/cnx016>
R. Sarmento (2019) <arXiv:1904.12593>

License GPL-2

Encoding UTF-8

LazyData false

Imports Rcpp (>= 0.12.15), Rdpack, methods

Depends igraph

LinkingTo Rcpp

RcppModules DynCommRcpp

RdMacros Rdpack

SystemRequirements C++11

Suggests devtools

RoxygenNote 7.1.0

StagedInstall yes

NeedsCompilation yes

R topics documented:

`DynComm-package` *DynComm: Dynamic Network Communities Detection*

Description

Bundle of algorithms used for evolving network analysis regarding community detection.

Details

Implements several algorithms, using a common API, that calculate communities for graphs whose vertices and edges change over time. Edges, which can have new vertices, can be added or deleted, and changes in the communities are calculated without recalculating communities for the entire graph.

Referenced Work

This package uses the following work as reference material for the implementation of the algorithms.

Author(s)

poltergeist0

References

GitHub project source Cordeiro M, Sarmento RP, Gama J (2016). "Dynamic community detection in evolving networks using locality modularity optimization." *Social Network Analysis and Mining*, **6**(1), 1–20. Rossetti G, Pappalardo L, Pedreschi D, Giannotti F (2017). "Tiles: An Online Algorithm for Community Discovery in Dynamic Social Networks." *Mach. Learn.*, **106**(8), 1213–1241. ISSN 0885-6125, doi: 10.1007/s1099401655828, https://doi.org/10.1007/s10994-016-5582-8. Rossetti G (2017). "RDYN: Graph Benchmark handling Community Dynamics." *Journal of Complex Networks*. doi: 10.1093/comnet/cnx016, https://academic.oup.com/comnet/article/5/6/893/3925036?guestAccessKey=c3470adf-391d-4fad-b935-63e71e4df06a. Sarmento RP (2019). "Density-based Community Detection/Optimization." *arXiv*. 1904.12593, https://arxiv.org/abs/1904.12593.

See Also

DynComm , DynComm-package-dev

---

addRemoveEdges *addRemoveEdges*

Description

This method reads edges from either a matrix or a file and adds or removes them to/from the graph.

Usage

```
addRemoveEdges.DynComm(dyncomm,graphAddRemove)
addRemove.DynComm(dyncomm,graphAddRemove)
add.DynComm(dyncomm,graphAddRemove)
```

Arguments

dyncomm	A DynComm object, if not using the inline version of the function call
graphAddRemove	Either the matrix or the filename that contains the edges to add/remove

Details

If the weight is exactly zero, the edge is removed from the graph.

If a vertex, mentioned in the source or destination, does not exist it will be added to the graph.

If any post processing algorithm exists, it is automatically calculated after the main algorithm.

Matrix input The matrix must have at least two columns with the source and destination vertices.

If all edges are to be added with the default weight, a third column is optional.

If any edge is to be removed, the third column is mandatory.

File input The file must have only one edge per line, with values separated by a white space (both SPACE and TAB work in any amount and combination). The line must end with a newline character (also known as linefeed, LF or '\n').

The first value is the source vertex, the second is the destination vertex, and the third is the weight.

The weight can be ommited if the edge is to be added using the default weight of 1 (one), or if the parameter to ignore weights was set.

The method detects automatically if the weight is present on a row by row basis so some rows may have weights defined and others not.

Value

FALSE if any kind of error occurred. Otherwise, TRUE

Author(s)

poltergeist0

See Also

DynComm , postProcess

Examples

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
dc$addRemoveEdges("graphAddRemoveFile.txt")
```

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
addRemoveEdges(dc,"graphAddRemoveFile.txt")
```

ALGORITHM *List of available algorithms.*

Description

An algorithm mainly defines how vertices and/or communities are processed, when criterion is applyed (quality measurements occur) and what happens to the communities depending on the value of the quality obtained.

Usage

```
ALGORITHM
```

Format

A named list with the names of the available algorithms:

LOUVAIN is a greedy optimization method to extract communities from large networks by optimizing the density of edges inside communities to edges outside communities.
See ALGORITHM_LOUVAIN
@references Cordeiro M, Sarmento RP, Gama J (2016). "Dynamic community detection in evolving networks using locality modularity optimization." *Social Network Analysis and Mining*, **6**(1), 1–20.

Author(s)

poltergeist0

See Also

DynComm

Examples

```
ALGORITHM$LOUVAIN
```

ALGORITHM-dev *DynComm Documentation for Developers of new Main Algorithms*

Description

Instructions for adding new main algorithms to the DynComm package.

Steps

This section provides step by step instructions on how to implement a new main algorithm and how to integrate it into the DynComm package.

1. Go to the project source and get an appropriate template for your algorithm from the "dev" folder on the root of the project source.
   Different languages are distinguished by the extension of the file.
   If a template is not available use one from a language that is most related to the language you intend to use or use the R template to, at least, provide you with the function names, types of inputs and types of outputs.
2. Implement the new algorithm inside the template, preferably, inside a private function. The algorithm, that is the private function, must be called at the end of the addRemoveVertices function of the API.
   All functions, except the addRemoveVertices, must only be used to convert between data types and to return data. No calculations should be performed in them.

   R Temporarily add any library required by the algorithm in your source file. It will have to be removed later but, for now, it is useful for testing.

   Depending on the programming language used, do the following:

   R Choose the "TemplateDynCommMain.R" template.
   Save the source file of the new algorithm in the "R-CRAN/R" folder.

   C++11 Choose the "TemplateDynCommMain.h" template.
   Save the source file of the new algorithm in the "R-CRAN/src/base/Cpp" folder.

   Python Choose the "TemplateDynCommMain.py" template.
   Save the source file of the new algorithm in the "R-CRAN/src/base/Python" folder.

   The name of the file should reflect the name of the algorithm and start with the word "algorithm".
3. Test the algorithm independently of the DynComm package. If programming in...

   R You can "source" your R file.

   C++11 You might have to write a small program to test it, that can later be used to produce a standalone version. Or you can integrate it in the existing C++ standalone, modifying where needed.
   If you want to use the existing standalone to test your algorithm, the file is named "DynCommEXE.cpp". Follow the instructions in its developer notice.
   Continue to read these instructions to know how to integrate your code into the existing C++ source code. The steps are identical to the integration of your algorithm into this package except that you can skip the documentation and editing of any R files.
   Afterwards, you can resume reading from here to implement the missing steps to integrate into this package.

Python You can test it by invoking it directly from a python command line. Check Python documentation on how to do it.

Use the tests performed as examples and write them in the documentation when it is created.

4. If the algorithm you are creating has associated bibliography, add reference to it in the existing "REFERENCES.bib" file.

5. Create documentation for your algorithm. This involves adding documentation in three files.
   The first is the developer documentation on the same file you implemented your algorithm. If using a template, there is already a documentation template for you to modify.
   This documentation must have a detailed description of the algorithm, including a description of how it works, its parameters and contain examples that can be used for automatic testing.
   The second file is the "ALGORITHM.R" where the user friendly documentation is written. This documentation should have a high level description of the algorithm, acceptable parameters and supported criterion.
   The thrird file is the "DynCommMain.R" where it says "document new algorithms here". This documentation should use the same format used for other algorithms with a very small description of the algorithm, preferably just two lines, with a link to the user friendly documentation and references to publications.

6. Add the name of your algorithm to the ALGORITHM list in the "DynCommMain.R" file under the marker that says "list new algorithms here".
   Add your algorithm parameters to the matrix in the "DynComm.R" file under the marker that says "add parameters here".

   R Add a source command to the "DynCommMain.R" file under the marker that says "Include R sources here". Add any R libraries required by your algorithm to the "DynComm.R" file under the marker that says "List imports here".
   Remove all libraries from your algorithms' R source file.

   C++11 Edit the file "algorithm.h" and look for the several "...new algorithms here" markers. They will tell what needs to be done to integrate your algorithm.
   Then, if your algorithm requires any parameters, edit the file "program.h" and look for the several "...new algorithms here" markers. They will tell what needs to be done to add your algorithms' parameters to the C++ source.

   Python TO DO :(

7. You should now be able to build the package and, if everything went right, your algorithm is successfuly integrated into this package. **Congratulations :D**

Author(s)

poltergeist0

See Also

DynComm , DynComm-package-dev

ALGORITHM_LOUVAIN *ALGORITHM_LOUVAIN*

Description

Is a greedy optimization method to extract communities from large networks by optimizing the density of edges inside communities to edges outside communities.

Supported CRITERION

MODULARITY See CRITERION_MODULARITY

PARAMETERS

This algorithm does not require any parameters.

Author(s)

poltergeist0

See Also

DynComm

communities *communities*

Description

This method returns all communities from the selected post processing algorithm or the main algorithm, after the last iteration.

Usage

```
communities.DynComm(dyncomm)
```

Arguments

dyncomm	A DynComm object, if not using the inline version of the function call

Value

a list of all communities

Author(s)

poltergeist0

See Also

DynComm , postProcess

Examples

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
dc$communities()
```

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
communities(dc)
```

---

communitiesEdgeCount *communitiesEdgeCount*

Description

This method returns the number of community to community edges in the graph from the selected post processing algorithm or the main algorithm, after the last iteration.

Usage

```
communitiesEdgeCount.DynComm(dyncomm)
```

Arguments

dyncomm	A DynComm object, if not using the inline version of the function call

Value

the number of community to community edges in the graph

Author(s)

poltergeist0

See Also

DynComm , postProcess

Examples

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
dc$communitiesEdgeCount()
```

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
communitiesEdgeCount(dc)
```

`community` *community*

Description

Get the community of the given vertex from the selected post processing algorithm or the main algorithm, after the last iteration.

Usage

```
community.DynComm(dyncomm,vertex)
```

Arguments

`dyncomm`	A DynComm object, if not using the inline version of the function call
`vertex`	The name of the intended vertex

Value

an unsigned integer with the community of the given vertex

Author(s)

poltergeist0

See Also

`DynComm` , `postProcess`

Examples

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
dc$community(8)
```

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
```

```
community(dc,8)
```

---

communityCount *communityCount*

Description

Get the number of communities from the selected post processing algorithm or the main algorithm, after the last iteration.

Usage

```
communityCount.DynComm(dyncomm)
```

Arguments

dyncomm	A DynComm object, if not using the inline version of the function call

Value

an unsigned integer value with the number of communities

Author(s)

poltergeist0

See Also

DynComm , postProcess

Examples

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
dc$communityCount()
```

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
communityCount(dc)
```

communityEdgeWeight *communityEdgeWeight*

Description

Get the weight of the edge that goes from source community to destination community from the selected post processing algorithm or the main algorithm, after the last iteration.

Usage

```
communityEdgeWeight.DynComm(dyncomm,source,destination)
```

Arguments

dyncomm	A DynComm object, if not using the inline version of the function call
source	The name of the source community that is part of the edge
destination	The name of the destination community that is part of the edge

Value

a floating point number with the weight

Author(s)

poltergeist0

See Also

DynComm , postProcess

Examples

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
dc$communityEdgeWeight(12,42)
```

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
communityEdgeWeight(dc,12,42)
```

communityInnerEdgesWeight *communityInnerEdgesWeight*

Description

Get the sum of weights of the inner edges of the given community from the selected post processing algorithm or the main algorithm, after the last iteration.

Usage

```
communityInnerEdgesWeight.DynComm(dyncomm,community)
```

Arguments

dyncomm	A DynComm object, if not using the inline version of the function call
community	The name of the intended community

Value

a floating point number with the weight

Author(s)

poltergeist0

See Also

DynComm , postProcess

Examples

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
dc$communityInnerEdgesWeight(1)
```

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
communityInnerEdgesWeight(dc,1)
```

communityMapping *communityMapping*

Description

Get the community mapping for all communities from the selected post processing algorithm or the main algorithm, after the last iteration.

Usage

```
communityMapping.DynComm(dyncomm,differential, file)
```

Arguments

dyncomm	A DynComm object, if not using the inline version of the function call
differential	If TRUE, only values that have changed in the latest run will be returned
file	If given, outputs the community mapping to the given file instead of the console

Details

If file is not given, returns a two column matrix with vertices in the first column and the communities in the second.

If file is given, returns a single row, single column matrix with TRUE or FALSE, depending whether if writing to file succeeded or failed, respectively.

When writing to file, if the Community-Vertex program parameter is TRUE, each line of the file will have the community first, followed by a list of vertices that belong to the community. If that parameter is FALSE, each line will have a single vertex followed by its community. All values are separated by a white character.

Value

a matrix with either the community mapping or a boolean value

Author(s)

poltergeist0

See Also

DynComm , postProcess

Examples

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
dc$communityMapping()
```

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
communityMapping(dc)
```

---

communityNeighbours *communityNeighbours*

Description

Get all neighbours (communities connected through direct edges) of the given community in the graph from the selected post processing algorithm or the main algorithm, after the last iteration.

Usage

```
communityNeighbours.DynComm(dyncomm,community)
```

Arguments

dyncomm	A DynComm object, if not using the inline version of the function call
community	The community to get neighbours from

Details

The return value is a matrix with two columns. The first is the neighbour and the second is the weight of the edge that connects them.

Value

a matrix of all communities in the graph that are neighbours of the given community and their edge weight

Author(s)

poltergeist0

See Also

DynComm , postProcess

Examples

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
dc$communityNeighbours(community)
```

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
communityNeighbours(dc,community)
```

`communityTotalWeight` *communityTotalWeight*

Description

Get the sum of weights of all edges of the given community from the selected post processing algorithm or the main algorithm, after the last iteration.

Usage

```
communityTotalWeight.DynComm(dyncomm,community)
```

Arguments

`dyncomm`	A DynComm object, if not using the inline version of the function call
`community`	The name of the intended community

Value

a floating point number with the weight

Author(s)

poltergeist0

See Also

`DynComm` , `postProcess`

Examples

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
dc$communityTotalWeight(1)
```

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
```

```
communityTotalWeight(dc,1)
```

`communityVertexCount` *communityVertexCount*

Description

Get the amount of vertices in the given community from the selected post processing algorithm or the main algorithm, after the last iteration.

Usage

```
communityVertexCount.DynComm(dyncomm,community)
communityNodeCount.DynComm(dyncomm,community)
```

Arguments

`dyncomm`	A DynComm object, if not using the inline version of the function call
`community`	The name of the intended community

Value

an unsigned integer with the number of vertices in the given community

Author(s)

poltergeist0

See Also

`DynComm` , `postProcess`

Examples

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
dc$communityVertexCount(3)
```

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
communityVertexCount(dc,3)
```

CRITERION *List of available CRITERION (quality measurement functions).*

Description

A criterion is used to indicate the proximity of the current grouping of vertices (communities) to the optimum one.

Usage

```
CRITERION
```

Format

A named list with the names of the available CRITERION:

MODULARITY Newman-Girvan
See CRITERION_MODULARITY

Details

Theoretically, the bigger the value returned by the criterion, the closer the current grouping is to the best possible grouping.

Each CRITERION internally defines two functions. One is used to evaluate if moving a vertex from one group (community) to another possibly yields a better overall result. The other is used to measure the actual overall quality of the entire grouping (current community mapping).

Not all criterion might be available for all algorithms. See each algorithms' help to find which criterion is supported

Author(s)

poltergeist0

See Also

DynComm

Examples

```
CRITERION$MODULARITY
```

CRITERION-dev *DynComm Documentation for Developers of new Criterion for Main Algorithms*

Description

Instructions for adding new criterion for main algorithms to the DynComm package.

Details

Due to criterion being tightly connected to main algorithms and the multiple programming language nature of this package, it is not possible to completely separate criterion from main algorithms to have them be independent.

This is due to the fact that criterion require access to the graph and, being completely independent, would mean that another copy of the graph would have to be stored at all times in the criterion to account for the possibilty of, as an example, using a main algorithm written in Python with a criterion written in C++.

Even if there could be a way for all algorithms and criterion to share the same graph through R, performance would suffer because of all calls being redirected through R.

So, currently, the best way is to write the criterion in each language. That has the obvious downside of having to know several languages but, on the other side, criterion are fairly simple when compared to main algorithms.

Also, if you are not able to implement the criterion in every language, you can document which main algorithms support it by adding the name of the criterion to the documentation of the main algorithm under the "Supported CRITERION" section.

Depending on the programming language, follow the instructions below:

C++11 It might be easier to just copy the source code of an existing criterion an modify it as required.
Implement a class that extends class "CriterionBase" and implements all functions of class "CriterionInterface". Follow the directions in the file "criterion.h" under the markers that start with the word "TODO", to add your criterion to C++.

Document your criterion in the file "CRITERION.R" following the instructions stated in the header. Follow the example of other documented criterion and add a section that states the supported main algorithms.

Add your criterion to the list of criterion in the file "DynCommMain.R" under the marker that says "list new criterions here" and add a link to its documentation in the documentation above that marker.

Add links to the criterion documentation on each of the supported main algorithms in the file "ALGORITHM.R"

Author(s)

poltergeist0

See Also

DynComm , DynComm-package-dev

CRITERION_MODULARITY *CRITERION_MODULARITY*

Description

Newman-Girvan criterion.

Supported in ALGORITHM

LOUVAIN See ALGORITHM_LOUVAIN

PARAMETERS

This criterion does not require any parameters.

Author(s)

poltergeist0

See Also

DynComm

DynComm *DynComm*

Description

Provides a single interface for all algorithms in the different languages.

Usage

```
DynComm(Algorithm,Criterion,Parameters)
```

Arguments

Algorithm	One of the available ALGORITHM. Default ALGORITHM$LOUVAIN. See ALGORITHM
Criterion	One of the available CRITERION. Default CRITERION$MODULARITY. See CRITERION
Parameters	A two column matrix defining additional parameters. Default NULL. See the PARAMETERS section on this page

Details

Includes methods to get results of processing and to interact with the vertices, edges and communities. Provided methods to return information on the graph are divided into two layers. A lower level layer that interacts with vertices and how they connect. And a higher level layer that interacts with communities and how they connect. Besides the main algorithm, also accepts post processing algorithms that are used mainly to filter the results. Post processing algorithms can use additional computational resources so check the Performance section of the help page of each algorithm you intend to use.

Value

`DynComm` object

PARAMETERS

A two column matrix defining additional parameters to be passed to the selected ALGORITHM and CRITERION. The first column names the parameter and the second defines its value.

c Owsinski-Zadrozny quality function parameter. Values [0.0:1.0]. Default: 0.5

k Shi-Malik quality function kappa_min value. Value > 0 . Default 1

w Treat graph as weighted. In other words, do not ignore weights for edges that define them when inserting edges in the graph. A weight of exactly zero removes the edge instead of inserting so its weight is never ignored. Without this parameter defined or for edges that do not have a weight defined, edges are assigned the default value of 1 (one). As an example, reading from a file may define weights (a third column) for some edges (defined in rows, one per row) and not for others. With this parameter defined, the edges that have weights that are not exactly zero, have their weight replaced by the default value. Values TRUE,FALSE. Default FALSE

e Stops when, on a cycle of the algorithm, the quality is increased by less than the value given in this parameter. Value > 0 . Default 0.01

cv Community-Vertex. Boolean parameter that indicates if sending community mapping to a file prints the community first, if true, or the vertex first, if false. See `communityMapping` for details. Default TRUE

Methods

postProcess(actions) Set a list of post processing steps. See `postProcess`

select(postProcessing,id) Select between getting the results of the algorithm or one of the post processing steps. See `select`

results(differential) Get additional results of the algorithm or the currently selected post processing steps. See `results`

addRemoveEdges(graphAddRemove) Add and remove edges read from a matrix or file. See `addRemoveEdges`

addRemove(graphAddRemove) Alias for addRemoveEdges(). See `addRemoveEdges`

add(graphAddRemove) Alias for addRemoveEdges(). See `addRemoveEdges`

quality() Get the quality measurement of the graph after the last iteration. See `quality`

communityCount() Get the number of communities after the last iteration. See `communityCount`

communities() Get all communities after the last iteration. See communities

communitiesEdgeCount() Get the number of community to community edges in the graph. See communitiesEdgeCount

communityNeighbours(community) Get the neighbours of the given community after the last iteration. See communityNeighbours

communityInnerEdgesWeight(community) Get the sum of weights of the inner edges of the given community after the last iteration. See communityInnerEdgesWeight

communityTotalWeight(community) Get the sum of weights of all edges of the given community after the last iteration. See communityTotalWeight

communityEdgeWeight(source,destination) Get the weight of the edge that goes from source community to destination community after the last iteration. See communityEdgeWeight

communityVertexCount(community) Get the amount of vertices in the given community after the last iteration. See communityVertexCount

communityNodeCount(community) Alias for communityVertexCount(). See communityVertexCount

community(vertex) Get the community of the given vertex after the last iteration. See community

vertexCount() Get the total number of vertices after the last iteration. See vertexCount

nodesCount() Alias for vertexCount(). See vertexCount

verticesAll() Get all vertices in the graph after the last iteration. See verticesAll

nodesAll() Alias for verticesAll(). See verticesAll

neighbours(vertex) Get the neighbours of the given vertex after the last iteration. See neighbours

edgeWeight(source,destination) Get the weight of the edge that goes from source vertex to destination vertex after the last iteration. See edgeWeight

edge(source,destination) Alias for edgeWeight(). See edgeWeight

vertices(community) Get all vertices belonging to the given community after the last iteration. See vertices

nodes(community) Alias for vertices(community). See vertices

edgeCount() Get the number of vertex to vertex edges in the graph. See edgeCount

communityMapping(differential, file) Get the community mapping for all communities after the last iteration. See communityMapping

mytime(differential) Get the cumulative time spent on processing after the last iteration. See mytime

version() Get the source code versions of the different sources. See version

Author(s)

poltergeist0

See Also

DynComm-package

Examples

```
Parameters<-matrix(c("e","0.1","w", "FALSE"),ncol=2, byrow=TRUE)
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,Parameters)
dc$addRemoveEdges(
matrix(
c(10,20,10,30,20,30,30,60,40,60,40,50,50,70,60,70)
,ncol=2,byrow=TRUE)
)
## or
## dc$addRemoveEdges("initial_graph.txt")
dc$communityCount()
## You can use the non inline version of the functions
communities.DynComm(dc)
## Several alias have been defined.
## In this case, communityNodeCount is alias of communityVertexCount
dc$communityNodeCount(10)
dc$communityNeighbours(10)
dc$communityInnerEdgesWeight(10)
dc$communityTotalWeight(10)
dc$communityEdgeWeight(10,40)
dc$community(10) ##this parameter is a vertex not a community. Do not confuse them
dc$vertices(10)
dc$communityMapping(TRUE)
dc$quality()
dc$mytime()
## lets add post processing :)
dc$postProcess(
list(
list(POSTPROCESSING$DENSOPT)
)
)
## the results of the last step of post processing are selected automatically
## densopt post processing algorithm may change the community mapping so...
## check it
dc$communityMapping(TRUE)
## densopt post processing algorithm may change quality so check it
dc$quality()
## time is now the total time of the main algorithm plus the time of every...
## post processing algorithm up to the one selected
dc$mytime()
## get back to main algorithm results to check they haven't changed
dc$select(POSTPROCESSING$NONE)
dc$communityMapping(TRUE)
dc$quality()
dc$mytime()
## add and remove edges. Notice that there is one more column to give...
## weights of zero on the edges to remove. In this case, all other weights...
## are ignored because the graph is set to ignore weights (parameter w is...
## false).
dc$addRemoveEdges(
matrix(
c(30,60,0,40,60,0.23,10,80,2342,80,90,3.1415)
```

```
,ncol=3,byrow=TRUE)
)
## since the post processing was not reset, it will be automatically...
## calculated and results switched to the last step. In this case, to the...
## densopt algorithm
dc$communityMapping(TRUE)
dc$quality()
dc$mytime()
## get back to main algorithm results to check them
dc$select(POSTPROCESSING$NONE)
dc$communityMapping(TRUE)
dc$quality()
dc$mytime()
## lets reset/remove post processing
dc$postProcess()
```

DynComm-package-dev *DynComm Documentation for Developers*

Description

Instructs delevopers how to add new algorithms, criterion and post processing algorithms to the DynComm package.

Details

Implementing new algorithms in new packages is a lot of work.

With this package, we try to accomplish two things: make the addition of new algorithms easier and concentrate dynamic community detection algorithms in a single package, no matter the language used to write them.

Always read the entirety of the instructions even if they do not seem to apply to your case. Care was taken to make the instructions as general as possible, mentioning specificities only when they differ from the general case.

Most of the instructions described are for algorithms writen in R, since it is the language used for the user interface and is the easiest to integrate.

Algorithms writen in other languages will also need this information in order to know the types of the inputs and outputs of the functions.

It is advisable to always read the "Developer Notice" on the beginning of the files mentioned in these instructions. It will contain useful information about the source code on the file and where new code can be added.

Whenever "Project", "Project Page" or "Project source" is mentioned, the developer should know that it refers to the project source code page on GitHub (GitHub project source).

The project source has the following organization:

Root This is the root folder of the project source code. It contains files about the project source code and the folders "dev", "R-CRAN", "test" and "standalone".

dev Folder with templates for developers of new main algorithms, new criterion and new post processing algorithms. Also contains these instructions in text format.

R-CRAN Contains the source code for the actual DynComm package. Internally, has the same organization as required by any R package project. The most important folders are named "inst", "src" and "R".

inst Contains a file named "REFERENCES.bib" where bibliographic references are stored using the bibtex format.

src The root of this folder contains files in other languages that implement an interaction layer between R and the respective programming language. The actual source code that implements a certain algorithm is placed inside a sub-folder named after the programming language inside the folder "base".

As an example, the Dynamic Louvain algorithm used in this package was implemented in C++11. There is a file named "DynCommRcpp.cpp" which implements the interaction layer using Rcpp. This layer only converts data types from R to C++, instantiates a Louvain object and redirects calls to methods of that object on the C++ source file named "DynCommBase.h".

R This is the folder that contains all R source code files where the architecture of the package is implemented, along with some main algorithms and post processing algorithms, and all the documentation.

Some of the adaptation layers for some programming languages, like Python, are in this folder since they must be implemented inside an R source file, as opposed to programming languages like C++ where Rcpp must be inside a C++ source code file.

test Folder with a few sample files with data that can be used to run examples and test the code.

standalone Contains the standalone (command line) versions of the algorithms, for the algorithms that provide them, in case anyone wants to run the algorithms outside of the R environment.

Each program is inside a folder with the name of the programming language used to implement it. As an example, C++ programs are inside a folder named "Cpp".

Not all algorithms may be implemented and some functionality might be slightly different from the one used in the R environment. Post processing algorithms are not provided.

In case of doubt, missing information or if you are implementing in a language that is still not supported, contact the maintainer of the package.

Follow the instructions of the links below in order to add your main algorithm, criterion or post processing algorithm, respectively.

I am implementing a

Main algorithm See ALGORITHM-dev

Criterion See CRITERION-dev

Post processing algorithm See POSTPROCESSING-dev

Author(s)

poltergeist0

See Also

DynComm , DynComm-package

---

edgeCount *edgeCount*

Description

This method returns the number of vertex to vertex edges in the graph from the selected post processing algorithm or the main algorithm, after the last iteration.

Usage

```
edgeCount.DynComm(dyncomm)
```

Arguments

dyncomm	A DynComm object, if not using the inline version of the function call

Value

the number of vertex to vertex edges in the graph

Author(s)

poltergeist0

See Also

DynComm , postProcess

Examples

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
dc$edgeCount()
```

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
edgeCount(dc)
```

edgeWeight *edgeWeight*

Description

Get the weight of the edge that goes from source vertex to destination vertex from the selected post processing algorithm or the main algorithm, after the last iteration.

Usage

```
edgeWeight.DynComm(dyncomm,source,destination)
```

Arguments

dyncomm	A DynComm object, if not using the inline version of the function call
source	The name of the source vertex that is part of the edge
destination	The name of the destination vertex that is part of the edge

Value

a floating point number with the weight

Author(s)

poltergeist0

See Also

DynComm , postProcess

Examples

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
dc$edgeWeight(12,42)
```

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
edgeWeight(dc,12,42)
```

mytime *mytime*

Description

Get the time, in nanoseconds, spent on processing after the last iteration.

Usage

```
mytime.DynComm(dyncomm,differential)
```

Arguments

dyncomm	A DynComm object, if not using the inline version of the function call
differential	Select between differential and accumulated time.

Details

If the differential parameter is set, the time taken by the last iteration will be returned. Otherwise, the default behaviour is to, return the accumulated time spent on processing since the creation of the DynComm object.

If post processing exists, the time returned by this function will include the processing time of all post processing algorithms up to the selected one.

Value

an unsigned integer with the total processing time

Author(s)

poltergeist0

See Also

DynComm , postProcess

Examples

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
dc$mytime()
## 2.3


dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
mytime(dc)
## 2.3
```

neighbours *neighbours*

Description

Get all neighbours (vertices connected through direct edges) of the given vertex in the graph from the selected post processing algorithm or the main algorithm, after the last iteration.

Usage

```
neighbours.DynComm(dyncomm,vertex)
```

Arguments

dyncomm	A DynComm object, if not using the inline version of the function call
vertex	The vertex to get neighbours from

Value

a matrix of all vertices in the graph that are neighbours of the given vertex and their edge weight

Author(s)

poltergeist0

See Also

DynComm , postProcess

Examples

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
dc$neighbours(vertex)
```

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
neighbours(dc,vertex)
```

postProcess *postProcess*

Description

This method receives a list of actions to perform in post processing in the same order they are listed from left to right.

Usage

```
postProcess.DynComm(dyncomm,actions)
```

Arguments

dyncomm	A DynComm object, if not using the inline version of the function call
actions	A list of post processing actions/steps

Details

Several actions of the same type are allowed. They receive an internal ID number that starts at one and increments by one unit with each action of the same type. Later, this ID can be used to select the intended action and get results from it.

Post processing can be reset (removed) be setting actions to NULL (default value) or passing an empty list.

The format of the actions is a list of action. Each action is a list of the action name (see POSTPROCESSING) and parameters. The parameters is a matrix of two columns, the first having the name of the parameter and, the second, the value of the parameter. The parameters is optional, and may be missing, in which case default values are used, if required at all.

The parameters accepted by each post processing algorithm can be found on the help page of each respective algorithm.

This slighty awkward syntax is due to R not supporting matrix of matrices.

Value

FALSE if any kind of error occurred. Otherwise, TRUE

Author(s)

poltergeist0

See Also

DynComm , select , POSTPROCESSING

Examples

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
dc$postProcess(
  list(
    list(POSTPROCESSING$WEIGHTTOP,matrix(data=c("n",10),ncol=2,byrow=TRUE))
    ,list(POSTPROCESSING$DENSOPT)
    ,list(POSTPROCESSING$WEIGHTTOP,matrix(data=c("n",3),ncol=2,byrow=TRUE))
  )
)
# first POSTPROCESSING$WEIGHTTOP gets ID=1 and second gets ID=2
# POSTPROCESSING$DENSOPT uses default parameters
dc$select(POSTPROCESSING$WEIGHTTOP,1)  #selects the results of the first WEIGHTTOP
dc$select(POSTPROCESSING$WEIGHTTOP,2)  #selects the results of the second WEIGHTTOP
dc$select(POSTPROCESSING$NONE)  #selects the main algorithm results
dc$select(POSTPROCESSING$DENSOPT)  #selects the results of densopt
dc$postProcess(NULL)  #remove post processing
## or just
## dc$postProcess()


dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
postProcess(dc,
  list(
    list(POSTPROCESSING$WEIGHTTOP,matrix(data=c("n",10),ncol=2,byrow=TRUE))
    ,list(POSTPROCESSING$DENSOPT)
    ,list(POSTPROCESSING$WEIGHTTOP,matrix(data=c("n",3),ncol=2,byrow=TRUE))
  )
)
# first POSTPROCESSING$WEIGHTTOP gets ID=1 and second gets ID=2
# POSTPROCESSING$DENSOPT uses default parameters
select(dc,POSTPROCESSING$WEIGHTTOP,1)  #selects the results of the first WEIGHTTOP
select(dc,POSTPROCESSING$WEIGHTTOP,2)  #selects the results of the second WEIGHTTOP
select(dc,POSTPROCESSING$NONE)  #selects the main algorithm results
select(dc,POSTPROCESSING$DENSOPT)  #selects the results of densopt
postProcess(dc,NULL)  #remove post processing
## or just
## postProcess(dc)
```

POSTPROCESSING *List of available post processing algorithms.*

Description

A post processing algorithm is a function that modifies the results presented to the user, allowing for limited result manipulation, but does not internally modify the results obtained by the algorithm.

Usage

```
POSTPROCESSING
```

Format

A named list with the names of the available algorithms:

DENSOPT Density optimization is an algorithm that provides a community structure based on the increase of the average community density. See postProcessDensOpt
@references Sarmento RP (2019). "Density-based Community Detection/Optimization." *arXiv*. 1904.12593, https://arxiv.org/abs/1904.12593.

Details

As an example, we are only interested in viewing communities larger than some value but do not want to actually remove the smaller ones from the graph, invalidating possible future processing over them. A post processing algorithm can filter the unwanted values from the results or present a more compact version of them.

Author(s)

poltergeist0

See Also

DynComm

Examples

```
POSTPROCESSING$DENSOPT
```

POSTPROCESSING-dev *DynComm Documentation for Developers of new Post Processing Algorithms*

Description

Instructions for adding new Post Processing Algorithms to the DynComm package.

Steps

This section provides step by step instructions on how to implement a new post processing algorithm and how to integrate it into the DynComm package.

1. Go to the project source and get an appropriate template for your algorithm from the "dev" folder on the root of the project source.
   Different languages are distinguished by the extension of the file.
   If a template is not available use one from a language that is most related to the language you intend to use or use the R template to, at least, provide you with the function names, types of inputs and types of outputs.
2. Implement the new algorithm inside the template, preferably, inside a private function. The algorithm, that is the private function, must be called at the end of the constructor.
   All API functions must only be used to convert between data types and to return data. No calculations should be performed in them.

   R Temporarily add any library required by the algorithm in your source file. It will have to be removed later but, for now, it is useful for testing.

   Depending on the programming language used, do the following:

   R Choose the "TemplateDynCommPostProcess.R" template.
   Save the source file of the new algorithm in the "R-CRAN/R" folder.

   The name of the file should reflect the name of the algorithm and start with the word "postProcess".
3. Test the algorithm independently of the DynComm package. If programming in...

   R You can "source" your R file.

   Use an applicable main algorithm of the DynComm package to generate data to your post processing algorithm. Use the tests performed as examples and write them in the documentation when it is created.
4. If the algorithm you are creating has associated bibliography, add reference to it in the existing "REFERENCES.bib" file.
5. Create documentation for your algorithm. This involves adding documentation in three files.
   The first is the developer documentation on the same file you implemented your algorithm. If using a template, there is already a documentation template for you to modify.
   This documentation must have a detailed description of the algorithm, including a description of how it works, its parameters and contain examples that can be used for automatic testing.
   The second file is the "POSTPROCESSING.R" where the user friendly documentation is written. This documentation should have a high level description of the algorithm, acceptable parameters and resource utilization.
   The thrird file is the "DynCommPostProcess.R" where it says "document new algorithms here". This documentation should use the same format used for other algorithms with a very small description of the algorithm, preferably just two lines, with a link to the user friendly documentation and references to publications.
6. Add the name of your algorithm to the POSTPROCESSING list in the "DynCommPostProcess.R" file under the marker that says "list new algorithms here".
   Add your algorithm parameters to the matrix in the "DynComm.R" file under the marker that says "add parameters here".

   R Add a source command to the "DynCommPostProcess.R" file under the marker that says "Include R sources here". Add any R libraries required by your algorithm to the "DynComm.R" file under the marker that says "List imports here".
   Remove all libraries from your algorithms' R source file.
7. You should now be able to build the package and, if everything went right, your algorithm is successfuly integrated into this package. **Congratulations :D**

Author(s)

poltergeist0

See Also

DynComm , DynComm-package-dev

POSTPROCESSING_DENSOPT *POSTPROCESSING_DENSOPT*

Description

Implementation of the density optimization algorithm as a post processing algorithm.

Performance

Initialization Uses a matrix with three columns and a maximum of verticelAll()^2 rows with the edges between vertices and their weight (vertex<->vertex<->weight) of the original graph. Temporarily stores a copy of the graph to calculate a new community mapping.

Results Uses a matrix with two columns and verticesAll() rows with the new community mapping (vertex<->community). Uses a matrix with three columns and a maximum of communityCount()^2+communityCount() rows with the edges between communities and their weight (community<->community<->weight).

#' @section PARAMETERS: This post processing algorithm does not require any parameters.

Author(s)

poltergeist0

See Also

DynComm

---

quality *quality*

Description

Get the quality measurement of the graph from the selected post processing algorithm or the main algorithm, after the last iteration.

Usage

```
quality.DynComm(dyncomm)
```

Arguments

dyncomm	A DynComm object, if not using the inline version of the function call

Value

a floating point number

Author(s)

poltergeist0

See Also

DynComm , postProcess

Examples

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
dc$quality()
```

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
quality(dc)
```

---

results *results*

Description

This method returns additional results from the selected post processing algorithm or the main algorithm. See select to know how to select an algorithm.

Usage

```
results.DynComm(dyncomm,differential)
```

Arguments

dyncomm	A DynComm object, if not using the inline version of the function call
differential	If TRUE, only values that have changed in the latest run will be returned

Details

Additional results are any results other than those returned by other existing functions like quality, mytime and communityMapping. Passing the parameter differential set to TRUE, will return only results that have changed from the previous to last iteration.

Value

a two column matrix where, the first column is the name of the result and, the second column is its value.

Author(s)

poltergeist0

See Also

DynComm , postProcess

Examples

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
dc$results()
```

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
results(dc)
```

`select` *select*

Description

This method allows for the selection of which result should be shown. Any of the post processing algorithms and the main algorithm can be choosen.

Usage

```
select.DynComm(dyncomm,postProcessing, id)
```

Arguments

`dyncomm`	A DynComm object, if not using the inline version of the function call
`postProcessing`	The name of the post processing algorithm. Default POSTPROCESSING$NONE. See POSTPROCESSING
`id`	The ID of the post processing algorithm. Default value is 1

Details

The ID parameter is used to distinguish between several post processing algorithms of the same type. It is not required for neither the main algorithm nor any post processing algorithm type that only appears one time.

The main algorithm can be selected with POSTPROCESSING$NONE (default value) and the ID is ignored. See POSTPROCESSING for other available algorithms.

If there are no actions defined for post processing, this function fails.

Value

FALSE if the algorithm does not exist in the chain. Otherwise, TRUE

Author(s)

poltergeist0

See Also

DynComm , postProcess

Examples

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
dc$postProcess(
  list(
    list(POSTPROCESSING$WEIGHTTOP,matrix(data=c("n",10),ncol=2,byrow=TRUE))
    ,list(POSTPROCESSING$DENSOPT)
    ,list(POSTPROCESSING$WEIGHTTOP,matrix(data=c("n",3),ncol=2,byrow=TRUE))
  )
)
# first POSTPROCESSING$WEIGHTTOP gets ID=1 and second gets ID=2
# POSTPROCESSING$DENSOPT uses default parameters
dc$select(POSTPROCESSING$WEIGHTTOP,1)  #selects the results of the first WEIGHTTOP
dc$select(POSTPROCESSING$WEIGHTTOP,2)  #selects the results of the second WEIGHTTOP
dc$select(POSTPROCESSING$NONE)  #selects the main algorithm results


dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
postProcess(dc,
  list(
    list(POSTPROCESSING$WEIGHTTOP,matrix(data=c("n",10),ncol=2,byrow=TRUE))
    ,list(POSTPROCESSING$DENSOPT)
    ,list(POSTPROCESSING$WEIGHTTOP,matrix(data=c("n",3),ncol=2,byrow=TRUE))
  )
)
# first POSTPROCESSING$WEIGHTTOP gets ID=1 and second gets ID=2
# POSTPROCESSING$DENSOPT uses default parameters
select(dc,POSTPROCESSING$WEIGHTTOP,1)  #selects the results of the first WEIGHTTOP
select(dc,POSTPROCESSING$WEIGHTTOP,2)  #selects the results of the second WEIGHTTOP
select(dc,POSTPROCESSING$NONE)  #selects the main algorithm results
```

vertexCount *vertexCount*

Description

Get the total number of vertices from the selected post processing algorithm or the main algorithm, after the last iteration.

Usage

```
vertexCount.DynComm(dyncomm)
nodesCount.DynComm(dyncomm)
```

Arguments

dyncomm	A DynComm object, if not using the inline version of the function call

Details

It can be useful since vertices can be added, if an edge being added has vertices that do not exist in the graph, or removed, if they are not part of any edge after removing an edge.

Value

an unsigned integer with the number of vertices in the graph

Author(s)

poltergeist0

See Also

DynComm , postProcess

Examples

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
dc$vertexCount()
```

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
vertexCount(dc)
```

vertices *vertices*

Description

Get all vertices belonging to the given community from the selected post processing algorithm or the main algorithm, after the last iteration.

Usage

```
vertices.DynComm(dyncomm,community)
nodes.DynComm(dyncomm,community)
```

Arguments

dyncomm	A DynComm object, if not using the inline version of the function call
community	The name of the intended community

Value

a list of vertices belonging to the given community

Author(s)

poltergeist0

See Also

DynComm , postProcess

Examples

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
dc$vertices(6)
```

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
vertices(dc,6)
```

verticesAll *verticesAll*

Description

Get all vertices in the graph from the selected post processing algorithm or the main algorithm, after the last iteration.

Usage

```
verticesAll.DynComm(dyncomm)
nodesAll.DynComm(dyncomm)
```

Arguments

dyncomm	A DynComm object, if not using the inline version of the function call

Value

a list of all vertices in the graph

Author(s)

poltergeist0

See Also

DynComm , postProcess

Examples

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
dc$verticesAll()
```

```
dc<-DynComm(ALGORITHM$LOUVAIN,CRITERION$MODULARITY,parameters)
verticesAll(dc)
```

Index