`Network` — library for graph theory

Table of contents

`Network::addEdge` – **adds one or several edges to a network**

`Network::addEdge` augments an existing network by new edges

Call(s):

- `Network::addEdge(G, e<, Eweight=c><, Capacity=t>)`
- `Network::addEdge(G, l<, Eweight=lc><, Capacity=lt>)`

Parameters:

`lc,lt` — lists of numbers
`c,t` — numbers
`l` — list of edges
`e` — edge
`G` — network

Options:

`Eweight` — The weight(s) for the new edge(s). Default is 1.
`Capacity` — The capacity/capacities for the new edge(s). Default is 1.

Return Value: The augmented network

Details:

- `Network::addEdge` adds one or several edges to an already existing network. An edge is represented by a list containing two nodes of the network. An error is raised if the specified edge is already contained in the network.
- `Network::addEdge(G,e)` adds the edge `e` to the network `G`. The endpoints of the edge must be nodes of the network. Otherwise an error is raised. A weight and a capacity can be set for the new edge with `Network::addEdge(G,e,Eweight=c,Capacity=t)`. If these specifications are missing, the default values 1 are assumed.
- Several edges can be added with `Network::addEdge(G,l)`, where `l` is a list of edges. For every edge in the list the endpoints have to be nodes of the network. With `Network::addEdge(G,l,Eweight=lc,Capacity=lt)` weights and capacities can be specified. Here `lc` and `lt` are numerical lists with exactly the same number of elements as `l`.

Example 1. We construct a cyclic network and add a few edges.

```
>> N1 := Network::cycle([v1,v2,v3,v4]):
   Network::edge(N1)

            [[v1, v2], [v2, v3], [v3, v4], [v4, v1]]

>> N2 := Network::addEdge(N1, [v1,v3]):
   Network::edge(N2)

        [[v1, v2], [v2, v3], [v3, v4], [v4, v1], [v1, v3]]
```

Now, both `v2` and `v3` are direct successors of `v1`.

```
>> Network::epost(N2)[v1];

                              [v2, v3]
```

`Network::addEdge` can augment a network with weights and capacities.

```
>> N3 := Network::addEdge(N1, [[v1,v3],[v1,v4]], Capacity = [3,5]):
   Network::eCapacity(N3);

                          table(
                            [v1, v4] = 5,
                            [v1, v3] = 3,
                            [v4, v1] = 1,
                            [v3, v4] = 1,
                            [v2, v3] = 1,
                            [v1, v2] = 1
                          )
```

Changes:

- `Network::addEdge` used to be `Network::AddEdges`.

`Network::addVertex` – adds one or several vertices to a network

`Network::addVertex(G,v)` adds the vertex or list of vertices `v` to the network `G`.

Call(s):

- `Network::addVertex(G, v<, Vweight=c>)`
- `Network::addVertex(G, l<, Vweight=lc>)`

Parameters:

`c` — number
`l` — list of expressions
`v` — expression
`lc` — list of numbers
`G` — network

Options:

`Vweight` — The weight of the vertex.

Return Value: `Network::addVertex` returns the augmented network.

Details:

- `Network::addVertex` adds one or several nodes to an already existing network. A node is to be assumed an arbitrary expression. If the specified node is already contained in the network an error is raised.
- `Network::addVertex(G,v)` adds the node `v` to the network `G`. A weight can be defined for the new vertex with `Network::addVertex(G,v,`*`Vweight`*`=c)`. If these specification is missing, the default value 0 is assumed.
- Several nodes can be added with `Network::addVertex(G,l)`, where `l` is a list of nodes. None of these nodes is allowed to be already contained in the network. Weights can be specified by `Network::addVertex(G,l,`*`Vweight`*`=lc)` where `lc` is a numerical list with exactly the same number of elements as `l`.

Example 1. Starting from a cyclic network with four nodes, we add three more nodes with non-zero weights.

```
>> N1 := Network::cycle([v1,v2,v3,v4]):
   Network::vertex(N1)

                          [v1, v2, v3, v4]

>> N2 := Network::addVertex(N1, [v5,v6,v7], Vweight=[2,3,4]):
   Network::vertex(N2)

                    [v1, v2, v3, v4, v5, v6, v7]

>> Network::vWeight(N2)
```

```
table(
  v7 = 4,
  v6 = 3,
  v5 = 2,
  v4 = 0,
  v3 = 0,
  v2 = 0,
  v1 = 0
)
```

Changes:

- `Network::addVertex` used to be `Network::AddVertex`.

`Network::admissibleFlow` – checks a flow for admissibility in a network

`Network::admissibleFlow(N,f)` checks if the flow `f` is admissible in the network `N` according to its vertices and their capacities.

Call(s):

- `Network::admissibleFlow(N, f)`

Parameters:

`N` — network
`f` — flow (a table)

Return Value: either `TRUE` or `FALSE`

Details:

- `Network::admissibleFlow` checks whether a given flow is an admissible flow in the specified network. A flow in a network is a table `t`, where `t[[i,j]]` gives the number of units flowing from node `i` to node `j`. `Network::admissibleFlow` returns `TRUE` if the flow is admissible. Otherwise `FALSE` is returned.

- `Network::admissibleFlow` does not check whether the flow is admissible if a flow from node `i` to node `j` is allowed to pass through other nodes. See example 2.

Example 1. In a cyclic network with default capacities (1), the flow with one unit flowing from each node to its successor is certainly admissible.

```
>> N1 := Network::cycle([v1,v2,v3,v4]):
   Network::admissibleFlow( N1, table([v1,v2]=1,
                       [v2,v3]=1, [v3,v4]=1, [v4,v1]=1))

                                TRUE
```

Example 2. The flow must give each connection to use directly. `Network::admissibleFlow` does not introduce "hops".

```
>> Network::admissibleFlow( Network::cycle([v1,v2,v3]),
                  table([v1,v3]=1))

                                FALSE
```

Changes:

- `Network::admissibleFlow` used to be `Network::AdmissibleFlow`.

`Network::allShortPath` – shortest paths for all pairs of nodes

`Network::allShortPath(G)` finds shortest paths in the network `G`.

Call(s):

- `Network::allShortPath(G<, Length><, Path>)`

Parameters:

`G` — network

Options:

`Length` — Return the lengths of shortest paths. This is the default unless *`Path`* is given.

`Path` — Return a table of shortest paths.

Return Value: A table or a sequence of two tables

Details:

- `Network::allShortPath(G)` gives a table with the length of a shortest path between two nodes `i` and `j` for every `i` and `j`.
- `Network::allShortPath(G, Path)` returns a table which contains a shortest path for every pair `(i,j)`. If there is no entry for a pair `(i,j)`, then either `i=j` or `j` cannot be reached from `i` in the network.
- `Network::allShortPath(G, Length)` returns a table which contains the length of a shortest path for every pair `(i,j)`. The entry `infinity` for `(i,j)` represents the fact, that `j` cannot be reached from `i` in the network. If both *`Length`* and *`Path`* are specified, then the distance table and the path table are returned. If the option *`Path`* is not given, the behaviour of `Network::allShortPath` with the option *`Length`* is the default behaviour.

Example 1. In a cyclic network with three vertices, each node can be reached in at most two steps.

```
>> N1 := Network::cycle([v1, v2, v3]):
   Network::allShortPath(N1)

                          table(
                            (v3, v3) = 0,
                            (v3, v2) = 2,
                            (v3, v1) = 1,
                            (v2, v3) = 1,
                            (v2, v2) = 0,
                            (v2, v1) = 2,
                            (v1, v3) = 2,
                            (v1, v2) = 1,
                            (v1, v1) = 0
                          )
```

Adding a vertex which has no connection to the three nodes, we get the expected entries `infinity`. The table of paths contains no entries referring to the newly added vertex.

```
>> N1 := Network::addVertex( N1, v4 ):
   Network::allShortPath(N1, Length, Path)

       table(
         (v4, v4) = 0,
         (v4, v3) = infinity,
         (v4, v2) = infinity,
         (v4, v1) = infinity,
```

```
    (v3, v4) = infinity,  table(
    (v3, v3) = 0,           (v3, v2) = [v3, v1, v2],
    (v3, v2) = 2,           (v3, v1) = [v3, v1],
    (v3, v1) = 1,       ,   (v2, v3) = [v2, v3],
    (v2, v4) = infinity,    (v2, v1) = [v2, v3, v1],
    (v2, v3) = 1,           (v1, v3) = [v1, v2, v3],
    (v2, v2) = 0,           (v1, v2) = [v1, v2]
    (v2, v1) = 2,         )
    (v1, v4) = infinity,
    (v1, v3) = 2,
    (v1, v2) = 1,
    (v1, v1) = 0
  )
```

Background:

- The implemented algorithm is taken from Floyd, "Algorithm 97, Shortest Path", Comm. ACM, Vol. 5, 1962. The running time is $O(n^3)$, where n is the number of nodes.

Changes:

- `Network::allShortPath` used to be `Network::AllShortPath`.

`Network::changeEdge` – changes weight and capacity of one or several edges

`Network::changeEdge(G, e, Eweight=c, Capacity=l)` changes the weight of edge `e` in network `G` to `c` and its capacity to `l`.

Call(s):

- `Network::changeEdge(G, e<, Eweight=c><, Capacity=t>)`
- `Network::changeEdge(G, l<, Eweight=lc><, Capacity=lt>)`

Parameters:

`lc,lt` — lists of numbers
`c,t` — numbers
`l` — list of edges
`e` — edge
`G` — network

Options:

Eweight — change the weight of the edge
Capacity — change the capacity of the edge

Return Value: The altered network

Details:

- `Network::changeEdge` changes the weight and capacity of one or several edges in a network. An edge is given as a list containing two nodes of the network. An error is raised if the specified edge is not contained in the network.
- `Network::changeEdge(G, e, Eweight=c)` changes the weight of edge `e` in the network `G` to the new value `c`.
- `Network::changeEdge(G, e, Capacity=t)` changes the capacity of edge `e` in the network `G` to the new value `t`.
- `Network::changeEdge(G, e, Eweight=c, Capacity=t)` changes the weight and capacity of edge `e` simultaneously.
- Instead of changing the values for one edge, they can be changed for several edges simultaneously. For this, `ChangeEdge(G,l,Eweight=lc,Capacity=lt)` has to be given where `l` is a list of edges and `lc` and `lt` are numerical lists with exactly the same number of elements as `l`.

Example 1. We construct a cyclic network with default weights. Then, those weights are changed.

```
>> N1 := Network::cycle([v1, v2, v3, v4]):
   Network::eWeight(N1)
```

```
                          table(
                            [v4, v1] = 1,
                            [v3, v4] = 1,
                            [v2, v3] = 1,
                            [v1, v2] = 1
                          )
```

```
>> N2 := Network::changeEdge(N1, [[v1,v2], [v2,v3]], Eweight=[2,2]):
   Network::eWeight(N2)
```

```
                          table(
                            [v4, v1] = 1,
                            [v3, v4] = 1,
                            [v2, v3] = 2,
                            [v1, v2] = 2
                          )
```

Changes:

- `Network::changeEdge` used to be `Network::ChangeEdge`.

`Network::changeVertex` – changes the weight of one or several vertices in a network

`Network::changeVertex(G, v, Vweight=c)` sets the weight of vertex `v` in network `G` to `c`.

Call(s):

- `Network::changeVertex(G, v<, Vweight=c>)`
- `Network::changeVertex(G, l<, Vweight=lc>)`

Parameters:

`c` — a number
`l` — a list of nodes
`v` — a node of the network `G`
`lc` — a list of numbers
`G` — a network

Options:

`Vweight` — the new weight(s) of the vertices

Return Value: the augmented network

Details:

- `Network::changeVertex` changes the weight of one or several nodes in a network. An error is raised if the specified node is not contained in the network.
- `Network::changeVertex(G,v,Vweight=c)` changes the weight of node `v` in the network `G` to the new value `c`.
- Instead of changing the vertex weight for one single node, the weight of several nodes can be changed with `Network::changeVertex(G,l,Vweight=lc)` where `l` is a list of nodes and `lc` is a numerical list with exactly the same number of elements as `l`. If one of the specified nodes is not contained in the network an error is raised.

Example 1. We generate a cyclic path with default weights. Then, the vertex weights are changed.

```
>> N1 := Network::cycle([v1,v2,v3,v4]):
   Network::vWeight(N1)
                                table(
                                  v4 = 0,
                                  v3 = 0,
                                  v2 = 0,
                                  v1 = 0
                                )

>> N2 := Network::changeVertex(N1, [v1,v2,v3,v4],
                                   Vweight=[1,2,3,4]):
   Network::vWeight(N2)
                                table(
                                  v4 = 4,
                                  v3 = 3,
                                  v2 = 2,
                                  v1 = 1
                                )
```

Changes:

- `Network::changeVertex` used to be `Network::ChangeVertex`.

`Network::complete` – generates a complete network

`Network::complete(n)` generates the complete network with `n` vertices.

Call(s):

- `Network::complete(n)`

Parameters:

`n` — non negative integer

Return Value: a network

Details:

- `Network::complete(n)` generates the complete network with `n` vertices. A complete network has a connection between each pair of vertices.
- The network generated by `Network::complete` uses the default values of 1 for vertex weights, edge weights and edge capacities.
- The vertices of the generated network are labeled with the numbers from `1` to `n`.

Example 1. The complete network with three vertices has $3! = 6$ edges.

```
>> N1 := Network::complete(3):
   Network::printGraph(N1)

                      Vertices: [1, 2, 3]

     Edges: [[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]]

               Vertex weights: table(3=0,2=0,1=0)

 Edge capacities: table([3, 2]=1,[3, 1]=1,[2, 3]=1,[2, 1]=1,[1,\
  3]=1,[1, 2]=1)

 Edge weights: table([3, 2]=1,[3, 1]=1,[2, 3]=1,[2, 1]=1,[1, 3]\
 =1,[1, 2]=1)

     Adjacency list (out): table(3=[1, 2],2=[1, 3],1=[2, 3])

     Adjacency list (in): table(3=[1, 2],2=[1, 3],1=[2, 3])
```

Changes:

- `Network::complete` used to be `Network::Complete`.

`Network::convertSSQ` – converts a network into a single source single sink network

`Network::convertSSQ(G, q, s)` augments the network `G` so that `q` is the single source and `s` is the single sink of the new network.

Call(s):

- `Network::convertSSQ(G, q, s)`

Parameters:

`q,s` — nodes not contained in the network
`G` — a network

Return Value: the augmented network

Details:

- `Network::convertSSQ(G, q, s)` converts the network `G` into a single source single sink network. The specified nodes `q` and `s` are added to the network. It is an error if they are already contained. Otherwise they are connected to the other nodes of the network in the following way:

  A new edge `[q,i]` is added for every vertex `i` with a positive weight. A new edge `[i,s]` is added for every vertex `i` with a negative weight. The capacities of these edges are in each case the weight of node `i`. The edge weights are zero.

Example 1. This is an ugly example. We should make up a better one and explain it.

```
>> V := [1,2,3,4]:
   Vw := [4,0,0,-4]:
   Ed := [[1,2], [1,3], [2,3], [2,4], [3,4]]:
   Ew := [2,2,1,3,1]:
   Ecap := [4,2,2,3,5]:
   N1 := Network(V,Ed,Vweight=Vw,Capacity=Ecap,Eweight=Ew):
   N2 := Network::convertSSQ(N1,q,s):
   Network::printGraph(N2)

                  Vertices: [1, 2, 3, 4, q, s]

 Edges: [[1, 2], [1, 3], [2, 3], [2, 4], [3, 4], [q, 1], [4, s]]

         Vertex weights: table(s=-4,q=4,4=0,3=0,2=0,1=0)

 Edge capacities: table([4, s]=4,[q, 1]=4,[3, 4]=5,[2, 4]=3,[2,\
  3]=2,[1, 3]=2,[1, 2]=4)

 Edge weights: table([4, s]=0,[q, 1]=0,[3, 4]=1,[2, 4]=3,[2, 3]\
 =1,[1, 3]=2,[1, 2]=2)

 Adjacency list (out): table(s=[],q=[1],4=[s],3=[4],2=[3, 4],1=\
 [2, 3])
```

```
Adjacency list (in): table(s=[4],q=[],4=[2, 3],3=[1, 2],2=[1],\
1=[q])
```

Changes:

- `Network::convertSSQ` used to be `Network::ConvertSSQ`.

`Network::cycle` – generates a cycle

`Network::cycle(L)` generates the cyclic network consisting of the nodes in `L`.

Call(s):

- `Network::cycle(L)`

Parameters:

`L` — list of expressions

Return Value: a network

Details:

- `Network::cycle([v1,...,vn])` generates a new network which is the cycle `[v1,v2]`, `[v2,v3]`, ..., `[vn,v1]`. The values for the edge weights, edge capacities and vertex weights are the default values 1, 1 and 0 respectively.

Example 1. The cyclic network with four vertices:

```
>> N1 := Network::cycle([$1..4]):
   Network::printGraph(N1)

                    Vertices: [1, 2, 3, 4]

              Edges: [[1, 2], [2, 3], [3, 4], [4, 1]]

              Vertex weights: table(4=0,3=0,2=0,1=0)

   Edge capacities: table([4, 1]=1,[3, 4]=1,[2, 3]=1,[1, 2]=1)

    Edge weights: table([4, 1]=1,[3, 4]=1,[2, 3]=1,[1, 2]=1)
```

```
Adjacency list (out): table(4=[1],3=[4],2=[3],1=[2])

 Adjacency list (in): table(4=[3],3=[2],2=[1],1=[4])
```

Changes:

- `Network::cycle` used to be `Network::Cycle`.

`Network::delEdge` – **deletes one or several edges from a network**

`Network::delEdge(G, e)` deletes edge `e` from network `G`.

Call(s):

- `Network::delEdge(G, e)`
- `Network::delEdge(G, l)`

Parameters:

`l` — a list of edges
`e` — an edge
`G` — a network

Return Value: the new network

Details:

- `Network::delEdge` deletes one or several edges from a network. An edge is represented by a list containing two nodes of the network. An error is raised if the specified edge is not contained in the network.
- `Network::delEdge(G, e)` deletes the edge `e` from the network `G`.
- `Network::delEdge(G, l)` deletes all edges in the list `l` from the network `G`.

Example 1. Deleting an edge from a cyclic network results in a (degenerated) tree.

```
>> N1 := Network::cycle([v1,v2,v3]):
   Network::printGraph(N1)
```

```
                    Vertices: [v1, v2, v3]

                Edges: [[v1, v2], [v2, v3], [v3, v1]]

                Vertex weights: table(v3=0,v2=0,v1=0)

     Edge capacities: table([v3, v1]=1,[v2, v3]=1,[v1, v2]=1)

       Edge weights: table([v3, v1]=1,[v2, v3]=1,[v1, v2]=1)

       Adjacency list (out): table(v3=[v1],v2=[v3],v1=[v2])

        Adjacency list (in): table(v3=[v2],v2=[v1],v1=[v3])
>> N2 := Network::delEdge(N1, [v2,v3]):
   Network::printGraph(N2)
                    Vertices: [v1, v2, v3]

                  Edges: [[v1, v2], [v3, v1]]

                Vertex weights: table(v3=0,v2=0,v1=0)

           Edge capacities: table([v3, v1]=1,[v1, v2]=1)

            Edge weights: table([v3, v1]=1,[v1, v2]=1)

        Adjacency list (out): table(v3=[v1],v2=[],v1=[v2])

         Adjacency list (in): table(v3=[],v2=[v1],v1=[v3])
```

Changes:

- `Network::delEdge` used to be `Network::DelEdge`.

`Network::delVertex` – deletes one or several vertices from a network

`Network::delVertex(G, v)` deletes the vertex `v` from network `G`.

Call(s):

- `Network::delVertex(G, v)`
- `Network::delVertex(G, l)`

Parameters:

`l` — list of expressions
`v` — expression
`G` — network

Return Value: the smaller network

Details:

- `Network::delVertex(G, v)` deletes the node `v` from the network `G`.
- `Network::delVertex(G, [v1, ..., vn])` deletes the nodes `v1, ..., vn` from the network `G`.
- The attempt to delete a node which is not in `G` causes an error.

Example 1. Deleting a vertex from a network also deletes all edges connected to it:

```
>> N1 := Network::cycle([v1,v2,v3]):
   Network::printGraph(N1)

                    Vertices: [v1, v2, v3]

               Edges: [[v1, v2], [v2, v3], [v3, v1]]

               Vertex weights: table(v3=0,v2=0,v1=0)

     Edge capacities: table([v3, v1]=1,[v2, v3]=1,[v1, v2]=1)

       Edge weights: table([v3, v1]=1,[v2, v3]=1,[v1, v2]=1)

       Adjacency list (out): table(v3=[v1],v2=[v3],v1=[v2])

        Adjacency list (in): table(v3=[v2],v2=[v1],v1=[v3])

>> N2 := Network::delVertex(N1, v3):
   Network::printGraph(N2)

                       Vertices: [v1, v2]

                        Edges: [[v1, v2]]

                 Vertex weights: table(v2=0,v1=0)

                Edge capacities: table([v1, v2]=1)

                  Edge weights: table([v1, v2]=1)
```

```
Adjacency list (out): table(v2=[],v1=[v2])

 Adjacency list (in): table(v2=[v1],v1=[])
```

Changes:

- `Network::delVertex` used to be `Network::DelVertex`.

`Network::eCapacity` – returns the table of capacities

`Network::eCapacity(G)` returns the table of capacities of the network `G`.

Call(s):

- `Network::eCapacity(G)`

Parameters:

`G` — a network

Return Value: a table

Details:

- `Network::eCapacity(G)` returns a table with the capacity of the network `G`. Thus `Network::eCapacity(G)[[v,w]]` is the capacity of the edge `[v,w]` in the network `G`.

Example 1.

```
>> V := [1,2,3,4,5]:
   Ed := [[1,2], [1,3], [2,3], [2,4], [3,4], [3,5], [4,5]]:
   Ecap := [30, 20, 25, 10, 20, 25, 20]:
   N1 := Network(V, Ed, Capacity=Ecap):
   Network::eCapacity(N1)
```

```
                        table(
                          [4, 5] = 20,
                          [3, 5] = 25,
                          [3, 4] = 20,
                          [2, 4] = 10,
                          [2, 3] = 25,
                          [1, 3] = 20,
                          [1, 2] = 30
                        )
```

The default capacity of edges is 1.

```
>> N1 := Network::complete(3):
   Network::eCapacity(N1)
```

```
                          table(
                            [3, 2] = 1,
                            [3, 1] = 1,
                            [2, 3] = 1,
                            [2, 1] = 1,
                            [1, 3] = 1,
                            [1, 2] = 1
                          )
```

Changes:

- `Network::eCapacity` used to be `Network::ECapacity`.

`Network::eWeight` – returns the table of edge weights

`Network::eWeight(G)` returns the table of the edge weights of the network `G`.

Call(s):

- `Network::eWeight(G)`

Parameters:

`G` — a network

Return Value: a table

Details:

- `Network::eWeight(G)` returns a table with the edge weight of the network `G`. Thus `Network::eWeight(G)[[v,w]]` is the weight of the edge `[v,w]` in the network `G`.

Example 1.

```
>> V := [1,2,3,4,5]:
   Ed := [[1,2], [1,3], [2,3], [2,4], [3,4], [3,5], [4,5]]:
   Ew := [30, 20, 25, 10, 20, 25, 20]:
   N1 := Network(V, Ed, Eweight=Ew):
   Network::eWeight(N1)
```

```
                        table(
                          [4, 5] = 20,
                          [3, 5] = 25,
                          [3, 4] = 20,
                          [2, 4] = 10,
                          [2, 3] = 25,
                          [1, 3] = 20,
                          [1, 2] = 30
                        )
```

The default weight is 1.

```
>> N1 := Network::complete(3):
   Network::eWeight(N1)
```

```
                         table(
                           [3, 2] = 1,
                           [3, 1] = 1,
                           [2, 3] = 1,
                           [2, 1] = 1,
                           [1, 3] = 1,
                           [1, 2] = 1
                         )
```

Changes:

- `Network::eWeight` used to be `Network::EWeight`.

`Network::edge` – returns a list with all edges

`Network::edge(G)` returns the list of all edges of the network `G`.

Call(s):

- `Network::edge(G)`

Parameters:

`G` — a network

Return Value: the list of all edges, a list of lists

Details:

- `Network::edge(G)` returns a list with all edges of the network `G`. Each edge is represented by a list containing the two connected vertices.

Example 1. `Network::edge` only returns the edges, without their capacities.

```
>> N1 := Network::cycle([v1,v2,v3,v4]):
   Network::edge(N1)

            [[v1, v2], [v2, v3], [v3, v4], [v4, v1]]

>> N2 := Network::complete(3):
   Network::edge(N2)

         [[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]]
```

Changes:

- `Network::edge` used to be `Network::Edge`.

`Network::epost, Network::epre` – adjacency lists

`Network::epost(G)` returns the direct successors of each vertex in the network `G`.

`Network::epre(G)` returns the direct predecessors of each vertex in the network `G`.

Call(s):

- `Network::epost(G)`
- `Network::epre(G)`

Parameters:

`G` — a network

Return Value: a table

Details:

- `Network::epost(G)` returns a table with the adjacency lists for outgoing edges. Thus `Network::epost(G)[v]` is a list containing all those nodes `w` for which there is an edge `[v,w]` in the network.
- `Network::epre(G)` returns a table with the adjacency lists for incoming edges. Thus `Network::epre(G)[v]` is a list containing all those nodes `w` for which there is an edge `[w,v]` in the network.

Example 1. Since networks are directed, the output of `epost` and `epre` may differ:

```
>> V := [1,2,3,4,5]:
   Ed := [[1,2], [1,3], [2,3], [2,4], [3,4], [3,5], [4,5]]:
   N1 := Network(V, Ed):
   Network::epost(N1), Network::epre(N1)
```

```
                table(          table(
                  5 = [],         5 = [3, 4],
                  4 = [5],        4 = [2, 3],
                  3 = [4, 5],,    3 = [1, 2],
                  2 = [3, 4],     2 = [1],
                  1 = [2, 3]      1 = []
                )               )
```

Of course, it is possible to model undirected graphs with the `Network` package:

```
>> N2 := Network::complete(4):
   Network::epost(N2), Network::epre(N2)
```

```
              table(             table(
                4 = [1, 2, 3],     4 = [1, 2, 3],
                3 = [1, 2, 4],,    3 = [1, 2, 4],
                2 = [1, 3, 4],     2 = [1, 3, 4],
                1 = [2, 3, 4]      1 = [2, 3, 4]
              )                  )
```

Changes:

- `Network::epost` used to be `Network::Epost`.
- `Network::epre` used to be `Network::Epre`.

`Network::inDegree` – the indegree of nodes

`Network::inDegree(G, v)` returns the number of edges coming into the node `v` of the network `G`.

Call(s):

- `Network::inDegree(G<, v, ...>)`

Parameters:

`G` — a network
`v` — expression

Return Value: either an integer or a table

Details:

- `Network::inDegree(G, v)` returns the indegree of the node `v` in network `G`, i.e., the number of edges `[w,v]`.
- `Network::inDegree(G, v1, v2, ...)` returns a table in which the keys are `v1, v2, ...` and the corresponding values are the indegrees, i.e., `Network::inDegree(G, v1, v2)[v1] = Network::inDegree(G, v1)`.
- `Network::inDegree(G)` returns a table in which each node of `G` is mapped to its indegree. If `G` contains more than one vertex, `Network::inDegree(G)` is equivalent to `Network::inDegree(G, op(Network::vertex(G)))`.

Example 1. In a complete network of n nodes, each vertex has indegree $n-1$:

```
>> N1 := Network::complete(3):
   Network::inDegree(N1)

                              table(
                                3 = 2,
                                2 = 2,
                                1 = 2
                              )
```

Changes:

- `Network::inDegree` used to be `Network::InDegree`.

`Network::isEdge, Network::isVertex` – checks whether an edge or vertex is contained in a network

`Network::isEdge(G, e)` checks if `e` is an edge of network `G`.

`Network::isVertex(G, v)` checks if `v` is a vertex of network `G`.

Call(s):

- `Network::isEdge(G, e)`
- `Network::isVertex(G, v)`

Parameters:

`G` — a network
`e` — a list of two expressions
`v` — an expression

Return Value: `TRUE` or `FALSE`

Details:

- `Network::isEdge(G, e)` gives `TRUE` if `e` is an edge in network `G`.
- `Network::isVertex(G, v)` return `TRUE` if `v` is a vertex in network `G`.

Example 1. Some examples for the use of these two functions:

```
>> N1 := Network::cycle([v1, v2, v3, v4]):
   Network::isEdge(N1, [v1, v2]), Network::isEdge(N1, [v1, v3])

                              TRUE, FALSE

>> Network::isVertex(N1, v1), Network::isVertex(N1, v5)

                              TRUE, FALSE
```

Changes:

- `Network::isEdge` used to be `Network::IsEdge`.
- `Network::isVertex` used to be `Network::IsVertex`.

`Network::longPath` – longest paths from one single node

`Network::longPath(G, v)` finds the longest path in network `G` starting from vertex `v`.

Call(s):

- `Network::longPath(G, v<, w><, Length><, Path>)`

Parameters:

`G` — a network
`v,w` — nodes in `G`

Options:

`Length` — Return a table with the lengths of shortest paths
`Path` — Return a table with the paths themselves

Return Value: a table, an integer or a list of nodes

Details:

- `Network::longPath(G, v)` returns a table with the length of longest paths from `v` to all other nodes in the network with respect to the edge weight.
- `Network::longPath(G, v, w)` returns the length of a longest path from `v` to `w`.
- If the optional argument *`Path`* is given, a table with longest paths is returned. If both *`Length`* and *`Path`* are given, then both the length of the longest paths and the paths are returned. Paths are given as lists of nodes in reverse order.
- If *`Path`* is not given, the option *`Length`* has no effect.
- `G` should not contain cycles.

Example 1. We construct a network and try a few calls to `Network::longPath`:

```
>> V := [1,2,3,4,5]:
   Ed := [[1,2], [1,3], [2,3], [2,4], [3,4], [3,5], [4,5]]:
   Ew := [7, 6, 5, 4, 2, 2, 1]:
   N1 := Network(V, Ed, Eweight=Ew):
   Network::longPath(N1,1)
```

```
                          table(
                            5 = 15,
                            4 = 14,
                            3 = 12,
                            2 = 7,
                            1 = 0
                          )
```

```
>> Network::longPath(N1,1,Path)
```

```
                      table(
                        5 = [2, 1, 2, 3, 4],
                        4 = [2, 1, 2, 3],
                        3 = [2, 1, 2],
                        2 = [2, 1]
                      )
```

Background:

- The implemented algorithm is a variation of the algorithm of Bellman.

Changes:

- `Network::longPath` used to be `Network::LongPath`.

`Network::maxFlow` – computes a maximal flow through a network

`Network::maxFlow(G, q, s)` computes a maximal flow through network `G` from node `q` to node `s`.

Call(s):

- `Network::maxFlow(G, q, s)`

Parameters:

`G` — network
`q, s` — expressions (nodes in `G`)

Return Value: a list, containing a number and a table

Details:

- `Network::maxFlow(G,q,s)` computes a maximal flow from `q` to `s` in `G` with respect to the edge capacities. `q` and `s` must be nodes in `G`.
- `Network::maxFlow(G,q,s)` returns a sequence containing the flow value, that is the inflow of `s`, which equals the outflow of `q`, and the flow itself in form of table `t` with the flow from node `v` to node `w` is `t[[v,w]]`.

Example 1. In the complete network with four vertices and default capacities of 1, the maximum flow from one vertex to another one consists of sending one unit through each of the remaining vertices and one directly, which makes three units altogether.

```
>> N1 := Network::complete(4):
   Network::maxFlow(N1,1,4)
```

```
                          table(
                            [4, 3] = 0,
                            [4, 2] = 0,
                            [4, 1] = 0,
                            [3, 4] = 1,
                            [3, 2] = 0,
                       3,   [3, 1] = 0,
                            [2, 4] = 1,
                            [2, 3] = 0,
                            [2, 1] = 0,
                            [1, 4] = 1,
                            [1, 3] = 1,
                            [1, 2] = 1
                          )
```

Example 2. A more complex example, the following network shows that this function also finds flows through multiple edges, unlike `Network::admissibleFlow`, which only works on completely described flows.

```
>> V := [1,2,3,q,s]:
   Edge := [[q,1], [q,2], [1,2], [1,3], [2,3], [3,s]]:
   up := [5, 5, 2, 6, 6, 1]:
   N2 := Network(V, Edge, Capacity=up):
   Network::maxFlow(N2, q, s)
```

```
    table(
      [3, s] = 1,
      [2, 3] = 1,
1,    [1, 3] = 0,
      [1, 2] = 0,
      [q, 2] = 1,
      [q, 1] = 0
    )
```

Background:

- The implemented algorithm is the preflow-push algorithm of Goldberg & Tarjan with the FIFO selection strategy and an exact distance labeling ("A new approach to the maximum-flow problem", Journal of the ACM 35(4), 1988).
- The running time is $O(n^3)$, where n is the number of vertices in the network.

Changes:

- `Network::maxFlow` used to be `Network::MaxFlow`.

`Network::minCost` – computes a minimal cost flow

`Network::minCost(G)` computes a minimal cost flow for the network `G`, taking into consideration supply and demand, capacities and transportation cost.

Call(s):

- `Network::minCost(G)`

Parameters:

`G` — network

Return Value: a sequence, consisting of a number and two tables

---

Details:

- `Network::minCost(G)` computes a minimal cost flow in `G` with respect to the edge capacities, the edge weights and the vertex weights of `G`.

The vertex weights are interpreted as supply and demand. The edge capacities give restrictions for the flow on every edge. The edge weights are the cost for one unit flow over an edge.

The algorithm computes a flow, if there is any, which is possible and satisfactory, i.e., it is within the supply and demand range, which respects the capacities and which has minimal cost.

- The result of `Network::minCost(G)` is a sequence of the price of a minimal cost flow, the minimal cost flow itself (in form of table), and a table with the dual prices.

Example 1. We construct a network with five nodes and seven edges. One of the nodes is a pure source (1), another one is a pure sink (5). No other nodes supply or demand any goods, they only serve as transportation junctions.

```
>> V := [1,2,3,4,5]:
   Vw := [25,0,0,0,-25]:
   Ed := [[1,2], [1,3], [2,3], [2,4], [3,4], [3,5], [4,5]]:
   Ew := [7, 6, 5, 4, 2, 2, 1]:
   Ecap := [30, 20, 25, 10, 20, 25, 20]:
   N1 := Network(V,Ed,Eweight=Ew, Capacity=Ecap, Vweight=Vw):
   Network::minCost(N1)
```

```
                      table(
                        [4, 5] = 5,   table(
                        [3, 5] = 20,    5 = 2,
                        [3, 4] = 0,     4 = 3,
                 220,   [2, 4] = 5, ,   3 = 4,
                        [2, 3] = 0,     2 = 7,
                        [1, 3] = 20,    1 = 14
                        [1, 2] = 5    )
                      )
```

All 25 units could be transported from node 1 to node 5, for a total cost of 220.

Background:

- The implemented algorithm is the relaxation algorithm due to Bertsekas (taken from Bertsekas, "Linear Network Optimization", MIT Press, Cambridge(Mass.)-London, 1991) which is known to be one of the fastest algorithms in practice.

Changes:

- `Network::minCost` used to be `Network::MinCost`.

`Network::minCut` – **computes a minimal cut**

`Network::minCut(G, q, s)` computes a minimal cut in `G` separating node `q` from node `s`.

Call(s):

- `Network::minCut(G,q,s)`

Parameters:

`q,s` — expressions (nodes in the network)
`G` — network

Details:

- `Network::minCut(G,q,s)` computes a minimal cut in `G` that separates `q` from `s`, i.e., a subset `T` of the set `S` of edges of `G` such that every path from `q` to `s` contains at least one edge in `T`. The cut is minimal with respect the capacities of the edges.
- `Network::minCut(G,q,s)` returns a sequence consisting of the cut value (the sum of the edge weights of the cut edges) and a list with the edges of the cut.
- Note that `q` is separated from `s`, not vice versa.

Example 1. In a complete network, a node can be separated from another one only by cutting all edges starting at the first node.

```
>> N1 := Network::complete(4):
   Network::minCut(N1, 1, 4)

                  3, [[1, 2], [1, 3], [1, 4]]
```

Example 2. In the following example, the edge from node `q` to node `1` could have been used as well, but its edge capacity is higher than that of the edge used, so the minimality condition precludes this choice:

```
>> V := [1, 2, 3, q, s]:
   Edge := [[q, 1], [1, 2], [1, 3], [2, 3], [3, s]]:
   up := [5, 2, 6, 6, 1]:
   N2 := Network(V, Edge, Capacity=up):
   Network::minCut(N2, q, s)
```

```
                          1, [[3, s]]
```

There is no path from node `s` to node `q` (or any other vertex of the network), so no cut is necessary to separate `s` from `q`:

```
>> Network::minCut(N2, s, q)
```

```
                              0, []
```

Changes:

- `Network::minCut` used to be `Network::MinCut`.

`Network::outDegree` – returns the out-degrees for nodes

`Network::outDegreeG` returns the "out-degrees" for the nodes of the network `G`. An out-degree is the number of edges leaving the node.

Call(s):

- `Network::outDegree(G<, v, ...>)`

Parameters:

`G` — a `Network`
`v` — expression

Return Value: Depending on the number of arguments, either a table or a non-negative integer.

Details:

- `Network::outDegree` returns the out-degree of one or several nodes of a network, i.e. the number of leaving edges.
- With `outDegree(G,v)`, where `G` is a network and `v` is a node in `G`, the out-degree of `v` is returned, i.e. the number of edges that are starting in `v`.

- `outDegree(G, v1, v2, ...)`, where `v1, v2, ...` are nodes in `G`, a table `ll` is returned, where `ll[i]` is the out-degree of node `l[i]` in `G`.
- `OutDegree(G)` gives a table `lg` with the out-degree of all nodes of `G`, i.e. `lg[i]` is the out-degree of node `Vertex(G)[i]` in `G`.

Example 1. In a complete network with n vertices, each vertex has out-degree $n - 1$.

```
>> N1 := Network::complete(4):
   Network::outDegree(N1)
```

```
table(
  4 = 3,
  3 = 3,
  2 = 3,
  1 = 3
)
```

Changes:

- `Network::outDegree` used to be `Network::OutDegree`.

`Network::printGraph` – print all information about a network

`Network::printGraphG` prints all information about the network `G` on the screen.

Call(s):

- `Network::printGraph(G)`

Parameters:

`G` — network

Return Value: The value of type `DOM_NULL`

Details:

- `Network::printGraph` prints all known information about a network.

Example 1.

```
>> V := [1,2,3,q,s]:
   Edge := [[q,1], [1,2], [1,3], [2,3], [3,s]]:
   up := [5, 4, 4, 2, 5]:
   N1 := Network(V,Edge,Capacity=up):
   Network::printGraph(N1);

                    Vertices: [1, 2, 3, q, s]

          Edges: [[q, 1], [1, 2], [1, 3], [2, 3], [3, s]]

            Vertex weights: table(s=0,q=0,3=0,2=0,1=0)

 Edge capacities: table([3, s]=5,[2, 3]=2,[1, 3]=4,[1, 2]=4,[q,\
  1]=5)

 Edge weights: table([3, s]=1,[2, 3]=1,[1, 3]=1,[1, 2]=1,[q, 1]\
 =1)

  Adjacency list (out): table(s=[],q=[1],3=[s],2=[3],1=[2, 3])

   Adjacency list (in): table(s=[3],q=[],3=[1, 2],2=[1],1=[q])
```

Changes:

- `Network::printGraph` used to be `Network::PrintGraph`.

`Network::random` – generates a random network

`Network::random` generates a random network.

Call(s):

- `Network::random()`
- `Network::random(In,D,Kn)`

Parameters:

`In` — list with two integers
`D` — real number between 0 and 1
`Kn` — list with two integers

Return Value: A `Network`

Details:

- `Network::random(In, D, Kn)` generates a random network and returns a list `[N, q, s]` where `N` is the network and `q` and `s` are two nodes in `N`.
- Networks generated by `Network::random` are undirected, i.e., if `[s, q]` is an edge of the network, so is `[q, s]`.
- If `In = [a, b]`, then number of nodes `K` in `N` is between `a` and `b`.
- `D` specifies the density of `N`, i.e. the number of edges in `N` is $K^2 * D * 2$.
- With `Kn = [a, b]`, the capacities of the edges in the Network generated my `Network::random` will be between `a` and `b`, inclusive.
- `Network::random() = Network::random([5, 10], 0.5, [1, 10])`.

Example 1. Your results may differ in the following example.

```
>> N1 := Network::random():
   Network::printGraph(N1[1])

                Vertices: [1, 2, 3, 4, 5, 6, 7, 8]

 Edges: [[7, 2], [2, 7], [1, 3], [3, 1], [1, 2], [2, 1], [8, 4]\
 , [4, 8], [6, 5], [5, 6], [4, 6], [6, 4], [7, 3], [3, 7], [6, \
 7], [7, 6], [8, 2], [2, 8], [4, 3], [3, 4], [3, 6], [6, 3], [7\
 , 1], [1, 7], [6, 8], [8, 6], [5, 4], [4, 5], [2, 6], [6, 2], \
 [1, 4], [4, 1]]

     Vertex weights: table(8=0,7=0,6=0,5=0,4=0,3=0,2=0,1=0)

 Edge capacities: table([2, 8]=1,[8, 2]=1,[3, 7]=10,[7, 3]=10,[\
 6, 4]=7,[4, 6]=7,[4, 5]=4,[5, 4]=4,[6, 3]=1,[3, 6]=1,[2, 7]=4,\
 [7, 2]=4,[6, 2]=8,[2, 6]=8,[1, 7]=4,[7, 1]=4,[3, 4]=4,[4, 3]=4\
 ,[4, 1]=6,[1, 4]=6,[3, 1]=6,[1, 3]=6,[2, 1]=10,[1, 2]=10,[8, 6\
 ]=2,[6, 8]=2,[7, 6]=7,[6, 7]=7,[4, 8]=8,[8, 4]=8,[5, 6]=2,[6, \
 5]=2)

 Edge weights: table([2, 8]=1,[8, 2]=1,[3, 7]=1,[7, 3]=1,[6, 4]\
 =1,[4, 6]=1,[4, 5]=1,[5, 4]=1,[6, 3]=1,[3, 6]=1,[2, 7]=1,[7, 2\
 ]=1,[6, 2]=1,[2, 6]=1,[1, 7]=1,[7, 1]=1,[3, 4]=1,[4, 3]=1,[4, \
 1]=1,[1, 4]=1,[3, 1]=1,[1, 3]=1,[2, 1]=1,[1, 2]=1,[8, 6]=1,[6,\
  8]=1,[7, 6]=1,[6, 7]=1,[4, 8]=1,[8, 4]=1,[5, 6]=1,[6, 5]=1)

 Adjacency list (out): table(8=[4, 2, 6],7=[2, 3, 6, 1],6=[5, 4\
```

```
, 7, 3, 8, 2],5=[6, 4],4=[8, 6, 3, 5, 1],3=[1, 7, 4, 6],2=[7, \
1, 8, 6],1=[3, 2, 7, 4])

Adjacency list (in): table(8=[4, 2, 6],7=[2, 3, 6, 1],6=[5, 4,\
 7, 3, 8, 2],5=[6, 4],4=[8, 6, 3, 5, 1],3=[1, 7, 4, 6],2=[7, 1\
, 8, 6],1=[3, 2, 7, 4])
```

Changes:

- `Network::random` used to be `Network::Random`.

`Network::residualNetwork` – computes the residual network

`Network::residualNetwork`G, `f` computes the residual of the network `G` with respect to the flow `f`, i.e., loosely speaking, the network that remains when the flow `f` is "subtracted" from `G`.

Call(s):

- `Network::residualNetwork(G, f<, Extended>)`

Parameters:

`G` — network
`f` — flow

Options:

`Extended` — include edges with zero capacities

Return Value: A `Network`

Details:

- `Network::residualNetwork` computes the residual network with respect to a given flow. A flow in a network is a table `t`, where `t[[i,j]]` gives the number of units flowing from node `i` to node `j`.
- If the optional argument `Extended` is given, then also those edges with a zero residual capacity are contained. Otherwise those edges are omitted.

Example 1.

```
>> N1 := Network::complete(3):
   N2 := Network::residualNetwork(N1,
        table( [1, 2] = 1, [2, 1] = 1/2,
               [1, 3] = 0, [3, 1] = 0.5,
               [2, 3] = 1, [3, 2] = 0 ) ):
   Network::eCapacity(N1), Network::eCapacity(N2)

                table(
                  [3, 2] = 1,  table(
                  [3, 1] = 1,    [3, 2] = 1,
                  [2, 3] = 1,,   [3, 1] = 0.5,
                  [2, 1] = 1,    [2, 1] = 1/2,
                  [1, 3] = 1,    [1, 3] = 1
                  [1, 2] = 1   )
                )
```

Example 2.

```
>> V := [1,2,3,q,s]:
   Edge := [[q,1], [1,2], [1,3], [2,3], [3,s]]:
   up := [5, 4, 4, 2, 5]:
   N := Network(V,Edge,Capacity=up):
   flow := table([q, 1]=5,[3, s]=5,[1, 2]=1,[1, 3]=4,[2, 3]=1):
   N1 := Network::residualNetwork(N, flow):
   Network::printGraph(N1);

                    Vertices: [1, 2, 3, q, s]

 Edges: [[1, 2], [2, 3], [1, q], [2, 1], [3, 1], [3, 2], [s, 3]]

            Vertex weights: table(s=0,q=0,3=0,2=0,1=0)

 Edge capacities: table([s, 3]=5,[3, 2]=1,[3, 1]=4,[2, 1]=1,[1,\
  q]=5,[2, 3]=1,[1, 2]=3)

 Edge weights: table([s, 3]=-1,[3, 2]=-1,[3, 1]=-1,[2, 1]=-
1,[1\
 , q]=-1,[2, 3]=1,[1, 2]=1)

 Adjacency list (out): table(s=[3],q=[],3=[1, 2],2=[3, 1],1=[2,\
  q])

 Adjacency list (in): table(s=[],q=[1],3=[2, s],2=[1, 3],1=[2, \
 3])
```

```
>> N1 := Network::residualNetwork(N, flow, Extended):
   Network::printGraph(N1);

                    Vertices: [1, 2, 3, q, s]

 Edges: [[q, 1], [1, 2], [1, 3], [2, 3], [3, s], [1, q], [2, 1]\
 , [3, 1], [3, 2], [s, 3]]

           Vertex weights: table(s=0,q=0,3=0,2=0,1=0)

 Edge capacities: table([s, 3]=5,[3, 2]=1,[3, 1]=4,[2, 1]=1,[1,\
  q]=5,[3, s]=0,[2, 3]=1,[1, 3]=0,[1, 2]=3,[q, 1]=0)

 Edge weights: table([s, 3]=-1,[3, 2]=-1,[3, 1]=-1,[2, 1]=-
1,[1\
 , q]=-1,[3, s]=1,[2, 3]=1,[1, 3]=1,[1, 2]=1,[q, 1]=1)

 Adjacency list (out): table(s=[3],q=[1],3=[s, 1, 2],2=[3, 1],1\
 =[2, 3, q])

 Adjacency list (in): table(s=[3],q=[1],3=[1, 2, s],2=[1, 3],1=\
 [q, 2, 3])
```

Changes:

- `Network::residualNetwork` used to be `Network::ResidualNetwork`.

`Network::shortPath` – shortest paths from one single node

`Network::shortPath(G, v)` returns a table with the length of shortest paths from `v` to all other nodes in the network with respect to the edge weight.

`Network::shortPath(G, v, w)` gives the length of a shortest path from `v` to `w`.

Call(s):

- `Network::shortPath(G, v<, w><, Length><, Path>)`

Parameters:

`G` — network
`v, w` — nodes in the network

Options:

Length — include table of path lengths; this is the default case, if *Path* is not given.

Path — return table of paths

Return Value: a number, a table or a sequence of two tables.

Details:

- If the optional argument `Path` is given, then a table with shortest pithes is returned.
- If `Length` and `Path` are given, then both the length of the shortest paths and the paths are returned.

Example 1.

```
>> V := [1, 2, 3, 4, 5]:
   Vw := [25, 0, 0, 0, -25]:
   Ed := [[1, 2], [1, 3], [2, 3],
          [2, 4], [3, 4], [3, 5], [4,  5]]:
   Ew := [7, 6, 5, 4, 2, 2, 1]:
   Ecap := [30, 20, 25, 10, 20, 25, 20]:
   N1 := Network(V, Ed, Eweight=Ew, Capacity=Ecap, Vweight=Vw):

>> Network::shortPath(N1,1)

                              table(
                                5 = 8,
                                4 = 8,
                                3 = 6,
                                2 = 7,
                                1 = 0
                              )

>> Network::shortPath(N1,1,Path)

                            table(
                              5 = [1, 3, 5],
                              4 = [1, 3, 4],
                              3 = [1, 3],
                              2 = [1, 2]
                            )
```

Background:

- If there are only non-negative edge weights the algorithm of Dijkstra is used. If there are positive and negative edge weights the algorithm of Bellman is used.

Changes:

- `Network::shortPath` used to be `Network::ShortPath`.

`Network::shortPathTo` – shortest paths to one single node

`Network::shortPathTo(G, v)` finds the shortest paths in the network `G` ending at node `v`.

Call(s):

- `Network::shortPathTo(G,v<, w><, Length><, Path>)`

Parameters:

`G` — network
`v, w` — nodes in the network

Options:

`Length` — include table of path lengths; default if *`Path`* not given

Return Value: a number, a table or a sequence of two tables

Details:

- `Network::shortPathTo(G, v)` returns a table with the length of shortest paths to `v` from all other nodes in the network with respect to the edge weight.
- `Network::shortPathTo(G, v, w)` gives the length of a shortest path from `w` to `v`.
- If the optional argument *`Path`*is given, then a table with shortest paths is returned. If *`Length`*and *`Path`*are given, then both the length of the shortest paths and the paths are returned.

Example 1.

```
>> V := [1,2,3,4,5]:
   Vw := [25,0,0,0,-25]:
   Ed := [[1,2], [1,3], [2,3], [2,4], [3,4], [3,5], [4,5]]:
   Ew := [7, 6, 5, 4, 2, 2, 1]:
   Ecap := [30, 20, 25, 10, 20, 25, 20]:
   N1 := Network(V,Ed,Eweight=Ew, Capacity=Ecap, Vweight=Vw):

>> Network::shortPathTo(N1, 5)

                              table(
                                5 = 0,
                                4 = 1,
                                3 = 2,
                                2 = 5,
                                1 = 8
                              )

>> Network::shortPathTo(N1, 5, Path)

                            table(
                              4 = [4, 5],
                              3 = [3, 5],
                              2 = [2, 4, 5],
                              1 = [1, 3, 5]
                            )
```

Background:

- If there are only non negative edge weights the algorithm of Dijkstra is used. If there are positive and negative edge weights the algorithm of Bellman is used.

Changes:

- `Network::shortPathTo` used to be `Network::ShortPathTo`.

`Network::showGraph` – plots a network

`Network::showGraph(N)` plots the network `N`.

Call(s):

- `Network::showGraph(N)`

Parameters:

N — network

Return Value: the value of type DOM_NULL

Details:

- Network::showGraph(N) gives s simple visual representation of the network N. Up to now no optimization with respect to a minimal number of intersection points of the edges is done. Actually, all nodes of the network are drawn at equal intervals around a circle.

Example 1.

```
>> Network::showGraph(Network::complete(4))
>> Network::showGraph(Network::random()[1])
```

Changes:

- Network::showGraph used to be Network::ShowGraph.

Network::topSort – topological sorting of the nodes

Network::topSort(G) computes a topological sorting of the network G, i.e., a numbering T of the nodes, such that $T[i] < T[j]$ whenever there is an edge $[i, j]$ in the network.

Call(s):

- Network::topSort(G)

Parameters:

G — network

Return Value: a table of nodes.

Details:

- If G contains any cycle then a topological sorting does not exist and the call of Network::topSort results in an error.

Example 1.

```
>> Network([1,2,3,4],[[1,2],[2,4],[3,4]]):
   Network::topSort(%)
                                table(
                                  4 = 4,
                                  2 = 3,
                                  3 = 2,
                                  1 = 1
                                )

>> Network::topSort(Network::complete(3))

 Error: Network contains cycle [Network::topSort]
```

Background:

- Note that the returned ordering is hardly ever unique.

Changes:

- `Network::topSort` used to be `Network::TopSort`.

`Network::vWeight` – returns the table of vertex weights

`Network::vWeight(G)` yields the table of the vertex weights of the network `G`.

Call(s):

- `Network::vWeight(G)`

Parameters:

`G` — network

Return Value: a table

Details:

- `Network::vWeight(G)` returns a table with the vertex weight of the network `G`. Thus `Network::vWeight(G)[v]` is the weight of vertex `v` in the network `G`.

Example 1.

```
>> V := [1,2,3,4,5]:
   Vw := [25,0,0,0,-25]:
   Ed := [[1,2], [1,3], [2,3], [2,4], [3,4], [3,5], [4,5]]:
   N1 := Network(V,Ed, Vweight=Vw):

>> Network::vWeight(N1)
```

```
table(
  5 = -25,
  4 = 0,
  3 = 0,
  2 = 0,
  1 = 25
)
```

Example 2.

```
>> N2 := Network::complete(5):
   Network::vWeight(N2)
```

```
table(
  5 = 0,
  4 = 0,
  3 = 0,
  2 = 0,
  1 = 0
)
```

Changes:

- `Network::vWeight` used to be `Network::VWeight`.

`Network::vertex` – returns a list with all vertices

`Network::vertex(G)` returns the list of all vertices of the network `G`.

Call(s):

- `Network::vertex(G)`

Parameters:

`G` — network

Return Value: a list

Details:

- `Network::vertex(G)` returns a list with all vertices of the network `G`.

Example 1.

```
>> N1 := Network::complete(10):
   Network::vertex(N1)

                [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>> N2 := Network::cycle([x.i $ i=1..12]):
   Network::vertex(N2)

        [x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12]
```

Changes:

- `Network::vertex` used to be `Network::Vertex`.

`Network::new` – generates a new network

`Network(V, E)` is used to generate a new `Network` with vertices `V` and edges `E`, which can be manipulated using the functions in the `Network` library.

Call(s):

- `Network::new(V, E)`
- `Network::new(V, E<, Eweight=lc><, Capacity=lt><, Vweight=lv>)`

Parameters:

`V` — list of expressions (nodes)
`E` — list of edges

Options:

`Eweight=lc`	— The weights of the edges, given as a list of numbers
`Capacity=lt`	— The capacity of the edges, given as a list of numbers
`Vweight=lv`	— The weights of the vertices, given as a list of numbers

Return Value: A `Network`

Details:

- `Network::new(V, E)` generates a new network. A network consists of a list of nodes and a list of edges connecting the nodes. These lists must be specified for the definition of a new network. If one of them is missing an error occurs.
- Instead of calling `Network::new(`*args*`)` the short form `Network(`*args*`)` can be used. The examples in the description below use this short form.
- `Network(V,E)` where `V` is a list of nodes and `E` a list of edges generates a new network with exactly this set of nodes and edges respectively. A node in a network can be an arbitrary expression. An edge is a list, which contains the start point and the endpoint of the edge. Therefor, if there are edges specified for which the incident nodes are not contained in the list `V` an error occurs.
- It is possible to assign a weight to each node and a capacity and a weight to each edge in the network. This can be done directly in the definition of a network via `Network(V, E, `*`Eweight`*`=lc, `*`Capacity`*`=lt, `*`Vweight`*`=lv)`. Here `lc`, `lt` and `lv` are numerical lists with exactly as many items as `E` and `V` respectively. For example, the capacity `lt[i]` is assigned to edge `E[i]`. If these specifications are missing, the default values `1` for edge weight and capacity and `0` for vertex weight are assumed.

Example 1.

```
>> V := [1,2,3,q,s]:
   Edge := [[q,1], [1,2], [1,3], [2,3], [3,s]]:
   up := [5, 4, 4, 2, 5]:
   N1 := Network(V,Edge,Capacity=up):
   Network::printGraph(N1);

                    Vertices: [1, 2, 3, q, s]

          Edges: [[q, 1], [1, 2], [1, 3], [2, 3], [3, s]]

            Vertex weights: table(s=0,q=0,3=0,2=0,1=0)

 Edge capacities: table([3, s]=5,[2, 3]=2,[1, 3]=4,[1, 2]=4,[q,\
  1]=5)

 Edge weights: table([3, s]=1,[2, 3]=1,[1, 3]=1,[1, 2]=1,[q, 1]\
 =1)

  Adjacency list (out): table(s=[],q=[1],3=[s],2=[3],1=[2, 3])
```

```
   Adjacency list (in): table(s=[3],q=[],3=[1, 2],2=[1],1=[q])

>> V := [1,2,3,4,5]:
   Vw := [25,0,0,0,-25]:
   Ed := [[1,2], [1,3], [2,3], [2,4], [3,4], [3,5], [4,5]]:
   Ew := [7, 6, 5, 4, 2, 2, 1]:
   Ecap := [30, 20, 25, 10, 20, 25, 20]:
   N2 := Network(V,Ed,Eweight=Ew, Capacity=Ecap, Vweight=Vw):
   Network::printGraph(N2)

                    Vertices: [1, 2, 3, 4, 5]

 Edges: [[1, 2], [1, 3], [2, 3], [2, 4], [3, 4], [3, 5], [4, 5]]

          Vertex weights: table(5=-25,4=0,3=0,2=0,1=25)

 Edge capacities: table([4, 5]=20,[3, 5]=25,[3, 4]=20,[2, 4]=10\
 ,[2, 3]=25,[1, 3]=20,[1, 2]=30)

 Edge weights: table([4, 5]=1,[3, 5]=2,[3, 4]=2,[2, 4]=4,[2, 3]\
 =5,[1, 3]=6,[1, 2]=7)

 Adjacency list (out): table(5=[],4=[5],3=[4, 5],2=[3, 4],1=[2,\
  3])

 Adjacency list (in): table(5=[3, 4],4=[2, 3],3=[1, 2],2=[1],1=\
 [])
```

Changes:

- No changes.