R topics documented:

astarSearch ........................................... 2
bandwidth ............................................. 3
betweenness.centrality.clustering ..................... 4
bellman.ford.sp ..................................... 5
brandes.betweenness.centrality ....................... 6
bfs .................................................... 7
biConnComp ......................................... 8
boyerMyrvoldPlanarityTest ............................. 9
chrobakPayneStraightLineDrawing .................... 10
clusteringCoefAppr .................................. 10
clusteringCoef ....................................... 12
sequential.vertex.coloring ............................ 13
connectedComp ...................................... 14
dag.sp ................................................ 15
dijkstra.sp .......................................... 16
dominatorTree ....................................... 17
edgeConnectivity .................................... 18
edmondsMaxCardinalityMatching ....................... 19
extractPath ......................................... 20
FileDep ............................................... 21
floyd.warshall.all.pairs.sp .......................... 21
gprofile ............................................. 22
graphGenerator ...................................... 23
highlyConnSG ....................................... 24
incremental.components ............................. 26
isKuratowskiSubgraph ................................ 27
isomorphism ......................................... 28
isStraightLineDrawing ................................ 29
is.triangulated .................................... 29
johnson.all.pairs.sp .................................. 30
kCliques ............................................ 31
kCores ............................................. 32
kolmogorovMaxFlow .................................. 33
lambdaSets .......................................... 34
layout ................................................ 35
lengauerTarjanDominatorTree ......................... 36
makeBiconnectedPlanar ............................... 37
Description

Compute astarSearch for a graph

Usage

astarSearch(g)

Arguments

g an instance of the graph class

Details

NOT IMPLEMENTED YET. TO BE FILLED IN

Author(s)

Li Long <li.long@isb-sib.ch>
bandwidth

References

Boost Graph Library ( www.boost.org/libs/graph/doc/index.html )


Examples

con <- file(system.file("XML/dijkex.gxl",package="RBGL"), open="r")
coex <- fromGXL(con)
close(con)
astarSearch(coex)

bandwidth

Compute bandwidth for an undirected graph

Description

Compute bandwidth for an undirected graph

Usage

bandwidth(g)

Arguments

g an instance of the graph class with edgemode "undirected"

Details

The bandwidth of an undirected graph G=(V, E) is the maximum distance between two adjacent vertices. See documentation on bandwidth in Boost Graph Library for more details.

Value

bandwidth the bandwidth of the given graph

Author(s)

Li Long <li.long@isb-sib.ch>

References

Boost Graph Library ( www.boost.org/libs/graph/doc/index.html )

betweeness.centrality.clustering

Examples

con <- file(system.file("XML/dijkex.gxl", package="RBGL"), open="r")
coex <- fromGXL(con)
close(con)
coex <- ugraph(coex)
bandwidth(coex)

betweenness.centrality.clustering

Description

Graph clustering based on edge betweenness centrality

Usage

betweenness.centrality.clustering(g, threshold = -1, normalize = T)

Arguments

g an instance of the graph class with edgemode "undirected"
threshold threshold to terminate clustering process
normalize boolean, when TRUE, the edge betweenness centrality is scaled by 2/((n-1)(n-2)) where n is the number of vertices in g; when FALSE, the edge betweenness centrality is the absolute value

Details

To implement graph clustering based on edge betweenness centrality.

The algorithm is iterative, at each step it computes the edge betweenness centrality and removes the edge with maximum betweenness centrality when it is above the given threshold. When the maximum betweenness centrality falls below the threshold, the algorithm terminates.

See documentation on Clustering algorithms in Boost Graph Library for details.

Value

A list of

no.of.edges number of remaining edges after removal
edges remaining edges
dge.betweenness.centrality betweenness centrality of remaining edges

Author(s)

Li Long <li.long@isb-sib.ch>


bellman.ford.sp

References

Boost Graph Library (www.boost.org/libs/graph/doc/index.html)


See Also

brandes.betweenness.centrality

Examples

con <- file(system.file("XML/conn.gxl", package="RBGL"))
coex <- fromGXL(con)
close(con)
coex <- ugraph(coex)
betweenness.centrality.clustering(coex, 0.5, TRUE)

bellman.ford.sp Bellman-Ford shortest paths using boost C++

Description

Algorithm for the single-source shortest paths problem for a graph with both positive and negative edge weights.

Usage

bellman.ford.sp(g, start=nodes(g)[1])

Arguments

g instance of class graph
start character: node name for start of path

Details

This function interfaces to the Boost graph library C++ routines for Bellman-Ford shortest paths. Choose the appropriate algorithm to calculate the shortest path carefully based on the properties of the given graph. See documentation on Bellman-Ford algorithm in Boost Graph Library for more details.

Value

A list with elements:
all edges minimized true if all edges are minimized, false otherwise.
distance The vector of distances from start to each node of g; includes Inf when there is no path from start.
penult A vector of indices (in nodes(g)) of predecessors corresponding to each node on the path from that node back to start
start The start node that was supplied in the call to bellman.ford.sp.
Author(s)

Li Long <li.long@isb-sib.ch>

References

Boost Graph Library (www.boost.org/libs/graph/doc/index.html)

See Also

dag.sp, dijkstra.sp, johnson.all.pairs.sp, sp.between

Examples

```r
con <- file(system.file("XML/conn2.gxl",package="RBGL"), open="r")
dd <- fromGXL(con)
close(con)
bellman.ford.sp(dd)
bellman.ford.sp(dd,nodes(dd)[2])
```

---

`brandes.betweenness.centrality`

*Compute betweenness centrality for an undirected graph*

Description

Compute betweenness centrality for an undirected graph

Usage

`brandes.betweenness.centrality(g)`

Arguments

`g` an instance of the `graph` class with `edgemode` “undirected”

Details

`Brandes.betweenness.centrality` computes the betweenness centrality of each vertex or each edge in the graph, using an algorithm by U. Brandes.

Betweenness centrality of a vertex `v` is calculated as follows: \( N_{st}(v) = \text{no. of shortest paths from } s \text{ to } t \text{ that pass through } v, N_{st} = \text{no. of shortest paths from } s \text{ to } t \), betweenness centrality of \( v = \sum (N_{st}(v)/N_{st}) \) for all vertices \( s \neq v \neq t \).

Betweenness centrality of an edge is calculated similarly.

The relative betweenness centrality for a vertex is to scale the betweenness centrality of the given vertex by \( 2/(n^2 - 3n + 2) \) where \( n \) is the no. of vertices in the graph.

Central point dominance measures the maximum betweenness of any vertex in the graph.

See documentation on `brandes.betweenness.centrality` in Boost Graph Library for more details.
Value

A list of

- `betweenness.centrality.vertices`
  betweenness centrality of each vertex
- `betweenness.centrality.edges`
  betweenness centrality of each edge
- `relative.betweenness.centrality.vertices`
  relative betweenness centrality of each vertex
- `dominance`
  maximum betweenness of any point in the graph

Author(s)

Li Long <li.long@isb-sib.ch>

References

Boost Graph Library ( www.boost.org/libs/graph/doc/index.html )


See Also

- `betweenness.centrality.clustering`

Examples

```r
con <- file(system.file("XML/conn.gxl", package="RBGL"), open="r")
coex <- fromGXL(con)
close(con)
coex <- ugraph(coex)
brandes.betweenness.centrality(coex)
```

bfs  

**Breadth and Depth-first search**

Description

These functions return information on graph traversal by breadth and depth first search using routines from the BOOST library.

Usage

```r
bfs(object, node, checkConn=TRUE)
dfs(object, node, checkConn=TRUE)
```
bfs

Arguments

- **object**: instance of class graph from Bioconductor graph class
- **node**: node name where search starts; defaults to the node in first position in the node vector.
- **checkConn**: logical for backwards compatibility; this parameter has no effect as of RBGL 1.7.9 and will be removed in future versions.

Details

These two functions are interfaces to the BOOST graph library functions for breadth first and depth first search. Both methods handle unconnected graphs by applying the algorithms over the connected components.

Cormen et al note (p 542) that ‘results of depth-first search may depend upon the order in which the vertices are examined ... These different visitation orders tend not to cause problems in practice, as any DFS result can usually be used effectively, with essentially equivalent results’.

Value

- For **bfs** a vector of node indices in order of BFS visit.
- For **dfs** a list of two vectors of nodes, with elements **discover** (order of DFS discovery), and **finish** (order of DFS completion).

Author(s)

VJ Carey <stvjc@channing.harvard.edu>

References

- Boost Graph Library ( [www.boost.org/libs/graph/doc/index.html](http://www.boost.org/libs/graph/doc/index.html) )


Examples

```r
con1 <- file(system.file("XML/bfsex.gxl",package="RBGL"), open="r")
dd <- fromGXL(con1)
close(con1)

bfs(dd, "r")
bfs(dd, "s")

con2 <- file(system.file("XML/dfsex.gxl",package="RBGL"), open="r")
dd2 <- fromGXL(con2)
close(con2)

dfs(dd2, "u")
```
Compute biconnected components for a graph

Usage

```r
biConnComp(g)
articulationPoints(g)
```

Arguments

- `g`: an instance of the `graph` class

Details

A biconnected graph is a connected graph that remains connected when any one of its vertices, and all the edges incident on this vertex, is removed and the graph remains connected. A biconnected component of a graph is a subgraph which is biconnected. An integer label is assigned to each edge to indicate which biconnected component it’s in.

A vertex in a graph is called an articulation point if removing it increases the number of connected components.

See the documentation for the Boost Graph Library for more details.

Value

- For `biConnComp`: a vector whose length is no. of biconnected components, each entry is a list of nodes that are on the same biconnected components.
- For `articulationPoints`: a vector of articulation points in the graph.

Author(s)

Li Long <li.long@isb-sib.ch>

References

- Boost Graph Library (www.boost.org/libs/graph/doc/index.html)

Examples

```r
con <= file(system.file("XML/conn.gxl",package="RBGL"), open="r")
coex <= fromGXL(con)
close(con)

biConnComp(coex)
articulationPoints(coex)
```
boyerMyrvoldPlanarityTest

Description
boyerMyrvoldPlanarityTest description

Usage
boyerMyrvoldPlanarityTest(g)

Arguments
g instance of class graphNEL from Bioconductor graph class

Author(s)
Li Long <li.long@isb-sib.ch>

References
Boost Graph Library (www.boost.org/libs/graph/doc/index.html)

chrobakPayneStraightLineDrawing

Description
chrobakPayneStraightLineDrawing description

Usage
chrobakPayneStraightLineDrawing(g)

Arguments
g instance of class graphNEL from Bioconductor graph class

Author(s)
Li Long <li.long@isb-sib.ch>
clusteringCoefAppr

References
Boost Graph Library ( www.boost.org/libs/graph/doc/index.html )

Description
Approximate clustering coefficient for an undirected graph

Usage
clusteringCoefAppr(g, k=length(nodes(g)), Weighted=FALSE, vW=degree(g))

Arguments

  g            an instance of the graph class
  Weighted     calculate weighted clustering coefficient or not
  vW           vertex weights to use when calculating weighted clustering coefficient
  k            parameter controls total expected runtime

Details
It is quite expensive to compute cluster coefficient and transitivity exactly for a large graph by computing the number of triangles in the graph. Instead, clusteringCoefAppr samples triples with appropriate probability, returns the ratio between the number of existing edges and the number of samples.
MORE ABOUT CHOICE OF K.
See reference for more details.

Value
Approximated clustering coefficient for graph g.

Author(s)
Li Long <li.long@isb-sib.ch>

References

See Also
clusteringCoef, transitivity, graphGenerator
Examples

```r
con <- file(system.file("XML/conn.gxl", package="RBGL"))
g <- fromGXL(con)
close(con)

k = length(nodes(g))
cc <- clusteringCoefAppr(g, k)
ccw1 <- clusteringCoefAppr(g, k, Weighted=TRUE)
vW <- c(1, 1, 1, 1, 1, 1)
ccw2 <- clusteringCoefAppr(g, k, Weighted=TRUE, vW)
```

clusteringCoef  
*Calculate clustering coefficient for an undirected graph*

Description

Calculate clustering coefficient for an undirected graph

Usage

```r
clusteringCoef(g, Weighted=FALSE, vW=degree(g))
```

Arguments

- `g`: an instance of the `graph` class
- `Weighted`: calculate weighted clustering coefficient or not
- `vW`: vertex weights to use when calculating weighted clustering coefficient

Details

For an undirected graph $G$, let $\delta(v)$ be the number of triangles with $v$ as a node, let $\tau(v)$ be the number of triples, i.e., paths of length 2 with $v$ as the center node.

Let $V'$ be the set of nodes with degree at least 2.

Define clustering coefficient for $v$, $c(v) = (\delta(v) / \tau(v))$.

Define clustering coefficient for $G$, $C(G) = \sum(c(v)) / |V'|$, for all $v$ in $V'$.

Define weighted clustering coefficient for $G$, $C_w(G) = \sum(w(v) * c(v)) / \sum(w(v))$, for all $v$ in $V'$.

Value

Clustering coefficient for graph $G$.

Author(s)

Li Long (li.long@isb-sib.ch)

References

sequential.vertex.coloring

See Also

clusteringCoefAppr, transitivity, graphGenerator

Examples

```r
c <- file(system.file("XML/conn.gxl", package="RBGL"))
g <- fromGXL(c)
close(c)
cc <- clusteringCoef(g)
ccw1 <- clusteringCoef(g, Weighted=TRUE)
vW <- c(1, 1, 1, 1, 1, 1, 1, 1)
ccw2 <- clusteringCoef(g, Weighted=TRUE, vW)
```

sequential.vertex.coloring

*Compute a vertex coloring for a graph*

Description

Compute vertex coloring for a graph

Usage

```r
sequential.vertex.coloring(g)
```

Arguments

- `g`: an instance of the graph class

Details

A vertex coloring for a graph is to assign a color for each vertex so that no two adjacent vertices are of the same color. We designate the colors as sequential integers: 1, 2, ...

For ordered vertices, v1, v2, ..., vn, for k = 1, 2, ..., n, this algorithm assigns vk to the smallest possible color. It does NOT guarantee to use minimum number of colors.

See documentations on these algorithms in Boost Graph Library for more details.

Value

- no. of colors needed: how many colors to use to color the graph
- colors of nodes: color label for each vertex

Author(s)

Li Long <li.long@isb-sib.ch>
connectedComp

References

Boost Graph Library ( www.boost.org/libs/graph/doc/index.html )

Examples

```r
con <- file(system.file("XML/dijkex.gxl",package="RBGL"), open="r")
coex <- fromGXL(con)
close(con)
sequential.vertex.coloring(coex)
```

connectedComp  Identify Connected Components in an Undirected Graph

Description

The connected components in an undirected graph are identified. If the graph is directed then the weakly connected components are identified.

Usage

```r
connectedComp(g)
```

Arguments

- `g` graph with edgemode “undirected”

Details

Uses a depth first search approach to identifying all the connected components of an undirected graph. If the input, `g`, is a directed graph it is first transformed to an undirected graph (using `ugraph`).

Value

A list of length equal to the number of connected components in `g`. Each element of the list contains a vector of the node labels for the nodes that are connected.

Author(s)

Vince Carey <stvjc@channing.harvard.edu>

References

Boost Graph Library ( www.boost.org/libs/graph/doc/index.html )
dag.sp

See Also

connComp, strongComp, ugraph, same.component

Examples

con <- file(system.file("GXL/kmstEx.gxl", package="graph"), open="r")
km <- fromGXL(con)
close(con)
km <- graph::addNode(c("F", "G", "H"), km)
km <- addEdge("G", "H", km, 1)
km <- addEdge("H", "G", km, 1)
ukm <- ugraph(km)
ukm
edges(ukm)
connectedComp(ukm)

---

**dag.sp**  

**DAG shortest paths using boost C++**

Description

Algorithm for the single-source shortest-paths problem on a weighted, directed acyclic graph (DAG)

Usage

`dag.sp(g, start=nodes(g)[1])`

Arguments

- **g**
  - instance of class graph
- **start**
  - source node for start of paths

Details

These functions are interfaces to the Boost graph library C++ routines for single-source shortest-paths on a weighted directed acyclic graph. Choose appropriate shortest-path algorithms carefully based on the properties of the input graph. See documentation in Boost Graph Library for more details.

Value

A list with elements:

- **distance**
  - The vector of distances from `start` to each node of `g`; includes `Inf` when there is no path from `start`.
- **penult**
  - A vector of indices (in `nodes(g)`) of predecessors corresponding to each node on the path from that node back to `start`. For example, if the element one of this vector has value 10, that means that the predecessor of node 1 is node 10. The next predecessor is found by examining `penult[10]`.
- **start**
  - The start node that was supplied in the call to `dag.sp`. 
dijkstra.sp

Author(s)

Li Long <li.long@isb-sib.ch>

References

Boost Graph Library ( www.boost.org/libs/graph/doc/index.html )
The Boost Graph Library: User Guide and Reference Manual; by Jeremy G. Siek, Lie-Quan Lee, and

See Also

bellman.ford.sp, dijkstra.sp, johnson.all.pairs.sp, sp.between

Examples

con <- file(system.file("XML/conn2.gxl",package="RBGL"), open="r")
dd <- fromGXL(con)
close(con)
dag.sp(dd)
dag.sp(dd,nodes(dd)[2])

dijkstra.sp Dijkstra's shortest paths using boost C++

Description

dijkstra's shortest paths

Usage

dijkstra.sp(g,start=nodes(g)[1], eW=unlist(edgeWeights(g)))

Arguments

g instance of class graph
start character: node name for start of path
eW numeric: edge weights.

Details

These functions are interfaces to the Boost graph library C++ routines for Dijkstra's shortest paths.
For some graph subclasses, computing the edge weights can be expensive. If you are calling
dijkstra.sp in a loop, you can pass the edge weights explicitly to avoid the edge weight creation cost.
dominatorTree

Value

A list with elements:

- **distance**: The vector of distances from `start` to each node of `g`; includes `Inf` when there is no path from `start`.
- **penult**: A vector of indices (in `nodes(g)`) of predecessors corresponding to each node on the path from that node back to `start`.
- **start**: The start node that was supplied in the call to `dijkstra.sp`.

Author(s)

VJ Carey <stvjc@channing.harvard.edu>

References

Boost Graph Library (www.boost.org/libs/graph/doc/index.html)


See Also

- `bellman.ford.sp`, `dag.sp`, `johnson.all.pairs.sp`, `sp.between`

Examples

```r
con1 <- file(system.file("XML/dijkex.gxl",package="RBGL"), open="r")
dd <- fromGXL(con1)
close(con1)
dijkstra.sp(dd)
dijkstra.sp(dd,nodes(dd)[2])

con2 <- file(system.file("XML/ospf.gxl",package="RBGL"), open="r")
ospf <- fromGXL(con2)
close(con2)
dijkstra.sp(ospf,nodes(ospf)[6])
```

dominatorTree  Compute dominator tree from a vertex in a directed graph

Description

Compute dominator tree from a vertex in a directed graph

Usage

```r
dominatorTree(g, start=nodes(g)[1])
```

Arguments

- **g**: a directed graph, one instance of the `graph` class
- **start**: a vertex in graph `g`
As stated in documentation on Lengauer Tarjan dominator tree in Boost Graph Library:
A vertex u dominates a vertex v, if every path of directed graph from the entry to v must go through u.
This function builds the dominator tree for a directed graph.

Value
Output is a vector, giving each node its immediate dominator.

Author(s)
Li Long <li.long@isb-sib.ch>

References
Boost Graph Library (www.boost.org/libs/graph/doc/index.html)

Examples
```r
con1 <- file(system.file("XML/dominator.gxl",package="RBGL"), open="r")
g1 <- fromGXL(con1)
close(con1)
dominatorTree(g1)
```

---

**edgeConnectivity**

**computed edge connectivity and min disconnecting set for an undirected graph**

**Description**
computed edge connectivity and min disconnecting set for an undirected graph

**Usage**

```r
edgeConnectivity(g)
```

**Arguments**

```r
g an instance of the graph class with edgemode "undirected"
```

**Details**
Consider a graph G consisting of a single connected component. The edge connectivity of G is the minimum number of edges in G that can be cut to produce a graph with two (disconnected) components. The set of edges in this cut is called the minimum disconnecting set.
Value

A list:

- `connectivity` the integer describing the number of edges that must be severed to obtain two components
- `minDisconSet` a list (of length `connectivity`) of pairs of node names describing the edges that need to be cut to obtain two components

Author(s)

Vince Carey <stvjc@channing.harvard.edu>

References

Boost Graph Library (www.boost.org/libs/graph/doc/index.html)


See Also

`minCut`, `edmunds.karp.max.flow`, `push.relabel.max.flow`

Examples

```r
con <- file(system.file("XML/conn.gxl",package="RBGL"), open="r")
coex <- fromGXL(con)
close(con)
edgeConnectivity(coex)
```

Description

edmondsMaxCardinalityMatching description

Usage

`edmondsMaxCardinalityMatching(g)`

Arguments

- `g` instance of class `graphNEL` from Bioconductor graph class

Author(s)

Li Long <li.long@isb-sib.ch>
References

Boost Graph Library ( www.boost.org/libs/graph/doc/index.html )


extractPath  

convert a dijkstra.sp predecessor structure into the path joining two nodes

Description

determine a path between two nodes in a graph, using output of dijkstra.sp.

Usage

extractPath(s, f, pens)

Arguments

s  
index of starting node in nodes vector of the graph from which pens was derived

f  
index of ending node in nodes vector

pens  
predecessor index vector as returned in the preds component of dijkstra.sp output

Author(s)

Vince Carey <stvjc@channing.harvard.edu>

References

Boost Graph Library ( www.boost.org/libs/graph/doc/index.html )


See Also

allShortestPaths

Examples

data(FileDep)
dd <- dijkstra.sp(FileDep)
extractPath(1,9,dd$pen)
FileDep

*FileDep: a graphNEL object representing a file dependency dataset example in boost graph library*

**Description**

FileDep: a graphNEL object representing a file dependency dataset example in boost graph library

**Usage**

```r
#data(FileDep)
```

**References**

Boost Graph Library ( [www.boost.org/libs/graph/doc/index.html](http://www.boost.org/libs/graph/doc/index.html) )


**Examples**

```r
# this is how the graph of data(FileDep) was obtained
library(graph)
fd <- file(system.file("XML/FileDep.gxl",package="RBGL"), open="r")
show(fromGXL(fd))
if (require(Rgraphviz))
{
  data(FileDep)
  plot(FileDep)
}
close(fd)
```

---

**floyd.warshall.all.pairs.sp**

*compute shortest paths for all pairs of nodes*

**Description**

compute shortest paths for all pairs of nodes

**Usage**

```r
floyd.warshall.all.pairs.sp(g)
```

**Arguments**

- `g` graph object with edge weights given
Details

Compute shortest paths between every pair of vertices for a dense graph. It works on both undirected and directed graph. The result is given as a distance matrix. The matrix is symmetric for an undirected graph, and asymmetric (very likely) for a directed graph. For a sparse graph, the johnson.all.pairs.sp functions should be used instead.
See documentation on these algorithms in Boost Graph Library for more details.

Value

A matrix of shortest path lengths between all pairs of nodes in the graph.

Author(s)

Li Long <li.long@isb-sib.ch>

References

Boost Graph Library ( www.boost.org/libs/graph/doc/index.html )

See Also

johnson.all.pairs.sp

Examples

```r
con <- file(system.file("XML/conn.gxl", package="RBGL"), open="r")
coex <- fromGXL(con)
close(con)
floyd.warshall.all.pairs.sp(coex)
```

---

**gprofile**

Compute profile for a graph

Description

Compute profile for a graph

Usage

```r
gprofile(g)
```

Arguments

```r
g an instance of the graph class
```

Details

The profile of a given graph is the sum of bandwidths for all the vertices in the graph.
See documentation on this function in Boost Graph Library for more details.
Value

profile the profile of the graph

Author(s)

Li Long <li.long@isb-sib.ch>

References

Boost Graph Library ( www.boost.org/libs/graph/doc/index.html )


Examples

```
con <- file(system.file("XML/dijkex.gxl",package="RBGL"), open="r")
coex <- fromGXL(con)
close(con)
gprofile(coex)
```

---

**graphGenerator**

**Generate an undirected graph with adjustable clustering coefficient**

Description

Generate an undirected graph with adjustable clustering coefficient

Usage

```
graphGenerator(n, d, o)
```

Arguments

- **n**
  - no. of nodes in the generated graph
- **d**
  - parameter for preferential attachment
- **o**
  - parameter for triple generation

Details

The graph generator works according to the preferential attachment rule. It also generates graphs with adjustable clustering coefficient. Parameter `d` specifies how many preferred edges a new node has. Parameter `o` limits how many triples to add to a new node.

See reference for details.
Description

Compute highly connected subgraphs for an undirected graph

Usage

highlyConnSG(g, sat=3, ldv=c(3,2,1))

Arguments

g  an instance of the graph class with edgemode “undirected”
sat  singleton adoption threshold, positive integer
ldv  heuristics to remove lower degree vertex, a decreasing sequence of positive integer
Details

A graph G with n vertices is highly connected if its connectivity \( k(G) > n/2 \). The HCS algorithm partitions a given graph into a set of highly connected subgraphs, by using minimum-cut algorithm recursively. To improve performance, the approach is refined by adopting singletons, removing low degree vertices and merging clusters.

On singleton adoption: after each round of partition, some highly connected subgraphs could be singletons (i.e., a subgraph contains only one node). To reduce the number of singletons, therefore reduce number of clusters, we try to get "normal" subgraphs to "adopt" them. If a singleton, s, has n neighbours in a highly connected subgraph c, and n > sat, we add s to c. To adapt to the modified subgraphs, this adoption process is repeated until no further such adoption.

On lower degree vertices: when the graph has low degree vertices, minimum-cut algorithm will just repeatedly separate these vertices from the rest. To reduce such expensive and non-informative computation, we "remove" these low degree vertices first before applying minimum-cut algorithm. Given \( l dv = (d_1, d_2, ..., d_p) \), \( d[i] > d[i+1] > 0 \), we repeat the following (i from 1 to p): remove all the highly-connected-subgraph found so far; remove vertices with degrees < \( d_i \); find highly-connected-subgraphs; perform singleton adoptions.

The Boost implementation does not support self-loops, therefore we signal an error and suggest that users remove self-loops using the function `removeSelfLoops` function. This change does affect degree, but the original article makes no specific reference to self-loops.

Value

A list of clusters, each is given as vertices in the graph.

Author(s)

Li Long <li.long@isb-sib.ch>

References

A Clustering Algorithm based on Graph Connectivity by E. Hartuv, R. Shamir, 1999.

See Also

`edgeConnectivity`, `minCut`, `removeSelfLoops`

Examples

```r
con <- file(system.file("XML/hcs.gxl",package="RBGL"))
coex <- fromGXL(con)
close(con)

highlyConnSG(coex)
```
incremental.components

Compute connected components for an undirected graph

Description

Compute connected components for an undirected graph

Usage

init.incremental.components(g)
incremental.components(g)
same.component(g, node1, node2)

Arguments

g an instance of the graph class
node1 one vertex of the given graph
node2 another vertex of the given graph

Details

This family of functions work together to calculate the connected components of an undirected graph. The algorithm is based on the disjoint-sets. It works where the graph is growing by adding new edges. Call "init.incremental.components" to initialize the calculation on a new graph. Call "incremental.components" to re-calculate connected components after growing the graph. Call "same.component" to learn if two given vertices are in the same connected components. Currently, the codes can only handle ONE incremental graph at a time. When you start working on another graph by calling "init.incremental.components", the disjoint-sets info on the previous graph is lost. See documentation on Incremental Connected Components in Boost Graph Library for more details.

Value

Output from init.incremental.components is a list of component numbers for each vertex in the graph.
Output from incremental.components is a list of component numbers for each vertex in the graph.
Output from same.component is true if both nodes are in the same connected component, otherwise it’s false.

Author(s)

Li Long <li.long@isb-sib.ch>

References

Boost Graph Library ( www.boost.org/libs/graph/doc/index.html )
isKuratowskiSubgraph

See Also

connComp, connectedComp, strongComp

Examples

con <- file(system.file("XML/conn2.gxl", package="RBGL"), open="r")
coex <- fromGXL(con)
close(con)

init.incremental.components(coex)
incremental.components(coex)
v1 <- 1
v2 <- 5
same.component(coex, v1, v2)

Description

isKuratowskiSubgraph description

Usage

isKuratowskiSubgraph(g)

Arguments

  g instance of class graphNEL from Bioconductor graph class

Author(s)

Li Long <li.long@isb-sib.ch>

References

Boost Graph Library (www.boost.org/libs/graph/doc/index.html)
isomorphism

Compute isomorphism from vertices in one graph to those in another graph

Description

Compute isomorphism from vertices in one graph to those in another graph

Usage

\texttt{isomorphism(g1, g2)}

Arguments

- \texttt{g1} one instance of the \texttt{graph} class
- \texttt{g2} one instance of the \texttt{graph} class

Details

As stated in documentation on isomorphism in Boost Graph Library: An isomorphism is a 1-to-1 mapping of the vertices in one graph to the vertices of another graph such that adjacency is preserved. Another words, given graphs \( G_1 = (V_1,E_1) \) and \( G_2 = (V_2,E_2) \) an isomorphism is a function \( f \) such that for all pairs of vertices \( a,b \) in \( V_1 \), edge \( (a,b) \) is in \( E_1 \) if and only if edge \( (f(a),f(b)) \) is in \( E_2 \).

Value

Output is true if there exists an isomorphism between \( g_1 \) and \( g_2 \), otherwise it's false.

Author(s)

Li Long <li.long@isb-sib.ch>

References

Boost Graph Library (www.boost.org/libs/graph/doc/index.html)


Examples

```r
con1 <- file(system.file("XML/dijkex.gxl",package="RBGL"), open="r")
g1 <- fromGXL(con1)
close(con1)

con2 <- file(system.file("XML/conn2.gxl",package="RBGL"), open="r")
g2 <- fromGXL(con2)
close(con2)

isomorphism(g1, g2)
```
isStraightLineDrawing

Description
isStraightLineDrawing description

Usage
isStraightLineDrawing(g)

Arguments

g instance of class graphNEL from Bioconductor graph class

Author(s)
Li Long <li.long@isb-sib.ch>

References
Boost Graph Library (www.boost.org/libs/graph/doc/index.html)

is.triangulated

Description
Decide if a graph is triangulated

Usage
is.triangulated(g)

Arguments

g an instance of the graph class

Details
An undirected graph $G = (V, E)$ is triangulated (i.e. chordal) if all cycles $[v_1, v_2, ..., v_k]$ of length 4 or more have a chord, i.e., an edge $[v_i, v_j]$ with $j \neq i \pm 1 \pmod{k}$

An equivalent definition of chordal graphs is:
$G$ is chordal iff either $G$ is an empty graph, or there is an $v$ in $V$ such that
1. the neighborhood of $v$ (i.e., $v$ and its adjacent nodes) forms a clique, and
2. recursively, $G-v$ is chordal
The return value is TRUE if \( g \) is triangulated and FALSE otherwise. An error is thrown if the graph is not undirected; you might use `ugraph` to compute the underlying graph.

**Author(s)**
Li Long <li.long@isb-sib.ch>

**References**
Combinatorial Optimization: algorithms and complexity (p. 403) by C. H. Papadimitriou, K. Steiglitz

**Examples**
```r
con1 <- file(system.file("XML/conn.gxl", package="RBGL"), open="r")
coex <- fromGXL(con1)
close(con1)

is.triangulated(coex)

con2 <- file(system.file("XML/hcs.gxl", package="RBGL"), open="r")
coex <- fromGXL(con2)
close(con2)

is.triangulated(coex)
```

**johnson.all.pairs.sp**

*compute shortest path distance matrix for all pairs of nodes*

**Description**
compute shortest path distance matrix for all pairs of nodes

**Usage**

`johnson.all.pairs.sp(g)`

**Arguments**

- `g` graph object for which edgeMatrix and edgeWeights are defined

**Details**
Uses BGL algorithm.

**Value**
matrix of shortest path lengths, read from row node to col node
**k Cliques**

Find all the k-cliques in an undirected graph

**Description**

Find all the k-cliques in an undirected graph

**Usage**

`kCliques(g)`

**Arguments**

- `g`: an instance of the `graph` class

**Details**

Notice that there are different definitions of k-clique in different context.

In computer science, a k-clique of a graph is a clique, i.e., a complete subgraph, of k nodes.

In Social Network Analysis, a k-clique in a graph is a subgraph where the distance between any two nodes is no greater than k.

Here we take the definition in Social Network Analysis.

Let D be a matrix, D[i][j] is the shortest path from node i to node j. Algorithm is outlined as following: (1) use Johnson’s algorithm to fill D; let N = max(D[i][j]) for all i, j; (2) each edge is a 1-clique by itself; (3) for k = 2, ..., N, try to expand each (k-1)-clique to k-clique: (3.1) consider a (k-1)-clique the current k-clique KC; (3.2) repeat the following: if for all nodes j in KC, D[v][j] <= k, add node v to KC; (3.3) eliminate duplicates; (4) the whole graph is N-clique.
kCores

Value
A list of length N; k-th entry (k = 1, ..., N) is a list of all the k-cliques in graph g.

Author(s)
Li Long <li.long@isb-sib.ch>

References

Examples
con <- file(system.file("XML/snacliqueex.gxl", package="RBGL"))
coex <- fromGXL(con)
close(con)
kClique(coex)

kCores

Find all the k-cores in a graph

Description
Find all the k-cores in a graph

Usage
kCores(g, EdgeType=c("in", "out"))

Arguments
g an instance of the graph class
EdgeType what types of edges to be considered when g is directed

Details
A k-core in a graph is a subgraph where each node is adjacent to at least a minimum number, k, of the other nodes in the subgraph.
A k-core in a graph may not be connected.
The core number for each node is the highest k-core this node is in. A node in a k-core will be, by definition, in a (k-1)-core.
The implementation is based on the algorithm by V. Batagelj and M. Zaversnik, 2002.
The example snacoreex.gxl is in the paper by V. Batagelj and M. Zaversnik, 2002.

Value
A vector of the core numbers for all the nodes in g.
Author(s)
Li Long <li.long@isb-sib.ch>

References

Examples
con1 <- file(system.file("XML/snacoreex.gxl",package="RBGL"))
kcoex <- fromGXL(con1)
close(con1)

kCores(kcoex)

con2 <- file(system.file("XML/conn2.gxl",package="RBGL"))
kcoex2 <- fromGXL(con2)
close(con2)

kCores(kcoex2)
kCores(kcoex2, "in")
kCores(kcoex2, "out")

Description
kolmogorovMaxFlow description

Usage
kolmogorovMaxFlow(g)

Arguments

  g           instance of class graphNEL from Bioconductor graph class

Author(s)
Li Long <li.long@isb-sib.ch>

References
Boost Graph Library ( www.boost.org/libs/graph/doc/index.html )
Find all the lambda-sets in an undirected graph

Usage

lambdaSets(g)

Arguments

g an instance of the graph class

Details

From reference (1), p. 270: A set of nodes is a lambda-set if any pair of nodes in the lambda set has larger edge connectivity than any pair of nodes consisting of one node from within the lambda set and a second node from outside the lambda set.

As stated in reference (2), a lambda set is a maximal subset of nodes who have more edge-independent paths connecting them to each other than to outsiders.

A lambda set could be characterized by the minimum edge connectivity $k$ among its members, and could be called lambda-$k$ sets.

Let $N$ be maximum edge connectivity of graph $g$, we output all the lambda-$k$ set for all $k = 1, ..., N$.

Value

Maximum edge connectivity, $N$, of the graph $g$, and A list of length $N$; k-th entry ($k = 1, ..., N$) is a list of all the lambda-$k$ sets in graph $g$.

Author(s)

Li Long <li.long@isb-sib.ch>

References

(1) Social Network Analysis: Methods and Applications. By S. Wasserman and K. Faust, pp. 269.
(2) LS sets, lambda sets and other cohesive subsets. By S. P. Borgatti, M. G. Everett, P. R. Shirey, Social Networks 12 (1990) p. 337-357

Examples

con <- file(system.file("XML/snalambdaex.gxl",package="RBGL"))
coex <- fromGXL(con)
close(con)

lambdaSets(coex)
Layout an undirected graph in 2D

**Usage**

- `circleLayout(g, radius=1)`
- `kamadaKawaiSpringLayout(g, edge_or_side=1, es_length=1)`
- `fruchtermanReingoldForceDirectedLayout(g, width=1, height=1)`
- `randomGraphLayout(g, minX=0, maxX=1, minY=0, maxY=1)`

**Arguments**

- `g`: an instance of the `graph` class with `edgemode` “undirected”
- `radius`: radius of a regular n-polygon
- `edge_or_side`: boolean indicating the length is for an edge or for a side, default is for an edge
- `es_length`: the length of an edge or a side for layout
- `width`: the width of the display area, all x coordinates fall in [-width/2, width/2]
- `height`: the height of the display area, all y coordinates fall in [-height/2, height/2]
- `minX`: minimum x coordinate
- `maxX`: maximum x coordinate
- `minY`: minimum y coordinate
- `maxY`: maximum y coordinate

**Details**

If you want to simply draw a graph, you should consider using package `Rgraphviz`. The layout options in package `Rgraphviz`: `neato`, `circo` and `fdp`, correspond to `kamadaKawaiSpringLayout`, `circleLayout` and `fruchtermanReingoldForceDirectedLayout`, respectively.

Function `circleLayout` layouts the graph with the vertices at the points of a regular n-polygon. The distance from the center of the polygon to each point is determined by the `radius` parameter.

Function `kamadaKawaiSpringLayout` provides Kamada-Kawai spring layout for connected, undirected graphs. User provides either the unit length `e` of an edge in the layout or the length of a side `s` of the display area.

Function `randomGraphLayout` places the points of the graph at random locations.

Function `fruchtermanReingoldForceDirectedLayout` performs layout of unweighted, undirected graphs. It’s a force-directed algorithm. The BGL implementation doesn’t handle disconnected graphs very well, since it doesn’t explicitly give each connected component a region proportional to its size.

See documentation on this function in Boost Graph Library for more details.

**Value**

A (2 x n) matrix, where n is the number of nodes in the graph, each column gives the (x, y)-coordinates for the corresponding node.
Author(s)
Li Long <li.long@isb-sib.ch>

References
Boost Graph Library ( www.boost.org/libs/graph/doc/index.html )

See Also
layoutGraph

Examples
con <- file(system.file("XML/conn.gxl",package="RBGL"), open="r")
coex <- fromGXL(con)
close(con)

coex <- ugraph(coex)
circleLayout(coex)
kamadaKawaiSpringLayout(coex)
randomGraphLayout(coex)
fruchtermanReingoldForceDirectedLayout(coex, 10, 10)

lengauerTarjanDominatorTree
dlengauerTarjanDominatorTree

Description
lengauerTarjanDominatorTree description

Usage
lengauerTarjanDominatorTree(g)

Arguments
g instance of class graphNEL from Bioconductor graph class

Author(s)
Li Long <li.long@isb-sib.ch>
**makeBiconnectedPlanar**

**References**

Boost Graph Library (www.boost.org/libs/graph/doc/index.html)


**makeBiconnectedPlanar description**

**Usage**

```r
makeBiconnectedPlanar(g)
```

**Arguments**

`g`  
instance of class graphNEL from Bioconductor graph class

**Author(s)**

Li Long <li.long@isb-sib.ch>

**makeConnected**

**makeConnected description**

**Usage**

```r
makeConnected(g)
```

**Arguments**

`g`  
instance of class graphNEL from Bioconductor graph class

**References**

Boost Graph Library (www.boost.org/libs/graph/doc/index.html)

Author(s)
Li Long <li.long@isb-sib.ch>

References
Boost Graph Library (www.boost.org/libs/graph/doc/index.html)

makeMaximalPlanar

Description
makeMaximalPlanar description

Usage
makeMaximalPlanar(g)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>an instance of class graphNEL from Bioconductor graph class</td>
</tr>
</tbody>
</table>

Author(s)
Li Long <li.long@isb-sib.ch>

References
Boost Graph Library (www.boost.org/libs/graph/doc/index.html)

maxClique

Find all the cliques in a graph

Description
Find all the cliques in a graph

Usage
maxClique(g)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>an instance of the graph class</td>
</tr>
</tbody>
</table>
Details

Notice the maximum clique problem is NP-complete, which means it cannot be solved by any known polynomial algorithm.

We implemented the algorithm by C. Bron and J. Kerbosch.

Value

maxClique list of all cliques in g

Author(s)

Li Long <li.long@isb-sib.ch>

References


Examples

con1 <- file(system.file("XML/conn.gxl",package="RBGL"), open="r")
coex <- fromGXL(con1)
close(con1)
maxClique(coex)

con2 <- file(system.file("XML/hcs.gxl",package="RBGL"), open="r")
coex <- fromGXL(con2)
close(con2)
maxClique(coex)

max.flow Compute max flow for a directed graph

Description

Compute max flow for a directed graph

Usage

edmunds.karp.max.flow(g, source, sink)
push.relabel.max.flow(g, source, sink)

Arguments

g an instance of the graph class with edgemode “directed”
source node name (character) or node number (int) for the source of the flow
sink node name (character) or node number (int) for the sink of the flow
Details

Given a directed graph $G=(V, E)$ of a single connected component with a vertex source and a vertex sink. Each arc has a positive real valued capacity, currently it’s equivalent to the weight of the arc. The flow of the network is the net flow entering the vertex sink. The maximum flow problem is to determine the maximum possible value for the flow to the sink and the corresponding flow values for each arc.

See documentation on these algorithms in Boost Graph Library for more details.

Value

A list of

- `maxflow` the max flow from source to sink
- `edges` the nodes of the arcs with non-zero capacities
- `flows` the flow values of the arcs with non-zero capacities

Author(s)

Li Long <li.long@isb-sib.ch>

References

Boost Graph Library ( www.boost.org/libs/graph/doc/index.html )


See Also

`minCut`, `edgeConnectivity`

Examples

```r
c <- file(system.file("XML/dijkex.gxl",package="RBGL"), open="r")
g <- fromGXL(c)
close(c)
ans1 <- edmunds.karp.max.flow(g, "B", "D")
ans2 <- edmunds.karp.max.flow(g, 3, 2)  # 3 and 2 equivalent to "C" and "B"
ans3 <- push.relabel.max.flow(g, 2, 4)  # 2 and 4 equivalent to "B" and "D"
ans4 <- push.relabel.max.flow(g, "C", "B")
```
maximumCycleRatio

Description
maximumCycleRatio description

Usage
maximumCycleRatio(g)

Arguments

  g                instance of class graphNEL from Bioconductor graph class

Author(s)
Li Long <li.long@isb-sib.ch>

References
Boost Graph Library ( www.boost.org/libs/graph/doc/index.html )

minCut

Compute min-cut for an undirected graph

Description
Compute min-cut for an undirected graph

Usage
minCut(g)

Arguments

  g                an instance of the graph class with edgemode “undirected”

Details
Given an undirected graph G=(V, E) of a single connected component, a cut is a partition of the set of vertices into two non-empty subsets S and V-S, a cost is the number of edges that are incident on one vertex in S and one vertex in V-S. The min-cut problem is to find a cut (S, V-S) of minimum cost.
For simplicity, the returned subset S is the smaller of the two subsets.
**minimumCycleRatio**

**Value**

A list of
- **mincut** the number of edges to be severed to obtain the minimum cut
- **S** the smaller subset of vertices in the minimum cut
- **V-S** the other subset of vertices in the minimum cut

**Author(s)**

Li Long <li.long@isb-sib.ch>

**References**

Boost Graph Library (www.boost.org/libs/graph/doc/index.html)


**See Also**

`edgeConnectivity`

**Examples**

```r
con <- file(system.file("XML/conn.gxl",package="RBGL"), open="r")
coex <- fromGXL(con)
close(con)

minCut(coex)
```

**Discussion**

minimumCycleRatio description

**Usage**

`minimumCycleRatio(g)`

**Arguments**

- **g** instance of class graphNEL from Bioconductor graph class

**Author(s)**

Li Long <li.long@isb-sib.ch>
References

Boost Graph Library ( www.boost.org/libs/graph/doc/index.html )


mstree.kruskal  
Kruskal’s minimum spanning tree in boost

Description

compute the minimum spanning tree (MST) for a graph and return a representation in matrices

Usage

mstree.kruskal(x)

Arguments

x  instance of class graph

Details

calls to kruskal minimum spanning tree algorithm of Boost graph library

Value

a list

edgeList  a matrix m of dimension 2 by number of edges in the MST, with m[i,j] the jth node in edge i

weights  a vector of edge weights corresponding to the columns of edgeList

nodes  the vector of nodes of the input graph x

Author(s)

VJ Carey <stvjc@channing.harvard.edu>

References

Boost Graph Library ( www.boost.org/libs/graph/doc/index.html )


See Also

prim.minST
Examples

```r
con1 <- file(system.file("XML/kmstEx.gxl", package="RBGL"), open="r")
km <- fromGXL(con1)
close(con1)

mstree.kruskal(km)
edgeData(km, "B", "D", "weight") <- 1.1
edgeData(km, "B", "E", "weight") <- .95
mstree.kruskal(km)

con2 <- file(system.file("XML/telenet.gxl", package="RBGL"), open="r")
km2 <- fromGXL(con2)
close(con2)

m <- mstree.kruskal(km2)
print(sum(m[[2]]))
```

---

**mstree.prim**

*Compute minimum spanning tree for an undirected graph*

**Description**

Compute minimum spanning tree for an undirected graph

**Usage**

```r
mstree.prim(g)
prim.minST(g)
```

**Arguments**

- `g` an instance of the `graph` class with edgedmo “undirected”

**Details**

This is Prim’s algorithm for solving the minimum spanning tree problem for an undirected graph with weighted edges.

The function `prim.minST` is depricated and renamed, use `mstree.prim` instead.

See documentations on this function in Boost Graph Library for more details.

**Value**

A list of

- `edges` the edges that form the minimum spanning tree
- `weights` the total weight of the minimum spanning tree

**Author(s)**

Li Long <li.long@isb-sib.ch>
Ordering

References

Boost Graph Library ( www.boost.org/libs/graph/doc/index.html )


See Also

mstree.kruskal

Examples

```r
con <- file(system.file("XML/conn2.gxl",package="RBGL"))
coex <- fromGXL(con)
close(con)
mstree.prim(coex)
```

Ordering

Compute vertex ordering for an undirected graph

Description

Compute vertex ordering for an undirected graph

Usage

```r
cuthill.mckee.ordering(g)
minDegreeOrdering(g, delta=0)
sloan.ordering(g, w1=1, w2=2)
```

Arguments

- `g`: an instance of the graph class with edgemode “undirected”
- `delta`: Multiple elimination control variable. If it is larger than or equal to zero then multiple elimination is enabled. The value of delta specifies the difference between the minimum degree and the degree of vertices that are to be eliminated.
- `w1`: First heuristic weight for the Sloan algorithm.
- `w2`: Second heuristic weight for the Sloan algorithm.

Details

The following details were obtained from the documentation of these algorithms in Boost Graph Library and readers are referred their for even more detail. The goal of the Cuthill-Mckee (and reverse Cuthill-Mckee) ordering algorithm is to reduce the bandwidth of a graph by reordering the indices assigned to each vertex.

The minimum degree ordering algorithm is a fill-in reduction matrix reordering algorithm.

The goal of the Sloan ordering algorithm is to reduce the profile and the wavefront of a graph by reordering the indices assigned to each vertex.

The goal of the King ordering algorithm is to reduce the bandwidth of a graph by reordering the indices assigned to each vertex.
Value

cuthill.mckee.ordering
returns a list with elements:
reverse cuthill.mckee.ordering
the vertices in the new ordering
original bandwidth
bandwidth before reordering vertices
new bandwidth
bandwidth after reordering of vertices
minDegreeOrdering
return a list with elements:
inverse_permutation
the new vertex ordering, given as the mapping from the new indices to the old indices
permutation
the new vertex ordering, given as the mapping from the old indices to the new indices
sloan.ordering
returns a list with elements:
sloan.ordering
the vertices in the new ordering
bandwidth
bandwidth of the graph after reordering
profile
profile of the graph after reordering
maxWavefront maxWavefront of the graph after reordering
aver.wavefront
aver.wavefront of the graph after reordering
rms.wavefront
rms.wavefront of the graph after reordering

Author(s)
Li Long <li.long@isb-sib.ch>

References
Boost Graph Library ( www.boost.org/libs/graph/doc/index.html )

Examples
con <- file(system.file("XML/dijkex.gxl",package="RBGL"), open="r")
coex <- fromGXL(con)
close(con)

coex <- ugraph(coex)
cuthill.mckee.ordering(coex)
minDegreeOrdering(coex)
sloan.ordering(coex)
planarCanonicalOrdering

Description
planarCanonicalOrdering description

Usage
planarCanonicalOrdering(g)

Arguments

\( g \) instance of class graphNEL from Bioconductor graph class

Author(s)
Li Long <li.long@isb-sib.ch>

References

Boost Graph Library (www.boost.org/libs/graph/doc/index.html)

planarFaceTraversal

Description
planarFaceTraversal description

Usage
planarFaceTraversal(g)

Arguments

\( g \) instance of class graphNEL from Bioconductor graph class

Author(s)
Li Long <li.long@isb-sib.ch>
References

Boost Graph Library ( www.boost.org/libs/graph/doc/index.html )


Description

The RBGL package consists of a number of interfaces to the Boost C++ library for graph algorithms. This page follows, approximately, the chapter structure of the monograph on the Boost Graph Library by Siek et al., and gives hyperlinks to documentation on R functions currently available, along with the names of formal parameters to these functions.

**basicAlgs**

<table>
<thead>
<tr>
<th>Functions</th>
<th>parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>bandwidth</td>
<td>g</td>
</tr>
<tr>
<td>bfs</td>
<td>object,node,checkConn</td>
</tr>
<tr>
<td>dfs</td>
<td>object,node,checkConn</td>
</tr>
<tr>
<td>edgeConnectivity</td>
<td>g</td>
</tr>
<tr>
<td>gprofile</td>
<td>g</td>
</tr>
<tr>
<td>isomorphism</td>
<td>g1,g2</td>
</tr>
<tr>
<td>minCut</td>
<td>g</td>
</tr>
<tr>
<td>transitive.closure</td>
<td>g</td>
</tr>
<tr>
<td>tsort</td>
<td>x</td>
</tr>
</tbody>
</table>

**ShortestPaths**

<table>
<thead>
<tr>
<th>Functions</th>
<th>parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>bellman.ford.sp</td>
<td>g,start</td>
</tr>
<tr>
<td>dag.sp</td>
<td>g,start</td>
</tr>
<tr>
<td>dijkstra.sp</td>
<td>g,start</td>
</tr>
<tr>
<td>extractPath</td>
<td>s,f,pens</td>
</tr>
<tr>
<td>johnson.all.pairs.sp</td>
<td>g</td>
</tr>
<tr>
<td>sp.between</td>
<td>g,start,finish</td>
</tr>
<tr>
<td>sp.between.old</td>
<td>g,start,finish</td>
</tr>
<tr>
<td>sp.between.scalar</td>
<td>g,start,finish</td>
</tr>
</tbody>
</table>

**MinimumSpanningTree**

<table>
<thead>
<tr>
<th>Functions</th>
<th>parameters</th>
</tr>
</thead>
</table>
### ConnectedComponents

<table>
<thead>
<tr>
<th>Functions</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>connectedComp</td>
<td>g</td>
</tr>
<tr>
<td>highlyConnSG</td>
<td>g,sat,ldv</td>
</tr>
<tr>
<td>incremental.components</td>
<td>g</td>
</tr>
<tr>
<td>init.incremental.components</td>
<td>g</td>
</tr>
<tr>
<td>same.component</td>
<td>g</td>
</tr>
<tr>
<td>strongComp</td>
<td>g</td>
</tr>
</tbody>
</table>

### MaximumFlow

<table>
<thead>
<tr>
<th>Functions</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>edmunds.karp.max.flow</td>
<td>g,source,sink</td>
</tr>
<tr>
<td>push.relabel.max.flow</td>
<td>g,source,sink</td>
</tr>
</tbody>
</table>

### SparseMatrixOrdering

<table>
<thead>
<tr>
<th>Functions</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>cuthill.mckee.ordering</td>
<td>g</td>
</tr>
<tr>
<td>minDegreeOrdering</td>
<td>g,delta</td>
</tr>
<tr>
<td>sloan.ordering</td>
<td>g,w1,w2</td>
</tr>
</tbody>
</table>

### LayoutAlgorithms

<table>
<thead>
<tr>
<th>Functions</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle.layout</td>
<td>g,radius</td>
</tr>
<tr>
<td>kamada.kawai.spring.layout</td>
<td>g,edge_or_side,es_length</td>
</tr>
</tbody>
</table>

### GraphClustering

<table>
<thead>
<tr>
<th>Functions</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>betweenness.centrality.clustering</td>
<td>g,threshold,normalize</td>
</tr>
</tbody>
</table>
removeSelfLoops

Betweenness

Functions parameters
brandes.betweenness.centrality g

Wavefront

Functions parameters
aver.wavefront g
ith.wavefront g, start
maxWavefront g
rms.wavefront g

References

Boost Graph Library (www.boost.org/libs/graph/doc/index.html)

Description

remove self loops in a graph

Usage

removeSelfLoops(g)

Arguments

g one instance of the graph class

Details

If a given graph contains self-loop(s), removeSelfLoops removes them. This is for those functions that cannot handle graphs with self-loops.

Value

A new graph without self loops.

Author(s)

Li Long <li.long@isb-sib.ch>
separates

Examples

    con <- file(system.file("XML/dijkex.gxl",package="RBGL"))
    g1 <- fromGXL(con)
    close(con)

    g2 <- ugraph(g1)
    removeSelfLoops(g2)

separates  A function to test whether a subset of nodes separates two other subsets of nodes.

Description

The function tests to see whether a set of nodes, S1, separates all nodes in a from all nodes in b.

Usage

    separates(a, b, S1, g)

Arguments

    a  The names of the nodes in the from set.
    b  The names of the nodes in the to set.
    S1 The names of the nodes in the separation set.
    g  An instance of the graph class. All nodes named in the other arguments must be nodes of this graph.

Details

The algorithm is quite simple. A subgraph is created by removing the nodes named in S1 from g. Then all paths between elements of a to elements of b are tested for. If any path exists the function returns FALSE, otherwise it returns TRUE.

Value

Either TRUE or FALSE depending on whether S1 separates a from b in g1.

Author(s)

R. Gentleman

References

    S. Lauritzen, Graphical Models, OUP.

See Also

    johnson.all.pairs.sp
Examples

```r
con <- file(system.file("XML/kmstEx.gxl",package="RBGL"))
km <- fromGXL(con)
close(con)

separates("B", "A", "E", km)
separates("B", "A", "C", km)
```

---

**sloanStartEndVertices**

**Description**

sloanStartEndVertices description

**Usage**

```r
sloanStartEndVertices(g)
```

**Arguments**

- `g` instance of class graphNEL from Bioconductor graph class

**Author(s)**

Li Long <li.long@isb-sib.ch>

**References**

Boost Graph Library ( www.boost.org/libs/graph/doc/index.html )


---

**sp.between**

Dijkstra’s shortest paths using boost C++

**Description**

dijkstra’s shortest paths

**Usage**

```r
sp.between(g,start,finish, detail=TRUE)
```
**Arguments**

- **g**: instance of class graph
- **start**: node name(s) for start of path(s)
- **finish**: node name(s) for end of path(s)
- **detail**: if TRUE, output additional info on the shortest path

**Details**

These functions are interfaces to the Boost graph library C++ routines for Dijkstra’s shortest paths. Function `sp.between.scalar` is obsolete.

**Value**

When `start` and/or `finish` are vectors, we use the normal cycling rule in R to match both vectors and try to find the shortest path for each pair.

Function `sp.between` returns a list of info on the shortest paths. Each such shortest path is designated by its starting node and its ending node. Each element in the returned list contains:

- **length**: total length (using edge weights) of this shortest path
- **path_detail**: if requested, a vector of names of the nodes on the shortest path
- **length_detail**: if requested, a list of edge weights of this shortest path

See `pathWeights` for caveats about undirected graph representation.

**Author(s)**

VJ Carey <stvjc@channing.harvard.edu>, Li Long <li.long@isb-sib.ch>

**See Also**

`bellman.ford.sp`, `dag.sp`, `dijkstra.sp`, `johnson.all.pairs.sp`

**Examples**

```r
con <- file(system.file("XML/ospf.gxl",package="RBGL"), open="r")
ospf <- fromGXL(con)
close(con)

dijkstra.sp(ospf,nodes(ospf)[6])
sp.between(ospf, "RT6", "RT1")
sp.between(ospf, c("RT6", "RT2"), "RT1", detail=FALSE)
sp.between(ospf, c("RT6", "RT2"), c("RT1","RT5"))

# see NAs for query on nonexistent path
sp.between(ospf,"N10", "N13")
```
Description

The strongly connected components in a directed graph are identified and returned as a list.

Usage

\[ \text{strongComp}(g) \]

Arguments

- \( g \) graph with edgemode “directed”.

Details

Tarjan’s algorithm is used to determine all strongly connected components of a directed graph.

Value

A list whose length is the number of strongly connected components in \( g \). Each element of the list is a vector of the node labels for the nodes in that component.

Author(s)

Vince Carey <stvjc@channing.harvard.edu>

References

- Boost Graph Library ( www.boost.org/libs/graph/doc/index.html )

See Also

\text{connComp, connectedComp, same.component}

Examples

```r
con <- file(system.file("XML/kmstEx.gxl",package="RBGL"), open="r")
km <- fromGXL(con)
close(con)

km<- graph::addNode(c("F","G","H"), km)
km<- addEdge("G", "H", km, 1)
km<- addEdge("H", "G", km, 1)
strongComp(km)

connectedComp(ugraph(km))
```
transitive.closure

Compute transitive closure of a directed graph

Description

Compute transitive closure of a directed graph

Usage

transitive.closure(g)

Arguments

g an instance of the graph class

Details

This function calculates the transitive closure of a directed graph. See documentation on this function in Boost Graph Library for more details.

Value

An object of class graphNEL.

Author(s)

Li Long <li.long@isb-sib.ch>

References

Boost Graph Library (www.boost.org/libs/graph/doc/index.html)


Examples

con <- file(system.file("XML/dijkex.gxl",package="RBGL"))
coex <- fromGXL(con)
close(con)

transitive.closure(coex)
transitivity

Calculate transitivity for an undirected graph

Description

Calculate transitivity for an undirected graph

Usage

transitivity(g)

Arguments

g  an instance of the graph class

Details

For an undirected graph $G$, let $\delta(v)$ be the number of triangles with $v$ as a node, let $\tau(v)$ be the number of triples, i.e., paths of length 2 with $v$ as the center node.

Define transitivity $T(G) = \frac{\sum \delta(v)}{\sum \tau(v)}$, for all $v$ in $V$.

Value

Transitivity for graph $g$.

Author(s)

Li Long (li.long@isb-sib.ch)

References


See Also

clusteringCoef, clusteringCoefAppr, graphGenerator

Examples

con <- file(system.file("XML/conn.gxl",package="RBGL"))
g <- fromGXL(con)
close(con)

tc <- transitivity(g)
tsort

**topological sort of vertices of a digraph**

**Description**
returns vector of zero-based indices of vertices of a DAG in topological sort order

**Usage**
```r
tsord(x) # now x assumed to be Bioconductor graph graphNEL
```

**Arguments**
- `x` instance of class graphNEL from Bioconductor graph class

**Details**
calls to the topological_sort algorithm of BGL. will check in BGL whether the input is a DAG and return a vector of zeroes (of length length(nodes(x))) if it is not. Thus this function can be used to check for cycles in a digraph.

**Value**
a numerical vector enumerating vertices in the topological sort sequence, 0-based

**Author(s)**
VJ Carey <stvjc@channing.harvard.edu>

**References**

Boost Graph Library ( [www.boost.org/libs/graph/doc/index.html](http://www.boost.org/libs/graph/doc/index.html) )


**Examples**
```r
data(FileDep)
tsord <- tsort(FileDep)
tsort
FD2 <- FileDep
# now introduce a cycle
FD2 <- addEdge("bar_o", "dax_h", FD2, 1)
tsort(FD2)
```
Compute the i-th/max/average/rms wavefront for a graph

**Description**

Compute the i-th/max/average/rms wavefront for a graph

**Usage**

ith.wavefront(g, start)
maxWavefront(g)
aver.wavefront(g)
rms.wavefront(g)

**Arguments**

start  
a vertex of the graph class


g  
an instance of the graph class

**Details**

Assorted functions on wavefront of a graph.

**Value**

ith.wavefront  
wavefront of the given vertex

maxWavefront  
maximum wavefront of a graph

aver.wavefront  
average wavefront of a graph

rms.wavefront  
root mean square of all wavefronts

**Author(s)**

Li Long <li.long@isb-sib.ch>

**References**

Boost Graph Library (www.boost.org/libs/graph/doc/index.html)


**See Also**

dgeConnectivity
Examples

```r
con <- file(system.file("XML/dijkstra.gxl", package="RBGL"), open="r")
coex <- fromGXL(con)
close(con)

ss <- 1
ith.wavefront(coex, ss)
maxWavefront(coex)
aver.wavefront(coex)
rms.wavefront(coex)
```
Index

*Topic graphs
bellman.ford.sp, 4
bfs, 6
boyerMyrvoldPlanarityTest, 8
chrobakPayneStraightLineDrawing, 9
dag.sp, 14
dijkstra.sp, 15
edmondsMaxCardinalityMatching, 18
FileDep, 20
isKuratowskiSubgraph, 26
isStraightLineDrawing, 28
kolmogorovMaxFlow, 32
lengauerTarjanDominatorTree, 35
makeBiconnectedPlanar, 36
makeConnected, 36
makeMaximalPlanar, 37
maximumCycleRatio, 40
minimumCycleRatio, 41
mstree.kruskal, 42
planarCanonicalOrdering, 46
planarFaceTraversal, 46
sloanStartEndVertices, 51
sp.between, 51
tsort, 56
floyd.warshall.all.pairs.sp, 20
gprofile, 21
graphGenerator, 22
highlyConnSG, 23
incremental.components, 25
is.triangulated, 28
isomorphism, 27
johnson.all.pairs.sp, 29
kCliqu,es, 30
kCores, 31
lambdaSets, 33
layout, 34
max.flow, 38
maxClique, 37
minCut, 40
mstree.prim, 43
Ordering, 44
RBGL.overview, 47
removeSelfLoops, 49
sequential.vertex.coloring, 12
strongComp, 53
transitiveclosure, 54
transitivity, 55
wavefront, 57
allShortestPaths, 19
articulationPoints(biConnComp), 7
astarSearch, 1
aver.wavefront, 49
aver.wavefront(wavefront), 57
bandwidth, 2, 47
bellman.ford.sp, 4, 15, 16, 30, 47, 52
betweenness.centrality.clustering, 3, 6, 48
bfs, 6, 47
bfs, graph, ANY, ANY-method(bfs), 6
bfs, graph, character, logical-method (bfs), 6
bfs, graph, character, missing-method (bfs), 6

*Topic manip
separates, 50

*Topic models
astarSearch, 1
bandwidth, 2
betweenness.centrality.clustering, 3
biConnComp, 7
brandes.betweenness.centrality, 5
clusteringCoe, 11
clusteringCoeAppr, 9
connectedComp, 13
dominatorTree, 16
dominate, 17
extractPath, 19

60
INDEX

bfs, graph, character-method (bfs), 6
bfs, graph, missing, logical-method (bfs), 6
bfs, graph, missing, missing-method (bfs), 6
bfs, graph-method (bfs), 6
biConnComp, 7
boyerMyrvoldPlanarityTest, 8
brandes.betweenness.centrality, 3, 5, 49
chrobakPayneStraightLineDrawing, 9
circle.layout, 48
circle.layout (layout), 34
circleLayout (layout), 34
clusteringCoef, 11
clusteringCoefAppr, 9
connComp, 13, 26, 53
connectedComp, 13, 26, 48, 53
cuthill.mckee.ordering, 48
cuthill.mckee.ordering (Ordering), 44
dag.sp, 5, 14, 16, 30, 47, 52
dfs, 47
dfs (bfs), 6
dfs, graph, character, ANY-method (bfs), 6
dfs, graph, character, logical-method (bfs), 6
dfs, graph, character, missing-method (bfs), 6
dfs, graph, character-method (bfs), 6
dfs, graph, missing, missing-method (bfs), 6
dijkstra.sp, 5, 15, 19, 30, 47, 52
dominatorTree, 16
dgeConnectivity, 17, 24, 39, 41, 47, 57
edmondsMaxCardinalityMatching, 18
edmunds.karp.max.flow, 18, 48
edmunds.karp.max.flow (max.flow), 38
extractPath, 19, 47
FileDep, 20
floyd.warshall.all.pairs.sp, 20
fruchtermanReingoldForceDirectedLayout (layout), 34
gprofile, 21, 47
graphGenerator, 22
gursoyAtunLayout (layout), 34
highlyConnSG, 23, 48
incremental.components, 25, 48
init.incremental.components, 48
init.incremental.components (incremental.components), 25
is.triangulated, 28
isKuratowskiSubgraph, 26
isomorphism, 27, 47
isStraightLineDrawing, 28
ith.wavefront, 49
ith.wavefront (wavefront), 57
johnson.all.pairs.sp, 5, 15, 16, 21, 29, 47, 50, 52
kamada.kawai.spring.layout, 48
kamada.kawai.spring.layout (layout), 34
kamadaKawaiSpringLayout (layout), 34
kClique, 30
kCores, 31
kingOrdering (Ordering), 44
kolmogorovMaxFlow, 32
lambdaSets, 33
layout, 34
layoutGraph, 35
lengauerTarjanDominatorTree, 35
makeBiconnectedPlanar, 36
makeConnected, 36
makeMaximalPlanar, 37
max.flow, 38
maxClique, 37
maximumCycleRatio, 40
maxWavefront, 49
maxWavefront (wavefront), 57
minCut, 18, 24, 39, 40, 47
minDegreeOrdering, 48
minDegreeOrdering (Ordering), 44
minimumCycleRatio, 41
mstree.kruskal, 42, 44, 48
mstree.prim, 43
pathWeights, 52
planarCanonicalOrdering, 46
planarFaceTraversal, 46
prim.minST, 42, 48
prim.minST (mstree.prim), 43
push.relabel.max.flow, 18, 48
push.relabel.max.flow (max.flow), 38
randomGraphLayout (layout), 34
RBGL.overview, 47
removeSelfLoops, 24, 49
rms.wavefront, 49
rms.wavefront (wavefront), 57
same.component, 13, 48, 53
same.component
  (incremental.components), 25
separates, 50
sequential.vertex.coloring, 12
sloan.ordering, 48
sloan.ordering (Ordering), 44
sloanStartEndVertices, 51
sp.between, 5, 15, 16, 30, 47, 51
sp.between.old, 47
sp.between.scalar, 47
strongComp, 13, 26, 48, 53
transitive.closure, 47, 54
transitivity, 55
tsort, 47, 56
ugraph, 13, 29
wavefront, 57