Package ‘RBGL’

December 21, 2024

Version 1.83.0

Title An interface to the BOOST graph library

Depends graph, methods

Imports methods

Suggests Rgraphviz, XML, RUnit, BiocGenerics, BiocStyle, knitr

Description A fairly extensive and comprehensive interface to the graph algorithms contained in the BOOST library.

License Artistic-2.0

URL http://www.bioconductor.org

LazyLoad yes

biocViews GraphAndNetwork, Network

LinkingTo BH

VignetteBuilder knitr

git_url https://git.bioconductor.org/packages/RBGL

git_branch devel

git_last_commit e0f422c

git_last_commit_date 2024-11-02

Repository Bioconductor 3.21

Date/Publication 2024-12-20

Author Vince Carey [aut],
Li Long [aut],
R. Gentleman [aut],
Emmanuel Taiwo [ctb] (Converted RBGL vignette from Sweave to RMarkdown / HTML.),
Bioconductor Package Maintainer [cre]

Maintainer Bioconductor Package Maintainer <maintainer@bioconductor.org>

Contents

astarSearch . . . 3
bandwidth . . . 4
bellman.ford.sp . . . 5
betweenness.centrality.clustering . . . 6
bfs . . . 7
biConnComp . . . 8
boyerMyrvoldPlanarityTest . . . 10
brandes.betweenness.centrality . . . 11
chrobakPayneStraightLineDrawing . . . 12
clusteringCoef . . . 13
clusteringCoefAppr . . . 14
connectedComp . . . 16
dag.sp . . . 17
dijkstra.sp . . . 18
dominatorTree . . . 20
edgeConnectivity . . . 21
edmondsMaxCardinalityMatching . . . 22
edmondsOptimumBranching . . . 23
extractPath . . . 24
FileDep . . . 25
floyd.warshall.all.pairs.sp . . . 26
gprofile . . . 27
graphGenerator . . . 28
highlyConnSG . . . 29
incremental.components . . . 30
is.triangulated . . . 32
isKuratowskiSubgraph . . . 33
isomorphism . . . 34
isStraightLineDrawing . . . 35
johnson.all.pairs.sp . . . 36
kCliques . . . 37
kCores . . . 38
lambdaSets . . . 39
layout . . . 40
makeBiconnectedPlanar . . . 42
makeConnected . . . 44
makeMaximalPlanar . . . 45
max.flow . . . 46
maxClique . . . 47
maximumCycleRatio . . . 48
minCut . . . 49
minimumCycleRatio . . . 50
mstree.kruskal . . . 51
mstree.prim . . . 52
Ordering . . . 53
planarCanonicalOrdering . . . 55

planarFaceTraversal 56
RBGL-defunct 57
RBGL.overview 58
removeSelfLoops 61
separates 62
sequential.vertex.coloring 63
sloanStartEndVertices 64
sp.between 65
strongComp 66
transitive.closure 67
transitivity 68
tsort 69
wavefront 70

Index **72**

`astarSearch` *Compute astarSearch for a graph*

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

Value

a string indicating non-implementation

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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

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)

The Boost Graph Library: User Guide and Reference Manual; by Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

Examples

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

`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`

. 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 `bellman.ford.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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

See Also

dag.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)
bellman.ford.sp(dd)
bellman.ford.sp(dd,nodes(dd)[2])
```

---

betweenness.centrality.clustering

Graph clustering based on edge betweenness centrality

Description

Graph clustering based on edge betweenness centrality

Usage

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

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
`edge.betweenness.centrality`	betweenness centrality of remaining edges

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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

See Also

`brandes.betweenness.centrality`

`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

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

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)

The Boost Graph Library: User Guide and Reference Manual; by Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

Examples

```
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")
```

`biConnComp` *Compute biconnected components for a graph*

Description

Compute biconnected components for a graph

Usage

```
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)

The Boost Graph Library: User Guide and Reference Manual; by Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

Examples

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

biConnComp(coex)
articulationPoints(coex)
```

boyerMyrvoldPlanarityTest

boyerMyrvoldPlanarityTest

Description

boyerMyrvoldPlanarityTest description

Usage

```
boyerMyrvoldPlanarityTest(g)
```

Arguments

g	instance of class graphNEL from Bioconductor graph class

Value

logical, TRUE if test succeeds

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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

Examples

```
con <- file(system.file("XML/dijkex.gxl",package="RBGL"), open="r")
coex <- fromGXL(con)
edgemode(coex) = "undirected"
boyerMyrvoldPlanarityTest(coex)  # only shows runnability, need better case
```

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) = no. of shortest paths from s to t that pass through v, N_st = no. of shortest paths from s to t, betweenness centrality of v = sum(N_st(v)/N_st) for all vertices s != v != 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)

The Boost Graph Library: User Guide and Reference Manual; by Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

See Also

betweenness.centrality.clustering

---

chrobakPayneStraightLineDrawing *chrobakPayneStraightLineDrawing*

Description

chrobakPayneStraightLineDrawing description

Usage

```
chrobakPayneStraightLineDrawing(g)
```

Arguments

g	instance of class graphNEL from Bioconductor graph class

Details

M. Chrobak, T. Payne, A Linear-time Algorithm for Drawing a Planar Graph on the Grid, Information Processing Letters 54: 241-246, 1995.

Value

A matrix with rows 'x' and 'y', and columns corresponding to graph nodes.

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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

Examples

```
V <- LETTERS[1:7]
g <- new("graphNEL", nodes=V, edgemode="undirected")
g <- addEdge(V[1+0], V[1+1], g)
g <- addEdge(V[1+1], V[2+1], g)
g <- addEdge(V[1+2], V[3+1], g)
g <- addEdge(V[1+3], V[0+1], g)
g <- addEdge(V[1+3], V[4+1], g)
g <- addEdge(V[1+4], V[5+1], g)
g <- addEdge(V[1+5], V[6+1], g)
g <- addEdge(V[1+6], V[3+1], g)
g <- addEdge(V[1+0], V[4+1], g)
g <- addEdge(V[1+1], V[3+1], g)
g <- addEdge(V[1+3], V[5+1], g)
g <- addEdge(V[1+2], V[6+1], g)
g <- addEdge(V[1+1], V[4+1], g)
g <- addEdge(V[1+1], V[5+1], g)
g <- addEdge(V[1+1], V[6+1], g)

x3 <- chrobakPayneStraightLineDrawing(g)
x3
plot(t(x3))
el = edgeL(g)
for (i in seq_len(length(nodes(g))))
  segments(
   rep(x3["x",i], length(el[[i]]$edges)),
   rep(x3["y",i], length(el[[i]]$edges)),
   x3["x", nodes(g)[el[[i]]$edges]],
   x3["y", nodes(g)[el[[i]]$edges]]
  )
```

`clusteringCoef` *Calculate clustering coefficient for an undirected graph*

Description

Calculate clustering coefficient for an undirected graph

Usage

```
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, Cw(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

Approximating Clustering Coefficient and Transitivity, T. Schank, D. Wagner, Journal of Graph Algorithms and Applications, Vol. 9, No. 2 (2005).

See Also

clusteringCoefAppr, transitivity, graphGenerator

Examples

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

---

clusteringCoefAppr *Approximate clustering coefficient for an undirected graph*

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

Approximating Clustering Coefficient and Transitivity, T. Schank, D. Wagner, Journal of Graph Algorithms and Applications, Vol. 9, No. 2 (2005).

See Also

clusteringCoef, transitivity, graphGenerator

Examples

```
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, 1, 1)
ccw2 <- clusteringCoefAppr(g, k, Weighted=TRUE, vW)
```

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

```
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)

The Boost Graph Library: User Guide and Reference Manual; by Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

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`.

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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

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.

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 `dijkstra.sp`.

Author(s)

VJ Carey <stvjc@channing.harvard.edu>

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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

See Also

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

Examples

```
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

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

Arguments

`g`	a directed graph, one instance of the `graph` class
`start`	a vertex in graph g

Details

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

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

dominatorTree(g1)
lengauerTarjanDominatorTree(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

```
edgeConnectivity(g)
```

Arguments

`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)

The Boost Graph Library: User Guide and Reference Manual; by Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

See Also

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

Examples

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

edgeConnectivity(coex)
```

---

edmondsMaxCardinalityMatching

edmondsMaxCardinalityMatching

Description

edmondsMaxCardinalityMatching description

Usage

```
edmondsMaxCardinalityMatching(g)
```

Arguments

g	instance of class graphNEL from Bioconductor graph class

Value

a list with two components: a logical named 'Is max matching' and a character matrix named 'Matching' with two rows 'vertex' and 'matched vertex', entries are node labels.

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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

Examples

```
V <- LETTERS[1:18]
g <- new("graphNEL", nodes=V, edgemode="undirected")
g <- addEdge(V[1+0], V[4+1], g);
g <- addEdge(V[1+1], V[5+1], g);
g <- addEdge(V[1+2], V[6+1], g);
g <- addEdge(V[1+3], V[7+1], g);
g <- addEdge(V[1+4], V[5+1], g);
```

```
g <- addEdge(V[1+6], V[7+1], g);
g <- addEdge(V[1+4], V[8+1], g);
g <- addEdge(V[1+5], V[9+1], g);
g <- addEdge(V[1+6], V[10+1], g);
g <- addEdge(V[1+7], V[11+1], g);
g <- addEdge(V[1+8], V[9+1], g);
g <- addEdge(V[1+10], V[11+1], g);
g <- addEdge(V[1+8], V[13+1], g);
g <- addEdge(V[1+9], V[14+1], g);
g <- addEdge(V[1+10], V[15+1], g);
g <- addEdge(V[1+11], V[16+1], g);
g <- addEdge(V[1+14], V[15+1], g);

x9 <- edmondsMaxCardinalityMatching(g)
x9

g <- addEdge(V[1+12], V[13+1], g);
g <- addEdge(V[1+16], V[17+1], g);

x10 <- edmondsMaxCardinalityMatching(g)
x10
```

`edmondsOptimumBranching`
edmondsOptimumBranching

Description

edmondsOptimumBranching description

Usage

```
edmondsOptimumBranching(g)
```

Arguments

g	instance of class graphNEL from Bioconductor graph class

Details

This is an implementation of Edmonds' algorithm to find optimum branching in a directed graph. See references for details.

Value

A list with three elements: edgeList, weights, and nodes for the optimum branching traversal

Author(s)

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

References

See Edmonds' Algorithm on https://github.com/atofigh/edmonds-alg

Examples

```
V <- LETTERS[1:4]
g <- new("graphNEL", nodes=V, edgemode="directed")
g <- addEdge(V[1+0],V[1+1],g, 3)
g <- addEdge(V[1+0],V[2+1],g, 1.5)
g <- addEdge(V[1+0],V[3+1],g, 1.8)
g <- addEdge(V[1+1],V[2+1],g, 4.3)
g <- addEdge(V[1+2],V[3+1],g, 2.2)

x11 <- edmondsOptimumBranching(g)
x11
```

---

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

Value

numeric vector of indices of nodes along shortest path

Author(s)

Vince Carey <stvjc@channing.harvard.edu>

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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

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

```
data(FileDep)
```

Value

an instance of graphNEL

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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

Examples

```
# 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

```
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)

The Boost Graph Library: User Guide and Reference Manual; by Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

See Also

`johnson.all.pairs.sp`

Examples

```
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

```
gprofile(g)
```

Arguments

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)

The Boost Graph Library: User Guide and Reference Manual; by Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

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 prefential 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.

Value

`no. of nodes`	No. of nodes in the generated graph
`no. of edges`	No. of edges in the generated graph
`edges`	Edges in the generated graph

Author(s)

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

References

Approximating Clustering Coefficient and Transitivity, T. Schank, D. Wagner, Journal of Graph Algorithms and Applications, Vol. 9, No. 2 (2005).

See Also

clusteringCoef, transitivity, clusteringCoefAppr

Examples

```
n <- 20
d <- 6
o <- 3
gg <- graphGenerator(n, d, o)
```

`highlyConnSG` *Compute highly connected subgraphs for an undirected graph*

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 vertice, 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 ldv = (d1, d2, ..., dp), (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 < di; 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

```
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)

The Boost Graph Library: User Guide and Reference Manual; by Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

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)
```

`is.triangulated` *Decide if a graph 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 [v1, v2, ..., vk] of length 4 or more have a chord, i.e., an edge [vi, vj] with j != i +/- 1 (mod 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

Value

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

```
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)
```

---

isKuratowskiSubgraph *isKuratowskiSubgraph*

Description

isKuratowskiSubgraph description

Usage

```
isKuratowskiSubgraph(g)
```

Arguments

g	instance of class graphNEL from Bioconductor graph class

Value

a list with three elements: 'Is planar' (logical), 'Is there a Kuratowski subgraph' (logical), and a two-row character matrix 'Edges of Kuratowski subgraph' with rows 'from' and 'two' and node names as entries.

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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

Examples

```
V <- LETTERS[1:6]
g <- new("graphNEL", nodes=V, edgemode="undirected")
g <- addEdge(V[1+0], V[1+1], g)
g <- addEdge(V[1+0], V[2+1], g)
g <- addEdge(V[1+0], V[3+1], g)
g <- addEdge(V[1+0], V[4+1], g)
g <- addEdge(V[1+0], V[5+1], g)
g <- addEdge(V[1+1], V[2+1], g)
g <- addEdge(V[1+1], V[3+1], g)
g <- addEdge(V[1+1], V[4+1], g)
```

```
g <- addEdge(V[1+1], V[5+1], g)
g <- addEdge(V[1+2], V[3+1], g)
g <- addEdge(V[1+2], V[4+1], g)
g <- addEdge(V[1+2], V[5+1], g)
g <- addEdge(V[1+3], V[4+1], g)
g <- addEdge(V[1+3], V[5+1], g)
g <- addEdge(V[1+4], V[5+1], g)

x4 <- isKuratowskiSubgraph(g)
x4
```

`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

```
isomorphism(g1, g2)
```

Arguments

g1	one instance of the `graph` class
g2	one instance of the `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 G1 = (V1,E1) and G2 = (V2,E2) an isomorphism is a function f such that for all pairs of vertices a,b in V1, edge (a,b) is in E1 if and only if edge (f(a),f(b)) is in E2.

Value

Output is true if there exists an isomorphism between g1 and g2, 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)

The Boost Graph Library: User Guide and Reference Manual; by Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

Examples

```
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` *isStraightLineDrawing*

Description

isStraightLineDrawing description

Usage

```
isStraightLineDrawing(g, drawing)
```

Arguments

`g`	instance of class graphNEL from Bioconductor graph class
`drawing`	coordinates of node positions

Value

logical, TRUE if `drawing` is a straight line layout for `g`

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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

Examples

```
V <- LETTERS[1:7]
g <- new("graphNEL", nodes=V, edgemode="undirected")
g <- addEdge(V[1+0], V[1+1], g)
g <- addEdge(V[1+1], V[2+1], g)
g <- addEdge(V[1+2], V[3+1], g)
g <- addEdge(V[1+3], V[0+1], g)
g <- addEdge(V[1+3], V[4+1], g)
g <- addEdge(V[1+4], V[5+1], g)
g <- addEdge(V[1+5], V[6+1], g)
g <- addEdge(V[1+6], V[3+1], g)
g <- addEdge(V[1+0], V[4+1], g)
g <- addEdge(V[1+1], V[3+1], g)
g <- addEdge(V[1+3], V[5+1], g)
g <- addEdge(V[1+2], V[6+1], g)
g <- addEdge(V[1+1], V[4+1], g)
g <- addEdge(V[1+1], V[5+1], g)
g <- addEdge(V[1+1], V[6+1], g)


x3 <- chrobakPayneStraightLineDrawing(g)

x8 <- isStraightLineDrawing(g, x3)
x8
```

`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

Author(s)

Vince Carey <stvjc@channing.harvard.edu>

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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

See Also

bellman.ford.sp, dag.sp, dijkstra.sp, sp.between

Examples

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

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

---

kCliques — *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.

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

Social Network Analysis: Methods and Applications. By S. Wasserman and K. Faust, pp. 258.

Examples

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

kCliques(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

Social Network Analysis: Methods and Applications. By S. Wasserman and K. Faust, pp. 266. An O(m) Algorithm for Cores decomposition of networks, by V. Batagelj and M. Zaversnik, 2002.

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")
```

---

lambdaSets *Find all the lambda-sets in an undirected graph*

Description

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 lamda set and a second node from outside the lamda 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 — *Layout an undirected graph in 2D – suspended june 16 2012*

Description

Layout an undirected graph in 2D – suspended june 16 2012

Usage

```
circleLayout(g, radius=1) # does not compile with boost 1.49
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 dislay 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)

The Boost Graph Library: User Guide and Reference Manual; by Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

See Also

layoutGraph

Examples

```
## Not run:
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)

## End(Not run)
```

---

makeBiconnectedPlanar *makeBiconnectedPlanar*

Description

makeBiconnectedPlanar description

Usage

```
makeBiconnectedPlanar(g)
```

Arguments

g	instance of class graphNEL from Bioconductor graph class

Details

From

http://www.boost.org/doc/libs/1_49_0/libs/graph/doc/planar_graphs.html

An undirected graph is connected if, for any two vertices u and v, there's a path from u to v. An undirected graph is biconnected if it is connected and it remains connected even if any single vertex is removed. Finally, a planar graph is maximal planar (also called triangulated) if no additional edge (with the exception of self-loops and parallel edges) can be added to it without creating a non-planar graph. Any maximal planar simple graph on n > 2 vertices has exactly 3n - 6 edges and 2n - 4 faces, a consequence of Euler's formula. If a planar graph isn't connected, isn't biconnected, or

isn't maximal planar, there is some set of edges that can be added to the graph to make it satisfy any of those three properties while preserving planarity. Many planar graph drawing algorithms make at least one of these three assumptions about the input graph, so there are functions in the Boost Graph Library that can help:

make_connected adds a minimal set of edges to an undirected graph to make it connected.

make_biconnected_planar adds a set of edges to a connected, undirected planar graph to make it biconnected while preserving planarity.

make_maximal_planar adds a set of edges to a biconnected, undirected planar graph to make it maximal planar.

The function documented here implements the second approach.

Value

A list with two elements: 'Is planar' is a logical indicating achievement of planarity, and 'new graph', a graphNEL instance that is biconnected and planar.

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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

Examples

```
V <- LETTERS[1:11]
g <- new("graphNEL", nodes=V, edgemode="undirected")
g <- addEdge(V[1+0], V[1+1], g)
g <- addEdge(V[1+2], V[3+1], g)
g <- addEdge(V[1+3], V[0+1], g)
g <- addEdge(V[1+3], V[4+1], g)
g <- addEdge(V[1+4], V[5+1], g)
g <- addEdge(V[1+5], V[3+1], g)
g <- addEdge(V[1+5], V[6+1], g)
g <- addEdge(V[1+6], V[7+1], g)
g <- addEdge(V[1+7], V[8+1], g)
g <- addEdge(V[1+8], V[5+1], g)
g <- addEdge(V[1+8], V[9+1], g)
g <- addEdge(V[1+0], V[10+1], g)

x6 <- makeBiconnectedPlanar(g)
x6
```

makeConnected *makeConnected*

Description

makeConnected description

Usage

```
makeConnected(g)
```

Arguments

g	instance of class graphNEL from Bioconductor graph class

Details

a graphNEL instance with a minimal set of edges added to achieve connectedness.

Value

an instance of graphNEL

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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

Examples

```
V <- LETTERS[1:11]
g <- new("graphNEL", nodes=V, edgemode="undirected")
g <- addEdge(V[1+0], V[1+1], g)
g <- addEdge(V[1+2], V[3+1], g)
g <- addEdge(V[1+3], V[4+1], g)
g <- addEdge(V[1+5], V[6+1], g)
g <- addEdge(V[1+6], V[7+1], g)
g <- addEdge(V[1+8], V[9+1], g)
g <- addEdge(V[1+9], V[10+1], g)
g <- addEdge(V[1+10], V[8+1], g)


x5 <- makeConnected(g)
```

```
x5
```

---

makeMaximalPlanar *makeMaximalPlanar*

Description

makeMaximalPlanar description

Usage

```
makeMaximalPlanar(g)
```

Arguments

g	instance of class graphNEL from Bioconductor graph class

Details

see http://www.boost.org/doc/libs/1_49_0/libs/graph/doc/planar_graphs.html

Value

a list with two elements, 'Is planar:', a logical indicating state of graph, and 'new graph', a graphNEL instance

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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

Examples

```
V <- LETTERS[1:10]
g <- new("graphNEL", nodes=V, edgemode="undirected")
g <- addEdge(V[1+0], V[1+1], g)
g <- addEdge(V[1+1], V[2+1], g)
g <- addEdge(V[1+2], V[3+1], g)
g <- addEdge(V[1+3], V[4+1], g)
g <- addEdge(V[1+4], V[5+1], g)
g <- addEdge(V[1+5], V[6+1], g)
```

```
g <- addEdge(V[1+6], V[7+1], g)
g <- addEdge(V[1+7], V[8+1], g)
g <- addEdge(V[1+8], V[9+1], g)

x7 <- makeMaximalPlanar(g)
x7
```

`max.flow` *Compute max flow for a directed graph*

Description

Compute max flow for a directed graph

Usage

```
edmonds.karp.max.flow(g, source, sink)
push.relabel.max.flow(g, source, sink)
kolmogorov.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)

The Boost Graph Library: User Guide and Reference Manual; by Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

See Also

minCut, edgeConnectivity

Examples

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

ans1 <- edmonds.karp.max.flow(g, "B", "D")
ans2 <- edmonds.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")

# error in the following  now, 14 june 2014
#ans5 <- kolmogorov.max.flow(g, "B", "D")
#ans6 <- kolmogorov.max.flow(g, 3, 2)
```

---

maxClique *Find all the cliques in a graph*

Description

Find all the cliques in a graph

Usage

```
maxClique(g, nodes=NULL, edgeMat=NULL)
```

Arguments

g	an instance of the graph class
nodes	vector of node names, to be supplied if g is not
edgeMat	2 x p matrix with indices of edges in nodes, one-based, only to be supplied if codeg is not

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,

It is an error to supply both g and either of the other arguments.

If g is not supplied, no checking of the consistency of `nodes` and `edgeMat` is performed.

Value

`maxClique`	list of all cliques in `g`

Author(s)

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

References

Finding all cliques of an undirected graph, by C. Bron and J. Kerbosch, Communication of ACM, Sept 1973, Vol 16, No. 9.

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)
```

`maximumCycleRatio` *maximumCycleRatio*

Description

maximumCycleRatio description

Usage

```
maximumCycleRatio(g)
```

Arguments

`g`	instance of class graphNEL from Bioconductor graph class

Details

NOT IMPLEMENTED

Value

A list with message indicating not implemented

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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

`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.

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)

The Boost Graph Library: User Guide and Reference Manual; by Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

See Also

edgeConnectivity

Examples

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

minCut(coex)
```

---

minimumCycleRatio *minimumCycleRatio*

Description

minimumCycleRatio description

Usage

```
minimumCycleRatio(g)
```

Arguments

g	instance of class graphNEL from Bioconductor graph class

Details

Not yet implemented.

Value

a list with message

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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

`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)

The Boost Graph Library: User Guide and Reference Manual; by Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

Examples

```
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

```
mstree.prim(g)
```

Arguments

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

Details

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

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>

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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

See Also

mstree.kruskal

Examples

```
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

```
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)

The Boost Graph Library: User Guide and Reference Manual; by Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

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 *planarCanonicalOrdering*

Description

planarCanonicalOrdering description

Usage

```
planarCanonicalOrdering(g)
```

Arguments

g	instance of class graphNEL from Bioconductor graph class

Details

see http://www.boost.org/doc/libs/1_49_0/libs/graph/doc/planar_graphs.html

Value

A vector of ordered node names

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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

Examples

```
V <- LETTERS[1:6]
g <- new("graphNEL", nodes=V, edgemode="undirected")
g <- addEdge(V[1+0], V[1+1], g)
g <- addEdge(V[1+1], V[2+1], g)
g <- addEdge(V[1+2], V[3+1], g)
g <- addEdge(V[1+3], V[4+1], g)
g <- addEdge(V[1+4], V[5+1], g)
g <- addEdge(V[1+5], V[0+1], g)
g <- addEdge(V[1+0], V[2+1], g)
g <- addEdge(V[1+0], V[3+1], g)
g <- addEdge(V[1+0], V[4+1], g)
g <- addEdge(V[1+1], V[3+1], g)
g <- addEdge(V[1+1], V[4+1], g)
g <- addEdge(V[1+1], V[5+1], g)

x2 <- planarCanonicalOrdering(g)
x2
```

---

planarFaceTraversal *planarFaceTraversal*

Description

planarFaceTraversal description

Usage

```
planarFaceTraversal(g)
```

Arguments

g	instance of class graphNEL from Bioconductor graph class

Details

see http://www.boost.org/doc/libs/1_49_0/libs/graph/doc/planar_graphs.html

Value

A list of character vectors with ordered sequences of node names

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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

Examples

```
V <- LETTERS[1:9]
g <- new("graphNEL", nodes=V, edgemode="undirected")
g <- addEdge(V[1+0],V[1+1],g)
g <- addEdge(V[1+1],V[1+2],g)
g <- addEdge(V[1+3],V[1+4],g)
g <- addEdge(V[1+4],V[1+5],g)
g <- addEdge(V[1+6],V[1+7],g)
g <- addEdge(V[1+7],V[1+8],g)
g <- addEdge(V[1+0],V[1+3],g)
g <- addEdge(V[1+3],V[1+6],g)
g <- addEdge(V[1+1],V[1+4],g)
g <- addEdge(V[1+4],V[1+7],g)
g <- addEdge(V[1+2],V[1+5],g)
g <- addEdge(V[1+5],V[1+8],g)

x1 <- planarFaceTraversal(g)
x1
```

RBGL-defunct *Defunct Functions in Package* **RBGL**

Description

The functions or variables listed here are no longer part of the RBGL package.

Usage

```
prim.minST()
```

Value

none

See Also

Defunct

RBGL.overview *RBGL.overview*

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

Functions	parameters
bandwidth	g
bfs	object,node,checkConn
dfs	object,node,checkConn
edgeConnectivity	g
gprofile	g
isomorphism	g1,g2
minCut	g
transitive.closure	g
tsort	x

ShortestPaths

Functions	parameters
bellman.ford.sp	g,start
dag.sp	g,start
dijkstra.sp	g,start
extractPath	s,f,pens
johnson.all.pairs.sp	g
sp.between	g,start,finish
sp.between.old	g,start,finish
sp.between.scalar	g,start,finish

MinimumSpanningTree

Functions	parameters
mstree.kruskal	x

ConnectedComponents

Functions	parameters
connectedComp	g
highlyConnSG	g,sat,ldv
incremental.components	g
init.incremental.components	g
same.component	g,node1,node2
strongComp	g

MaximumFlow

Functions	parameters
edmonds.karp.max.flow	g,source,sink
push.relabel.max.flow	g,source,sink

SparseMatrixOrdering

Functions	parameters
cuthill.mckee.ordering	g
minDegreeOrdering	g,delta
sloan.ordering	g,w1,w2

LayoutAlgorithms

Functions	parameters
circle.layout	g,radius
kamada.kawai.spring.layout	g,edge_or_side,es_length

GraphClustering

Functions	parameters
betweenness.centrality.clustering	g,threshold,normalize

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)

The Boost Graph Library: User Guide and Reference Manual; by Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

`removeSelfLoops` *remove self loops in a graph*

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>

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

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

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

`sequential.vertex.coloring`
Compute a vertex coloring for a graph

Description

Compute vertex coloring for a graph

Usage

```
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>

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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

Examples

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

---

sloanStartEndVertices *sloanStartEndVertices*

Description

sloanStartEndVertices description

Usage

```
sloanStartEndVertices(g)
```

Arguments

g	instance of class graphNEL from Bioconductor graph class

Details

not used

Value

message

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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

`sp.between` *Dijkstra's shortest paths using boost C++*

Description

dijkstra's shortest paths

Usage

```
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

```
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")
```

---

strongComp *Identify Strongly Connected Components*

Description

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

Usage

```
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)

The Boost Graph Library: User Guide and Reference Manual; by Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

See Also

connComp,connectedComp, same.component

Examples

```
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)

The Boost Graph Library: User Guide and Reference Manual; by Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

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) = 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

Approximating Clustering Coefficient and Transitivity, T. Schank, D. Wagner, Journal of Graph Algorithms and Applications, Vol. 9, No. 2 (2005).

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

```
tsort(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 character vector of vertices in the topological sort sequence.

Author(s)

VJ Carey <stvjc@channing.harvard.edu>

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 Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

Examples

```
data(FileDep)
tsind <- tsort(FileDep)
tsind
FD2 <- FileDep
# now introduce a cycle
FD2 <- addEdge("bar_o", "dax_h", FD2, 1)
tsort(FD2)
```

`wavefront` *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)

The Boost Graph Library: User Guide and Reference Manual; by Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine; (Addison-Wesley, Pearson Education Inc., 2002), xxiv+321pp. ISBN 0-201-72914-8

See Also

edgeConnectivity

Examples

```
con <- file(system.file("XML/dijkex.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

∗ **graphs**
- bellman.ford.sp, 5
- bfs, 7
- boyerMyrvoldPlanarityTest, 10
- chrobakPayneStraightLineDrawing, 12
- dag.sp, 17
- dijkstra.sp, 18
- edmondsMaxCardinalityMatching, 22
- edmondsOptimumBranching, 23
- FileDep, 25
- isKuratowskiSubgraph, 33
- isStraightLineDrawing, 35
- makeBiconnectedPlanar, 42
- makeConnected, 44
- makeMaximalPlanar, 45
- maximumCycleRatio, 48
- minimumCycleRatio, 50
- mstree.kruskal, 51
- planarCanonicalOrdering, 55
- planarFaceTraversal, 56
- sloanStartEndVertices, 64
- sp.between, 65
- tsort, 69

∗ **manip**
- separates, 62

∗ **models**
- astarSearch, 3
- bandwidth, 4
- betweenness.centrality.clustering, 6
- biConnComp, 8
- brandes.betweenness.centrality, 11
- clusteringCoef, 13
- clusteringCoefAppr, 14
- connectedComp, 16
- dominatorTree, 20
- edgeConnectivity, 21
- extractPath, 24
- floyd.warshall.all.pairs.sp, 26
- gprofile, 27
- graphGenerator, 28
- highlyConnSG, 29
- incremental.components, 30
- is.triangulated, 32
- isomorphism, 34
- johnson.all.pairs.sp, 36
- kCliques, 37
- kCores, 38
- lambdaSets, 39
- layout, 40
- max.flow, 46
- maxClique, 47
- minCut, 49
- mstree.prim, 52
- Ordering, 53
- RBGL.overview, 58
- removeSelfLoops, 61
- sequential.vertex.coloring, 63
- strongComp, 66
- transitive.closure, 67
- transitivity, 68
- wavefront, 70

allShortestPaths, *25*
articulationPoints *(*biConnComp*)*, 8
astarSearch, 3
aver.wavefront, *60*
aver.wavefront *(*wavefront*)*, 70

bandwidth, 4, *58*
bellman.ford.sp, 5, *18, 19*, *37*, *58*, *66*
betweenness.centrality.clustering, 6, *12*, *60*
bfs, 7, *58*
bfs,graph,ANY,ANY-method *(*bfs*)*, 7
bfs,graph,character,logical-method *(*bfs*)*, 7

bfs,graph,character,missing-method (bfs), 7
bfs,graph,character-method (bfs), 7
bfs,graph,missing,logical-method (bfs), 7
bfs,graph,missing,missing-method (bfs), 7
bfs,graph-method (bfs), 7
biConnComp, 8
boyerMyrvoldPlanarityTest, 10
brandes.betweenness.centrality, *7*, 11, *60*

chrobakPayneStraightLineDrawing, 12
circle.layout, *59*
circle.layout (layout), 40
circleLayout (layout), 40
clusteringCoef, 13
clusteringCoefAppr, 14
connComp, *16*, *31*, *67*
connectedComp, 16, *31*, *59*, *67*
cuthill.mckee.ordering, *59*
cuthill.mckee.ordering (Ordering), 53

dag.sp, *6*, 17, *19*, *37*, *58*, *66*
Defunct, *57*
dfs, *58*
dfs (bfs), 7
dfs,graph,character,ANY-method (bfs), 7
dfs,graph,character,logical-method (bfs), 7
dfs,graph,character,missing-method (bfs), 7
dfs,graph,character-method (bfs), 7
dfs,graph,missing,missing-method (bfs), 7
dijkstra.sp, *6*, *18*, 18, *24*, *37*, *58*, *66*
dominatorTree, 20

edgeConnectivity, 21, *30*, *47*, *50*, *58*, *71*
edmonds.karp.max.flow, *21*, *59*
edmonds.karp.max.flow (max.flow), 46
edmondsMaxCardinalityMatching, 22
edmondsOptimumBranching, 23
extractPath, 24, *58*

FileDep, 25
floyd.warshall.all.pairs.sp, 26
fruchtermanReingoldForceDirectedLayout (layout), 40

gprofile, 27, *58*
graphGenerator, 28
gursoyAtunLayout (layout), 40

highlyConnSG, 29, *59*

incremental.components, 30, *59*
init.incremental.components, *59*
init.incremental.components (incremental.components), 30
is.triangulated, 32
isKuratowskiSubgraph, 33
isomorphism, 34, *58*
isStraightLineDrawing, 35
ith.wavefront, *60*
ith.wavefront (wavefront), 70

johnson.all.pairs.sp, *6*, *18*, *19*, *26*, 36, *58*, *62*, *66*

kamada.kawai.spring.layout, *59*
kamada.kawai.spring.layout (layout), 40
kamadaKawaiSpringLayout (layout), 40
kCliques, 37
kCores, 38
kingOrdering (Ordering), 53
kolmogorov.max.flow (max.flow), 46

lambdaSets, 39
layout, 40
layoutGraph, *42*
lengauerTarjanDominatorTree (dominatorTree), 20

makeBiconnectedPlanar, 42
makeConnected, 44
makeMaximalPlanar, 45
max.flow, 46
maxClique, 47
maximumCycleRatio, 48
maxWavefront, *60*
maxWavefront (wavefront), 70
minCut, *21*, *30*, *47*, 49, *58*
minDegreeOrdering, *59*
minDegreeOrdering (Ordering), 53
minimumCycleRatio, 50
mstree.kruskal, 51, *53*, *59*
mstree.prim, 52

Ordering, 53

pathWeights, *65*
planarCanonicalOrdering, 55
planarFaceTraversal, 56
prim.minST *(*RBGL-defunct*)*, 57
push.relabel.max.flow, *21*, *59*
push.relabel.max.flow *(*max.flow*)*, 46

randomGraphLayout *(*layout*)*, 40
RBGL-defunct, 57
RBGL.overview, 58
removeSelfLoops, *29, 30*, 61
rms.wavefront, *60*
rms.wavefront *(*wavefront*)*, 70

same.component, *16*, *59*, *67*
same.component
*(*incremental.components*)*, 30
separates, 62
sequential.vertex.coloring, 63
sloan.ordering, *59*
sloan.ordering *(*Ordering*)*, 53
sloanStartEndVertices, 64
sp.between, *6*, *18, 19*, *37*, *58*, 65
sp.between.old, *58*
sp.between.scalar, *58*
strongComp, *16*, *31*, *59*, 66

transitive.closure, *58*, 67
transitivity, 68
tsort, *58*, 69

ugraph, *16*, *32*

wavefront, 70