Implement a New Graph

The BRAPH 2 Developers

August 14, 2024

This is the developer tutorial for implementing a new graph. In this tutorial, you will learn 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, you 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

Unilayer binary directed graph (`GraphBD`)

You 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 can be either 0 (absence of connection) or 1 (existence of connection).

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

```
1  %% iheader!
2  GraphBD < Graph (g, binary directed graph) is a binary directed graph.  (1)
3
4  %%% idescription!  (2)
5  In a binary directed (BD) graph, the edges are directed and they can be
       either 0 (absence of connection) or 1 (existence of connection).
6
7  %%% ibuild!  (3)
8  1
```

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

(2) provides a description of this graph.

(3) defines the build number of the graph element.

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

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

```
28  %%%% idefault!
29  'GraphBD notes'
30
31  %%% iprop!  ①
32  GRAPH_TYPE (constant, scalar) returns the graph type __Graph.GRAPH__.
33  %%%% idefault!
34  Graph.GRAPH
35
36  %%% iprop!  ②
37  CONNECTIVITY_TYPE (query, smatrix) returns the connectivity type __Graph.
        BINARY__.
38  %%%% idefault!
39  value = Graph.BINARY;
40
41  %%% iprop!  ③
42  DIRECTIONALITY_TYPE  (query, smatrix) returns the directionality type
        __Graph.DIRECTED__.
43  %%%% idefault!
44  value = Graph.DIRECTED;
45
46  %%% iprop!  ④
47  SELFCONNECTIVITY_TYPE (query, smatrix) returns the self-connectivity type
        __Graph.NONSELFCONNECTED__.
48  %%%% idefault!
49  value = Graph.NONSELFCONNECTED;
50
51  %%% iprop!  ⑤
52  NEGATIVITY_TYPE (query, smatrix) returns the negativity type __Graph.
        NONNEGATIVE__.
53  %%%% idefault!
54  value = Graph.NONNEGATIVE;
55
56  %%% iprop!  ⑥
57  A (result, cell) is the binary adjacency matrix of the binary directed graph
        .
58  %%%% icalculate!
59  B = g.get('B');  ⑦
60
61  B = dediagonalize(B);  ⑧
62  B = semipositivize(B, 'SemipositivizeRule', g.get('SEMIPOSITIVIZE_RULE'));
        ⑨
63  B = binarize(B);  ⑩
64
65  A = {B};  ⑪
66  if g.get('RANDOMIZE')  ⑫
67    random_A = g.get('RANDOMIZATION', A);
68    A = {random_A};
69  end
70  value = A;  ⑬
71  %%%% igui!  ⑭
72  pr = PanelPropCell('EL', g, 'PROP', GraphBD.A, ...
73    'TABLE_HEIGHT', s(40), ...
74    'XSLIDERSHOW', false, ...
75    'YSLIDERSHOW', false, ...
76    'ROWNAME' , g.getCallback('ANODELABELS'), ...
77    'COLUMNNAME', g.getCallback('ANODELABELS'));
```

① defines the *graph type*: Graph.GRAPH (single layer), Graph.MULTIGRAPH (multiple unconnected layers), Graph.MULTILAYER (multiple layers), Graph.ORDERED_MULTILAYER (multiple 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 icalculate!.

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

⑧, ⑨, and ⑩ condition the adjaciency 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 calculated.

⑫ randomizes adjacency matrix when 'RANDOMIZE' is true by calling the function of the graph named RANDOMIZATION

⑬ returns the calculated graph A assigning it to the output variable value.

⑭ employs the property panel PanelPropCell to be employed to visualize A, setting also its properties.

```
78
79  %%% iprop! (15)
80  COMPATIBLE_MEASURES (constant, classlist) is the list of compatible measures
        .
81  %%%% idefault!
82  getCompatibleMeasures('GraphBD')
```

(15) determines the list of compatible figures.

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

```
1   %% iprops!
2
3   %%% iprop! (1)
4   B (data, smatrix) is the input graph adjacency matrix.
5   %%%% igui! (2)
6   pr = PanelPropMatrix('EL', g, 'PROP', GraphBD.B, ...
7     'TABLE_HEIGHT' , s(40), ...
8     'ROWNAME' , g.getCallback('ANODELABELS'), ...
9     'COLUMNNAME', g.getCallback('ANODELABELS'), ...
10    varargin{:});
11
12  %%% iprop! (3)
13  SEMIPOSITIVIZE_RULE (parameter, option) determines how to remove the
        negative edges.
14  %%%% isettings!
15  {'zero', 'absolute'}
16
17  %%% iprop! (4)
18  ATTEMPTSPEREDGE (parameter, scalar) is the attempts to rewire each edge.
19  %%%% idefault!
20  5
21
22  %%% iprop! (5)
23  RANDOMIZATION (query, cell) randomizes matrix contained in the cell.
24  %%%% icalculate!
25  rng(g.get('RANDOM_SEED'), 'twister')
26
27  if isempty(varargin) (6)
28    value = {};
29    return
30  end
31
32  A = cell2mat(varargin{1});
33  attempts_per_edge = g.get('ATTEMPTSPEREDGE');
34  % remove self connections
35  A(1:length(A)+1:numel(A)) = 0;
36  [I_edges, J_edges] = find(A);  (7)
37  E = length(I_edges);  (8)
38
39  if E == 0  (9)
40    value = A;
41    swaps = 0;
42    return
43  end
44
45  if E == 1  (10)
46    r_ab = A(I_edges(1), J_edges(1));
```

(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).

(4) defines the number of attempts that will be used for each edge when calling RANDOMIZATION.

(5) randomizes the adjacency matrix contained in cell.

(6) returns empty cell is the input is an empty cell.

(7) finds number of edges in the matrix (different from zero).

(8) returns number of edges in the matrix (different from zero).

(9) returns same input matrix if it is all zeros.

(10) randomizes the edge when there is only one edge in the input matrix.

```
47    A(I_edges(1), J_edges(1)) = 0;
48    selected_nodes = randperm(size(A, 1), 2);
49    A(selected_nodes(1), selected_nodes(2)) = r_ab;
50    value = A;
51    swaps = 1;
52    return
53  end
54
55  random_A = A;
56  swaps = 0; % number of successful edge swaps
57  for attempt = 1:1:attempts_per_edge*E  (11)
58
59    selected_edges = randperm(E,2);  (12)
60    node_start_1 = I_edges(selected_edges(1));
61    node_end_1 = J_edges(selected_edges(1));
62    node_start_2 = I_edges(selected_edges(2));
63    node_end_2 = J_edges(selected_edges(2));
64
65    r_1 = random_A(node_start_1, node_end_1);  (13)
66    r_2 = random_A(node_start_2, node_end_2);
67
68    if ~random_A(node_start_1, node_end_2) && ...
69    ~random_A(node_start_2, node_end_1) && ...
70    node_start_1~=node_start_2 && ...
71    node_end_1~=node_end_2 && ...
72    node_start_1~=node_end_2 && ...
73    node_start_2~=node_end_1
74
75      % erase old edges  (14)
76      random_A(node_start_1, node_end_1) = 0;
77      random_A(node_start_2, node_end_2) = 0;
78
79      % write new edges  (15)
80      random_A(node_start_1, node_end_2) = r_1;
81      random_A(node_start_2, node_end_1) = r_2;
82
83      % update edge list
84      J_edges(selected_edges(1)) = node_end_2;
85      J_edges(selected_edges(2)) = node_end_1;
86
87      swaps = swaps+1;
88    end
89  end
90  value = random_A;
```

(11) randomizes edges in the matrix when more than one edge (non-zero) were found in the input matrix.

(12) takes two random edges.

(13) saves the values of the selected random edges (this is important when the property RANDOMIZATION is used by weighted graphs).

(14) deletes edges in the old positions.

(15) sets values of edges in the new random positions.

Code 4: **GraphBD element tests.** The tests section in 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 defined rules (one test per rule).

```
1  %% !test!
2
3  %%% !excluded_props!  (1)
4  [GraphBD.PFGA GraphBD.PFGH]
5
```

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

```
6   %%% ¡test!
7   %%%% ¡name!
8   Constructor - Empty   (2)
9   %%%% ¡probability!   (3)
10  .01
11  %%%% ¡code!
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  %%% ¡test!
23  %%%% ¡name!
24  Constructor - Full   (9)
25  %%%% ¡probability!
26  .01
27  %%%% ¡code!
28  B = randn(randi(10));   (10)
29  g = GraphBD('B', B);
30
31  g.get('A_CHECK')
32
33  A = {binarize(semipositivize(dediagonalize(B)))};
34  assert(isequal(g.get('A'), A), ...
35    [BRAPH2.STR ':GraphBD:' BRAPH2.FAIL_TEST], ...
36    'GraphBD is not constructing well.')
37
38  %%% ¡test!
39  %%%% ¡name!
40  Semipositivize Rules   (11)
41  %%%% ¡probability!
42  .01   (3)
43  %%%% ¡code!
44  B = [   (12)
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
50    ];
51
52  g0 = GraphBD('B', B);   (13)
53  A0 = {[   (14)
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    ]};
60  assert(isequal(g0.get('A'), A0), ...
61    [BRAPH2.STR ':GraphBD:' BRAPH2.FAIL_TEST], ...
```

(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 applying the corresponding functions.

(8) tests that the value of generated graph calculated by applying the 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.

(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.

```
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    ]};
72  assert(isequal(g_zero.get('A'), A_zero), ...
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    ]};
84  assert(isequal(g_absolute.get('A'), A_absolute), ...
85    [BRAPH2.STR ':GraphBD:' BRAPH2.FAIL_TEST], ...
86    'GraphBD is not constructing well.')
87
88  %%% ¡test!
89  %%%% ¡name!
90  Randomize Rules  17
91  %%%% ¡probability!
92  .01
93  %%%% ¡code!
94  B = randn(10);
95
96  g = GraphBD('B', B);
97  g.set('RANDOMIZE', true);
98  g.set('ATTEMPTSPEREDGE', 4);
99
100 A = g.get('A');
101
102 assert(isequal(size(A{1}), size(B)), ...  18
103 [BRAPH2.STR ':GraphBD:' BRAPH2.FAIL_TEST], ...
104 'GraphBD Randomize is not functioning well.')
105
106 g2 = GraphBD('B', B);
107 g2.set('RANDOMIZE', false);
108 g2.set('ATTEMPTSPEREDGE', 4);
109 A2 = g2.get('A');
110 random_A = g2.get('RANDOMIZATION', A2);
111
112 if all(A2{1}==0, "all")  19
113   assert(isequal(A2{1}, random_A), ...
114   [BRAPH2.STR ':GraphBD:' BRAPH2.FAIL_TEST], ...
115   'GraphBD Randomize is not functioning well.')
116 elseif isequal((length(A2{1}).^2)- length(A2{1}), sum(A2{1}==1, "all"))  20
117   assert(isequal(A2{1}, random_A), ...
118   [BRAPH2.STR ':GraphBD:' BRAPH2.FAIL_TEST], ...
119   'GraphBD Randomize is not functioning well.')
```

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.

17 tests that RANZOMIZATION works properly.

18 tests that RANZOMIZATION returns a matrix with same size.

19 tests that RANZOMIZATION returns a matrix of zeros when input matrix is all zeros.

20 tests that RANZOMIZATION returns a matrix of ones when input matrix is all ones (except diagonal).

```
120  else  ⃝21
121    assert(~isequal(A2{1}, random_A), ...
122    [BRAPH2.STR ':GraphBD:' BRAPH2.FAIL_TEST], ...
123    'GraphBD Randomize is not functioning well.')
124  end
125
126  assert(isequal(numel(find(A2{1})), numel(find(random_A))), ...  ⃝22
127  [BRAPH2.STR ':GraphBD:' BRAPH2.FAIL_TEST], ...
128  'GraphBD Randomize is not functioning well.')
129
130  deg_A = sum(A2{1}, 2);
131  deg_B = sum(random_A, 2);
132  [h, p, ks2stat] = kstest2(deg_A, deg_B);
133
134  assert(isequal(0, h), ... %  ⃝23
135  [BRAPH2.STR ':GraphBD:' BRAPH2.FAIL_TEST], ...
136  'GraphBD Randomize is not functioning well.')
```

⃝21 tests that new random matrix is different from original one.

⃝22 tests that new random matrix has the same number of nodes as the original one

⃝23 tests that new random matrix has the same degree distribution as the original one

Implementation of multilayer graphs

Weighted directed multilayer graph (`MultilayerWD`)

You 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 in `_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
        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.
6
7  %%% ibuild!
8  1
```

Code 6: MultilayerWD element prop update. The `props_update` section of the generator code in `_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 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.'
12
13  %%% iprop!
14  TEMPLATE (parameter, item) is the template of the multilayer weighted
        directed graph.
15
16  %%% iprop!
17  ID (data, string) is a few-letter code of the multilayer weighted directed
        graph.
```

```matlab
18 %%%% idefault!
19 'MultilayerWD ID'
20
21 %%% iprop!
22 LABEL (metadata, string) is an extended label of the multilayer weighted
        directed graph.
23 %%%% idefault!
24 'MultilayerWD label'
25
26 %%% iprop!
27 NOTES (metadata, string) are some specific notes about the multilayer
        weighted directed graph.
28 %%%% idefault!
29 'MultilayerWD notes'
30
31 %%% iprop!
32 GRAPH_TYPE (constant, scalar) returns the graph type __Graph.MULTILAYER__.
33 %%%% idefault!
34 Graph.MULTILAYER
35
36 %%% iprop!
37 CONNECTIVITY_TYPE (query, smatrix) returns the connectivity type __Graph.
        WEIGHTED__ * ones(layernumber).
38 %%%% icalculate!
39 if isempty(varargin)
40 layernumber = 1;
41 else
42 layernumber = varargin{1};
43 end
44 value = Graph.WEIGHTED * ones(layernumber);
45
46 %%% iprop!
47 DIRECTIONALITY_TYPE (query, smatrix) returns the directionality type __Graph
        .DIRECTED__ * ones(layernumber).
48 %%%% icalculate!
49 if isempty(varargin)
50    layernumber = 1;
51 else
52    layernumber = varargin{1};
53 end
54 value = Graph.DIRECTED * ones(layernumber);
55
56 %%% iprop!
57 SELFCONNECTIVITY_TYPE (query, smatrix) returns the self-connectivity type
        __Graph.NONSELFCONNECTED__ on the diagonal and __Graph.SELFCONNECTED__
        off diagonal.
58 %%%% icalculate!
59 if isempty(varargin)
60    layernumber = 1;
61 else
62    layernumber = varargin{1};
63 end
64 value = Graph.SELFCONNECTED * ones(layernumber);
65 value(1:layernumber+1:end) = Graph.NONSELFCONNECTED;
66
67 %%% iprop!
68 NEGATIVITY_TYPE (query, smatrix) returns the negativity type __Graph.
        NONNEGATIVE__ * ones(layernumber).
69 %%%% icalculate!
70 if isempty(varargin)
71    layernumber = 1;
```

```matlab
72   else
73     layernumber = varargin{1};
74   end
75   value = Graph.NONNEGATIVE * ones(layernumber);
76
77   %%% iprop!
78   A (result, cell) is the cell containing the within-layer weighted adjacency
79   matrices of the multilayer weighted directed graph and the connections
80   between layers.
81
82   %%%% icalculate!
83   B = g.get('B');
84   L = length(B);
85   A = cell(L, L);
86   for i = 1:1:L         ①
87     M = dediagonalize(B{i,i});
88     M = semipositivize(M, 'SemipositivizeRule', g.get('SEMIPOSITIVIZE_RULE'));
89     M = standardize(M, 'StandardizeRule', g.get('STANDARDIZE_RULE'));
90     A(i, i) = {M};
91     if ~isempty(A{i, i})
92       for j = i+1:1:L
93         M = semipositivize(B{i,j}, 'SemipositivizeRule',  g.get('
           SEMIPOSITIVIZE_RULE'));
94         M = standardize(M, 'StandardizeRule',   g.get('STANDARDIZE_RULE'));
95         A(i, j) = {M};
96         M = semipositivize(B{j,i}, 'SemipositivizeRule',  g.get('
           SEMIPOSITIVIZE_RULE'));
97         M = standardize(M, 'StandardizeRule',   g.get('STANDARDIZE_RULE'));
98         A(j, i) = {M};
99       end
100    end
101  end
102  if g.get('RANDOMIZE')
103    A = g.get('RANDOMIZATION', A);
104  end
105  value = A;
106  %%%% igui!
107  pr = PanelPropCell('EL', g, 'PROP', MultilayerWD.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  value = ones(1, g.get('LAYERNUMBER'));
122
123  %%% iprop!   ②
124  ALAYERLABELS (query, stringlist) returns the layer labels to be used by the
         slider.
125  %%%% icalculate!
126  alayerlabels = g.get('LAYERLABELS');   ③
127  if isempty(alayerlabels) && ~isa(g.getr('A'), 'NoValue') % ensures that it's
```

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 the graph adjacency matrix A used in its visualization. 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 nodal labels for each layer)).

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

```
           not unecessarily calculated
128    alayerlabels = cellfun(@num2str, num2cell([1:1:g.get('LAYERNUMBER')]), '
           uniformoutput', false);  (4)
129 end
130 value = alayerlabels;
131
132 %%% iprop!
133 COMPATIBLE_MEASURES (constant, classlist) is the list of compatible measures
        .
134 %%%% idefault!
135 getCompatibleMeasures('MultilayerWD')
```

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

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

```
1  %% iprops!
2
3  %%% iprop!
4  B (data, cell) is the input cell containing the multilayer adjacency
        matrices.
5  %%%% idefault!
6  {[] []; [] []}
7  %%%% igui!  (1)
8  pr = PanelPropCell('EL', g, 'PROP', MultilayerWD.B, ...
9    'TABLE_HEIGHT', s(40), ...
10   'XSLIDERSHOW', true, ...
11   'XSLIDERLABELS', g.get('LAYERLABELS'), ...
12   'XSLIDERHEIGHT', s(3.5), ...
13   'YSLIDERSHOW', false, ...
14   'ROWNAME', g.getCallback('ANODELABELS'), ...
15   'COLUMNNAME', g.getCallback('ANODELABELS'), ...
16   varargin{:});
17
18
19 %%% iprop!
20 SEMIPOSITIVIZE_RULE (parameter, option) determines how to remove the
        negative edges.
21 %%%% isettings!
22 {'zero', 'absolute'}
23
24 %%% iprop!  (2)
25 STANDARDIZE_RULE (parameter, option) determines how to normalize the weights
        between 0 and 1.
26 %%%% isettings!
27 {'threshold' 'range'}
28
29 %%% iprop!
30 ATTEMPTSPEREDGE (parameter, scalar) is the attempts to rewire each edge.
31 %%%% idefault!
32 5
33
34 %%% iprop!
35 NUMBEROFWEIGHTS (parameter, scalar) specifies the number of weights sorted
        at the same time.  (3)
36 %%%% idefault!
37 10
38
39 %%% iprop!
```

(1) Same as in note (2) of Code 3.

(2) Same as in note (3) of Code 3.

(3) defines the number of weights that will be sorted at the same time when using RANDOMIZATION.

```
40  RANDOMIZATION (query, cell) is the attempts to rewire each edge.
41  %%%% !calculate!
42  rng(g.get('RANDOM_SEED'), 'twister')
43
44  if isempty(varargin)
45    value = {};
46    return
47  end
48
49  A = varargin{1};
50  attempts_per_edge = g.get('ATTEMPTSPEREDGE');
51
52  for i = 1:length(A) ④
53    tmp_a = A{i,i};
54
55    tmp_g = GraphWD(); ⑤
56    tmp_g.set('ATTEMPTSPEREDGE', g.get('ATTEMPTSPEREDGE'));
57    tmp_g.set('NUMBEROFWEIGHTS', g.get('NUMBEROFWEIGHTS'));
58    random_A = tmp_g.get('RANDOMIZATION', {tmp_a});
59    A{i, i} = random_A;
60  end
61  value = A;
```

④ iterates over each layer in `MultilayerWD` to randomize it.

⑤ initizalizes empty `GraphWD` to get `RANDOMIZATION` property from it.

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

```
1   %% !tests!
2
3   %%% !excluded_props!
4   [MultilayerWD.PFGA MultilayerWD.PFGH]
5
6   %%% !test!
7   %%%% !name!
8   Constructor - Full
9   %%%% !probability!
10  .01
11  %%%% !code!
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));
```

```matlab
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.')
50
51  %%% itest!
52  %%%% iname!
53  Randomize Rules
54  %%%% iprobability!
55  .01
56  %%%% icode!
57  B1 = rand(randi(10));
58  B2 = rand(randi(10));
59  B3 = rand(randi(10));
60  B12 = rand(size(B1, 1),size(B2, 2));
61  B13 = rand(size(B1, 1),size(B3, 2));
62  B23 = rand(size(B2, 1),size(B3, 2));
63  B21 = rand(size(B2, 1),size(B1, 2));
64  B31 = rand(size(B3, 1),size(B1, 2));
65  B32 = rand(size(B3, 1),size(B2, 2));
66  B = {
67    B1                        B12                       B13
68    B21                       B2                        B23
69    B31                       B32                       B3
70  };
71  g = MultilayerWD('B', B);
72  g.set('RANDOMIZE', true);
73  g.set('ATTEMPTSPEREDGE', 4);
74  g.get('A_CHECK')
75
76  A = g.get('A')
77
78  assert(isequal(size(A{1}), size(B{1})), ...
79  [BRAPH2.STR 'MultilayerWD:' BRAPH2.FAIL_TEST], ... 'MultilayerWD Randomize
         is not functioning well.')
80
81  g2 = MultilayerWD('B', B);
82  g2.set('RANDOMIZE', true);
83  g2.set('ATTEMPTSPEREDGE', 4);
84  g2.get('A_CHECK')
85  A2 = g2.get('A');
86  random_A = g2.get('RANDOMIZATION', A2);
87
88  for i = 1:length(A2)  ①
89    if all(A2{i, i}==0, "all") %if all edges are zero, the new random matrix
         is all zeros
90      assert(isequal(A2{i, i}, random_A{i, i}), ...
91      [BRAPH2.STR ':MultilayerWD:' BRAPH2.FAIL_TEST], ...
92      'MultilayerWD Randomize is not functioning well.')
93    elseif isequal((length(A2{i, i}).^2)- length(A2{i, i}), sum(A2{i, i}==1, "
```

① tests RANDOMIZATION as in Code 4 for each layer in A2.

```
         all")) %if all nodes (except diagonal) are one, the random matrix is
          the same as original
 94       assert(isequal(A2{i, i}, random_A{i, i}), ...
 95       [BRAPH2.STR ':MultilayerWD:' BRAPH2.FAIL_TEST], ...
 96       'MultilayerWD Randomize is not functioning well.')
 97    else
 98       assert(~isequal(A2{i, i}, random_A{i, i}), ...
 99       [BRAPH2.STR ':MultilayerWD:' BRAPH2.FAIL_TEST], ...
100       'MultilayerWD Randomize is not functioning well.')
101    end
102 assert(isequal(numel(find(A2{i, i})), numel(find(random_A{i, i}))), ... %
        check same number of nodes
103 [BRAPH2.STR ':MultilayerWD:' BRAPH2.FAIL_TEST], ...
104 'MultilayerWD Randomize is not functioning well.')
105 end
```

Binary undirected multilayer graph with fixed thresholds (`MultiplexBUT`)

Now you will implement the `MultiplexBUT` graph building on the previous codes `GraphBD` and `MultilayerWD`. Again, the differences will be highlighted. A multiplex graph is a type of multilayer graph where interlayer edges are allowed only 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 the generator code in `_MultiplexBUT.gen.m` provides the general information about the `MultiplexBUT` element. ← Code 1

```
1  %% iheader!
2  MultiplexBUT < MultiplexWU (g, binary undirected multiplex with fixed
       thresholds) is a binary undirected multiplex with fixed thresholds.  (1)
3
4  %%% idescription!
5  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.
6
7  %%% ibuild!
8  1
```

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

Code 10: **MultiplexBUT element prop update.** The `props_update` section of the generator code in `_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
       multiplex with fixed thresholds.
10 %%%% idefault!
11 '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.'
12
13 %%% iprop!
14 TEMPLATE (parameter, item) is the template of the binary undirected
       multiplex with fixed thresholds.
15
16 %%% iprop!
17 ID (data, string) is a few-letter code of the binary undirected multiplex
       with fixed thresholds.
18 %%%% idefault!
```

```matlab
19  'MultiplexBUT ID'
20
21  %%% iprop!
22  LABEL (metadata, string) is an extended label of the binary undirected
        multiplex with fixed thresholds.
23  %%%% idefault!
24  'MultiplexBUT label'
25
26  %%% iprop!
27  NOTES (metadata, string) are some specific notes about the binary undirected
        multiplex with fixed thresholds.
28  %%%% idefault!
29  'MultiplexBUT notes'
30
31  %%% iprop!
32  GRAPH_TYPE (constant, scalar) returns the graph type __Graph.MULTIPLEX__.
33  %%%% idefault!
34  Graph.MULTIPLEX
35
36  %%% iprop!
37  CONNECTIVITY_TYPE (query, smatrix) returns the connectivity type __Graph.
        BINARY__  * ones(layernumber).
38  %%%% icalculate!
39  if isempty(varargin)
40    layernumber = 1;
41  else
42    layernumber = varargin{1};
43  end
44  value = Graph.BINARY * ones(layernumber);
45
46  %%% iprop!
47  DIRECTIONALITY_TYPE (query, smatrix) returns the directionality type __Graph
        .UNDIRECTED__ * ones(layernumber).
48  %%%% icalculate!
49  if isempty(varargin)
50    layernumber = 1;
51  else
52    layernumber = varargin{1};
53  end
54  value = Graph.UNDIRECTED * ones(layernumber);
55
56  %%% iprop!
57  SELFCONNECTIVITY_TYPE (query, smatrix) returns the self-connectivity type
        __Graph.NONSELFCONNECTED__ on the diagonal and __Graph.SELFCONNECTED__
        off diagonal.
58  %%%% icalculate!
59  if isempty(varargin)
60    layernumber = 1;
61  else
62    layernumber = varargin{1};
63  end
64  value = Graph.SELFCONNECTED * ones(layernumber);
65  value(1:layernumber+1:end) = Graph.NONSELFCONNECTED;
66
67  %%% iprop!
68  NEGATIVITY_TYPE (query, smatrix) returns the negativity type __Graph.
        NONNEGATIVE__ * ones(layernumber).
69  %%%% icalculate!
70  if isempty(varargin)
71    layernumber = 1;
72  else
```

```
73    layernumber = varargin{1};
74  end
75  value = Graph.NONNEGATIVE * ones(layernumber);
76
77  %%% !prop!
78  A (result, cell) is the cell containing multiplex binary adjacency matrices
        of the binary undirected multiplex.
79
80  %%%% !calculate!
81  A_WU = calculateValue@MultiplexWU(g, prop);  ①
82
83  thresholds = g.get('THRESHOLDS');  ②
84  L = length(A_WU); % number of layers  ③
85  A = cell(length(thresholds) * L);  ④
86
87  if L > 0 && ~isempty(cell2mat(A_WU))
88    A(:, :) = {eye(length(A_WU{1, 1}))};
89    for i = 1:1:length(thresholds)  ⑤
90      threshold = thresholds(i);
91      layer = 1;
92      for j = (i - 1) * L + 1:1:i * L  ⑥
93        A{j, j} = dediagonalize(binarize(A_WU{layer, layer}, 'threshold',
          threshold));  ⑦
94        layer = layer + 1;
95      end
96    end
97  end
98  if g.get('RANDOMIZE')
99    A = g.get('RANDOMIZATION', A);
100 end
101 value = A;
102
103 %%%% !gui!  ⑧
104 pr = PanelPropCell('EL', g, 'PROP', MultiplexBUT.A, ...
105   'TABLE_HEIGHT', s(40), ...
106   'XYSLIDERLOCK', true, ...
107   'XSLIDERSHOW', false, ...
108   'YSLIDERSHOW', true, ...
109   'YSLIDERLABELS', g.getCallback('ALAYERLABELS'), ...
110   'YSLIDERWIDTH', s(5), ...
111   'ROWNAME', g.getCallback('ANODELABELS'), ...
112   'COLUMNNAME', g.getCallback('ANODELABELS'), ...
113   varargin{:});
114
115 %%% !prop!
116 PARTITIONS (result, rvector) returns the number of layers in the partitions
        of the graph.
117 %%%% !calculate!
118 l = g.get('LAYERNUMBER');
119 thresholds = g.get('THRESHOLDS');
120 value = ones(1, length(thresholds)) * l / length(thresholds);
121
122 %%% !prop!
123 ALAYERLABELS (query, stringlist) returns the layer labels to be used by the
        slider.
124 %%%% !calculate!
125 alayerlabels = g.get('LAYERLABELS');
126 if ~isa(g.getr('A'), 'NoValue') && length(alayerlabels) ~= g.get('
        LAYERNUMBER') % ensures that it's not unecessarily calculated
```

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.

```
127    thresholds = cellfun(@num2str, num2cell(g.get('THRESHOLDS')), '
          uniformoutput', false);
128
129    if length(alayerlabels) == length(g.get('B'))
130      blayerlabels = alayerlabels;
131    else % includes isempty(layerlabels)
132      blayerlabels = cellfun(@num2str, num2cell([1:1:length(g.get('B'))]), '
          uniformoutput', false);
133    end
134
135    alayerlabels = {};
136    for i = 1:1:length(thresholds) ⑨
137      for j = 1:1:length(blayerlabels)
138        alayerlabels = [alayerlabels, [blayerlabels{j} '|' thresholds{i}]];
139      end
140    end
141  end
142  value = alayerlabels;
143
144  %%% iprop!
145  COMPATIBLE_MEASURES (constant, classlist) is the list of compatible measures
         .
146  %%%% idefault!
147  getCompatibleMeasures('MultiplexBUT')
148
149  %%% iprop!
150  ATTEMPTSPEREDGE (parameter, scalar) is the attempts to rewire each edge.
151  %%%% idefault!
152  5
153
154  %%% iprop!
155  RANDOMIZATION (query, cell) is the attempts to rewire each edge.
156  %%%% icalculate!
157  rng(g.get('RANDOM_SEED'), 'twister')
158
159  if isempty(varargin)
160    value = {};
161    return
162  end
163
164  A = varargin{1};
165  attempts_per_edge = g.get('ATTEMPTSPEREDGE');
166
167  for i = 1:length(A)
168    tmp_a = A{i,i};
169
170    random_g = GraphBU();   ⑩
171    random_g.set('ATTEMPTSPEREDGE', g.get('ATTEMPTSPEREDGE'));
172    random_A = random_g.get('RANDOMIZATION', {tmp_a});
173    A{i, i} = random_A;
174  end
175  value = A;
```

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

⑩ Same as in Code 6 but using GraphBU.

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

```
1  %% iprops!
2
```

```
3  %%% iprop!
4  THRESHOLDS (parameter, rvector) is the vector of thresholds.
5  %%%% igui! ①
6  pr = PanelPropRVectorSmart('EL', g, 'PROP', MultiplexBUT.THRESHOLDS, ...
7    'MAX', 1, ...
8    'MIN', -1, ...
9    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

```
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);
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
38  %%% itest!
39  %%%% iname!
40  Randomize Rules
41  %%%% iprobability!
42  .01
43  %%%% icode!
44  B11 = randn(10);
45
46  B12 = rand(size(B11,1),size(B11,2));
47
48  B= {B11 B12 B12;
49    B12 B11 B12;
```

① creates an example of the necessary input adjacency matrices.

② defines the thresholds.

```
50    B12 B12 B11};
51  thresholds = [0 .5 1];
52  g = MultilayerBUT('B', B, 'THRESHOLDS', thresholds);
53
54  g.set('RANDOMIZE', true);
55  g.set('ATTEMPTSPEREDGE', 4);
56  g.get('A_CHECK')
57
58  A = g.get('A');
59
60  assert(isequal(size(A{1}), size(B{1})), ...
61  [BRAPH2.STR ':MultilayerBUT:' BRAPH2.FAIL_TEST], ... 'MultilayerBUT
         Randomize is not functioning well.')
62
63  g2 = MultilayerBUT('B', B, 'THRESHOLDS', thresholds);
64  g2.set('RANDOMIZE', false);
65  g2.set('ATTEMPTSPEREDGE', 4);
66  A2 = g2.get('A');
67  random_A = g2.get('RANDOMIZATION', A2);
68
69  for i = 1:length(A2)
70    if all(A2{i, i}==0, "all") %if all edges are zero, the new random matrix
         is all zeros
71      assert(isequal(A2{i, i}, random_A{i, i}), ...
72      [BRAPH2.STR ':MultilayerBUT:' BRAPH2.FAIL_TEST], ...
73      'MultilayerBUT Randomize is not functioning well.')
74    elseif isequal((length(A2{i, i}).^2)- length(A2{i, i}), sum(A2{i, i}==1, "
         all")) %if all nodes (except diagonal) are one, the random matrix is
         the same as original
75      assert(isequal(A2{i, i}, random_A{i, i}), ...
76      [BRAPH2.STR ':MultilayerBUT:' BRAPH2.FAIL_TEST], ...
77      'MultilayerBUT Randomize is not functioning well.')
78    else
79      assert(~isequal(A2{i, i}, random_A{i, i}), ...
80      [BRAPH2.STR ':MultilayerBUT:' BRAPH2.FAIL_TEST], ...
81      'MultilayerBUT Randomize is not functioning well.')
82    end
83
84  assert(isequal(numel(find(A2{i, i})), numel(find(random_A{i, i}))), ... %
         check same number of nodes
85  [BRAPH2.STR ':MultilayerBUT:' BRAPH2.FAIL_TEST], ...
86  'MultilayerBUT Randomize is not functioning well.')
87
88  assert(issymmetric(random_A{i, i}), ... % check symmetry ③
89  [BRAPH2.STR ':MultilayerBUT:' BRAPH2.FAIL_TEST], ...
90  'MultilayerBUT Randomize is not functioning well.')
91
92  end
```

③ checks symmetry of each layer in the new random graph random_A since they are undirected.

Binary undirected ordinal multiplex graph with fixed thresholds (`OrdMxBUT`)

Finally, you will implement the `OrdMxBUT` graph based on the previous codes `GraphBD`, `MultilayerWD` and `MultiplexBUT`. Again the differences will be highlighted. An ordered multiplex is a type of multiplex graph that consists of a sequence of layers with edges between corresponding nodes in subsequent layers.

Code 13: **OrdMxBUT element header.** The `header` section of the generator code in `_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
        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.
6
7  %%% ibuild!
8  1
```

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

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

```
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), 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.'
12
13  %%% iprop!
14  TEMPLATE (parameter, item) is the template of the binary undirected ordinal
        multiplex with fixed thresholds.
15
16  %%% iprop!
```

```
17  ID (data, string) is a few-letter code of the binary undirected ordinal
        multiplex with fixed thresholds.
18  %%%% idefault!
19  'OrdMxBUT ID'

20
21  %%% iprop!
22  LABEL (metadata, string) is an extended label of the binary undirected
        ordinal multiplex with fixed thresholds.
23  %%%% idefault!
24  'OrdMxBUT label'

25
26  %%% iprop!
27  NOTES (metadata, string) are some specific notes about the binary undirected
        ordinal multiplex with fixed thresholds.
28  %%%% idefault!
29  'OrdMxBUT notes'

30
31  %%% iprop!
32  GRAPH_TYPE (constant, scalar) returns the graph type __Graph.
        ORDERED_MULTIPLEX__.
33  %%%% idefault!
34  Graph.ORDERED_MULTIPLEX

35
36  %%% iprop!
37  CONNECTIVITY_TYPE (query, smatrix) returns the connectivity type __Graph.
        BINARY__  * ones(layernumber).
38  %%%% icalculate!
39  if isempty(varargin)
40    layernumber = 1;
41  else
42    layernumber = varargin{1};
43  end
44  value = Graph.BINARY * ones(layernumber);

45
46  %%% iprop!
47  DIRECTIONALITY_TYPE (query, smatrix) returns the directionality type __Graph
        .UNDIRECTED__ * ones(layernumber).
48  %%%% icalculate!
49  if isempty(varargin)
50    layernumber = 1;
51  else
52    layernumber = varargin{1};
53  end
54  value = Graph.UNDIRECTED * ones(layernumber);

55
56  %%% iprop!
57  SELFCONNECTIVITY_TYPE (query, smatrix) returns the self-connectivity type
        __Graph.NONSELFCONNECTED__ on the diagonal and __Graph.SELFCONNECTED__
        off diagonal.
58  %%%% icalculate!
59  if isempty(varargin)
60    layernumber = 1;
61  else
62    layernumber = varargin{1};
63  end
64  value = Graph.SELFCONNECTED * ones(layernumber);
65  value(1:layernumber+1:end) = Graph.NONSELFCONNECTED;

66
67  %%% iprop!
68  NEGATIVITY_TYPE (query, smatrix) returns the negativity type __Graph.
        NONNEGATIVE__ * ones(layernumber).
```

```matlab
69  %%%% ¡calculate!
70  if isempty(varargin)
71    layernumber = 1;
72  else
73    layernumber = varargin{1};
74  end
75  value = Graph.NONNEGATIVE * ones(layernumber);
76
77  %%% ¡prop!
78  A (result, cell) is the cell containing binary supra-adjacency matrix of the
         binary undirected multiplex with fixed thresholds (BUT).
79
80  %%%% ¡calculate!
81  A_WU = calculateValue@OrdMxWU(g, prop);①
82
83  thresholds = g.get('THRESHOLDS');  ②
84  L = length(A_WU); % number of layers
85  A = cell(length(thresholds)*L);
86
87  if L > 0 && ~isempty(cell2mat(A_WU))
88    A(:, :) = {zeros(length(A_WU{1, 1}))};
89    for i = 1:1:length(thresholds) ③
90      threshold = thresholds(i);
91      layer = 1;
92      for j = (i - 1) * L + 1:1:i * L ④
93        for k = (i - 1) * L + 1:1:i * L
94          if j == k ⑤
95            A{j, j} = dediagonalize(binarize(A_WU{layer, layer}, 'threshold',
         threshold));
96          elseif (j-k)==1 || (k-j)==1 ⑥
97            A(j, k) = {eye(length(A{1, 1}))};
98          else ⑦
99            A(j, k) = {zeros(length(A{1, 1}))};
100          end
101        end
102        layer = layer + 1;
103      end
104    end
105  end
106  if g.get('RANDOMIZE')
107    A = g.get('RANDOMIZATION', A);
108  end
109  value = A;
110
111  %%%% ¡gui!
112  pr = PanelPropCell('EL', g, 'PROP', OrdMxBUT.A, ...
113    'TABLE_HEIGHT', s(40), ...
114    'XYSLIDERLOCK', true, ...
115    'XSLIDERSHOW', false, ...
116    'YSLIDERSHOW', true, ...
117    'YSLIDERLABELS', g.getCallback('ALAYERLABELS'), ...
118    'YSLIDERWIDTH', s(5), ...
119    'ROWNAME', g.getCallback('ANODELABELS'), ...
120    'COLUMNNAME', g.getCallback('ANODELABELS'), ...
121    varargin{:});
122
123  %%% ¡prop!
124  PARTITIONS (result, rvector) returns the number of layers in the partitions
         of the graph.
```

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.

```
125  %%%% ¡calculate!
126  l = g.get('LAYERNUMBER');
127  thresholds = g.get('THRESHOLDS');
128  value = ones(1, length(thresholds)) * l / length(thresholds);
129
130  %%% ¡prop!
131  ALAYERLABELS (query, stringlist) returns the layer labels to be used by the
        slider.
132  %%%% ¡calculate!
133  alayerlabels = g.get('LAYERLABELS');
134  if ~isa(g.getr('A'), 'NoValue') && length(alayerlabels) ~= g.get('
        LAYERNUMBER') % ensures that it's not unecessarily calculated
135    thresholds = cellfun(@num2str, num2cell(g.get('THRESHOLDS')), '
        uniformoutput', false);
136
137    if length(alayerlabels) == length(g.get('B'))
138      blayerlabels = alayerlabels;
139    else % includes isempty(layerlabels)
140      blayerlabels = cellfun(@num2str, num2cell([1:1:length(g.get('B'))]), '
        uniformoutput', false);
141    end
142
143    alayerlabels = {};
144    for i = 1:1:length(thresholds)
145      for j = 1:1:length(blayerlabels)
146        alayerlabels = [alayerlabels, [blayerlabels{j} '|' thresholds{i}]];
147      end
148    end
149  end
150  value = alayerlabels;
151
152  %%% ¡prop!
153  COMPATIBLE_MEASURES (constant, classlist) is the list of compatible measures
        .
154  %%%% ¡default!
155  getCompatibleMeasures('OrdMxBUT')
156
157  %%% ¡prop!
158  ATTEMPTSPEREDGE (parameter, scalar) is the attempts to rewire each edge.
159  %%%% ¡default!
160  5
161
162  %%% ¡prop!
163  RANDOMIZATION (query, cell) is the attempts to rewire each edge.  ⑧
164  %%%% ¡calculate!
165  rng(g.get('RANDOM_SEED'), 'twister')
166
167  if isempty(varargin)
168    value = {};
169    return
170  end
171
172  A = varargin{1};
173  attempts_per_edge = g.get('ATTEMPTSPEREDGE');
174
175  for i = 1:length(A)
176    tmp_a = A{i,i};
177
178    random_g = GraphBU();
179    random_g.set('ATTEMPTSPEREDGE', g.get('ATTEMPTSPEREDGE'));
180    random_A = random_g.get('RANDOMIZATION', {tmp_a});
```

⑧ same as in Code 10.

```
181    A{i, i} = random_A;
182  end
183  value = A;
```

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

```
1  %% iprops!
2
3  %%% iprop!
4  THRESHOLDS (parameter, rvector) is the vector of thresholds.
5  %%%% igui!
6  pr = PanelPropRVectorSmart('EL', g, 'PROP', OrdMxBUT.THRESHOLDS,  ...
7    'MAX', 1, ...
8    'MIN', -1, ...
9    varargin{:});
```

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

```
1  %% itests!
2
3  %%% iexcluded_props!
4  [OrdMxBUT.PFGA OrdMxBUT.PFGH]
5
6  %%% itest!
7  %%%% iname!
8  Constructor - Full
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')
24
25  A = g.get('A');
26  for i = 1:1:length(thresholds)
27    threshold = thresholds(i);
28    for j = (i - 1) * length(B) + 1:1:i * length(B)
29      for k = (i - 1) * length(B) + 1:1: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
```

```matlab
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
47 %%% !test!
48 %%%% !name!
49 Randomize Rules   ①
50 %%%% !probability!
51 .01
52 %%%% !code!
53 B1 = randn(10);
54 B = {B1, B1, B1};
55 thresholds = [0 .1 .2 .3 .4];
56 g = OrdMxBUT('B', B, 'THRESHOLDS', thresholds);
57
58 g.set('RANDOMIZE', true);
59 g.set('ATTEMPTSPEREDGE', 4);
60 g.get('A_CHECK')
61
62 A = g.get('A');
63
64 assert(isequal(size(A{1}), size(B{1})), ...
65 [BRAPH2.STR ':OrdMxBUT:' BRAPH2.FAIL_TEST], ...
66 'OrdMxBUT Randomize is not functioning well.')
67
68 g2 = OrdMxBUT('B', B, 'THRESHOLDS', thresholds);
69 g2.set('RANDOMIZE', false);
70 g2.set('ATTEMPTSPEREDGE', 4);
71 A2 = g2.get('A');
72 random_A = g2.get('RANDOMIZATION', A2);
73
74 for i = 1:length(A2)
75   if all(A2{i, i}==0, "all") %if all edges are zero, the new random matrix
        is all zeros
76     assert(isequal(A2{i, i}, random_A{i, i}), ...
77     [BRAPH2.STR ':OrdMxBUT:' BRAPH2.FAIL_TEST], ...
78     'OrdMxBUT Randomize is not functioning well.')
79     elseif isequal((length(A2{i, i}).^2)- length(A2{i, i}), sum(A2{i, i}==1,
        "all")) %if all nodes (except diagonal) are one, the random matrix is
        the same as original
80     assert(isequal(A2{i, i}, random_A{i, i}), ...
81     [BRAPH2.STR ':OrdMxBUT:' BRAPH2.FAIL_TEST], ...
82     'OrdMxBUT Randomize is not functioning well.')
83   else
84     assert(~isequal(A2{i, i}, random_A{i, i}), ...
85     [BRAPH2.STR ':OrdMxBUT:' BRAPH2.FAIL_TEST], ...
86     'OrdMxBUT Randomize is not functioning well.')
87   end
88
89 assert(isequal(numel(find(A2{i, i})), numel(find(random_A{i, i}))), ... %
        check same number of nodes
90 [BRAPH2.STR ':OrdMxBUT:' BRAPH2.FAIL_TEST], ...
91 'OrdMxBUT Randomize is not functioning well.')
92
93 assert(issymmetric(random_A{i, i}), ... % check symmetry
94 [BRAPH2.STR ':OrdMxBUT:' BRAPH2.FAIL_TEST], ...
95 'OrdMxBUT Randomize is not functioning well.')
```

① same as in Code 12.

```
96
97  end
```