Package 'optrees'

September 2, 2014

Type Package

Title Optimal Trees in Weighted Graphs

Version 1.0

Date 2014-09-01

Author Manuel Fontenla [aut, cre]

Maintainer Manuel Fontenla <manu.fontenla@gmail.com>

Depends R (>= 2.7.0), igraph (>= 0.7.1)

Description Finds optimal trees in weighted graphs. In particular, this package provides solving tools for minimum cost spanning tree problems, minimum cost arborescence problems, shortest path tree problems and minimum cut tree problem.

License GPL-3

NeedsCompilation no

Repository CRAN

Date/Publication 2014-09-02 06:14:11

R topics documented:

`optrees-package` *Optimal Trees in Weighted Graphs*

Description

Finds optimal trees in weighted graphs. In particular, this package provides solving tools for minimum cost spanning tree problems, minimum cost arborescence problems, shortest path tree problems and minimum cut tree problem.

Details

Package:	optrees
Type:	Package
Version:	1.0
Date:	2014-09-01
License:	GPL-3

The most important functions are getMinimumSpanningTree, getMinimumArborescence, getShortestPathTree and getMinimumCutTree. The other functions included in the package are auxiliary ones that can be used independently.

Author(s)

Manuel Fontenla <manu.fontenla@gmail.com>

`ArcList2Cmat` *Builds the cost matrix of a graph from its list of arcs*

Description

The `ArcList2Cmat` function constructs the cost matrix of a graph from a list that contains the arcs and its associated weights.

Usage

```
ArcList2Cmat(nodes, arcs, directed = TRUE)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`directed`	logical value indicating whether the graph is directed (`TRUE`) or not (`FALSE`).

Value

`ArcList2Cmat` returns a $n \times n$ matrix that contains the weights of the arcs. It means that the element (i, j) of the matrix returns the weight of the arc (i, j). If the value of an arc (i, j) is `NA` or `Inf`, then it means this arc does not exist in the graph.

`checkArbor` *Checks if there is at least one arborescence in the graph*

Description

Given a directed graph, `checkArbor` searchs for an arborescence from the list of arcs. An arborescence is a directed graph with a source node and such that there is a unique path from the source to any other node.

Usage

```
checkArbor(nodes, arcs, source.node = 1)
```

Arguments

nodes	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
arcs	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
source.node	source node of the graph. Its default value is 1.

Value

If checkArbor found an arborescence it returns TRUE, otherwise it returns FALSE. If there is an arborescence the function also returns the list of arcs of the arborescence.

See Also

This function is an auxiliar function used in msArborEdmonds and getMinimumArborescence.

---

checkGraph *Checks if the graph contains at least one tree or one arborescence*

Description

The checkGraph function checks if it is possible to find at least one tree (or arborescence, if it is the case) in the graph. It only happens when the graph is connected and it is posible to find a walk from the source to any other node.

Usage

```
checkGraph(nodes, arcs, source.node = 1, directed = TRUE)
```

Arguments

nodes	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
arcs	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
source.node	number pointing the source node of the graph.
directed	logical value indicating whether the graph is directed (TRUE) or not (FALSE).

Value

checkGraph returns the value TRUE if the graph meets the requirements and FALSE otherwise. If the graph is not acceptable this functions also prints the reason.

Examples

```
# Graph
nodes <- 1:4
arcs <- matrix(c(1,2,2, 1,3,15, 2,3,1, 2,4,9, 3,4,1),
               byrow = TRUE, ncol = 3)
# Check graph
checkGraph(nodes, arcs)
```

Cmat2ArcList *Builds the list of arcs of a graph from its cost matrix*

Description

The Cmat2ArcList function builds the list of arcs of a graph from a cost matrix that contains the weights of all the arcs.

Usage

```
Cmat2ArcList(nodes, Cmat, directed = TRUE)
```

Arguments

nodes	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
Cmat	$n \times n$ matrix that contains the weights or costs of the arcs. Row i and column j represents the endpoints of an arc, and the value of the index ij is its weight or cost. If this value is NA or Inf means that there is no arc ij.
directed	logical value indicating whether the graph is directed (TRUE) or not (FALSE).

Value

Cmat2ArcList returns a matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

---

compactCycle *Compacts the nodes in a cycle into a single node*

Description

Given a directed graph with a cycle, compactCycle compacts all the nodes in the cycle to a single node called supernode. The function uses the first and the last node of the cycle as a fusion point and obtains the costs of the incoming and outgoing arcs of the new node.

Usage

```
compactCycle(nodes, arcs, cycle)
```

Arguments

nodes	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
arcs	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
cycle	vector with the original nodes in the cycle.

Value

compactCycle returns the nodes and the list of arcs forming a new graph with the compressed cycle within a supernode. Also returns a list of the correspondences between the nodes of the new graph and the nodes of the previous graph.

See Also

This function is an auxiliar function used in msArborEdmonds and getMinimumArborescence.

---

findMinCut *Finds the minimum cut of a given graph*

Description

The findMinCut function can find the minimum cut of a given graph. For that, this function computes the maximum flow of the network and applies the max-flow min-cut theorem to determine the cut with minimum weight between the source and the sink nodes.

Usage

```
findMinCut(nodes, arcs, algorithm = "Ford-Fulkerson", source.node = 1,
  sink.node = nodes[length(nodes)], directed = FALSE)
```

Arguments

nodes	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
arcs	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
algorithm	denotes the algorithm used to compute the maximum flow: "Ford-Fulkerson".
source.node	number pointing to the source node of the graph. It's node 1 by default.
sink.node	number pointing to the sink node of the graph. It's the last node by default.
directed	logical value indicating whether the graph is directed (TRUE) or not (FALSE).

Details

The max-flow min-cut theorem proves that, in a flow network, the maximum flow between the source node and the sink node and the weight of any minimum cut between them is equal.

Value

`findMinCut` returns a list with:

`s.cut`	vector with the nodes of the `s` cut.
`t.cut`	vector with the nodes of the `t` cut.
`max.flow`	value with the maximum flow in the flow network.
`cut.set`	list of arcs of the cut set founded.

See Also

This function is an auxiliar function used in ghTreeGusfield and getMinimumCutTree.

Examples

```
# Graph
nodes <- 1:6
arcs <- matrix(c(1,2,1, 1,3,7, 2,3,1, 2,4,3, 2,5,2, 3,5,4, 4,5,1, 4,6,6,
                 5,6,2), byrow = TRUE, ncol = 3)
# Find minimum cut
findMinCut(nodes, arcs, source.node = 2, sink.node = 6)
```

`findstCut` *Determines the s-t cut of a graph*

Description

`findstCut` reviews a given graph with a cut between two nodes with the bread-first search strategy and determines the two cut set of the partition. The cut is marked in the arc list with an extra column that indicates the remaining capacity of each arc.

Usage

```
findstCut(nodes, arcs, s = 1, t = nodes[length(nodes)])
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`s`	number pointing one node of the `s` cut in a given graph. It's node 1 by default.
`t`	number pointing one node of the `t` cut in a given graph. It's the last node by default.

Value

`findstCut` returns a list with two elements:

`s.cut`	vector with the nodes of the `s` cut.
`t.cut`	vector with the nodes of the `t` cut.

See Also

This function is an auxiliar function used in ghTreeGusfield and getMinimumCutTree.

`getCheapArcs` *Substracts the minimum weight of the arcs pointing to each node*

Description

The `getCheapArcs` function substracts to each arc of a given graph the value of the minimum weight of the arcs pointing to the same node.

Usage

```
getCheapArcs(nodes, arcs)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Value

`getCheapArcs` returns a matrix with a new list of arcs.

See Also

This function is an auxiliar function used in msArborEdmonds and getMinimumArborescence.

getComponents *Connected components of a graph*

Description

The getComponents function returns all the connected components of a graph.

Usage

```
getComponents(nodes, arcs)
```

Arguments

nodes	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
arcs	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Value

getComponents returns a list with all the components and the nodes of each one ($components) and a matrix with all the arcs of the graph and its component ($components.arcs).

Examples

```
# Graph
nodes <- 1:4
arcs <- matrix(c(1,2,1, 1,6,1, 3,4,1, 4,5,1), ncol = 3, byrow = TRUE)
# Components
getComponents(nodes, arcs)
```

getMinCostArcs *Selects the minimum cost of the arcs pointing to each node*

Description

Given a directed graph, getMinCostArcs selects the minimum cost arcs entering each node and removes the others.

Usage

```
getMinCostArcs(nodes, arcs)
```

Arguments

nodes	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
arcs	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Value

The getMinCostArcs function returns a matrix with the list of the minimum cost arcs pointing to each node of the graph.

See Also

This function is an auxiliar function used in msArborEdmonds and getMinimumArborescence.

---

getMinimumArborescence

Computes a minimum cost arborescence

Description

Given a connected weighted directed graph, getMinimumArborescence computes a minimum cost arborescence. This function provides a method to find the minimum cost arborescence with Edmonds' algorithm.

Usage

```
getMinimumArborescence(nodes, arcs, source.node = 1, algorithm = "Edmonds",
  stages.data = FALSE, show.data = TRUE, show.graph = TRUE,
  check.graph = FALSE)
```

Arguments

nodes	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
arcs	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
source.node	number pointing to the source node of the graph. It's node 1 by default.
algorithm	denotes the algorithm used to find a minimum cost arborescence: "Edmonds".
check.graph	logical value indicating if it is necesary to check the graph. Is FALSE by default.
show.data	logical value indicating if the function displays the console output (TRUE) or not (FALSE). The default is TRUE.

show.graph	logical value indicating if the function displays a graphical representation of the graph and its minimum arborescence (TRUE) or not (FALSE). The default is TRUE.
stages.data	logical value indicating if the function returns data of each stage. The default is FALSE.

Details

Given a connected weighted directed graph, a minimum cost arborescence is an arborescence such that the sum of the weight of its arcs is minimum. In some cases, it is possible to find several minimum cost arborescences, but the proposed algorithm only finds one of them.

Edmonds' algorithm was developed by the mathematician and computer scientist Jack R. Edmonds in 1967. Although, it was previously proposed in 1965 by Yoeng-jin Chu and Tseng-hong Liu. This algorithm decreases the weights of the arcs in a graph and compacts cycles of zero weight until it can find an arborescence. This arborescence has to be a minimum cost arborescence of the graph.

Value

getMinimumArborescence returns a list with:

tree.nodes	vector containing the nodes of the minimum cost arborescence.
tree.arcs	matrix containing the list of arcs of the minimum cost arborescence.
weight	value with the sum of weights of the arcs.
stages	number of stages required.
time	time needed to find the minimum cost arborescence.

This function also represents the graph and the minimum arborescence and prints to the console the results with additional information (number of stages, computational time, etc.).

References

Chu, Y. J., and Liu, T. H., "On the Shortest Arborescence of a Directed Graph", Science Sinica, vol. 14, 1965, pp. 1396-1400.

Edmonds, J., "Optimum Branchings", Journal of Research of the National Bureau of Standards, vol. 71B, No. 4, October-December 1967, pp. 233-240.

Examples

```
# Graph
nodes <- 1:4
arcs <- matrix(c(1,2,2, 1,3,3, 1,4,4, 2,3,3, 2,4,4, 3,2,3,
                 3,4,1, 4,2,1, 4,3,2),byrow = TRUE, ncol = 3)
# Minimum cost arborescence
getMinimumArborescence(nodes, arcs)
```

getMinimumCutTree *getMinimumCutTree* ——————————————————-
Computes a minimum cut tree

Description

Given a connected weighted undirected graph, `getMinimumCutTree` computes a minimum cut tree, also called Gomory-Hu tree. This function uses the Gusfield's algorithm to find it.

Usage

```
getMinimumCutTree(nodes, arcs, algorithm = "Gusfield", show.data = TRUE,
  show.graph = TRUE, check.graph = FALSE)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`algorithm`	denotes the algorithm to use for find a minimum cut tree or Gomory-Hu tree: "Gusfield".
`check.graph`	logical value indicating if it is necesary to check the graph. Is `FALSE` by default.
`show.data`	logical value indicating if the function displays the console output (`TRUE`) or not (`FALSE`). The default is `TRUE`.
`show.graph`	logical value indicating if the function displays a graphical representation of the graph and its minimum cut tree (`TRUE`) or not (`FALSE`). The default is `TRUE`.

Details

The minimum cut tree or Gomory-Hu tree was introduced by R. E. Gomory and T. C. Hu in 1961. Given a connected weighted undirected graph, the Gomory-Hu tree is a weighted tree that contains the minimum s-t cuts for all s-t pairs of nodes in the graph. Gomory and Hu developed an algorithm to find this tree, but it involves maximum flow searchs and nodes contractions.

In 1990, Dan Gusfield proposed a new algorithm that can be used to find the Gomory-Hu tree without any nodes contraction and simplifies the implementation.

Value

`getMinimumCutTree` returns a list with:

`tree.nodes`	vector containing the nodes of the minimum cut tree.
`tree.arcs`	matrix containing the list of arcs of the minimum cut tree.
`weight`	value with the sum of weights of the arcs.

stages	number of stages required.
time	time needed to find the minimum cut tree.

This function also represents the graph and the minimum cut tree and prints in console the results whit additional information (number of stages, computational time, etc.).

References

R. E. Gomory, T. C. Hu. Multi-terminal network flows. Journal of the Society for Industrial and Applied Mathematics, vol. 9, 1961.

Dan Gusfield (1990). "Very Simple Methods for All Pairs Network Flow Analysis". SIAM J. Comput. 19 (1): 143-155.

Examples

```
# Graph
nodes <- 1:6
arcs <- matrix(c(1,2,1, 1,3,7, 2,3,1, 2,4,3, 2,5,2, 3,5,4, 4,5,1, 4,6,6,
                 5,6,2), byrow = TRUE, ncol = 3)
# Minimum cut tree
getMinimumCutTree(nodes, arcs)
```

---

getMinimumSpanningTree

Computes a minimum cost spanning tree

Description

Given a connected weighted undirected graph, `getMinimumSpanningTree` computes a minimum cost spanning tree. This function provides methods to find a minimum cost spanning tree with the three most commonly used algorithms: "Prim", "Kruskal" and "Boruvka".

Usage

```
getMinimumSpanningTree(nodes, arcs, algorithm, start.node = 1,
  show.data = TRUE, show.graph = TRUE, check.graph = FALSE)
```

Arguments

nodes	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
arcs	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
algorithm	denotes the algorithm used to find a minimum spanning tree: "Prim", "Kruskal" or "Boruvka".
check.graph	logical value indicating if it is necesary to check the graph. Is `FALSE` by default.

start.node	number which indicates the first node in Prim's algorithm. If none is specified node 1 is used by default.
show.data	logical value indicating if the function displays the console output (TRUE) or not (FALSE). The default is TRUE.
show.graph	logical value indicating if the function displays a graphical representation of the graph and its minimum spanning tree (TRUE) or not (FALSE). The default is TRUE.

Details

Given a connected weighted undirected graph, a minimum spanning tree is a spanning tree such that the sum of the weights of the arcs is minimum. There may be several minimum spanning trees of the same weight in a graph. Several algorithms were proposed to find a minimum spanning tree in a graph.

Prim's algorithm was developed in 1930 by the mathematician Vojtech Jarnik, independently proposed by the computer scientist Robert C. Prim in 1957 and rediscovered by Edsger Dijkstra in 1959. This is a greedy algorithm that can find a minimum spanning tree in a connected weighted undirected graph by adding minimum cost arcs leaving visited nodes recursively.

Kruskal's algorithm was published for first time in 1956 by mathematician Joseph Kruskal. This is a greedy algorithm that finds a minimum cost spanning tree in a connected weighted undirected graph by adding, without form cycles, the minimum weight arc of the graph in each iteration.

Boruvka's algorithm was published for first time in 1926 by mathematician Otakar Boruvka. This algorithm go through a connected weighted undirected graph, reviewing each component and adding the minimum weight arcs without repeat it until one minimum spanning tree is complete.

Value

getMinimumSpanningTree returns a list with:

tree.nodes	vector containing the nodes of the minimum cost spanning tree.
tree.arcs	matrix containing the list of arcs of the minimum cost spanning tree.
weight	value with the sum of weights of the arcs.
stages	number of stages required.
stages.arcs	stages in which each arc was added.
time	time needed to find the minimum cost spanning tree.

This function also represents the graph and the minimum spanning tree and prints to the console the results whit additional information (number of stages, computational time, etc.).

References

Prim, R. C. (1957), "Shortest Connection Networks And Some Generalizations", Bell System Technical Journal, 36 (1957), pp. 1389-1401.

Kruskal, Joshep B. (1956), "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem", Proceedings of the American Mathematical Society, Vol. 7, No. 1 (Feb., 1956), pp. 48-50.

Boruvka, Otakar (1926). "O jistem problemu minimalnim (About a certain minimal problem)". Prace mor. prirodoved. spol. v Brne III (in Czech, German summary) 3: 37-58.

Examples

```
# Graph
nodes <- 1:4
arcs <- matrix(c(1,2,2, 1,3,15, 1,4,3, 2,3,1, 2,4,9, 3,4,1),
               ncol = 3, byrow = TRUE)
# Minimum cost spanning tree with several algorithms
getMinimumSpanningTree(nodes, arcs, algorithm = "Prim")
getMinimumSpanningTree(nodes, arcs, algorithm = "Kruskal")
getMinimumSpanningTree(nodes, arcs, algorithm = "Boruvka")
```

---

getShortestPathTree *Computes a shortest path tree*

Description

Given a connected weighted graph, directed or not, getShortestPathTree computes the shortest path tree from a given source node to the rest of the nodes the graph, forming a shortest path tree. This function provides methods to find it with two known algorithms: "Dijkstra" and "Bellman-Ford".

Usage

```
getShortestPathTree(nodes, arcs, algorithm, check.graph = FALSE,
  source.node = 1, directed = TRUE, show.data = TRUE, show.graph = TRUE,
  show.distances = TRUE)
```

Arguments

nodes	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
arcs	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
algorithm	denotes the algorithm used to find a shortest path tree: "Dijkstra" or "Bellman-Ford".
check.graph	logical value indicating if it is necesary to check the graph. Is FALSE by default.
source.node	number indicating the source node of the graph. It's node 1 by default.
directed	logical value indicating wheter the graph is directed (TRUE) or not (FALSE).
show.data	logical value indicating if the function displays the console output (TRUE) or not (FALSE). The default is TRUE.
show.graph	logical value indicating if the function displays a graphical representation of the graph and its shortest path tree (TRUE) or not (FALSE). The default is TRUE.
show.distances	logical value indicating if the function displays in the console output the distances from source to all the other nodes. The default is TRUE.

Details

Given a connected weighted graph, directed or not, a shortest path tree rooted at a source node is a spanning tree such thtat the path distance from the source to any other node is the shortest path distance between them. Differents algorithms were proposed to find a shortest path tree in a graph.

One of these algorithms is Dijkstra's algorithm. Developed by the computer scientist Edsger Dijkstra in 1956 and published in 1959, it is an algorithm that can compute a shortest path tree from a given source node to the others nodes in a connected, directed or not, graph with non-negative weights.

The Bellman-Ford algorithm gets its name for two of the developers, Richard Bellman y Lester Ford Jr., and it was published by them in 1958 and 1956 respectively. The same algorithm also was published independently in 1957 by Edward F. Moore. This algorithm can compute the shortest path from a source node to the rest of nodes in a connected, directed or not, graph with weights that can be negatives. If the graph is connected and there isn't negative cycles, the algorithm always finds a shortest path tree.

Value

`getShortestPathTree` returns a list with:

`tree.nodes`	vector containing the nodes of the shortest path tree.
`tree.arcs`	matrix containing the list of arcs of the shortest path tree.
`weight`	value with the sum of weights of the arcs.
`distances`	vector with distances from source to other nodes
`stages`	number of stages required.
`time`	time needed to find the shortest path tree.

This function also represents the graph with the shortest path tree and prints to the console the results with additional information (number of stages, computational time, etc.).

References

Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs". Numerische Mathematik 1, 269-271.

Bellman, Richard (1958). "On a routing problem". Quarterly of Applied Mathematics 16, 87-90.

Ford Jr., Lester R. (1956). Network Flow Theory. Paper P-923. Santa Monica, California: RAND Corporation.

Moore, Edward F. (1959). "The shortest path through a maze". Proc. Internat. Sympos. Switching Theory 1957, Part II. Cambridge, Mass.: Harvard Univ. Press. pp. 285-292.

Examples

```
# Graph
nodes <- 1:5
arcs <- matrix(c(1,2,2, 1,3,2, 1,4,3, 2,5,5, 3,2,4, 3,5,3, 4,3,1, 4,5,0),
               ncol = 3, byrow = TRUE)
# Shortest path tree
```

```
getShortestPathTree(nodes, arcs, algorithm = "Dijkstra", directed=FALSE)
getShortestPathTree(nodes, arcs, algorithm = "Bellman-Ford", directed=FALSE)

# Graph with negative weights
nodes <- 1:5
arcs <- matrix(c(1,2,6, 1,3,7, 2,3,8, 2,4,5, 2,5,-4, 3,4,-3, 3,5,9, 4,2,-2,
                 5,1,2, 5,4,7), ncol = 3, byrow = TRUE)
# Shortest path tree
getShortestPathTree(nodes, arcs, algorithm = "Bellman-Ford", directed=TRUE)
```

`getZeroArcs` *Selects zero weight arcs of a graph*

Description

Given a directed graph, `getZeroArcs` returns the list of arcs with zero weight. Removes other arcs by assign them infinite value.

Usage

```
getZeroArcs(nodes, arcs)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Value

The `getZeroArcs` function returns a matrix with the list of zero weight arcs of the graph.

See Also

This function is an auxiliar function used in msArborEdmonds and getMinimumArborescence.

`ghTreeGusfield` *Gomory-Hu tree with the Gusfield's algorithm*

Description

Given a connected weighted and undirected graph, the `ghTreeGusfield` function builds a Gomory-Hu tree with the Gusfield's algorithm.

Usage

```
ghTreeGusfield(nodes, arcs)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Details

The Gomory-Hu tree was introduced by R. E. Gomory and T. C. Hu in 1961. Given a connected weighted and undirected graph, the Gomory-Hu tree is a weighted tree that contains the minimum s-t cuts for all s-t pairs of nodes in the graph. Gomory and Hu also developed an algorithm to find it that involves maximum flow searchs and nodes contractions.

In 1990, Dan Gusfield proposed a new algorithm that can be used to find a Gomory-Hu tree without nodes contractions and simplifies the implementation.

Value

`ghTreeGusfield` returns a list with:

`tree.nodes`	vector containing the nodes of the Gomory-Hu tree.
`tree.arcs`	matrix containing the list of arcs of the Gomory-Hu tree.
`stages`	number of stages required.

References

R. E. Gomory, T. C. Hu. Multi-terminal network flows. Journal of the Society for Industrial and Applied Mathematics, vol. 9, 1961.

Dan Gusfield (1990). "Very Simple Methods for All Pairs Network Flow Analysis". SIAM J. Comput. 19 (1): 143-155.

See Also

A more general function getMinimumCutTree.

`maxFlowFordFulkerson` *Maximum flow with the Ford-Fulkerson algorithm*

Description

The `maxFlowFordFulkerson` function computes the maximum flow in a given flow network with the Ford-Fulkerson algorithm.

Usage

```
maxFlowFordFulkerson(nodes, arcs, directed = FALSE, source.node = 1,
  sink.node = nodes[length(nodes)])
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`directed`	logical value indicating wheter the graph is directed (`TRUE`) or not (`FALSE`).
`source.node`	number pointing to the source node of the graph. It's node 1 by default.
`sink.node`	number pointing to the sink node of the graph. It's the last node by default.

Details

The Ford-Fulkerson algorithm was published in 1956 by L. R. Ford, Jr. and D. R. Fulkerson. This algorithm can compute the maximum flow between source and sink nodes of a flow network.

Value

`maxFlowFordFulkerson` returns a list with:

`s.cut`	vector with the nodes of the `s` cut.
`t.cut`	vector with the nodes of the `t` cut.
`max.flow`	value with the maximum flow in the flow network.

References

Ford, L. R.; Fulkerson, D. R. (1956). "Maximal flow through a network". Canadian Journal of Mathematics 8: 399.

See Also

This function is an auxiliar function used in ghTreeGusfield and getMinimumCutTree.

Examples

```
# Graph
nodes <- 1:6
arcs <- matrix(c(1,2,1, 1,3,7, 2,3,1, 2,4,3, 2,5,2, 3,5,4, 4,5,1, 4,6,6,
                 5,6,2), byrow = TRUE, ncol = 3)
# Maximum flow with Ford-Fulkerson algorithm
maxFlowFordFulkerson(nodes, arcs, source.node = 2, sink.node = 6)
```

`msArborEdmonds` *Minimum cost arborescence with Edmonds' algorithm*

Description

Given a connected weighted and directed graph, `msArborEdmonds` uses Edmonds' algorithm to find a minimum cost arborescence.

Usage

```
msArborEdmonds(nodes, arcs, source.node = 1, stages.data = FALSE)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`source.node`	source node of the graph. It's node 1 by default.
`stages.data`	logical value indicating if the function returns data of each stage. Is `FALSE` by default.

Details

Edmonds' algorithm was developed by the mathematician and computer scientist Jack R. Edmonds in 1967. Previously, it was proposed in 1965 by Yoeng-jin Chu and Tseng-hong Liu.

Value

`msArborEdmonds` returns a list with:

`tree.nodes`	vector containing the nodes of the minimum cost arborescence.
`tree.arcs`	matrix containing the list of arcs of the minimum cost arborescence.
`stages`	number of stages required.

References

Chu, Y. J., and Liu, T. H., "On the Shortest Arborescence of a Directed Graph", Science Sinica, vol. 14, 1965, pp. 1396-1400.

Edmonds, J., "Optimum Branchings", Journal of Research of the National Bureau of Standards, vol. 71B, No. 4, October-December 1967, pp. 233-240.

See Also

A more general function getMinimumSpanningTree.

`msTreeBoruvka` *Minimum cost spanning tree with Boruvka's algorithm.*

Description

`msTreeBoruvka` computes a minimum cost spanning tree of an undirected graph with Boruvka's algorithm.

Usage

```
msTreeBoruvka(nodes, arcs)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Details

Boruvka's algorithm was firstly published in 1926 by the mathematician Otakar Boruvka. This algorithm works in a connected, weighted and undirected graph, checking each component and adding the minimum weight arcs that connect the component to other components until one minimum spanning tree is complete.

Value

`msTreeBoruvka` returns a list with:

`tree.nodes`	vector containing the nodes of the minimum cost spanning tree.
`tree.arcs`	matrix containing the list of arcs of the minimum cost spanning tree.
`stages`	number of stages required.
`stages.arcs`	stages in which each arc was added.

References

Boruvka, Otakar (1926). "O jistem problemu minimalnim (About a certain minimal problem)". Prace mor. prirodoved. spol. v Brne III (in Czech, German summary) 3: 37-58.

See Also

A more general function getMinimumSpanningTree.

`msTreeKruskal` *Minimum cost spanning tree with Kruskal's algorithm*

Description

`msTreeKruskal` computes a minimum cost spanning tree of an undirected graph with Kruskal's algorithm.

Usage

```
msTreeKruskal(nodes, arcs)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Details

Kruskal's algorithm was published for first time in 1956 by mathematician Joseph Kruskal. This is a greedy algorithm that finds a minimum cost spanning tree in a connected weighted undirected graph by adding, without forming cycles, the minimum weight arc of the graph at each stage.

Value

`msTreeKruskal` returns a list with:

`tree.nodes`	vector containing the nodes of the minimum cost spanning tree.
`tree.arcs`	matrix containing the list of arcs of the minimum cost spanning tree.
`stages`	number of stages required.
`stages.arcs`	stages in which each arc was added.

References

Kruskal, Joshep B. (1956), "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem", Proceedings of the American Mathematical Society, Vol. 7, No. 1 (Feb., 1956), pp. 48-50

See Also

A more general function getMinimumSpanningTree.

`msTreePrim` *Minimum cost spanning tree with Prim's algorithm*

Description

`msTreePrim` computes a minimum cost spanning tree of an undirected graph with Prim's algorithm.

Usage

```
msTreePrim(nodes, arcs, start.node = 1)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`start.node`	number associated with the first node in Prim's algorithm. By default, node 1 is the first node.

Details

Prim's algorithm was developed in 1930 by the mathematician Vojtech Jarnik, later proposed by the computer scientist Robert C. Prim in 1957 and rediscovered by Edsger Dijkstra in 1959. This is a greedy algorithm that can find a minimum spanning tree in a connected, weighted and undirected graph by adding recursively minimum cost arcs leaving visited nodes.

Value

`msTreePrim` returns a list with:

`tree.nodes`	vector containing the nodes of the minimum cost spanning tree.
`tree.arcs`	matrix containing the list of arcs of the minimum cost spanning tree.
`stages`	number of stages required.
`stages.arcs`	stages in which each arc was added.

References

Prim, R. C. (1957), "Shortest Connection Networks And Some Generalizations", Bell System Technical Journal, 36 (1957), pp. 1389-1401

See Also

A more general function getMinimumSpanningTree.

removeLoops *Remove loops of a graph*

Description

This function reviews the arc list of a given graph and check if exists loops in it. A loop is an arc that connect a node with itself. If removeLoops find a loop remove it from the list of arcs.

Usage

```
removeLoops(arcs)
```

Arguments

arcs	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Value

removeLoops returns a new list of arcs without any of the loops founded.

removeMultiArcs *Remove multi-arcs with no minimum cost*

Description

The removeMultiArcs function go through the arcs list of a given graph and check if there are more than one arc between two nodes. If exist more than one, the function keeps one with minimum cost and remove the others.

Usage

```
removeMultiArcs(arcs, directed = TRUE)
```

Arguments

arcs	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
directed	logical value indicating whether the graph is directed (TRUE) or not (FALSE).

Value

removeMultiArcs returns a new list of arcs without any of the multi-arcs founded.

repGraph *Visual representation of a graph*

Description

The repGraph function uses igraph package to represent a graph.

Usage

```
repGraph(nodes, arcs, tree = NULL, directed = FALSE, plot.title = NULL,
  fix.seed = NULL)
```

Arguments

nodes	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
arcs	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
tree	matrix with the list of arcs of a tree, if there is one. Is NULL by default.
directed	logical value indicating whether the graph is directed (TRUE) or not (FALSE).
plot.title	string with main title of the graph. Is NULL by default.
fix.seed	number to set a seed for the representation.

Value

repGraph returns a plot with the given graph.

Examples

```
# Graph
nodes <- c(1:4)
arcs <- matrix(c(1,2,2, 1,3,15, 2,3,1, 2,4,9, 3,4,1),
               byrow = TRUE, ncol = 3)
# Plot graph
repGraph(nodes, arcs)
```

searchFlowPath *Find a maximum flow path*

Description

searchFlowPath go through a given graph and obtains a maximum flow path between source and sink nodes. The function uses a deep-first search estrategy.

Usage

```
searchFlowPath(nodes, arcs, source.node = 1,
  sink.node = nodes[length(nodes)])
```

Arguments

nodes	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
arcs	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
source.node	number pointing to the source node of the graph. It's node 1 by default.
sink.node	number pointing to the sink node of the graph. It's the last node by default.

Value

searchFlowPath returns a list with two elements:

path.nodes	vector with nodes of the path.
path.arcs	matrix with the list of arcs that form the maximum flow path.

See Also

This function is an auxiliar function used in ghTreeGusfield and getMinimumCutTree.

searchWalk *Finds an open walk in a graph*

Description

This function walks a given graph, directed or not, searching for a walk from a starting node to a final node. The searchWalk function uses a deep-first search strategy to returns the first open walk found, regardless it has formed cycles or repeated nodes.

Usage

```
searchWalk(nodes, arcs, directed = TRUE, start.node = nodes[1],
  end.node = nodes[length(nodes)], method = NULL)
```

Arguments

nodes	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
arcs	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
directed	logical value indicating whether the graph is directed (TRUE) or not (FALSE).
start.node	number with one node from which a walk start.
end.node	number with final node of the walk.
method	character string specifying which method use to select the arcs that will form the open walk: "min" if the function chooses the minimum weight arcs, "max" if chooses the maximum weight arcs, or NULL if chooses the arcs by their order in the list of arcs.

Value

If searchWalk found an open walk in the graph returns TRUE, a vector with the nodes of the walk and a matrix with the list of arcs of it.

---

searchZeroCycle *Zero weight cycle in a graph*

Description

Given a directed graph, searchZeroCycle search paths in it that forms a zero weight cycle. The function finishes when found one cycle.

Usage

```
searchZeroCycle(nodes, arcs)
```

Arguments

nodes	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
arcs	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Value

`searchZeroCycle` returns a vector with the nodes and a matrix with a list of arcs of the cycle found.

See Also

This function is an auxiliar function used in msArborEdmonds and getMinimumArborescence.

`spTreeBellmanFord` *Shortest path tree with Bellman-Ford algorithm*

Description

The `spTreeBellmanFord` function computes the shortest path tree of an undirected or directed graph with the Bellman-Ford algorithm.

Usage

```
spTreeBellmanFord(nodes, arcs, source.node = 1, directed = TRUE)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`source.node`	number pointing the source node of the graph. It's node 1 by default.
`directed`	logical value indicating whether the graph is directed (`TRUE`) or not (`FALSE`).

Details

The Bellman-Ford algorithm gets its name for two of the developers, Richard Bellman y Lester Ford Jr., that published it in 1958 and 1956 respectively. The same algorithm also was published independently in 1957 by Edward F. Moore.

The Bellman-Ford algorithm can compute the shortest path from a source node to the rest of nodes that make a connected graph, directed or not, with weights that can be negatives. If the graph is connected and there isn't negative cycles, the algorithm always finds a shortest path tree.

Value

`spTreeBellmanFord` returns a list with:

`tree.nodes`	vector containing the nodes of the shortest path tree.
`tree.arcs`	matrix containing the list of arcs of the shortest path tree.
`stages`	number of stages required.
`distances`	vector with distances from source to other nodes

References

Bellman, Richard (1958). "On a routing problem". Quarterly of Applied Mathematics 16, 87-90.

Ford Jr., Lester R. (1956). Network Flow Theory. Paper P-923. Santa Monica, California: RAND Corporation.

Moore, Edward F. (1959). "The shortest path through a maze". Proc. Internat. Sympos. Switching Theory 1957, Part II. Cambridge, Mass.: Harvard Univ. Press. pp. 285-292.

See Also

A more general function getShortestPathTree.

`spTreeDijkstra` *Shortest path tree with Dijkstra's algorithm*

Description

The `spTreeDijkstra` function computes the shortest path tree of an undirected or directed graph with Dijkstra's algorithm.

Usage

```
spTreeDijkstra(nodes, arcs, source.node = 1, directed = TRUE)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`source.node`	number pointing the source node of the graph. It's node 1 by default.
`directed`	logical value indicating whether the graph is directed (`TRUE`) or not (`FALSE`).

Details

Dijkstra's algorithm was developed by the computer scientist Edsger Dijkstra in 1956 and published in 1959. This is an algorithm that can computes a shortest path tree from a given source node to the others nodes that make a connected graph, directed or not, with non-negative weights.

Value

`spTreeDijkstra` returns a list with:

`tree.nodes`	vector containing the nodes of the shortest path tree.
`tree.arcs`	matrix containing the list of arcs of the shortest path tree.
`stages`	number of stages required.
`distances`	vector with distances from source to other nodes

References

Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs". Numerische Mathematik 1, 269-271.

See Also

A more general function getShortestPathTree.

`stepbackArbor` *Go back between two stages of the Edmond's algorithm*

Description

The `stepbackArbor` function rebuilds an arborescence present in earlier stage of Edmonds's algorithm to find a minimum cost arborescence.

Usage

```
stepbackArbor(before, after)
```

Arguments

`before`	list with elements of the previous stage
`after`	list with elements of the next stage

Value

A updated list of elements of the earlier stage with a new arborescence

See Also

This function is an auxiliar function used in msArborEdmonds and getMinimumArborescence.

Index