Implement a new Graph

The BRAPH 2 Developers

August 31, 2023

This is the developer tutorial for implementing a new graph. In this Tutorial, we will explain how to create the generator file `*.gen.m` for a new graph, which can then be compiled by `braph2genesis`. All graphs are (direct or indirect) extensions of the base element `Graph`. Here, we will use as examples the graphs `GraphBD` (Binary Directed graph), `MultilayerWU` (Weighted Undirected multilayer graph), `MultiplexBUT` (Binary Undirected multiplex at fixed Thresholds), and `OrdMxBUT` (Binary Undirected ordinal multiplex with fixed Thresholds).

Contents

Implementation of Unilayer Graphs 2
- *Unilayer Binary Directed Graph (GraphBD)* 2

Implementation of Multilayer Graphs 7
- *Weigthed Directed Multilayer Graph (MultilayerWD)* 7
- *Binary Undirected Multilayer Graph with fixed Thresholds (MultiplexBUT)* 12
- *Binary Undirected Ordinal Multiplex Graph with fixed Thresholds (OrdMxBUT)* 17

Implementation of Unilayer Graphs

Unilayer Binary Directed Graph (GraphBD)

We will start by implementing in detail `GraphBD`, which is a direct extension of `Graph`. A unilayer graph is constituted by nodes connected by edges, where the edges are directed and they can be either 0 (absence of connection) or 1 (existence of connection).

Code 1: **GraphBD element header.** The `header` section of the generator code for `_GraphBD.gen.m` provides the general information about the `GraphBD` element.

```
%% ¡header!
GraphBD < Graph (g, binary directed graph) is a binary directed graph. 1

%%% ¡description!
In a binary directed (BD) graph, the edges are directed and they can be
     either 0 (absence of connection) or 1 (existence of connection).
```

1 defines `GraphBD` as a subclass of `Graph`. The moniker will be `g`.

Code 2: **GraphBD element prop update.** The `props_update` section of the generator code for `GraphBD.gen.m` updates the properties of the `Graph` element. This defines the core properties of the graph.

```
%% ¡props_update!

%%% ¡prop!
NAME (constant, string) is the name of the binary directed graph.
%%%% ¡default!
'GraphBD'

%%% ¡prop!
DESCRIPTION (constant, string) is the description of the  binary directed
     graph.
%%%% ¡default!
'In a binary directed (BD) graph, the edges are directed and they can be
     either 0 (absence of connection) or 1 (existence of connection).'

%%% ¡prop!
TEMPLATE (parameter, item) is the template of the  binary directed graph.

%%% ¡prop!
ID (data, string) is a few-letter code of the  binary directed graph.
%%%% ¡default!
'GraphBD ID'

%%% ¡prop!
LABEL (metadata, string) is an extended label of the binary directed graph.
%%%% ¡default!
'GraphBD label'

%%% ¡prop!
NOTES (metadata, string) are some specific notes about the binary directed
     graph.
%%%% ¡default!
'GraphBD notes'

```

```
%%% ¡prop! (1)
GRAPH_TYPE (constant, scalar) returns the graph type __Graph.GRAPH__.
%%%% ¡default!
Graph.GRAPH

%%% ¡prop! (2)
CONNECTIVITY_TYPE (query, smatrix) returns the connectivity type __Graph.
      BINARY__.
%%%% ¡default!
value = Graph.BINARY;

%%% ¡prop! (3)
DIRECTIONALITY_TYPE  (query, smatrix) returns the directionality type
      __Graph.DIRECTED__.
%%%% ¡default!
value = Graph.DIRECTED;

%%% ¡prop! (4)
SELFCONNECTIVITY_TYPE (query, smatrix) returns the self-connectivity type
      __Graph.NONSELFCONNECTED__.
%%%% ¡default!
value = Graph.NONSELFCONNECTED;

%%% ¡prop! (5)
NEGATIVITY_TYPE (query, smatrix) returns the negativity type __Graph.
      NONNEGATIVE__.
%%%% ¡default!
value = Graph.NONNEGATIVE;

%%% ¡prop! (6)
A (result, cell) is the binary adjacency matrix of the binary directed graph
      .
%%%% ¡calculate!
B = g.get('B'); (7)

B = dediagonalize(B);  (8)
B = semipositivize(B, 'SemipositivizeRule', g.get('SEMIPOSITIVIZE_RULE'));
      (9)
B = binarize(B); (10)

A = {B}; (11)
value = A; (12)

%%%% ¡gui! (13)
pr = PanelPropCell('EL', g, 'PROP', GraphBD.A, ...
  'TABLE_HEIGHT', s(40), ...
  'XSLIDERSHOW', false, ...
  'YSLIDERSHOW', false, ...
  'ROWNAME' , g.getCallback('ANODELABELS'), ...
  'COLUMNNAME', g.getCallback('ANODELABELS'));

%%% ¡prop! (14)
COMPATIBLE_MEASURES (constant, classlist) is the list of compatible measures
      .
%%%% ¡default!
getCompatibleMeasures('GraphBD')
```

(1) defines the *graph type*: `Graph.GRAPH` (single layer), `Graph.MULTIGRAPH` (multiple unconnected layers), `Graph.MULTILAYER` (multiple layers), `Graph.ORDERED_MULTILAYER` (multiple layers with subsequent layers) `Graph.MULTIPLEX` (multilayer with connections between corresponding nodes), and `Graph.ORDERED_MULTIPLEX` (multilayer with connections between corresponding nodes in subsequent layers).

(2) defines the *graph connectivity*: `Graph.BINARY` (0 or 1) or `Graph.WEIGHTED`.

(3) defines the *edge directionality*: `Graph.DIRECTED` or `Graph.UNDIRECTED`.

(4) defines the *graph self-connectivity*: `Graph.NONSELFCONNECTED` or `Graph.SELFCONNECTED`.

(5) defines the *graph negativity*: `Graph.NONNEGATIVE` or `Graph.NEGATIVE`.

(6) The property `A` contains the supra-adjacency matrix of the graph, which is calculated by the code under `¡calculate!`.

(7) retrieves the adjacency matrix of the graph `B`, defined in the new properties below.

(8), (9), and (10) condition the adjacency matrix removing the diagonal elements, making it semidefinte positive, and binarizing it. A list of useful functions is: `diagonalize` (removes the off-diagonal), `dediagonalize` (removes the diagonal), `binarize` (binarizes with threshold=0), `semipositivize` (removes negative weights), `standardize` (normalizes between 0 and 1) or `symmetrize` (symmetrizes the matrix). Use the MatLab help to see additional functionalities.

(11) preallocates the adjacency matrix to be calcualted.

(12) returns the calculated graph `A` assigning it to the output variable `value`.

(13) employes the property panel `PanelPropCell` to be employed to visualize `A`, setting also its properties.

(14) determines the list of compatible figures.

Code 3: **GraphBD element props.** The `props` section of generator code for `GraphBD.gen.m` defines the properties to be used in `GraphBD`.

```
%% iprops!

%%% iprop! 1
B (data, smatrix) is the input graph adjacency matrix.
%%%% igui! 2
pr = PanelPropMatrix('EL', g, 'PROP', GraphBD.B, ...
  'TABLE_HEIGHT' , s(40), ...
  'ROWNAME' , g.getCallback('ANODELABELS'), ...
  'COLUMNNAME', g.getCallback('ANODELABELS'), ...
  varargin{:});

%%% iprop! 3
SEMIPOSITIVIZE_RULE (parameter, option) determines how to remove the negative edges.
%%%% isettings!
{'zero', 'absolute'}
```

1 contains the input adjacency matrix `B`, which is typically weighted and directed.

2 defines the property panel `PanelPropMatrix` to plot this property with a table.

3 defines the semi-positivation rule (i.e., how to remove the negative edges) to be used when generating the adjacency matrix `A` from the intput property `B`. The admissible options are: 'zero' (default, convert negative values to zeros) or 'absolute' (convert negative values to absolute value).

Code 4: **GraphBD element tests.** The `tests` section from the element generator `_GraphBD.gen.m`. A general test should be prepared to test the properties of the graph when it is empty and full. Furthermore, additional tests should be prepared for the rules defined (one test per rule).

```
%% ¡tests!

%%% ¡excluded_props! (1)
[GraphBD.PFGA GraphBD.PFGH]

%%% ¡test!
%%%% ¡name!
Constructor - Empty (2)
%%%% ¡probability! (3)
.01
%%%% ¡code!
B = []; (4)
g = GraphBD('B', B);(5)

g.get('A_CHECK'); (6)

A = {binarize(semipositivize(dediagonalize(B)))}; (7)
assert(isequal(g.get('A'), A), ...(8)
  [BRAPH2.STR ':GraphBD:' BRAPH2.FAIL_TEST], ...
  'GraphBD is not constructing well.')

%%% ¡test!
%%%% ¡name!
Constructor - Full (9)
%%%% ¡probability!
.01
%%%% ¡code!
B = randn(randi(10)); (10)
g = GraphBD('B', B);

g.get('A_CHECK')

A = {binarize(semipositivize(dediagonalize(B)))};
assert(isequal(g.get('A'), A), ...
  [BRAPH2.STR ':GraphBD:' BRAPH2.FAIL_TEST], ...
  'GraphBD is not constructing well.')

%%% ¡test!
%%%% ¡name!
Semipositivize Rules (11)
%%%% ¡probability!
.01 (3)
%%%% ¡code!
B = [ (12)
  -2 -1 0 1 2
  -1 0 1 2 -2
  0 1 2 -2 -1
  1 2 -2 -1 0
  2 -2 -1 0 1
  ];
```

(1) List of properties that are excluded from testing.

(2) checks that an empty `GraphBD` graph is constructing well.

(3) assigns a low test execution probability.

(4) initializes an empty input adjacency matrix `B`.

(5) constructs the `GraphBD` graph from the initialized `B`.

(6) performs the corresponding checks for the format of the adjacency matrix `A`: `GRAPH_TYPE`, `CONNECTIVITY_TYPE`, `DIRECTIONALITY_TYPE`, `SELFCONNECTIVITY_TYPE`, and `NEGATIVITY_TYPE`.

(7) calculates the value of the graph by apply the corresponding properties function.

(8) tests that the value of generated graph calculated by applying the properties functions coincides with the expected value.

(9) checks that a full `GraphBD` graph is constructing well.

(10) generates a random input adjacency matrix `B`.

(11) checks the `SEMIPOSITIVIZE_RULE` on the `GraphBD` graph.

(12) generates an input adjacency matrix with negative weights.

```
g0 = GraphBD('B', B);  13
A0 = {[  14
  0 0 0 1 1
  0 0 1 1 0
  0 1 0 0 0
  1 1 0 0 0
  1 0 0 0 0
  ]};
assert(isequal(g0.get('A'), A0), ...
  [BRAPH2.STR ':GraphBD:' BRAPH2.FAIL_TEST], ...
  'GraphBD is not constructing well.')

g_zero = GraphBD('B', B, 'SEMIPOSITIVIZE_RULE', 'zero');  15
A_zero = {[
  0 0 0 1 1
  0 0 1 1 0
  0 1 0 0 0
  1 1 0 0 0
  1 0 0 0 0
  ]};
assert(isequal(g_zero.get('A'), A_zero), ...
  [BRAPH2.STR ':GraphBD:' BRAPH2.FAIL_TEST], ...
  'GraphBD is not constructing well.')

g_absolute = GraphBD('B', B, 'SEMIPOSITIVIZE_RULE', 'absolute');  16
A_absolute = {[
  0 1 0 1 1
  1 0 1 1 1
  0 1 0 1 1
  1 1 1 0 0
  1 1 1 0 0
  ]};
assert(isequal(g_absolute.get('A'), A_absolute), ...
  [BRAPH2.STR ':GraphBD:' BRAPH2.FAIL_TEST], ...
  'GraphBD is not constructing well.')
...
```

13 constructs the `GraphBD` graph from the initialized `B` with default RULE for `SEMIPOSITIVIZE_RULE`.

14 provides the expected value of `A` calculated by external means.

15 constructs the `GraphBD` graph from the initialized `B` with RULE = 'zero' for `SEMIPOSITIVIZE_RULE`.

16 constructs the `GraphBD` graph from the initialized `B` with RULE = 'absolute' for `SEMIPOSITIVIZE_RULE`.

Implementation of Multilayer Graphs

Weigthed Directed Multilayer Graph (MultilayerWD)

We can now use `GraphBD` as the basis to implement the `MultilayerWD` graph. The parts of the code that are modified are highlighted. A multilayer graph allows connections between any nodes across the multiple layers, where all layers are interconnected following a categorical fashion.

Code 5: **MultilayerWD element header.** The `header` section of generator code for `_MultilayerWD.gen.m` provides the general information about the `MultilayerWD` element. ← Code 1

```
%% ¡header!
MultilayerWD < Graph (g, multilayer weighted directed graph) is a multilayer weighted directed graph.

%%% ¡description!
In a multilayer weighted directed (WD) graph, layers could have different number of nodes with within-layer weighted directed edges, associated with a real number between 0 and 1 and indicating the strength of the connection. The connectivity matrices are symmetric (within layer). All node connections are allowed between layers.
```

Code 6: **MultilayerWD element prop update.** The `props_update` section of generator code for `_MultilayerWD.gen.m` updates the properties of `MultilayerWD`. ← Code 2

```
%% ¡props_update!

%%% ¡prop!
NAME (constant, string) is the name of the multilayer weighted directed graph.
%%%% ¡default!
'MultilayerWD'

%%% ¡prop!
DESCRIPTION (constant, string) is the description of the multilayer weighted directed graph.
%%%% ¡default!
'In a multilayer weighted directed (WD) graph, layers could have different number of nodes with within-layer weighted directed edges, associated with a realnumber between 0 and 1 and indicating the strength of the connection. The connectivity matrices are symmetric (within layer). All node connections are allowed between layers.'

%%% ¡prop!
TEMPLATE (parameter, item) is the template of the multilayer weighted directed graph.

%%% ¡prop!
ID (data, string) is a few-letter code of the multilayer weighted directed graph.
%%%% ¡default!
'MultilayerWD ID'

```

```
%%% ¡prop!
LABEL (metadata, string) is an extended label of the multilayer weighted directed graph.
%%%% ¡default!
'MultilayerWD label'

%%% ¡prop!
NOTES (metadata, string) are some specific notes about the multilayer weighted directed graph.
%%%% ¡default!
'MultilayerWD notes'

%%% ¡prop!
GRAPH_TYPE (constant, scalar) returns the graph type __Graph.MULTILAYER__.
%%%% ¡default!
Graph.MULTILAYER

%%% ¡prop!
CONNECTIVITY_TYPE (query, smatrix) returns the connectivity type __Graph.WEIGHTED__ * ones(layernumber).
%%%% ¡calculate!
if isempty(varargin)
layernumber = 1;
else
layernumber = varargin{1};
end
value = Graph.WEIGHTED * ones(layernumber);

%%% ¡prop!
DIRECTIONALITY_TYPE (query, smatrix) returns the directionality type __Graph.DIRECTED__ * ones(layernumber).
%%%% ¡calculate!
if isempty(varargin)
  layernumber = 1;
else
  layernumber = varargin{1};
end
value = Graph.DIRECTED * ones(layernumber);

%%% ¡prop!
SELFCONNECTIVITY_TYPE (query, smatrix) returns the self-connectivity type __Graph.NONSELFCONNECTED__ on the diagonal and __Graph.SELFCONNECTED__ off diagonal.
%%%% ¡calculate!
if isempty(varargin)
  layernumber = 1;
else
  layernumber = varargin{1};
end
value = Graph.SELFCONNECTED * ones(layernumber);
value(1:layernumber+1:end) = Graph.NONSELFCONNECTED;

%%% ¡prop!
NEGATIVITY_TYPE (query, smatrix) returns the negativity type __Graph.NONNEGATIVE__ * ones(layernumber).
%%%% ¡calculate!
if isempty(varargin)
  layernumber = 1;
else
  layernumber = varargin{1};
end
```

```
value = Graph.NONNEGATIVE * ones(layernumber);

%%% ¡prop!
A (result, cell) is the cell containing the within-layer weighted adjacency matrices of the multilayer weighted directed graph and the connections between layers.

%%%% ¡calculate!
B = g.get('B');
L = length(B);
A = cell(L, L);
for i = 1:1:L 1
  M = dediagonalize(B{i,i});
  M = semipositivize(M, 'SemipositivizeRule', g.get('SEMIPOSITIVIZE_RULE'));
  M = standardize(M, 'StandardizeRule', g.get('STANDARDIZE_RULE'));
  A(i, i) = {M};
  if ~isempty(A{i, i})
    for j = i+1:1:L
      M = semipositivize(B{i,j}, 'SemipositivizeRule',  g.get('SEMIPOSITIVIZE_RULE'));
      M = standardize(M, 'StandardizeRule',   g.get('STANDARDIZE_RULE'));
      A(i, j) = {M};
      M = semipositivize(B{j,i}, 'SemipositivizeRule',  g.get('SEMIPOSITIVIZE_RULE'));
      M = standardize(M, 'StandardizeRule',   g.get('STANDARDIZE_RULE'));
      A(j, i) = {M};
    end
  end
end
value = A;
%%%% ¡gui!
pr = PanelPropCell('EL', g, 'PROP', MultilayerWD.A, ...
  'TABLE_HEIGHT', s(40), ...
  'XYSLIDERLOCK', true, ...
  'XSLIDERSHOW', false, ...
  'YSLIDERSHOW', true, ...
  'YSLIDERLABELS', g.getCallback('ALAYERLABELS'), ...
  'YSLIDERWIDTH', s(5), ...
  'ROWNAME', g.getCallback('ANODELABELS'), ...
  'COLUMNNAME', g.getCallback('ANODELABELS'), ...
  varargin{:});

%%% ¡prop!
PARTITIONS (result, rvector) returns the number of layers in the partitions of the graph.
%%%% ¡calculate!
value = ones(1, g.get('LAYERNUMBER'));

%%% ¡prop! 2
ALAYERLABELS (query, stringlist) returns the layer labels to be used by the slider.
%%%% ¡calculate!
alayerlabels = g.get('LAYERLABELS'); 3
if isempty(alayerlabels) && ~isa(g.getr('A'), 'NoValue') % ensures that it's not unecessarily calculated
  alayerlabels = cellfun(@num2str, num2cell([1:1:g.get('LAYERNUMBER')]), 'uniformoutput', false); 4
end
value = alayerlabels;
```

1 For each layer in `MultilayerWD` graph, the corresponding functions are applied as in the notes 8, 9, and 10 of Code 2.

2 These are some properties of graph adjacency matrix `A` that can be used in the gui to make the visualization user friendly. The list of properties that can be used are: `ALAYERTICKS` (to set ticks for each layer according to the layer number), `ALAYERLABELS` (to set labels for each layer), and `ANODELABELS` (to set the nodel labels for each layer)).

3 returns the labels of the graph layers provided by the user.

4 constructs the labels of the layers based on the number of the layer (in case no layer labels were provided by the user).

```
%%% ¡prop!
COMPATIBLE_MEASURES (constant, classlist) is the list of compatible measures
     .
%%%% ¡default!
getCompatibleMeasures('MultilayerWD')
```

Code 7: **MultilayerWD element props.** The `props` section of generator code for `MultilayerWD.gen.m` defines the properties to be used in `MultilayerWD`. ← Code 3

```
%% ¡props!

%%% ¡prop!
B (data, cell) is the input cell containing the multilayer adjacency
     matrices.
%%%% ¡default!
{[] []; [] []}
%%%% ¡gui! 1
pr = PanelPropCell('EL', g, 'PROP', MultilayerWD.B, ...
  'TABLE_HEIGHT', s(40), ...
  'XSLIDERSHOW', true, ...
  'XSLIDERLABELS', g.get('LAYERLABELS'), ...
  'XSLIDERHEIGHT', s(3.5), ...
  'YSLIDERSHOW', false, ...
  'ROWNAME', g.getCallback('ANODELABELS'), ...
  'COLUMNNAME', g.getCallback('ANODELABELS'), ...
  varargin{:});


%%% ¡prop!
SEMIPOSITIVIZE_RULE (parameter, option) determines how to remove the
     negative edges.
%%%% ¡settings!
{'zero', 'absolute'}

%%% ¡prop! 2
STANDARDIZE_RULE (parameter, option) determines how to normalize the weights
      between 0 and 1.
%%%% ¡settings!
{'threshold' 'range'}
```

1 Same as in note 2 of Code 3.

2 Same as in note 3 of Code 3.

Code 8: **MultilayerWD element tests.** The `tests` section from the element generator `_MultilayerWD.gen.m`. ← Code 4

```
%% ¡tests!

%%% ¡excluded_props!
[MultilayerWD.PFGA MultilayerWD.PFGH]

%%% ¡test!
%%%% ¡name!
Constructor - Full
%%%% ¡probability!
.01
%%%% ¡code!
B1 = rand(randi(10));
B2 = rand(randi(10));
B3 = rand(randi(10));
B12 = rand(size(B1, 1),size(B2, 2));
```

```
B13 = rand(size(B1, 1),size(B3, 2));
B23 = rand(size(B2, 1),size(B3, 2));
B21 = rand(size(B2, 1),size(B1, 2));
B31 = rand(size(B3, 1),size(B1, 2));
B32 = rand(size(B3, 1),size(B2, 2));
B = {
  B1                            B12                             B13
  B21                           B2                              B23
  B31                           B32                             B3
};
g = MultilayerWD('B', B);
g.get('A_CHECK')
A1 = standardize(semipositivize(dediagonalize(B1)));
A2 = standardize(semipositivize(dediagonalize(B2)));
A3 = standardize(semipositivize(dediagonalize(B3)));
A12 = standardize(semipositivize(B12));
A13 = standardize(semipositivize(B13));
A23 = standardize(semipositivize(B23));
A21 = standardize(semipositivize(B21));
A31 = standardize(semipositivize(B31));
A32 = standardize(semipositivize(B32));
B{1,1} = A1;
B{2,2} = A2;
B{3,3} = A3;
B{1,2} = A12;
B{1,3} = A13;
B{2,3} = A23;
B{2,1} = A21;
B{3,1} = A31;
B{3,2} = A32;
A = B;
assert(isequal(g.get('A'), A), ...
  [BRAPH2.STR ': MultilayerWD: ' BRAPH2.FAIL_TEST], ...
  'MultilayerWD  is not constructing well.')
```

Binary Undirected Multilayer Graph with fixed Thresholds (MultiplexBUT)

Now we implement the `MultiplexBUT` graph based on previous codes `GraphBD` and `MultilayerWD`, again highlighting the differences. A multiplex graph is a type of multilayer graph where only interlayer edges are allowed between homologous nodes. In this case, the layers follow a categorical architecture, which means that all layers are interconnected.

Code 9: **MultiplexBUT element header.** The `header` section of generator code for `_MultiplexBUT.gen.m` provides the general information about the `MultiplexBUT` element. ← Code 1

```
%% ¡header!
MultiplexBUT < MultiplexWU (g, binary undirected multiplex with fixed
    thresholds) is a binary undirected multiplex with fixed thresholds. 1

%%% ¡description!
In a binary undirected multiplex with fixed thresholds (BUT), the layers are
     those of binary undirected (BU) multiplex graphs derived from the same
     weighted supra-connectivity matrices binarized at different thresholds
    .The supra-connectivity matrix has a number of partitions equal to the
    number of thresholds.
```

1 MultiplexBUT is a child of `MultiplexWU`, which in turn derives from `Graph`.

Code 10: **MultiplexBUT element prop update.** The `props_update` section of generator code for `_MultiplexBUT.gen.m` updates the properties of `MultiplexBUT`. ← Code 2

```
%% ¡props_update!

%%% ¡prop!
NAME (constant, string) is the name of the binary undirected multiplex with
    fixed thresholds.
%%%% ¡default!
'MultiplexBUT'

%%% ¡prop!
DESCRIPTION (constant, string) is the description of the binary undirected
    multiplex with fixed thresholds.
%%%% ¡default!
'In a binary undirected multiplex with fixed thresholds (BUT), the layers
    are those of binary undirected (BU) multiplex graphs derived from the
    same weighted supra-connectivity matrices binarized at different
    thresholds. The supra-connectivity matrix has a number of partitions
    equal to the number of thresholds.'

%%% ¡prop!
TEMPLATE (parameter, item) is the template of the binary undirected
    multiplex with fixed thresholds.

%%% ¡prop!
ID (data, string) is a few-letter code of the binary undirected multiplex
    with fixed thresholds.
%%%% ¡default!
'MultiplexBUT ID'

%%% ¡prop!
```

```
LABEL (metadata, string) is an extended label of the binary undirected multiplex with fixed thresholds.
%%%% ¡default!
'MultiplexBUT label'

%%% ¡prop!
NOTES (metadata, string) are some specific notes about the binary undirected multiplex with fixed thresholds.
%%%% ¡default!
'MultiplexBUT notes'

%%% ¡prop!
GRAPH_TYPE (constant, scalar) returns the graph type __Graph.MULTIPLEX__.
%%%% ¡default!
Graph.MULTIPLEX

%%% ¡prop!
CONNECTIVITY_TYPE (query, smatrix) returns the connectivity type __Graph.BINARY__  * ones(layernumber).
%%%% ¡calculate!
if isempty(varargin)
  layernumber = 1;
else
  layernumber = varargin{1};
end
value = Graph.BINARY * ones(layernumber);

%%% ¡prop!
DIRECTIONALITY_TYPE (query, smatrix) returns the directionality type __Graph.UNDIRECTED__ * ones(layernumber).
%%%% ¡calculate!
if isempty(varargin)
  layernumber = 1;
else
  layernumber = varargin{1};
end
value = Graph.UNDIRECTED * ones(layernumber);

%%% ¡prop!
SELFCONNECTIVITY_TYPE (query, smatrix) returns the self-connectivity type __Graph.NONSELFCONNECTED__ on the diagonal and __Graph.SELFCONNECTED__ off diagonal.
%%%% ¡calculate!
if isempty(varargin)
  layernumber = 1;
else
  layernumber = varargin{1};
end
value = Graph.SELFCONNECTED * ones(layernumber);
value(1:layernumber+1:end) = Graph.NONSELFCONNECTED;

%%% ¡prop!
NEGATIVITY_TYPE (query, smatrix) returns the negativity type __Graph.NONNEGATIVE__ * ones(layernumber).
%%%% ¡calculate!
if isempty(varargin)
  layernumber = 1;
else
  layernumber = varargin{1};
end
value = Graph.NONNEGATIVE * ones(layernumber);
```

```

%%% ¡prop!
A (result, cell) is the cell containing multiplex binary adjacency matrices of the binary undirected multiplex.

%%%% ¡calculate!
A_WU = calculateValue@MultiplexWU(g, prop); 1

thresholds = g.get('THRESHOLDS'); 2
L = length(A_WU); % number of layers 3
A = cell(length(thresholds) * L); 4

if L > 0 && ~isempty(cell2mat(A_WU))
  A(:, :) = {eye(length(A_WU{1, 1}))};
  for i = 1:1:length(thresholds) 5
    threshold = thresholds(i);
    layer = 1;
    for j = (i - 1) * L + 1:1:i * L 6
      A{j, j} = dediagonalize(binarize(A_WU{layer, layer}, 'threshold', threshold)); 7
      layer = layer + 1;
    end
  end
end

value = A;

%%%% ¡gui! 8
pr = PanelPropCell('EL', g, 'PROP', MultiplexBUT.A, ...
  'TABLE_HEIGHT', s(40), ...
  'XYSLIDERLOCK', true, ...
  'XSLIDERSHOW', false, ...
  'YSLIDERSHOW', true, ...
  'YSLIDERLABELS', g.getCallback('ALAYERLABELS'), ...
  'YSLIDERWIDTH', s(5), ...
  'ROWNAME', g.getCallback('ANODELABELS'), ...
  'COLUMNNAME', g.getCallback('ANODELABELS'), ...
  varargin{:});

%%% ¡prop!
PARTITIONS (result, rvector) returns the number of layers in the partitions of the graph.
%%%% ¡calculate!
l = g.get('LAYERNUMBER');
thresholds = g.get('THRESHOLDS');
value = ones(1, length(thresholds)) * l / length(thresholds);

%%% ¡prop!
ALAYERLABELS (query, stringlist) returns the layer labels to be used by the slider.
%%%% ¡calculate!
alayerlabels = g.get('LAYERLABELS');
if ~isa(g.getr('A'), 'NoValue') && length(alayerlabels) ~= g.get('LAYERNUMBER') % ensures that it's not unecessarily calculated
  thresholds = cellfun(@num2str, num2cell(g.get('THRESHOLDS')), 'uniformoutput', false);

  if length(alayerlabels) == length(g.get('B'))
    blayerlabels = alayerlabels;
```

1 calculates the graph MultiplexWU calling its parent `MultiplexWU`.

2 gets the thresholds to be applied to `A_WU`.

3 gets the number of layers in graph `A_WU`.

4 The new `MultiplexBUT` graph will have `L` layers for each threshold applied.

5 iterates over all the thresholds to be applied.

6 iterates over all the layers in `A_WU`.

7 binarizes the present layer of the `A_WU` graph according to the present threshold.

8 Same as in note 2 of Code 2.

```
  else % includes isempty(layerlabels)
    blayerlabels = cellfun(@num2str, num2cell([1:1:length(g.get('B'))]), 'uniformoutput', false);
  end

  alayerlabels = {};
  for i = 1:1:length(thresholds) (9)
    for j = 1:1:length(blayerlabels)
      alayerlabels = [alayerlabels, [blayerlabels{j} '|' thresholds{i}]];
    end
  end
end
value = alayerlabels;

%%% iprop!
COMPATIBLE_MEASURES (constant, classlist) is the list of compatible measures.
%%%% idefault!
getCompatibleMeasures('MultiplexBUT')
```

(9) sets the labels of the layers considering the thresholds and the number of layers in each multiplex graph for each threshold

Code 11: **MultiplexBUT element props.** The `props` section of generator code for `MultiplexBUT.gen.m` defines the properties to be used in `MultiplexBUT`. ← Code 3

```
%% iprops!

%%% iprop!
THRESHOLDS (parameter, rvector) is the vector of thresholds.
%%%% igui! (1)
pr = PanelPropRVectorSmart('EL', g, 'PROP', MultiplexBUT.THRESHOLDS, ...
  'MAX', 1, ...
  'MIN', -1, ...
  varargin{:});
```

(1) `PanelPropRVectorSmart` plots the panel for a row vector with an edit field. Smart means that (almost) any MatLab expression leading to a correct row vector can be introduced in the edit field. Also, the value of the vector can be limited between some MIN and MAX.

Code 12: **MultiplexBUT element tests.** The `tests` section from the element generator `_MultiplexBUT.gen.m`. ← Code 4

```
%% itests!

%%% itest!
%%%% iname!
Constructor - Full
%%%% iprobability!
.01
%%%% icode!
B1 = [
  0 .1 .2 .3 .4
  .1 0 .1 .2 .3
  .2 .1 0 .1 .2
  .3 .2 .1 0 .1
  .4 .3 .2 .1 0
  ];
B = {B1, B1, B1}; (1)
thresholds = [0 .1 .2 .3 .4]; (2)
g = MultiplexBUT('B', B, 'THRESHOLDS', thresholds);

g.get('A_CHECK')

```

(1) creates an example of the necessary input adjacency matrices.

(2) defines the thresholds.

```
A = g.get('A');
for i = 1:1:length(B) * length(thresholds)
  for j = 1:1:length(B) * length(thresholds)
    if i == j
      threshold = thresholds(floor((i - 1) / length(B)) + 1);
      assert(isequal(A{i, i}, binarize(B1, 'threshold', threshold)), ...
      [BRAPH2.STR ':MultiplexBUT:' BRAPH2.FAIL_TEST], ...
      'MultiplexBUT is not constructing well.')
    else
      assert(isequal(A{i, j}, eye(length(B1))), ...
      [BRAPH2.STR ':MultiplexBUT:' BRAPH2.FAIL_TEST], ...
      'MultiplexBUT is not constructing well.')
    end
  end
end
```

Binary Undirected Ordinal Multiplex Graph with fixed Thresholds (OrdMxBUT)

Finally, we implement the `OrdMxBUT` graph based on previous codes `GraphBD`, `MultilayerWD` and `MultiplexBUT`, again highlighting the differences. An ordered multiplex is a type of multiplex graph that consists of a sequence of layers with ordinal edges between corresponding nodes in subsequent layers.

Code 13: **OrdMxBUT element header.** The `header` section of generator code for `_OrdMxBUT.gen.m` provides the general information about the `OrdMxBUT` element. ← Code 1

```
%% ¡header!
OrdMxBUT < OrdMxWU (g, ordinal multiplex binary undirected with fixed
    thresholds) is a binary undirected ordinal multiplex with fixed
    thresholds. (1)

%%% ¡description!
In a binary undirected ordinal multiplex with fixed thresholds (BUT), all
    the layers consist of binary undirected (BU) multiplex graphs derived
    from the same weighted supra-connectivity matrices binarized at
    different thresholds. The supra-connectivity matrix has a number of
    partitions equal to the number of thresholds. The layers are connected
    in an ordinal fashion, i.e., only consecutive layers are connected.
```

(1) `OrdMxBUT` is a child of `OrdMxWU`, which in turn derives from `Graph`.

Code 14: **OrdMxBUT element prop update.** The `props_update` section of generator code for `_OrdMxBUT.gen.m` updates the properties of `OrdMxBUT`. ← Code 10

```
%% ¡props_update!

%%% ¡prop!
NAME (constant, string) is the name of the binary undirected ordinal
    multiplex with fixed thresholds.s.
%%%% ¡default!
'OrdMxBUT'

%%% ¡prop!
DESCRIPTION (constant, string) is the description of the binary undirected
    ordinal multiplex with fixed thresholds..
%%%% ¡default!
'In a binary undirected ordinal multiplex with fixed thresholds (BUT), all
    the layers consist of binary undirected (BU) multiplex graphs derived
    from the same weighted supra-connectivity matrices binarized at
    different thresholds. The supra-connectivity matrix has a number of
    partitions equal to the number of thresholds. The layers are
    connectedin an ordinal fashion, i.e., only consecutive layers are
    connected.'

%%% ¡prop!
TEMPLATE (parameter, item) is the template of the binary undirected ordinal
    multiplex with fixed thresholds.

%%% ¡prop!
ID (data, string) is a few-letter code of the binary undirected ordinal
    multiplex with fixed thresholds.
```

```
%%%% ¡default!
'OrdMxBUT ID'

%%% ¡prop!
LABEL (metadata, string) is an extended label of the binary undirected ordinal multiplex with fixed thresholds.
%%%% ¡default!
'OrdMxBUT label'

%%% ¡prop!
NOTES (metadata, string) are some specific notes about the binary undirected ordinal multiplex with fixed thresholds.
%%%% ¡default!
'OrdMxBUT notes'

%%% ¡prop!
GRAPH_TYPE (constant, scalar) returns the graph type __Graph.ORDERED_MULTIPLEX__.
%%%% ¡default!
Graph.ORDERED_MULTIPLEX

%%% ¡prop!
CONNECTIVITY_TYPE (query, smatrix) returns the connectivity type __Graph.BINARY__ * ones(layernumber).
%%%% ¡calculate!
if isempty(varargin)
  layernumber = 1;
else
  layernumber = varargin{1};
end
value = Graph.BINARY * ones(layernumber);

%%% ¡prop!
DIRECTIONALITY_TYPE (query, smatrix) returns the directionality type __Graph.UNDIRECTED__ * ones(layernumber).
%%%% ¡calculate!
if isempty(varargin)
  layernumber = 1;
else
  layernumber = varargin{1};
end
value = Graph.UNDIRECTED * ones(layernumber);

%%% ¡prop!
SELFCONNECTIVITY_TYPE (query, smatrix) returns the self-connectivity type __Graph.NONSELFCONNECTED__ on the diagonal and __Graph.SELFCONNECTED__ off diagonal.
%%%% ¡calculate!
if isempty(varargin)
  layernumber = 1;
else
  layernumber = varargin{1};
end
value = Graph.SELFCONNECTED * ones(layernumber);
value(1:layernumber+1:end) = Graph.NONSELFCONNECTED;

%%% ¡prop!
NEGATIVITY_TYPE (query, smatrix) returns the negativity type __Graph.NONNEGATIVE__ * ones(layernumber).
%%%% ¡calculate!
if isempty(varargin)
```

```
  layernumber = 1;
else
  layernumber = varargin{1};
end
value = Graph.NONNEGATIVE * ones(layernumber);

%%% ¡prop!
A (result, cell) is the cell containingbinary supra-adjacency matrix of the binary undirected multiplex with fixed thresholds (BUT).

%%%% ¡calculate!
A_WU = calculateValue@OrdMxWU(g, prop);①

thresholds = g.get('THRESHOLDS'); ②
L = length(A_WU); % number of layers
A = cell(length(thresholds)*L);

if L > 0 && ~isempty(cell2mat(A_WU))
  A(:, :) = {zeros(length(A_WU{1, 1}))};
  for i = 1:1:length(thresholds) ③
    threshold = thresholds(i);
    layer = 1;
    for j = (i - 1) * L + 1:1:i * L ④
      for k = (i - 1) * L + 1:1:i * L
        if j == k ⑤
          A{j, j} = dediagonalize(binarize(A_WU{layer, layer}, 'threshold', threshold));
        elseif (j-k)==1 || (k-j)==1 ⑥
          A(j, k) = {eye(length(A{1, 1}))};
        else ⑦
          A(j, k) = {zeros(length(A{1, 1}))};
        end
      end
      layer = layer + 1;
    end
  end
end

value = A;

%%%% ¡gui!
pr = PanelPropCell('EL', g, 'PROP', OrdMxBUT.A, ...
  'TABLE_HEIGHT', s(40), ...
  'XYSLIDERLOCK', true, ...
  'XSLIDERSHOW', false, ...
  'YSLIDERSHOW', true, ...
  'YSLIDERLABELS', g.getCallback('ALAYERLABELS'), ...
  'YSLIDERWIDTH', s(5), ...
  'ROWNAME', g.getCallback('ANODELABELS'), ...
  'COLUMNNAME', g.getCallback('ANODELABELS'), ...
  varargin{:});

%%% ¡prop!
PARTITIONS (result, rvector) returns the number of layers in the partitions of the graph.
%%%% ¡calculate!
l = g.get('LAYERNUMBER');
thresholds = g.get('THRESHOLDS');
value = ones(1, length(thresholds)) * l / length(thresholds);
```

① calculates the graph OrdMxWU calling the parent `OrdMxWU`.

② Same as in notes ②, ③, and ④ of Code 10..

③ constructs an ordinal muliplex binary undirected graph for each threshold.

④ loops over the layers of `A_Wu` for each threshold.

⑤ sets the layers constructed by binarizing `A_Wu` according to the present threshold on the diagonal of the supra-adjacency matrix.

⑥ connects consecutive layers.

⑦ does NOT connect NON-consecutive layers.

```
%%% ¡prop!
ALAYERLABELS (query, stringlist) returns the layer labels to be used by the
    slider.
%%%% ¡calculate!
alayerlabels = g.get('LAYERLABELS');
if ~isa(g.getr('A'), 'NoValue') && length(alayerlabels) ~= g.get('
    LAYERNUMBER') % ensures that it's not unecessarily calculated
  thresholds = cellfun(@num2str, num2cell(g.get('THRESHOLDS')), '
    uniformoutput', false);

  if length(alayerlabels) == length(g.get('B'))
    blayerlabels = alayerlabels;
  else % includes isempty(layerlabels)
    blayerlabels = cellfun(@num2str, num2cell([1:1:length(g.get('B'))]), '
    uniformoutput', false);
  end

  alayerlabels = {};
  for i = 1:1:length(thresholds)
    for j = 1:1:length(blayerlabels)
      alayerlabels = [alayerlabels, [blayerlabels{j} '|' thresholds{i}]];
    end
  end
end
value = alayerlabels;

%%% ¡prop!
COMPATIBLE_MEASURES (constant, classlist) is the list of compatible measures
    .
%%%% ¡default!
getCompatibleMeasures('OrdMxBUT')
```

Code 15: **OrdMxBUT element props.** The `props` section of generator code for `OrdMxBUT.gen.m` defines the properties to be used in `MultiplexBUT`. ← Code 11

```
%% ¡props!

%%% ¡prop!
THRESHOLDS (parameter, rvector) is the vector of thresholds.
%%%% ¡gui!
pr = PanelPropRVectorSmart('EL', g, 'PROP', OrdMxBUT.THRESHOLDS, ...
  'MAX', 1, ...
  'MIN', -1, ...
  varargin{:});
```

Code 16: **OrdMxBUT element tests.** The `tests` section from the element generator `_OrdMxBUT.gen.m`. ← Code 12

```
%% ¡tests!

%%% ¡excluded_props!
[OrdMxBUT.PFGA OrdMxBUT.PFGH]

%%% ¡test!
%%%% ¡name!
Constructor - Full
%%%% ¡probability!
```

```
.01
%%%% ¡code!
B1 = [
  0 .1 .2 .3 .4
  .1 0 .1 .2 .3
  .2 .1 0 .1 .2
  .3 .2 .1 0 .1
  .4 .3 .2 .1 0
  ];
B = {B1, B1, B1};
thresholds = [0 .1 .2 .3 .4];
g = OrdMxBUT('B', B, 'THRESHOLDS', thresholds);

g.get('A_CHECK')

A = g.get('A');
for i = 1:1:length(thresholds)
  threshold = thresholds(i);
  for j = (i - 1) * length(B) + 1:1:i * length(B)
    for k = (i - 1) * length(B) + 1:1:i * length(B)
      if j == k
        assert(isequal(A{j, j}, binarize(B1, 'threshold', threshold)), ...
          [BRAPH2.STR ':OrdMxBUT:' BRAPH2.FAIL_TEST], ...
          'OrdMxBUT is not constructing well.')
      elseif (j-k)==1 || (k-j)==1
        assert(isequal(A{j, k}, eye(length(B1))), ...
          [BRAPH2.STR ':OrdMxBUT:' BRAPH2.FAIL_TEST], ...
          'OrdMxBUT is not constructing well.')
      else
        assert(isequal(A{j, k}, zeros(length(B1))), ...
          [BRAPH2.STR ':OrdMxBUT:' BRAPH2.FAIL_TEST], ...
          'OrdMxBUT is not constructing well.')
      end
    end
  end
end
```