

# *Implement a new Graph*

*The BRAPH 2 Developers*

*August 30, 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 be compiled by `braph2genesis`, using the graphs `GraphBD`, `MultilayerWU`, `MultiplexBUT` and `OrdMxBUT` as examples.

## *Contents*

<i>Implementation of Unilayer Graph</i>	<i>2</i>
<i>Unilayer Graph Binary Directed (GraphBD)</i>	<i>2</i>
<i>Implementation of Multilayer Graph</i>	<i>7</i>
<i>Multilayer Weigthed Directed Graph (MultilayerWD )</i>	<i>7</i>
<i>Multiplex Binary Undirected with fixed Thresholds Graph (MultiplexBUT)</i>	<i>12</i>
<i>Ordinal Multiplex Binary Undirected with fixed Thresholds Graph (OrdMxBUT)</i>	<i>17</i>

## Implementation of Unilayer Graph

### Unilayer Graph Binary Directed (GraphBD)

We will start by implementing in detail the graph GraphBD which is a direct extension of the element Graph.

A unilayer graph is represented by nodes connected with edges. This type of graph is used for single-layer weighted analysis (WU, WD).

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

---

```

1 %% iheader!
2 GraphBD < Graph (m, binary directed graph) is a binary directed graph. ①
3
4 %%% idescription!
5 In a binary directed (BD) graph, the edges are directed and they can be
6 either 0 (absence of connection) or 1 (existence of connection).
```

---

① The element GraphBD is defined as a subclass of Graph. The moniker will be m.

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

---

```

1 %% iprops_update!
2 %%% iprop!
3 NAME (constant, string) is the name of the binary directed graph.
4 %%% idefault!
5 'GraphBD'
6
7 %%% iprop!
8 DESCRIPTION (constant, string) is the description of the binary directed
9 graph.
10 %%% idefault!
11 'In a binary directed (BD) graph, the edges are directed and they can be
12 either 0 (absence of connection) or 1 (existence of connection).'
```

---

```

13 %%% iprop!
14 TEMPLATE (parameter, item) is the template of the binary directed graph.
15
16 %%% iprop!
17 ID (data, string) is a few-letter code of the binary directed graph.
18 %%% idefault!
19 'GraphBD ID'
20
21 %%% iprop!
22 LABEL (metadata, string) is an extended label of the binary directed graph.
23 %%% idefault!
24 'GraphBD label'
25
26 %%% iprop!
27 NOTES (metadata, string) are some specific notes about the binary directed
28 graph.
29 %%% idefault!
30 'GraphBD notes'
```

```

29 %% iprop! ①
30 GRAPH_TYPE (constant, scalar) returns the graph type __Graph.GRAPH__.
31 %%% idefault!
32 Graph.GRAPH
33
34 %% iprop! ②
35 CONNECTIVITY_TYPE (query, smatrix) returns the connectivity type __Graph.
    BINARY__.
36 %%% idefault!
37 value = Graph.BINARY;
38
39 %% iprop! ③
40 DIRECTIONALITY_TYPE (query, smatrix) returns the directionality type
    __Graph.DIRECTED__.
41 %%% idefault!
42 value = Graph.DIRECTED;
43
44 %% iprop! ④
45 SELFCONNECTIVITY_TYPE (query, smatrix) returns the self-connectivity type
    __Graph.NONSELFCONNECTED__.
46 %%% idefault!
47 value = Graph.NONSELFCONNECTED;
48
49 %% iprop! ⑤
50 NEGATIVITY_TYPE (query, smatrix) returns the negativity type __Graph.
    NONNEGATIVE__.
51 %%% idefault!
52 value = Graph.NONNEGATIVE;
53
54 %% iprop! ⑥
55 A (result, cell) is the binary adjacency matrix of the binary directed graph
    .
56 %%% icalculate!
57 B = g.get('B'); ⑦
58
59 B = dedagonalize(B); ⑧
60 B = semipositivize(B, 'SemipositivizeRule', g.get('SEMIPOSITIVIZE_RULE'));
61 B = binarize(B);
62
63 A = {B}; ⑨
64 value = A; ⑩
65
66 %%% igui! ⑪
67 pr = PanelPropCell('EL', g, 'PROP', GraphBD.A, ... ⑫
68 'TABLE_HEIGHT', s(40), ...
69 'XSLIDERSHOW', false, ...
70 'YSLIDERSHOW', false, ...
71 'ROWNAME', g.getCallback('ANODELABELS'), ...
72 'COLUMNNAME', g.getCallback('ANODELABELS'), ...
73 );
74
75
76
77
78
79
80

```

① We need to define the type of graph: `Graph.GRAPH` (consists of a single layer), `Graph.MULTIGRAPH` (multiple unconnected layers of graphs), `Graph.MULTILAYER` (multiple layers with categorical connections between any nodes), `Graph.ORDERED_MULTILAYER` (multiple layers with ordinal connections between any nodes), `Graph.MULTIPLEX` (multilayer graph where only interlayer edges are allowed between homologous nodes) and `Graph.ORDERED_MULTIPLEX` (multiplex graph that consists of a sequence of layers with ordinal edges between corresponding nodes in subsequent layers).

② Graphs have a `CONNECTIVITY_TYPE`: `Graph.BINARY` (Graph with binary, 0 or 1, connections) or `Graph.WEIGHTED` (Graph with weighted connections).

③ Graphs have a `DIRECTIONALITY_TYPE`: `Graph.DIRECTED` (graph with directed edges) or `Graph.UNDIRECTED` (graph with undirected edges).

④ Graphs have a `SELFCONNECTIVITY_TYPE`: `Graph.NONSELFCONNECTED` (Graph without self-connections) or `Graph.SELFCONNECTED` (Graph with self-connections).

⑤ Graphs have a `NEGATIVITY_TYPE`: `Graph.NONNEGATIVE` (Graph without negative edges) or `Graph.NEGATIVE` (Graph allowing negative edges).

⑥ The property A contains the code to be executed to calculate the graph.

⑦ Retrieves the cell with the adjacency matrix of the graph

⑧ Apply corresponding functions to define the properties of the graph: `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)

⑨ Preallocates the adjacency matrix that contains the result of the defined graph.

⑩ Returns the calculated graph A assigning it to the output variable `value`.

⑪ Each graph has a panel figure of the cell containing the graph adjacency matrix.

⑫ `PanelPropCell` plots the panel for a CELL property with a table and two sliders. It can be personalized with props, e.g., `TABLE_HEIGHT` (height in pixels), `XSLIDERSHOW` (whether to show the x-slider), or `COLUMNNAME` (string list with column names)

```

81 %%% iprop! (13)
82 COMPATIBLE_MEASURES (constant, classlist) is the list of compatible measures
83 .
84 %%% idefault!
85 getCompatibleMeasures('GraphBD')
86
87 %%% iprop! (14)
88 B (data, smatrix) is the input graph adjacency matrix.
89 %%% igui!
90 pr = PanelPropMatrix('EL', g, 'PROP', GraphBD.B, ... (15)
91 'TABLE_HEIGHT', s(40), ...
92 'ROWNAME', g.getCallback('ANODELABELS'), ...
93 'COLUMNNAME', g.getCallback('ANODELABELS'), ...
94 varargin{:});
95
96 %%% iprop! (16)
97 SEMIPOSITIVIZE_RULE (parameter, option) determines how to remove the
98 negative edges.
99 %%% isettings!
100 {'zero', 'absolute'}

```

---

(13) Each graph has a list of compatible measures.

(14) Each graph has a panel figure of the graph adjacency matrix.

(15) PanelPropMatrix plots the panel of a property matrix-like with a table. It can be personalized with props as in

(12).

(16) Each graph have different rules that need to be defined:  
 SYMMETRIZE\_RULE: symmetrizes the matrix A by the symmetrize rule specified by RULE and the admissible RULE options are: 'max' (default, maximum between inconnection and outconnection), 'sum' (convert negative values to absolute value), 'average' (average of inconnection and outconnection) or 'min' (minimum between inconnection and outconnection)  
 SEMIPOSITIVIZE\_RULE: determines how to remove the negative edges and the admissible RULE options are: 'zero' (default, convert negative values to zeros) or 'absolute' (convert negative values to absolute value)  
 STANDARDIZE\_RULE : determines how to normalize the weights between 0 and 1 and the admissible RULE options are: 'threshold' (default, normalizes the matrix A by converting negative values to zero and values larger than 1 to 1) or 'range' (normalizes the matrix A in order to have values scaled between 0 and 1 by using a linear function).

Code 3: **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).

```

1 %% itests!
2
3 %% iexcluded_props! ①
4 [GraphBD.PFGA GraphBD.PFGH]
5
6 %% itest!
7 %%% iname!
8 Constructor - Empty ②
9 %%% iprobability! ③
10 .01
11 %%% icode!
12 B = []; ④
13 g = GraphBD('B', B); ⑤
14
15 g.get('A-CHECK'); ⑥
16
17 A = {binarize(semipositivize(dediagonalize(B)))}; ⑦
18 assert(isequal(g.get('A'), A), ... ⑧
19 [BRAPH2.STR ':GraphBD:' BRAPH2.FAIL_TEST], ...
20 'GraphBD is not constructing well.')
21
22 %% itest!
23 %%% iname!
24 Constructor - Full ⑨
25 %%% iprobability! ③
26 .01 ③
27 %%% icode!
28 B = randn(randi(10)); ⑩
29 g = GraphBD('B', B); ⑤
30
31 g.get('A-CHECK') ⑥
32
33 A = {binarize(semipositivize(dediagonalize(B)))}; ⑦
34 assert(isequal(g.get('A'), A), ... ⑧
35 [BRAPH2.STR ':GraphBD:' BRAPH2.FAIL_TEST], ...
36 'GraphBD is not constructing well.')
37
38 %% itest!
39 %%% iname!
40 Semipositivize Rules ⑪
41 %%% iprobability!
42 .01 ③
43 %%% icode!
44 B = [
45 -2 -1 0 1 2
46 -1 0 1 2 -2
47 0 1 2 -2 -1
48 1 2 -2 -1 0
49 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 GraphBD graph

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 graph

⑪ Checks the SEMIPOSITIVIZE\_RULE on the GraphBD graph.

```

50 ]; (12)
51
52 g0 = GraphBD('B', B); (13)
53 A0 = {[
54   0 0 0 1 1
55   0 0 1 1 0
56   0 1 0 0 0
57   1 1 0 0 0
58   1 0 0 0 0
59 ]}; (14)
60 assert(isequal(g0.get('A'), A0), ... (8)
61 [BRAPH2.STR ':GraphBD:' BRAPH2.FAIL_TEST], ...
62 'GraphBD is not constructing well.')
63
64 g_zero = GraphBD('B', B, 'SEMIPOSITIVIZE_RULE', 'zero'); (15)
65 A_zero = {[
66   0 0 0 1 1
67   0 0 1 1 0
68   0 1 0 0 0
69   1 1 0 0 0
70   1 0 0 0 0
71 ]}; (14)
72 assert(isequal(g_zero.get('A'), A_zero), ... (8)
73 [BRAPH2.STR ':GraphBD:' BRAPH2.FAIL_TEST], ...
74 'GraphBD is not constructing well.')
75
76 g_absolute = GraphBD('B', B, 'SEMIPOSITIVIZE_RULE', 'absolute'); (16)
77 A_absolute = {[
78   0 1 0 1 1
79   1 0 1 1 1
80   0 1 0 1 1
81   1 1 1 0 0
82   1 1 1 0 0
83 ]}; (14)
84 assert(isequal(g_absolute.get('A'), A_absolute), ... (8)
85 [BRAPH2.STR ':GraphBD:' BRAPH2.FAIL_TEST], ...
86 'GraphBD is not constructing well.')
87
88 ...

```

(12) Generates an example graph with negative weights

(13) Constructs the GraphBD graph from the initialized B with default RULE for SEMIPOSITIVIZE\_RULE.

(14) Expected value of the graph 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 Graph

### Multilayer Weighted Directed 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 network allows connections between any nodes across the multiple layers, where all layers are interconnected following a categorical fashion.

**Code 4: MultilayerWD element header.** The header section of generator code for `_MultilayerWD.gen.m` provides the general information about the MultilayerWD element. [← Code 1](#)

---

```

1 %% iheader!
2 MultilayerWD < Graph (g, multilayer weighted directed graph) is a multilayer
  weighted directed graph.
3
4 %%% idescription!
5 In a multilayer weighted directed (WD) graph, layers could have different
6 number of nodes with within-layer weighted directed edges, associated with a
7 real number between 0 and 1 and indicating the strength of the connection.
8 The connectivity matrices are symmetric (within layer).
9 All node connections are allowed between layers.
```

---

**Code 5: MultilayerWD element prop update.** The `props_update` section of generator code for `_MultilayerWD.gen.m` updates the properties of MultilayerWD. [← Code 2](#)

---

```

1 %% iprops_update!
2
3 %%% iprop!
4 NAME (constant, string) is the name of the multilayer weighted directed
  graph.
5 %%% idefault!
6 'MultilayerWD'
7
8 %%% iprop!
9 DESCRIPTION (constant, string) is the description of the multilayer weighted
  directed graph.
10 %%% idefault!
11 'In a multilayer weighted directed (WD) graph, layers could have different
  number
12 of nodes with within-layer weighted directed edges, associated with a real
13 number between 0 and 1 and indicating the strength of the connection.
14 The connectivity matrices are symmetric (within layer).
15 All node connections are allowed between layers.'
```

---

```

16
17 %%% iprop!
18 TEMPLATE (parameter, item) is the template of the multilayer weighted
  directed graph.
19
20 %%% iprop!
21 ID (data, string) is a few-letter code of the multilayer weighted directed
  graph.
22 %%% idefault!
```

```

23 'MultilayerWD ID'
24
25 %%% iprop!
26 LABEL (metadata, string) is an extended label of the multilayer weighted
    directed graph.
27 %%%% ndefault!
28 'MultilayerWD label'
29
30 %%% iprop!
31 NOTES (metadata, string) are some specific notes about the multilayer
    weighted directed graph.
32 %%%% ndefault!
33 'MultilayerWD notes'
34
35 %%% iprop!
36 GRAPH_TYPE (constant, scalar) returns the graph type __Graph.MULTILAYER__.
37 %%%% ndefault!
38 Graph.MULTILAYER
39
40 %%% iprop!
41 CONNECTIVITY_TYPE (query, smatrix) returns the connectivity type __Graph.
    WEIGHTED__ * ones(layernumber).
42 %%%% ndefault!
43 if isempty(varargin)
44     layernumber = 1;
45 else
46     layernumber = varargin{1};
47 end
48 value = Graph.WEIGHTED * ones(layernumber);
49
50 %%% iprop!
51 DIRECTIONALITY_TYPE (query, smatrix) returns the directionality type __Graph.
    DIRECTED__ * ones(layernumber).
52 %%%% ndefault!
53 if isempty(varargin)
54     layernumber = 1;
55 else
56     layernumber = varargin{1};
57 end
58 value = Graph.DIRECTED * ones(layernumber);
59
60 %%% iprop!
61 SELFCONNECTIVITY_TYPE (query, smatrix) returns the self-connectivity type
    __Graph.NONSELFCONNECTED__ on the diagonal and __Graph.SELFCONNECTED__
    off diagonal.
62 %%%% ndefault!
63 if isempty(varargin)
64     layernumber = 1;
65 else
66     layernumber = varargin{1};
67 end
68 value = Graph.SELFCONNECTED * ones(layernumber);
69 value(1:layernumber+1:end) = Graph.NONSELFCONNECTED;
70
71 %%% iprop!
72 NEGATIVITY_TYPE (query, smatrix) returns the negativity type __Graph.
    NONNEGATIVE__ * ones(layernumber).
73 %%%% ndefault!
74 if isempty(varargin)
75     layernumber = 1;
76 else

```



```

77     layernumber = varargin{1};
78 end
79 value = Graph.NONNEGATIVE * ones(layernumber);
80
81 %%% iprop!
82 A (result, cell) is the cell containing the within-layer weighted adjacency
83 matrices of the multilayer weighted directed graph and the connections
84 between layers.
85
86 %%% icalculate!
87 B = g.get('B');
88 L = length(B);
89 A = cell(L, L);
90 for i = 1:L (1)
91     M = dedagonalize(B{i,i});
92     M = semipositivize(M, 'SemipositivizeRule', g.get('SEMIPOSITIVIZE_RULE'));
93     M = standardize(M, 'StandardizeRule', g.get('STANDARDIZE_RULE'));
94     A(i, i) = {M};
95     if ~isempty(A{i, i})
96         for j = i+1:L
97             M = semipositivize(B{i,j}, 'SemipositivizeRule', g.get('
SEMIPOSITIVIZE_RULE'));
98             M = standardize(M, 'StandardizeRule', g.get('STANDARDIZE_RULE'));
99             A(i, j) = {M};
100             M = semipositivize(B{j,i}, 'SemipositivizeRule', g.get('
SEMIPOSITIVIZE_RULE'));
101             M = standardize(M, 'StandardizeRule', g.get('STANDARDIZE_RULE'));
102             A(j, i) = {M};
103         end
104     end
105 end
106 value = A;
107
108 %%% igui!
109 pr = PanelPropCell('EL', g, 'PROP', MultilayerWD.A, ...
110 'TABLE_HEIGHT', s(40), ...
111 'XSLIDERLOCK', true, ...
112 'XSLIDERSHOW', false, ...
113 'YSLIDERSHOW', true, ...
114 'YSLIDERLABELS', g.getCallback('ALAYERLABELS'), ...
115 'YSLIDERWIDTH', s(5), ...
116 'ROWNAME', g.getCallback('ANODELABELS'), ...
117 'COLUMNNAME', g.getCallback('ANODELABELS'), ...
118 varargin{:});
119
120 %%% iprop!
121 PARTITIONS (result, rvector) returns the number of layers in the partitions
    of the graph.
122 %%% icalculate!
123 value = ones(1, g.get('LAYERNUMBER'));
124
125 %%% iprop!
126 ALAYERLABELS (query, stringlist) returns the layer labels to be used by the
    slider.
127 %%% icalculate!
128 alayerlabels = g.get('ALAYERLABELS');
129 if isempty(alayerlabels) && ~isa(g.get('A'), 'NoValue') % ensures that it's
    not unnecessarily calculated
130     alayerlabels = cellfun(@(num2str, num2cell([1:L:g.get('LAYERNUMBER')]), '
uniformoutput', false);
131 end

```

(1) For each layer in MultilayerWD graph the corresponding functions are applied as in ← Code 2 (8)

```

132 value = alayerlabels;
133
134 %% iprop!
135 COMPATIBLE_MEASURES (constant, classlist) is the list of compatible measures
136
137 %%%% idefault!
138 getCompatibleMeasures('MultilayerWD')
139
140 %% iprops!
141
142 %%%% idefault!
143 B (data, cell) is the input cell containing the multilayer adjacency
144 matrices.
145 {[] []; [] []}
146 %%%% igui! ②
147 pr = PanelPropCell('EL', g, 'PROP', MultilayerWD.B, ...
148 'TABLE_HEIGHT', s(40), ...
149 'XSLIDERSHOW', true, ...
150 'XSLIDERLABELS', g.get('LAYERLABELS'), ...
151 'XSLIDERHEIGHT', s(3.5), ...
152 'YSLIDERSHOW', false, ...
153 'ROWNAME', g.getCallback('ANODELABELS'), ...
154 'COLUMNNAME', g.getCallback('ANODELABELS'), ...
155 varargin{:});
156
157 %% iprop!
158 SEMIPOSITIVIZE_RULE (parameter, option) determines how to remove the
159 negative edges.
160 %%%% isettings!
161 {'zero', 'absolute'}
162
163 %%%% idefault! ③
164 STANDARDIZE_RULE (parameter, option) determines how to normalize the weights
165 between 0 and 1.
166 %%%% isettings!
167 {'threshold' 'range'}

```

② Same as in ← Code 2 ⑫

③ Same as in ← Code 2 ⑫

Code 6: **MultilayerWD element tests.** The tests section from the element generator `_MultilayerWD.gen.m`. ← Code 3

```

1 %% itests!
2
3 %% iexcluded_props!
4 [MultilayerWD.PFGA MultilayerWD.PFGH]
5
6 %% itest!
7 %%%% iname!
8 Constructor - Full
9 %%%% iprobability!
10 .01
11 %%%% icode!
12 B1 = rand(randi(10));
13 B2 = rand(randi(10));
14 B3 = rand(randi(10));
15 B12 = rand(size(B1, 1), size(B2, 2));
16 B13 = rand(size(B1, 1), size(B3, 2));
17 B23 = rand(size(B2, 1), size(B3, 2));
18 B21 = rand(size(B2, 1), size(B1, 2));

```

```

19 B31 = rand(size(B3, 1),size(B1, 2));
20 B32 = rand(size(B3, 1),size(B2, 2));
21 B = {
22     B1                B12                B13
23     B21                B2                B23
24     B31                B32                B3
25 };
26 g = MultilayerWD('B', B);
27 g.get('A_CHECK')
28 A1 = standardize(semipositivize(dediagonalize(B1)));
29 A2 = standardize(semipositivize(dediagonalize(B2)));
30 A3 = standardize(semipositivize(dediagonalize(B3)));
31 A12 = standardize(semipositivize(B12));
32 A13 = standardize(semipositivize(B13));
33 A23 = standardize(semipositivize(B23));
34 A21 = standardize(semipositivize(B21));
35 A31 = standardize(semipositivize(B31));
36 A32 = standardize(semipositivize(B32));
37 B{1,1} = A1;
38 B{2,2} = A2;
39 B{3,3} = A3;
40 B{1,2} = A12;
41 B{1,3} = A13;
42 B{2,3} = A23;
43 B{2,1} = A21;
44 B{3,1} = A31;
45 B{3,2} = A32;
46 A = B;
47 assert(isequal(g.get('A'), A), ...
48 [BRAPH2.STR ': MultilayerWD: ' BRAPH2.FAIL_TEST], ...
49 'MultilayerWD is not constructing well.')

```

---

### *Multiplex Binary Undirected with fixed Thresholds Graph (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 inter-layer edges are allowed between homologous nodes. In this case, the layers follow a categorical architecture, which means that all layers are interconnected.

**Code 7: MultiplexBUT element header.** The header section of generator code for `_MultiplexBUT.gen.m` provides the general information about the MultiplexBUT element. ← [Code 1](#)

---

```

1  %% iheader!
2  MultiplexBUT < MultiplexWU (g, binary undirected multiplex with fixed
3  thresholds) is a binary undirected multiplex with fixed thresholds. ①
4
5  %%% idescription!
6  In a binary undirected multiplex with fixed thresholds (BUT), the layers
7  are those of binary undirected (BU) multiplex graphs derived from the
8  same weighted supra-connectivity matrices binarized at different
9  thresholds. The supra-connectivity matrix has a number of partitions
10 equal to the number of thresholds.
```

---

① MultiplexBUT is a child of MultiplexWU graph

**Code 8: MultiplexBUT element prop update.** The `props_update` section of generator code for `_MultiplexBUT.gen.m` updates the properties of MultiplexBUT. ← [Code 2](#)

---

```

1  %% iprops_update!
2
3  %%% iprop!
4  NAME (constant, string) is the name of the binary undirected multiplex
   with fixed thresholds.
5  %%% idefault!
6  'MultiplexBUT'
7
8  %%% iprop!
9  DESCRIPTION (constant, string) is the description of the binary undirected
10 multiplex with fixed thresholds.
11 %%% idefault!
12 'In a binary undirected multiplex with fixed thresholds (BUT), the layers
13 are those of binary undirected (BU) multiplex graphs derived from the
14 same weighted supra-connectivity matrices binarized at different
15 thresholds. The supra-connectivity matrix has a number of partitions
16 equal to the number of thresholds.'
```

---

```

17
18 %%% iprop!
19 TEMPLATE (parameter, item) is the template of the binary undirected
   multiplex with fixed thresholds.
20
21 %%% iprop!
22 ID (data, string) is a few-letter code of the binary undirected multiplex
   with fixed thresholds.
23 %%% idefault!
24 'MultiplexBUT ID'
25
```

```

26  %%% iprop!
27  LABEL (metadata, string) is an extended label of the binary undirected
    multiplex with fixed thresholds.
28  %%% idefault!
29  'MultiplexBUT label'
30
31  %%% iprop!
32  NOTES (metadata, string) are some specific notes about the binary
    undirected multiplex with fixed thresholds.
33  %%% idefault!
34  'MultiplexBUT notes'
35
36  %%% iprop!
37  GRAPH_TYPE (constant, scalar) returns the graph type __Graph.MULTIPLEX__.
38  %%% idefault!
39  Graph.MULTIPLEX
40
41  %%% iprop!
42  CONNECTIVITY_TYPE (query, smatrix) returns the connectivity type __Graph.
    BINARY__ * ones(layernumber).
43  %%% icalculate!
44  if isempty(varargin)
45      layernumber = 1;
46  else
47      layernumber = varargin{1};
48  end
49  value = Graph.BINARY * ones(layernumber);
50
51  %%% iprop!
52  DIRECTIONALITY_TYPE (query, smatrix) returns the directionality type
    __Graph.UNDIRECTED__ * ones(layernumber).
53  %%% icalculate!
54  if isempty(varargin)
55      layernumber = 1;
56  else
57      layernumber = varargin{1};
58  end
59  value = Graph.UNDIRECTED * ones(layernumber);
60
61  %%% iprop!
62  SELFCONNECTIVITY_TYPE (query, smatrix) returns the self-connectivity type
    __Graph.NONSELFCONNECTED__ on the diagonal and __Graph.SELFCONNECTED__
    off diagonal.
63  %%% icalculate!
64  if isempty(varargin)
65      layernumber = 1;
66  else
67      layernumber = varargin{1};
68  end
69  value = Graph.SELFCONNECTED * ones(layernumber);
70  value(1:layernumber+1:end) = Graph.NONSELFCONNECTED;
71
72  %%% iprop!
73  NEGATIVITY_TYPE (query, smatrix) returns the negativity type __Graph.
    NONNEGATIVE__ * ones(layernumber).
74  %%% icalculate!
75  if isempty(varargin)
76      layernumber = 1;
77  else
78      layernumber = varargin{1};
79  end

```

```

80 value = Graph.NONNEGATIVE * ones(layernumber);
81
82 %%% iprop!
83 A (result, cell) is the cell containing multiplex binary adjacency
    matrices of the binary undirected multiplex.
84
85 %%% icalculate!
86 A_WU = calculateValue@MultiplexWU(g, prop); ①
87
88 thresholds = g.get('THRESHOLDS'); ②
89 L = length(A_WU); % number of layers ③
90 A = cell(length(thresholds)*L); ④
91
92 if L > 0 && ~isempty(cell2mat(A_WU))
93     A(:, :) = {eye(length(A_WU{1, 1}))};
94     for i = 1:length(thresholds) ⑤
95         threshold = thresholds(i);
96         layer = 1;
97         for j = (i - 1) * L + 1:i * L ⑥
98             A{j, j} = dediagonalize(binrize(A_WU{layer, layer}, 'threshold',
                threshold)); ⑦
99             layer = layer + 1;
100         end
101     end
102 end
103
104 value = A;
105
106 %%% igui! ⑧
107 pr = PanelPropCell('EL', g, 'PROP', MultiplexBUT.A, ...
108     'TABLE_HEIGHT', s(40), ...
109     'XYSLIDERLOCK', true, ...
110     'XSLIDERSHOW', false, ...
111     'YSLIDERSHOW', true, ...
112     'YSLIDERLABELS', g.getCallback('ALAYERLABELS'), ...
113     'YSLIDERWIDTH', s(5), ...
114     'ROWNAME', g.getCallback('ANODELABELS'), ...
115     'COLUMNNAME', g.getCallback('ANODELABELS'), ...
116     varargin{:});
117
118 %%% iprop!
119 PARTITIONS (result, rvector) returns the number of layers in the
    partitions of the graph.
120 %%% icalculate!
121 l = g.get('LAYERNUMBER');
122 thresholds = g.get('THRESHOLDS');
123 value = ones(1, length(thresholds)) * l / length(thresholds);
124
125 %%% iprop!
126 ALAYERLABELS (query, stringlist) returns the layer labels to be used by
    the slider.
127 %%% icalculate!
128 alayerlabels = g.get('LAYERLABELS');
129 if ~isa(g.getr('A'), 'NoValue') && length(alayerlabels) ~= g.get('
    LAYERNUMBER') % ensures that it's not unnecessarily calculated
130     thresholds = cellfun(@num2str, num2cell(g.get('THRESHOLDS')), '
        uniformoutput', false);
131
132     if length(alayerlabels) == length(g.get('B'))

```

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 ← Code 2 ⑫

```

133     blayerlabels = alayerlabels;
134 else % includes isempty(layerlabels)
135     blayerlabels = cellfun(@num2str, num2cell([1:1:length(g.get('B'))]), '
uniformoutput', false);
136 end
137
138 alayerlabels = {};
139 for i = 1:1:length(thresholds)
140     for j = 1:1:length(blayerlabels)
141         alayerlabels = [alayerlabels, [blayerlabels{j} '|' thresholds{i}]];
142     end
143 end
144 end
145 value = alayerlabels;
146
147 %%% iprop!
148 COMPATIBLE_MEASURES (constant, classlist) is the list of compatible
    measures.
149 %%%% idefault!
150 getCompatibleMeasures('MultiplexBUT')
151
152 %% iprops!
153
154 %%% iprop!
155 THRESHOLDS (parameter, rvector) is the vector of thresholds.
156 %%%% igui! ⑨
157 pr = PanelPropRVectorSmart('EL', g, 'PROP', MultiplexBUT.THRESHOLDS, 'MAX'
    , 1, 'MIN', -1, varargin{:});
158
159

```

⑨ 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 9: **MultiplexBUT element tests.** The tests section from the element generator `_MultiplexBUT.gen.m`. ← [Code 3](#)

```

1  %% itests!
2
3  %%% itest!
4  %%%% iname!
5  Constructor - Full
6  %%%% iprobability!
7  .01
8  %%%% icode!
9  B1 = [
10  0 .1 .2 .3 .4
11  .1 0 .1 .2 .3
12  .2 .1 0 .1 .2
13  .3 .2 .1 0 .1
14  .4 .3 .2 .1 0
15  ];
16  B = {B1, B1, B1}; ①
17  thresholds = [0 .1 .2 .3 .4]; ②
18  g = MultiplexBUT('B', B, 'THRESHOLDS', thresholds);
19
20  g.get('A_CHECK')
21
22  A = g.get('A');
23  for i = 1:1:length(B) * length(thresholds)
24      for j = 1:1:length(B) * length(thresholds)
25          if i == j
26              threshold = thresholds(floor((i - 1) / length(B)) + 1);

```

① creates and example MultiplexWU  
 ② defines some example thresholds

```
27     assert(isequal(A{i, i}, binarize(B1, 'threshold', threshold)), ...
28     [BRAPH2.STR ':MultiplexBUT:' BRAPH2.FAIL_TEST], ...
29     'MultiplexBUT is not constructing well.')
30 else
31     assert(isequal(A{i, j}, eye(length(B1))), ...
32     [BRAPH2.STR ':MultiplexBUT:' BRAPH2.FAIL_TEST], ...
33     'MultiplexBUT is not constructing well.')
34 end
35 end
36 end
37
```

---



### *Ordinal Multiplex Binary Undirected with fixed Thresholds Graph (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 10: OrdMxBUT element header.** The header section of generator code for `_OrdMxBUT.gen.m` provides the general information about the OrdMxBUT element. ← [Code 1](#)

---

```

1  %% iheader!
2  OrdMxBUT < OrdMxWU (g, ordinal multiplex binary undirected with fixed
    thresholds) is a binary undirected ordinal multiplex with fixed
    thresholds. ①
3
4  %%% idescription!
5  In a binary undirected ordinal multiplex with fixed thresholds (BUT), all
6  the layers consist of binary undirected (BU) multiplex graphs derived
7  from the same weighted supra-connectivity matrices binarized at
8  different thresholds. The supra-connectivity matrix has a number of
9  partitions equal to the number of thresholds. The layers are connected
10 in an ordinal fashion, i.e., only consecutive layers are connected.
11

```

---

① OrdMxBUT is a child of OrdMxWU graph

**Code 11: OrdMxBUT element prop update.** The `props_update` section of generator code for `_OrdMxBUT.gen.m` updates the properties of OrdMxBUT. ← [Code 2](#)

---

```

1  %% iprops_update!
2
3  %%% iprop!
4  NAME (constant, string) is the name of the binary undirected ordinal
    multiplex with fixed thresholds.s.
5  %%% idefault!
6  'OrdMxBUT'
7
8  %%% iprop!
9  DESCRIPTION (constant, string) is the description of the binary undirected
    ordinal multiplex with fixed thresholds..
10 %%% idefault!
11 'In a binary undirected ordinal multiplex with fixed thresholds (BUT),
12 all the layers consist of binary undirected (BU) multiplex graphs
13 derived from the same weighted supra-connectivity matrices
14 binarized at different thresholds. The supra-connectivity matrix has a
15 number of partitions equal to the number of thresholds. The layers are
16 connected in an ordinal fashion, i.e., only consecutive layers are
17 connected.'
18
19 %%% iprop!
20 TEMPLATE (parameter, item) is the template of the binary undirected
    ordinal multiplex with fixed thresholds.
21
22 %%% iprop!
23 ID (data, string) is a few-letter code of the binary undirected ordinal
    multiplex with fixed thresholds.

```

---

```

24 %%%% idefault!
25 'OrdMxBUT ID'
26
27 %%% iprop!
28 LABEL (metadata, string) is an extended label of the binary undirected
    ordinal multiplex with fixed thresholds.
29 %%%% idefault!
30 'OrdMxBUT label'
31
32 %%% iprop!
33 NOTES (metadata, string) are some specific notes about the binary
    undirected ordinal multiplex with fixed thresholds.
34 %%%% idefault!
35 'OrdMxBUT notes'
36
37 %%% iprop!
38 GRAPH_TYPE (constant, scalar) returns the graph type __Graph.
    ORDERED_MULTIPLEX__.
39 %%%% idefault!
40 Graph.ORDERED_MULTIPLEX
41
42 %%% iprop!
43 CONNECTIVITY_TYPE (query, smatrix) returns the connectivity type __Graph.
    BINARY__ * ones(layernumber).
44 %%%% icalculate!
45 if isempty(varargin)
46     layernumber = 1;
47 else
48     layernumber = varargin{1};
49 end
50 value = Graph.BINARY * ones(layernumber);
51
52 %%% iprop!
53 DIRECTIONALITY_TYPE (query, smatrix) returns the directionality type
    __Graph.UNDIRECTED__ * ones(layernumber).
54 %%%% icalculate!
55 if isempty(varargin)
56     layernumber = 1;
57 else
58     layernumber = varargin{1};
59 end
60 value = Graph.UNDIRECTED * ones(layernumber);
61
62 %%% iprop!
63 SELFCONNECTIVITY_TYPE (query, smatrix) returns the self-connectivity type
    __Graph.NONSELFCONNECTED__ on the diagonal and __Graph.SELFCONNECTED__
    off diagonal.
64 %%%% icalculate!
65 if isempty(varargin)
66     layernumber = 1;
67 else
68     layernumber = varargin{1};
69 end
70 value = Graph.SELFCONNECTED * ones(layernumber);
71 value(1:layernumber+1:end) = Graph.NONSELFCONNECTED;
72
73 %%% iprop!
74 NEGATIVITY_TYPE (query, smatrix) returns the negativity type __Graph.
    NONNEGATIVE__ * ones(layernumber).
75 %%%% icalculate!
76 if isempty(varargin)

```

```

77 layernumber = 1;
78 else
79 layernumber = varargin{1};
80 end
81 value = Graph.NONNEGATIVE * ones(layernumber);
82
83 %%% iprop!
84 A (result, cell) is the cell containing binary supra-adjacency matrix of
    the binary undirected multiplex with fixed thresholds (BUT).
85
86 %%% icalculate!
87 A_WU = calculateValue@OrdMxWU(g, prop); ①
88
89 thresholds = g.get('THRESHOLDS'); ②
90 L = length(A_WU); % number of layers
91 A = cell(length(thresholds)*L);
92
93 if L > 0 && ~isempty(cell2mat(A_WU))
94     A(:, :) = {zeros(length(A_WU{1, 1}))};
95     for i = 1:length(thresholds) ③
96         threshold = thresholds(i);
97         layer = 1;
98         for j = (i - 1) * L + 1:i * L ④
99             for k = (i - 1) * L + 1:i * L
100                 if j == k ⑤
101                     A{j, j} = dediagonalize(binarize(A_WU{layer, layer}, 'threshold'
                        , threshold));
102                     elseif (j-k)==1 || (k-j)==1 ⑥
103                         A{j, k} = {eye(length(A{1, 1}))};
104                     else ⑦
105                         A{j, k} = {zeros(length(A{1, 1}))};
106                     end
107                 end
108                 layer = layer + 1;
109             end
110         end
111     end
112
113 value = A;
114
115 %%% igui! ⑧
116 pr = PanelPropCell('EL', g, 'PROP', OrdMxBUT.A, ...
117     'TABLE-HEIGHT', s(40), ...
118     'XYSLIDERLOCK', true, ...
119     'XSLIDERSHOW', false, ...
120     'YSLIDERSHOW', true, ...
121     'YSLIDERLABELS', g.getCallback('ALAYERLABELS'), ...
122     'YSLIDERWIDTH', s(5), ...
123     'ROWNAME', g.getCallback('ANODELABELS'), ...
124     'COLUMNNAME', g.getCallback('ANODELABELS'), ...
125     varargin{:});
126
127 %%% iprop!
128 PARTITIONS (result, rvector) returns the number of layers in the
    partitions of the graph.
129 %%% icalculate!
130 l = g.get('LAYERNUMBER');
131 thresholds = g.get('THRESHOLDS');
132 value = ones(1, length(thresholds)) * l / length(thresholds);

```

① Calculates the graph OrdMxWU calling the parent OrdMxWU.

② Same as in ← Code 8 ②-④.

③ For each threshold we construct an ordinal multiplex binary undirected graph

④ We need to loop over the layers of A\_Wu for each threshold

⑤ In the diagonal of the supra-adjacency matrix we have the layers that are constructed by binarizing A\_Wu according to the present threshold

⑥ Consecutive layers are connected

⑦ Non-consecutive layers are not connected

⑧ Same as in ← Code 8

```

133
134   %%% iprop!
135   ALAYERLABELS (query, stringlist) returns the layer labels to be used by
      the slider.
136   %%% icalculate!
137   alayerlabels = g.get('LAYERLABELS');
138   if ~isa(g.get('A'), 'NoValue') && length(alayerlabels) ~= g.get('
      LAYERNUMBER') % ensures that it's not unnecessarily calculated
139   thresholds = cellfun(@num2str, num2cell(g.get('THRESHOLDS')), '
      uniformoutput', false);
140
141   if length(alayerlabels) == length(g.get('B'))
142       blayerlabels = alayerlabels;
143   else % includes isempty(layerlabels)
144       blayerlabels = cellfun(@num2str, num2cell([1:1:length(g.get('B'))]), '
      uniformoutput', false);
145   end
146
147   alayerlabels = {};
148   for i = 1:1:length(thresholds)
149       for j = 1:1:length(blayerlabels)
150           alayerlabels = [alayerlabels, [blayerlabels{j} '|' thresholds{i}]];
151       end
152   end
153 end
154 value = alayerlabels;
155
156 %%% iprop!
157 COMPATIBLE_MEASURES (constant, classlist) is the list of compatible
      measures.
158 %%% idefault!
159 getCompatibleMeasures('OrdMxBUT')
160

```

---

**Code 12: OrdMxBUT element tests.** The tests section from the  
element generator `_OrdMxBUT.gen.m`. ← [Code 3](#)

---

```

1   %% itests!
2
3   %%% iexcluded_props!
4   [OrdMxBUT.PFGA OrdMxBUT.PFGH]
5
6   %%% itest!
7   %%% iname!
8   Constructor - Full ①
9   %%% iprobability!
10  .01
11  %%% icode!
12  B1 = [
13  0 .1 .2 .3 .4
14  .1 0 .1 .2 .3
15  .2 .1 0 .1 .2
16  .3 .2 .1 0 .1
17  .4 .3 .2 .1 0
18  ];
19  B = {B1, B1, B1};
20  thresholds = [0 .1 .2 .3 .4];
21  g = OrdMxBUT('B', B, 'THRESHOLDS', thresholds);
22
23  g.get('A_CHECK')

```

① same as in ← [Code 9](#).

```

24
25 A = g.get('A');
26 for i = 1:length(thresholds)
27     threshold = thresholds(i);
28     for j = (i - 1) * length(B) + 1:i * length(B)
29         for k = (i - 1) * length(B) + 1:i * length(B)
30             if j == k
31                 assert(isequal(A{j, j}, binarize(B1, 'threshold', threshold)), ...
32                     [BRAPH2.STR ':OrdMxBUT:' BRAPH2.FAIL_TEST], ...
33                     'OrdMxBUT is not constructing well.')
34             elseif (j-k)==1 || (k-j)==1
35                 assert(isequal(A{j, k}, eye(length(B1))), ...
36                     [BRAPH2.STR ':OrdMxBUT:' BRAPH2.FAIL_TEST], ...
37                     'OrdMxBUT is not constructing well.')
38             else
39                 assert(isequal(A{j, k}, zeros(length(B1))), ...
40                     [BRAPH2.STR ':OrdMxBUT:' BRAPH2.FAIL_TEST], ...
41                     'OrdMxBUT is not constructing well.')
42             end
43         end
44     end
45 end
46

```

---