Package 'rlemon'

April 15, 2022

Type Package

Title R Access to LEMON Graph Algorithms

Version 0.2.0

Description Allows easy access to the LEMON Graph Library set of algorithms, written in C++.
See the LEMON project page at <https://lemon.cs.elte.hu/trac/lemon>.
Current LEMON version is 1.3.1.

License BSL-1.0

Imports Rcpp (>= 1.0.5)

LinkingTo Rcpp

URL https://errickson.net/rlemon/

BugReports https://github.com/josherrickson/rlemon/issues/

RoxygenNote 7.1.2

Encoding UTF-8

Suggests testthat, covr

Depends R (>= 2.10)

LazyData true

Language en-US

NeedsCompilation yes

Author Arav Agarwal [aut],
Aditya Tewari [aut],
Josh Errickson [cre, aut]

Maintainer Josh Errickson <jerrick@umich.edu>

Repository CRAN

Date/Publication 2022-04-15 15:40:02 UTC

R topics documented:

`AllPairsMinCut` *Solver for All-Pairs MinCut*

Description

Finds the all-pairs minimum cut tree, using the Gomory-Hu algorithm.

Usage

```
AllPairsMinCut(
  arcSources,
  arcTargets,
  arcWeights,
  numNodes,
  algorithm = "GomoryHu"
)
```

Arguments

`arcSources`	Vector corresponding to the source nodes of a graph's edges
`arcTargets`	Vector corresponding to the destination nodes of a graph's edges
`arcWeights`	Vector corresponding to the weights of a graph's arcs
`numNodes`	The number of nodes in the graph
`algorithm`	Choices of algorithm include "GomoryHu". "GomoryHu" is the default.

Details

For details on LEMON's implementation, including differences between the algorithms, see https://lemon.cs.elte.hu/pub/doc/1.3.1/a00182.html.

Value

A list containing three entries: 1) A vector of predecessor nodes of each node in the graph, and 2) A vector of weights of the predecessor edge of each node, and 3) A vector of distances from the root node to each node.

`CountBiEdgeConnectedComponents`
Count Number of Bi-Edge-Connected Components

Description

Counts the number of bi-edge-connected components in an undirected graph.

Usage

```
CountBiEdgeConnectedComponents(arcSources, arcTargets, numNodes)
```

Arguments

`arcSources`	Vector corresponding to the source nodes of a graph's edges
`arcTargets`	Vector corresponding to the destination nodes of a graph's edges
`numNodes`	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#ga4d5db78dc21099d075c3967484990954 for more information.

Value

An integer defining the number of bi-edge-connected components

---

`CountBiNodeConnectedComponents`
Count Number of Bi-Node-Connected Components

Description

Counts the number of bi-node-connected components in an undirected graph.

Usage

```
CountBiNodeConnectedComponents(arcSources, arcTargets, numNodes)
```

Arguments

`arcSources`	Vector corresponding to the source nodes of a graph's edges
`arcTargets`	Vector corresponding to the destination nodes of a graph's edges
`numNodes`	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#gaf7c5744b2175210b8ea67897aaa27885 for more information.

Value

An integer defining the number of bi-node-connected components

---

`CountConnectedComponents`

Count the Number of Connected Components

Description

The connected components are the classes of an equivalence relation on the nodes of an undirected graph. Two nodes are in the same class if they are connected with a path.

Usage

```
CountConnectedComponents(arcSources, arcTargets, numNodes)
```

Arguments

`arcSources`	Vector corresponding to the source nodes of a graph's edges
`arcTargets`	Vector corresponding to the destination nodes of a graph's edges
`numNodes`	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#ga33a9d9d4803cb15e83568b2526e978a5 for more information.

Value

An integer defining the number of connected components

`CountStronglyConnectedComponents`
Count the Number of Strongly Connected Components

Description

The strongly connected components are the classes of an equivalence relation on the nodes of a directed graph. Two nodes are in the same class if they are connected with directed paths in both direction.

Usage

```
CountStronglyConnectedComponents(arcSources, arcTargets, numNodes)
```

Arguments

`arcSources`	Vector corresponding to the source nodes of a graph's edges
`arcTargets`	Vector corresponding to the destination nodes of a graph's edges
`numNodes`	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#gad30bc47dfffb78234eeee903cb3766f4 for more information.

Value

An integer defining the number of strongly connected components

---

`FindBiEdgeConnectedComponents`
Find Bi-Edge-Connected Components

Description

The bi-edge-connected components are the classes of an equivalence relation on the nodes of an undirected graph. Two nodes are in the same class if they are connected with at least two edge-disjoint paths.

Usage

```
FindBiEdgeConnectedComponents(arcSources, arcTargets, numNodes)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
numNodes	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#ga76c1fdd1881d21677507100b7e96c983 for more information.

Value

A vector containing the node id of each bi-edge-connected component.

`FindBiEdgeConnectedCutEdges`

Find Bi-Edge-Connected Cut Edges

Description

The bi-edge-connected components are the classes of an equivalence relation on the nodes of an undirected graph. Two nodes are in the same class if they are connected with at least two edge-disjoint paths. The bi-edge-connected components are separted by the cut edges of the components.

Usage

```
FindBiEdgeConnectedCutEdges(arcSources, arcTargets, numNodes)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
numNodes	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#ga58d444eba448c5f1a53539bd1b69636e for more information.

Value

A list containing 1) A vector of cut edge sources, and 2) A vector of cut edge destinations.

`FindBiNodeConnectedComponents`

Find Bi-Node-Connected Components

Description

The bi-node-connected components are the classes of an equivalence relation on the edges of a undirected graph. Two edges are in the same class if they are on same circle.

Usage

```
FindBiNodeConnectedComponents(arcSources, arcTargets, numNodes)
```

Arguments

`arcSources`	Vector corresponding to the source nodes of a graph's edges
`arcTargets`	Vector corresponding to the destination nodes of a graph's edges
`numNodes`	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#ga9d70526ab54e10b4b6fe3762af8675dd for more information.

Value

A vector containing the arc id of each bi-node-connected component

---

`FindBiNodeConnectedCutNodes`

Find Bi-Node-Connected Cut Nodes

Description

The bi-node-connected components are the classes of an equivalence relation on the edges of a undirected graph. Two edges are in the same class if they are on same circle.

Usage

```
FindBiNodeConnectedCutNodes(arcSources, arcTargets, numNodes)
```

Arguments

`arcSources`	Vector corresponding to the source nodes of a graph's edges
`arcTargets`	Vector corresponding to the destination nodes of a graph's edges
`numNodes`	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#ga31461f33a748327ea3ef2a3199ffb6c7 for more information.

Value

A vector containing the cut nodes.

`FindConnectedComponents` *Find Connected Components*

Description

The connected components are the classes of an equivalence relation on the nodes of an undirected graph. Two nodes are in the same class if they are connected with a path.

Usage

```
FindConnectedComponents(arcSources, arcTargets, numNodes)
```

Arguments

`arcSources`	Vector corresponding to the source nodes of a graph's edges
`arcTargets`	Vector corresponding to the destination nodes of a graph's edges
`numNodes`	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#gaa467a3e0a8c2e9e762650fd01fadff89 for more information.

Value

A vector containing the node id of each connected component.

`FindStronglyConnectedComponents`
Find Strongly Connected Components

Description

The strongly connected components are the classes of an equivalence relation on the nodes of a directed graph. Two nodes are in the same class if they are connected with directed paths in both direction.

Usage

```
FindStronglyConnectedComponents(arcSources, arcTargets, numNodes)
```

Arguments

`arcSources`	Vector corresponding to the source nodes of a graph's edges
`arcTargets`	Vector corresponding to the destination nodes of a graph's edges
`numNodes`	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#ga46f8c22f3e2989c4689faa4c46ec9436 for more information.

Value

A vector containing the node id of each strongly connected component.

`FindStronglyConnectedCutArcs`
Find Strongly Connected Cut Arcs

Description

The strongly connected components are the classes of an equivalence relation on the nodes of a directed graph. Two nodes are in the same class if they are connected with directed paths in both direction. The strongly connected components are separated by the cut arcs.

Usage

```
FindStronglyConnectedCutArcs(arcSources, arcTargets, numNodes)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
numNodes	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#gad7af5c3a97453e37f251f0e86dbb83db for more information.

Value

A list containing 1) A vector of cut arc sources, and 2) A vector of cut arc destinations.

GetAndCheckTopologicalSort

Check if Graph is DAG, then Sorts Nodes into Topological Order

---

Description

Checks if a directed graph is a DAG and returns the topological order.

Usage

```
GetAndCheckTopologicalSort(arcSources, arcTargets, numNodes)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
numNodes	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#gaf10c5e1630e5720c20d83cfb77dbf024 for more information.

Value

A list containing 1) A `logical` stating if the graph is a dag, and 2) A vector of length `numNodes`, containing the index of vertex i in the ordering at location i

GetBipartitePartitions *Obtains (if possible) Bipartite Split*

Description

Checks if an undirected graph is bipartite and finds the bipartite partitions.

Usage

```
GetBipartitePartitions(arcSources, arcTargets, numNodes)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
numNodes	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#ga58ba1d00c569f0eb0deb42afca9f80bb for more information.

Value

A list containing 1) A logical stating if the graph is bipartite, and 2) A vector of length numNodes, containing the partition for each node

GetTopologicalSort *Sorts Nodes into Topological Order*

Description

Gives back the topological order of a DAG.

Usage

```
GetTopologicalSort(arcSources, arcTargets, numNodes)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
numNodes	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#gafc2cb20cf3859f157c0e12da7f310bb3 for more information.

Value

A vector of length numNodes, containing the index of vertex i in the ordering at location i.

GraphSearch *Solver for Graph Search*

Description

Runs a common graph search algorithm to find the minimum cardinality shortest path. Finds the shortest path from/to all vertices if a start/end node are not given.

Usage

```
GraphSearch(
  arcSources,
  arcTargets,
  numNodes,
  startNode = -1,
  endNode = -1,
  algorithm = "Bfs"
)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
numNodes	The number of nodes in the graph
startNode	Optional start node of the path
endNode	Optional end node of the path
algorithm	Choices of algorithm include "Bfs" (Breadth First Search) and "Dfs" (Depth First Search). Bfs is the default.

Details

For details on LEMON's implementation, including differences between the algorithms, see https://lemon.cs.elte.hu/pub/doc/1.3.1/a00608.html.

Value

A list containing three entries: 1) the predecessor of each vertex in its shortest path, 2) the distances from each node to the startNode, 3) a vector of logicals indicating whether a node was reached.

GrossoLocatelliPullanMcRunner
LEMON runners

Description

These "runner" functions provide a slightly lower-level access to LEMON.

Usage

```
GrossoLocatelliPullanMcRunner(arcSources, arcTargets, numNodes)

getBipartitePartitionsRunner(arcSources, arcTargets, numNodes)

getAndCheckTopologicalSortRunner(arcSources, arcTargets, numNodes)

getTopologicalSortRunner(arcSources, arcTargets, numNodes)

IsConnectedRunner(arcSources, arcTargets, numNodes)

IsAcyclicRunner(arcSources, arcTargets, numNodes)

IsTreeRunner(arcSources, arcTargets, numNodes)

IsBipartiteRunner(arcSources, arcTargets, numNodes)

IsStronglyConnectedRunner(arcSources, arcTargets, numNodes)

IsDAGRunner(arcSources, arcTargets, numNodes)

IsBiNodeConnectedRunner(arcSources, arcTargets, numNodes)

IsBiEdgeConnectedRunner(arcSources, arcTargets, numNodes)

IsLoopFreeRunner(arcSources, arcTargets, numNodes)

IsParallelFreeRunner(arcSources, arcTargets, numNodes)

IsSimpleGraphRunner(arcSources, arcTargets, numNodes)

IsEulerianRunner(arcSources, arcTargets, numNodes)

CountBiEdgeConnectedComponentsRunner(arcSources, arcTargets, numNodes)

CountConnectedComponentsRunner(arcSources, arcTargets, numNodes)

CountBiNodeConnectedComponentsRunner(arcSources, arcTargets, numNodes)
```

```
CountStronglyConnectedComponentsRunner(arcSources, arcTargets, numNodes)

FindStronglyConnectedComponentsRunner(arcSources, arcTargets, numNodes)

FindStronglyConnectedCutArcsRunner(arcSources, arcTargets, numNodes)

FindBiEdgeConnectedCutEdgesRunner(arcSources, arcTargets, numNodes)

FindBiNodeConnectedComponentsRunner(arcSources, arcTargets, numNodes)

FindBiNodeConnectedCutNodesRunner(arcSources, arcTargets, numNodes)

FindConnectedComponentsRunner(arcSources, arcTargets, numNodes)

FindBiEdgeConnectedComponentsRunner(arcSources, arcTargets, numNodes)

GraphCompatabilityConverter(nodesList, arcSources, arcTargets)

BfsRunner(arcSources, arcTargets, numNodes, startNode = -1L, endNode = -1L)

DfsRunner(arcSources, arcTargets, numNodes, startNode = -1L, endNode = -1L)

MaxCardinalitySearchRunner(
  arcSources,
  arcTargets,
  arcCapacities,
  numNodes,
  startNode = -1L
)

CirculationRunner(
  arcSources,
  arcTargets,
  arcLowerBound,
  arcUpperBound,
  nodeSupplies,
  numNodes
)

PreflowRunner(
  arcSources,
  arcTargets,
  arcDistances,
  sourceNode,
  destinationNode,
  numNodes
)
```

```
EdmondsKarpRunner(
  arcSources,
  arcTargets,
  arcDistances,
  sourceNode,
  destinationNode,
  numNodes
)

MaximumWeightPerfectMatchingRunner(
  arcSources,
  arcTargets,
  arcWeights,
  numNodes
)

MaximumWeightFractionalPerfectMatchingRunner(
  arcSources,
  arcTargets,
  arcWeights,
  numNodes
)

MaximumWeightFractionalMatchingRunner(
  arcSources,
  arcTargets,
  arcWeights,
  numNodes
)

MaximumWeightMatchingRunner(arcSources, arcTargets, arcWeights, numNodes)

MaximumCardinalityMatchingRunner(arcSources, arcTargets, numNodes)

MaximumCardinalityFractionalMatchingRunner(arcSources, arcTargets, numNodes)

CycleCancellingRunner(
  arcSources,
  arcTargets,
  arcCapacities,
  arcCosts,
  nodeSupplies,
  numNodes
)

CapacityScalingRunner(
  arcSources,
```

```
  arcTargets,
  arcCapacities,
  arcCosts,
  nodeSupplies,
  numNodes
)

CostScalingRunner(
  arcSources,
  arcTargets,
  arcCapacities,
  arcCosts,
  nodeSupplies,
  numNodes
)

NetworkSimplexRunner(
  arcSources,
  arcTargets,
  arcCapacities,
  arcCosts,
  nodeSupplies,
  numNodes
)

NagamochiIbarakiRunner(arcSources, arcTargets, arcWeights, numNodes)

HaoOrlinRunner(arcSources, arcTargets, arcWeights, numNodes)

GomoryHuTreeRunner(arcSources, arcTargets, arcWeights, numNodes)

HowardMmcRunner(arcSources, arcTargets, arcDistances, numNodes)

KarpMmcRunner(arcSources, arcTargets, arcDistances, numNodes)

HartmannOrlinMmcRunner(arcSources, arcTargets, arcDistances, numNodes)

KruskalRunner(arcSources, arcTargets, arcDistances, numNodes)

MinCostArborescenceRunner(
  arcSources,
  arcTargets,
  arcDistances,
  sourceNode,
  numNodes
)

PlanarCheckingRunner(arcSources, arcTargets, numNodes)
```

```
PlanarEmbeddingRunner(arcSources, arcTargets, numNodes)

PlanarColoringRunner(arcSources, arcTargets, numNodes, useFiveAlg = TRUE)

PlanarDrawingRunner(arcSources, arcTargets, numNodes)

SuurballeRunner(
  arcSources,
  arcTargets,
  arcDistances,
  numNodes,
  startNode,
  endNode
)

DijkstraRunner(arcSources, arcTargets, arcDistances, numNodes, startNode)

BellmanFordRunner(arcSources, arcTargets, arcDistances, numNodes, startNode)

ChristofidesRunner(
  arcSources,
  arcTargets,
  arcDistances,
  numNodes,
  defaultEdgeWeight = 999999L
)

GreedyTSPRunner(
  arcSources,
  arcTargets,
  arcDistances,
  numNodes,
  defaultEdgeWeight = 999999L
)

InsertionTSPRunner(
  arcSources,
  arcTargets,
  arcDistances,
  numNodes,
  defaultEdgeWeight = 999999L
)

NearestNeighborTSPRunner(
  arcSources,
  arcTargets,
  arcDistances,
```

```
  numNodes,
  defaultEdgeWeight = 999999L
)

Opt2TSPRunner(
  arcSources,
  arcTargets,
  arcDistances,
  numNodes,
  defaultEdgeWeight = 999999L
)

lemon_runners()
```

Arguments

arcSources	a vector corresponding to the source nodes of a graph's edges
arcTargets	a vector corresponding to the destination nodes of a graph's edges
numNodes	the number of nodes in the graph
nodesList	a vector of all the nodes in the graph
startNode	in path-based algorithms, the start node of the path
endNode	in path-based algorithms, the end node of the path
arcCapacities	vector corresponding to the capacities of nodes of a graph's edges
arcLowerBound	vector corresponding to the lower-bound capacities of nodes of a graph's edges
arcUpperBound	vector corresponding to the upper-bound capacities of nodes of a graph's edges
nodeSupplies	vector corresponding to the supplies of each node of the graph
arcDistances	vector corresponding to the distances of a graph's edges
sourceNode	in flow-based algorithms, the source node of the flow
destinationNode	in flow-based algorithms, the destination node of the flow
arcWeights	vector corresponding to the weights of a graph's arcs
arcCosts	vector corresponding to the costs of nodes of a graph's edges
useFiveAlg	if TRUE (default), run a 5-color algorithm. If FALSE, runs a faster 6-coloring algorithm instead.
defaultEdgeWeight	The default edge weight if an edge is not-specified (default value 999999)

Details

Internally, all exported rlemon functions call a "runner" function to interface with the C++, for example, `MaxFlow(...,algorithm = "PreFlow")` will call `PreFlowRunner(...)`.

In almost all cases, users will want to stick with the exported functions.

Runners differ from exported functions in two significant way and one minor way:

1. Exported functions provide input checking.
2. Exported functions provide slightly cleaner output, such as converting 0/1 boolean into `logical`.
3. The `arcWeights` argument is optional to `MaxMatching()`, automatically generating a constant weight if it is excluded. `arcWeights` is not optional in `MaxMatchingRunner()`.

Value

Algorithm results

`IsAcyclic` *Check if Graph is Acyclic.*

Description

A cycle is a path starting and ending in the same node and containing at least one other node. A acyclic graph contains no cycles.

Usage

```
IsAcyclic(arcSources, arcTargets, numNodes)
```

Arguments

`arcSources`	Vector corresponding to the source nodes of a graph's edges
`arcTargets`	Vector corresponding to the destination nodes of a graph's edges
`numNodes`	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#ga14c191b2133a1dd23e1527f074c821c0 for more information.

Value

A `logical` stating if the graph is acyclic

IsBiEdgeConnected *Chcek if Graph is Bi-Edge-Connected*

Description

Checks if an undirected graph is bi-edge-connected, that is if there are no edges that, if removed, would split the graph into two unconnected graphs.

Usage

```
IsBiEdgeConnected(arcSources, arcTargets, numNodes)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
numNodes	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#ga37d22a2ddd5a064a9203720f2b93518e for more information.

Value

A logical stating if the graph is bi-edge connected

IsBiNodeConnected *Checks if Graph is Bi-Node-Connected*

Description

Checks if an undirected graph is bi-node-connected, that is if there is are no nodes which, if removed, would split the graph into two unconnected graphs.

Usage

```
IsBiNodeConnected(arcSources, arcTargets, numNodes)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
numNodes	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#gac9257323ead7cbe64b7b4a628c4876b3 for more information.

Value

A `logical` stating if the graph is bi-node connected

`IsBipartite` *Checks if Graph is Bipartite*

Description

A bipartite graph is one whose nodes can be divided into two disjoint and independent sets such that edges only connecte between those two sets and not within a set.

Usage

```
IsBipartite(arcSources, arcTargets, numNodes)
```

Arguments

`arcSources`	Vector corresponding to the source nodes of a graph's edges
`arcTargets`	Vector corresponding to the destination nodes of a graph's edges
`numNodes`	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#ga577db110d33bd487aaad5bfffb31c6f5 for more information.

Value

A `logical` stating if the graph is bipartite

IsConnected *Check if Graph is Connected*

Description

A connected graph has a path between any two nodes in the graph.

Usage

```
IsConnected(arcSources, arcTargets, numNodes)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
numNodes	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#gad5c8d1b650f6b614a852f8430d90e184 for more information.

Value

A logical stating if the graph is connected

IsDAG *Check if Graph is a DAG.*

Description

A graph is a DAG if it is Directed and Acyclic.

Usage

```
IsDAG(arcSources, arcTargets, numNodes)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
numNodes	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#gaef2b43c8cd1d74e15fa5c7607bc5e396 for more information.

Value

A `logical` stating if the graph is DAG

`IsEulerian` *Check if Graph is Eulerian*

Description

A directed graph is Eulerian if and only if it is connected and the number of incoming and outgoing edges are the same for each node. An undirected graph is Eulerian if and only if it is connected and the number of incident edges is even for each node.

Usage

```
IsEulerian(arcSources, arcTargets, numNodes)
```

Arguments

`arcSources`	Vector corresponding to the source nodes of a graph's edges
`arcTargets`	Vector corresponding to the destination nodes of a graph's edges
`numNodes`	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#gafb5a4961cac4d877006869fc4cb6ea1d for more information.

Value

TRUE if graph is Eulerian, FALSE otherwise

IsLoopFree *Checks if Graph is Loop Free*

Description

A loop is an edge that starts and ends at the same node and passes through no other nodes.

Usage

```
IsLoopFree(arcSources, arcTargets, numNodes)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
numNodes	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#ga127f3963003cd532c79c226885fe1c8c for more information.

Value

TRUE if the graph is loop free, FALSE otherwise

IsParallelFree *Check if Graph is Parallel Free*

Description

Parallel edges occur when there are two edges between a single pair of nodes.

Usage

```
IsParallelFree(arcSources, arcTargets, numNodes)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
numNodes	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#gaa05e0683f90b69f31eb29fe7d09afde4 for more information.

Value

TRUE if the graph is parallel free, FALSE otherwise

IsSimpleGraph *Check if Graph is Simple*

Description

A graph is simple if it is both loop free, and parallel free. See also `IsLoopFree` and `IsParallelFree`.

Usage

```
IsSimpleGraph(arcSources, arcTargets, numNodes)
```

Arguments

`arcSources`	Vector corresponding to the source nodes of a graph's edges
`arcTargets`	Vector corresponding to the destination nodes of a graph's edges
`numNodes`	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#gae4c7ae734e2509ab78dc747d602c9236 for more information.

Value

TRUE if graph is simple, FALSE otherwise.

IsStronglyConnected *Check if Graph is Strongly Connected*

Description

A directed graph is strongly connected if any two nodes are connected via paths in both directions.

Usage

```
IsStronglyConnected(arcSources, arcTargets, numNodes)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
numNodes	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#gacd21b34d7b42b9835a204a57fcf15964 for more information.

Value

A logical stating if the graph is strongly connected

IsTree *Check if Graph is a Tree*

Description

A tree is an undirected graph in which any two nodes are connected by exactly one path, or equivalently is both connected and acyclic.

Usage

```
IsTree(arcSources, arcTargets, numNodes)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
numNodes	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00616.html#gad1e4de234e926958647905478415bd54 for more information.

Value

A `logical` stating if the graph is a tree

`MaxCardinalityMatching` *Solve for Maximum Cardinality Matching*

Description

Finds the maximum cardinality matching in graphs and bipartite graphs.

Usage

```
MaxCardinalityMatching(
  arcSources,
  arcTargets,
  numNodes,
  algorithm = "MaxMatching"
)
```

Arguments

`arcSources`	Vector corresponding to the source nodes of a graph's edges
`arcTargets`	Vector corresponding to the destination nodes of a graph's edges
`numNodes`	The number of nodes in the graph
`algorithm`	Choices of algorithm include "MaxMatching" and "MaxFractionalMatching". "MaxMatching" is the default.

Details

For details on LEMON's implementation, including differences between the algorithms, see https://lemon.cs.elte.hu/pub/doc/1.3.1/a00615.html.

Value

A list containing two entries: 1) The matching value, 2) The edges of the final graph, in a List of (node, node) pairs

`MaxCardinalitySearch` *Solver for Max Cardinality Search*

Description

Runs the maximum cardinality search algorithm on a directed graph. The maximum cardinality search first chooses any node of the digraph. Then every time it chooses one unprocessed node with maximum cardinality, i.e the sum of capacities on out arcs to the nodes which were previously processed. If there is a cut in the digraph the algorithm should choose again any unprocessed node of the digraph.

Usage

```
MaxCardinalitySearch(
  arcSources,
  arcTargets,
  arcCapacities,
  numNodes,
  startNode = -1,
  algorithm = "maxcardinalitysearch"
)
```

Arguments

`arcSources`	Vector corresponding to the source nodes of a graph's edges
`arcTargets`	Vector corresponding to the destination nodes of a graph's edges
`arcCapacities`	Vector corresponding to the distances of a graph's edges
`numNodes`	The number of nodes in the graph
`startNode`	Optional start node of the path
`algorithm`	Choices of algorithm include "maxcardinalitysearch". maxcardinalitysearch is the default.

Details

For details on LEMON's implementation, including differences between the algorithms, see https://lemon.cs.elte.hu/pub/doc/1.3.1/a00255.html.

Value

A list containing two entries: 1) the cardinality of each node , 2) a logical vector indicating whether a node was reached or not

MaxClique *Solver for Largest Complete Subgroup (All Nodes Connected)*

Description

Finds the largest complete subgraph (clique) in an undirected graph via approximation algorithms for the maximal clique problem.

Usage

```
MaxClique(
  arcSources,
  arcTargets,
  numNodes,
  algorithm = "GrossoLocatelliPullanMc"
)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
numNodes	The number of nodes in the graph
algorithm	Choices of algorithm include "GrossoLocatelliPullanMc". GrossoLocatelliPullanMc is the default.

Details

For details on LEMON's implementation, including differences between the algorithms, see https://lemon.cs.elte.hu/pub/doc/1.3.1/a00194.html.

Value

A list containing two entries: 1) the clique size, and 2) the members of the clique.

MaxFlow *Solver for MaxFlow*

Description

Finds the maximum flow of a directed graph, given a source and destination node.

Usage

```
MaxFlow(
  arcSources,
  arcTargets,
  arcCapacities,
  sourceNode,
  destNode,
  numNodes,
  algorithm = "Preflow"
)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
arcCapacities	Vector corresponding to the capacities of nodes of a graph's edges
sourceNode	The source node
destNode	The destination node
numNodes	The number of nodes in the graph
algorithm	Choices of algorithm include "Preflow" and "EdmondsKarp". "Preflow" is the default.

Details

For details on LEMON's implementation, including differences between the algorithms, see https://lemon.cs.elte.hu/pub/doc/1.3.1/a00611.html.

Value

A list containing three entries: 1) A vector corresponding to the flows of arcs in the graph, 2) A vector of cut-values of the graph's nodes, and 3) the total cost of the flows in the graph, i.e. the maxflow value.

MaxMatching *Solver for Maximum Weighted Matching*

Description

Finds the maximum weighted matching in graphs and bipartite graphs. Each algorithm in this set returns different outputs depending on different situations, like PerfectMatching or PerfectFractionalMathing.

Usage

```
MaxMatching(
  arcSources,
  arcTargets,
  arcWeights = NULL,
  numNodes,
  algorithm = "MaxWeightedMatching"
)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
arcWeights	Vector corresponding to the weights of a graph's edges. Default is NULL for unweight matching.
numNodes	The number of nodes in the graph
algorithm	Choices of algorithm include "MaxWeightedMatching", "MaxWeightedPerfectMatching", "MaxWeightedFractionalMatching", and "MaxWeightedPerfectFractionalMatching". "MaxWeightedMatching" is the default.

Details

For details on LEMON's implementation, including differences between the algorithms, see https://lemon.cs.elte.hu/pub/doc/1.3.1/a00615.html.

Value

A list containing two entries: 1) The matching value, 2) The edges of the final graph, in a list of (node, node) pairs

---

MinCostArborescence *Solver for Minimum Cost Arborescence*

Description

Finds the minimum cost arborescence of a graph, returning both the cost and the pairs of nodes for the edges in the arborescence.

Usage

```
MinCostArborescence(
  arcSources,
  arcTargets,
  arcDistances,
  sourceNode,
  numNodes,
  algorithm = "MinCostArborescence"
)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
arcDistances	Vector corresponding to the distances of nodes of a graph's edges
sourceNode	The source node
numNodes	The number of nodes in the graph
algorithm	Choices of algorithm include "MinCostArborescence". "MinCostArborescence" is the default.

Details

For details on LEMON's implementation, including differences between the algorithms, see https://lemon.cs.elte.hu/pub/doc/1.3.1/a00264.html.

Value

A list containing three entries: 1) A vector corresponding the source nodes of the edges in the tree, 2) a vector corresponding the target nodes of the edges in the tree, and 3) the total cost of the arborescence.

MinCostFlow *Solver for MinCostFlow*

Description

Finds the minimum cost flow of a directed graph.

Usage

```
MinCostFlow(
  arcSources,
  arcTargets,
  arcCapacities,
  arcCosts,
  nodeSupplies,
  numNodes,
  algorithm = "NetworkSimplex"
)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
arcCapacities	Vector corresponding to the capacities of nodes of a graph's edges
arcCosts	Vector corresponding to the capacities of nodes of a graph's edges

nodeSupplies	Vector corresponding to the supplies of each node
numNodes	The number of nodes in the graph
algorithm	Choices of algorithm include "NetworkSimplex", "CostScaling", "CapacityScaling", and "CycleCancelling". NetworkSimplex is the default.

Details

For details on LEMON's implementation, including differences between the algorithms, see https://lemon.cs.elte.hu/pub/doc/1.3.1/a00612.html.

Value

A list containing three entries: 1) A vector corresponding to the flows of arcs in the graph, 2) A vector of potentials of the graph's nodes, 3) the total cost of the flows in the graph, i.e. the mincostflow value, and 4) LEMON's feasibility type, demonstrating how feasible the graph problem is, one of "INFEASIBLE", "OPTIMAL", and "UNBOUNDED"

MinCut *Solver for MinCut*

Description

Finds the minimum cut on graphs. NagamochiIbaraki calculates the min cut value and edges in undirected graphs,while HaoOrlin calculates the min cut value and edges in directed graphs.

Usage

```
MinCut(
  arcSources,
  arcTargets,
  arcWeights,
  numNodes,
  algorithm = "NagamochiIbaraki"
)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
arcWeights	Vector corresponding to the weights of a graph's arcs
numNodes	The number of nodes in the graph
algorithm	Choices of algorithm include "NagamochiIbaraki" and "HaoOrlin". "NagamochiIbaraki" is the default.

Details

For details on LEMON's implementation, including differences between the algorithms, see https://lemon.cs.elte.hu/pub/doc/1.3.1/a00613.html.

Value

A list containing three entries: 1) The value of the minimum cut in the graph, and 2) A vector of nodes in the first partition, and 3) A vector of nodes in the second partition. GomoryHu calculates a Gomory-Hu Tree and returns a list containing three entries: 1) A vector of predecessor nodes of each node in the graph, and 2) A vector of weights of the predecessor edge of each node, and 3) A vector of distances from the root node to each node.

MinMeanCycle *Solver for Minimum Mean Cycle*

Description

Finds the Minimum Mean Cycle in directed graphs.

Usage

```
MinMeanCycle(
  arcSources,
  arcTargets,
  arcDistances,
  numNodes,
  algorithm = "Howard"
)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
arcDistances	Vector corresponding to the distances of a graph's edges
numNodes	The number of nodes in the graph
algorithm	Choices of algorithm include "Howard", "Karp", and "HartmannOrlin". "Howard" is the default.

Details

For details on LEMON's implementation, including differences between the algorithms, see https://lemon.cs.elte.hu/pub/doc/1.3.1/a00614.html.

Value

A list containing two entries: 1) A vector containing the costs of each edge in the MMC, and 2) the nodes in the MMC.

`MinSpanningTree` *Solver for Minimum Spanning Tree*

Description

Finds the minimum spanning tree of a graph. The minimum spanning tree is the minimal connected acyclic subgraph of a graph, assuming the graph is undirected.

Usage

```
MinSpanningTree(
  arcSources,
  arcTargets,
  arcDistances,
  numNodes,
  algorithm = "Kruskal"
)
```

Arguments

`arcSources`	Vector corresponding to the source nodes of a graph's edges
`arcTargets`	Vector corresponding to the destination nodes of a graph's edges
`arcDistances`	Vector corresponding to the distances of nodes of a graph's edges
`numNodes`	The number of nodes in the graph
`algorithm`	Choices of algorithm include "Kruskal". "Kruskal" is the default.

Details

For details on LEMON's implementation, including differences between the algorithms, see https://lemon.cs.elte.hu/pub/doc/1.3.1/a00610.html#ga233792b2c44a3581b85a775703e045af

Value

A list containing three entries: 1) A vector corresponding the source nodes of the edges in the tree, 2) a vector corresponding the target nodes of the edges in the tree, and 3) the total minimum spanning tree value.

`NetworkCirculation` *Solver for Network Circulation*

Description

Finds the solution to the network circulation problem via the push-relabel circulation algorithm.

Usage

```
NetworkCirculation(
  arcSources,
  arcTargets,
  arcLowerBound,
  arcUpperBound,
  nodeSupplies,
  numNodes,
  algorithm = "Circulation"
)
```

Arguments

`arcSources`	Vector corresponding to the source nodes of a graph's edges
`arcTargets`	Vector corresponding to the destination nodes of a graph's edges
`arcLowerBound`	Vector corresponding to the lower-bound capacities of nodes of a graph's edges
`arcUpperBound`	Vector corresponding to the upper-bound capacities of nodes of a graph's edges
`nodeSupplies`	Vector corresponding to the supplies of each node of the graph.
`numNodes`	The number of nodes in the graph
`algorithm`	Choices of algorithminclude "Circulation". "Circulation" is the default.

Details

For details on LEMON's implementation, including differences between the algorithms, see https://lemon.cs.elte.hu/pub/doc/1.3.1/a00078.html.

Value

A list containing two entries: 1) A vector corresponding to the flows of arcs in the graph, and 2) A vector of the graph's barrier nodes.

PlanarChecking *Check if Graph is Planar*

Description

Checks if an undirected graph is planar.

Usage

```
PlanarChecking(arcSources, arcTargets, numNodes)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
numNodes	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00617.html#ga230242aa2ee36f9b1b5a58f2c53016eb for more information.

Value

A logical stating if the graph is planar or not.

PlanarColoring *Solver for Planar Coloring*

Description

Checks if a graph is planar and returns the coloring of the graph

Usage

```
PlanarColoring(arcSources, arcTargets, numNodes, algorithm = "fiveColoring")
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
numNodes	The number of nodes in the graph
algorithm,	the algorithm to use. "sixColoring" generates a 6-coloring of the graph, while "fiveColoring" generates a 5-coloring.

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00306.html for more information.

Value

A list containing 1) A `logical` if the graph is planar, 2) the color of each vertex of the graph

---

`PlanarDrawing` *Solver for Planar Drawing*

Description

The planar drawing algorithm calculates positions for the nodes in the plane. These coordinates satisfy that if the edges are represented with straight lines, then they will not intersect each other.

Usage

```
PlanarDrawing(arcSources, arcTargets, numNodes)
```

Arguments

`arcSources`	Vector corresponding to the source nodes of a graph's edges
`arcTargets`	Vector corresponding to the destination nodes of a graph's edges
`numNodes`	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00307.html for more information.

Value

A list of 1) a `logical` of if the graph is planar, 2) the x-coordinate of the planar embedding, 3) the y-coordinate of the planar embedding

PlanarEmbedding *Solver for Planar Embedding*

Description

Checks if an undirected graph is planar and returns a list of outputs related to the planar embedding

Usage

```
PlanarEmbedding(arcSources, arcTargets, numNodes)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
numNodes	The number of nodes in the graph

Details

See https://lemon.cs.elte.hu/pub/doc/1.3.1/a00308.html for more information.

Value

A list containing 1) A logical indicating if the graph is planar, 2) the start nodes of the arcs of the embedding, 3) the end nodes of the arcs of the planar embedding, 4) the start nodes of the edges of the kuratowski subdivision, 5) the end nodes of the edges of the kuratowski subdivision.

ShortestPath *Solver for Shortest Path Between Two Nodes*

Description

Finds the shortest arc disjoint paths between two nodes in a directed graph. This implementation runs a variation of the successive shortest path algorithm.

Usage

```
ShortestPath(
  arcSources,
  arcTargets,
  arcDistances,
  numNodes,
  sourceNode,
  destNode,
  algorithm = "Suurballe"
)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
arcDistances	Vector corresponding to the distances of a graph's edges
numNodes	The number of nodes in the graph
sourceNode	The start node of the path
destNode	The end node of the path
algorithm	Choices of algorithm include "Suurballe". "Suurballe" is the default.

Details

For details on LEMON's implementation, including differences between the algorithms, see https://lemon.cs.elte.hu/pub/doc/1.3.1/a00609.html.

Value

A list containing two entries: 1) the number of paths from the start node to the end node and 2) a list of paths found. If there are multiple paths, then the second entry will have multiple paths.

ShortestPathFromSource

Solve for Shortest Path from Source Node to All Other Nodes

Description

Finds the shortest path from a source node to the rest of the nodes in a directed graph. These shortest path algorithms consider the distances present in the graph, as well as the number of edges.

Usage

```
ShortestPathFromSource(
  arcSources,
  arcTargets,
  arcDistances,
  numNodes,
  sourceNode,
  algorithm = "Dijkstra"
)
```

Arguments

arcSources	Vector corresponding to the source nodes of a graph's edges
arcTargets	Vector corresponding to the destination nodes of a graph's edges
arcDistances	Vector corresponding to the distances of a graph's edges
numNodes	The number of nodes in the graph
sourceNode	The source node
algorithm	Choices of algorithm include "Dijkstra" and "BellmanFord". "Dijkstra" is the default.

Details

For details on LEMON's implementation, including differences between the algorithms, see https://lemon.cs.elte.hu/pub/doc/1.3.1/a00609.html.

Value

A list containing two entries: 1) the distances from each node to the startNode and 2) the predecessor of each vertex in its shortest path.

---

small_graph_example *A small network graph example*

Description

A small network graph example

Usage

```
small_graph_example
```

Format

A list of length 5.

`TravelingSalesperson` *Solver for Traveling Salesperson Problem*

Description

Finds approximations for the travelling salesperson problem using approximation algorithms on graphs. NOTE: LEMON's TSP uses a complete graph in its backend, so expect less performance on sparse graphs.

Usage

```
TravelingSalesperson(
  arcSources,
  arcTargets,
  arcDistances,
  numNodes,
  defaultEdgeWeight = 999999,
  algorithm = "Christofides"
)

TravellingSalesperson(
  arcSources,
  arcTargets,
  arcDistances,
  numNodes,
  defaultEdgeWeight = 999999,
  algorithm = "Christofides"
)
```

Arguments

`arcSources`	Vector corresponding to the source nodes of a graph's edges
`arcTargets`	Vector corresponding to the destination nodes of a graph's edges
`arcDistances`	Vector corresponding to the distances of a graph's edges
`numNodes`	The number of nodes in the graph
`defaultEdgeWeight`	The default edge weight if an edge is not-specified (default value 999999)
`algorithm`	Choices of algorithm include "Christofides", "Greedy", "Insertion", "Nearest-Neighbor", and "Opt2". "Christofides" is the default.

Details

For details on LEMON's implementation, including differences between the algorithms, see https://lemon.cs.elte.hu/pub/doc/1.3.1/a00618.html.

Value

a List with 1) the vector of visited nodes in order, and 2) the total tour cost

Index