Implement a new Graph

The BRAPH 2 Developers

July 26, 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 the be compiled by `braph2genesis`, using the graphs `GraphBD`, `MultilayerWU`, `Multiplex??`, `OrdMl??` as examples. and `OrdMx??` as examples.

Contents

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`.

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

```
%% ¡header!
GraphBD < Graph (m, 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 The element `GraphBD` is defined as a subclass of `Graph`. The moniker will be `m`.

Code 2: **GraphBD element layout.** The `layout` section of generator code for `_GraphBD.gen.m` creates the general variables needed for the `GraphBD` element.

```
%% ¡layout!
%%% ¡prop!
%%%% ¡id!
GraphBD.ID
%%%% ¡title!
Graph ID

%%% ¡prop!
%%%% ¡id!
GraphBD.LABEL
%%%% ¡title!
Graph NAME

%%% ¡prop!
%%%% ¡id!
GraphBD.NODELABELS
%%%% ¡title!
NODES labels

%%% ¡prop!
%%%% ¡id!
GraphBD.B
%%%% ¡title!
Input ADJACENCY MATRIX

%%% ¡prop!
%%%% ¡id!
GraphBD.SEMIPOSITIVIZE_RULE
%%%% ¡title!
NEGATIVE WEIGHTS RULE

%%% ¡prop!
%%%% ¡id!
GraphBD.A
%%%% ¡title!
Binary Directed ADJACENCY MATRIX
```

```

%%% ¡prop!
%%%% ¡id!
GraphBD.PFGA
%%%% ¡title!
Adjacency Matrix Plot

%%% ¡prop!
%%%% ¡id!
GraphBD.PFGH
%%%% ¡title!
Graph Histogram

%%% ¡prop!
%%%% ¡id!
GraphBD.M_DICT
%%%% ¡title!
Graph MEASURES

%%% ¡prop!
%%%% ¡id!
GraphBD.NOTES
%%%% ¡title!
Graph NOTES
```

Code 3: **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.

```
%% ¡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'));
B = binarize(B);

A = {B}; ⑨
value = A; ⑩

%%%% ¡gui! ⑪
pr = PanelPropCell('EL', g, 'PROP', GraphBD.A, ... ⑫
'TABLE_HEIGHT', s(40), ...
'XSLIDERSHOW', false, ...
'YSLIDERSHOW', false, ...
'ROWNAME' , g.getCallback('ANODELABELS'), ...
'COLUMNNAME', g.getCallback('ANODELABELS'), ...
);
```

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 calcualted 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)

```


%%% ¡prop! (13)
COMPATIBLE_MEASURES (constant, classlist) is the list of compatible measures
     .
%%%% ¡default!
getCompatibleMeasures('GraphBD')

%%% ¡prop!(14)
B (data, smatrix) is the input graph adjacency matrix.
%%%% ¡gui!
pr = PanelPropMatrix('EL', g, 'PROP', GraphBD.B, ... (15)
'TABLE_HEIGHT' , s(40), ...
'ROWNAME' , g.getCallback('ANODELABELS'), ...
'COLUMNNAME', g.getCallback('ANODELABELS'), ...
varargin{:});

%%% ¡prop! (16)
SEMIPOSITIVIZE_RULE (parameter, option) determines how to remove the
     negative edges.
%%%% ¡settings!
{'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 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! ③
.01 ③
%%%% ¡code!
B = randn(randi(10)); ⑩
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!
Semipositivize Rules ⑩
%%%% ¡probability!
.01 ③
%%%% ¡code!
B = [
-2 -1 0 1 2
-1 0 1 2 -2
0 1 2 -2 -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

④ Generates a random graph

⑩ Checks the `SEMIPOSITIVIZE_RULE` on the `GraphBD` graph.

```
1 2 -2 -1 0
2 -2 -1 0 1
]; (11)

g0 = GraphBD('B', B);  (12)
A0 = {[
  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
  ]};  (13)
assert(isequal(g0.get('A'), A0), ...  (8)
[BRAPH2.STR ':GraphBD:' BRAPH2.FAIL_TEST], ...
'GraphBD is not constructing well.')

g_zero = GraphBD('B', B, 'SEMIPOSITIVIZE_RULE', 'zero'); (14)
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
  ]}; (13)
assert(isequal(g_zero.get('A'), A_zero), ... (8)
[BRAPH2.STR ':GraphBD:' BRAPH2.FAIL_TEST], ...
'GraphBD is not constructing well.')

g_absolute = GraphBD('B', B, 'SEMIPOSITIVIZE_RULE', 'absolute'); (15)
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
  ]}; (13)
assert(isequal(g_absolute.get('A'), A_absolute), ... (8)
[BRAPH2.STR ':GraphBD:' BRAPH2.FAIL_TEST], ...
'GraphBD is not constructing well.')

...
```

(11) Generates an example graph with negative weights

(12) Constructs the `GraphBD` graph from the initialized `B` with default RULE for `SEMIPOSITIVIZE_RULE`.

(13) Expected value of the graph calculated by external means

(14) Constructs the `GraphBD` graph from the initialized `B` with RULE = 'zero' for `SEMIPOSITIVIZE_RULE`.

(15) Constructs the `GraphBD` graph from the initialized `B` with RULE = 'absolute' for `SEMIPOSITIVIZE_RULE`

Implementation of Multilayer Graph

Multilayer Weigthed Directed Graph (MultilayerWD)

We can now use GraphBD as the basis to implement the global measure MultilayerWD. The parts of the code that are modified are highlighted.

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.

```
%% ¡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 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.'

%%% ¡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;
```

① For each layer in `MultilayerWD` graph the corresponding functions are applied as in Code 3 ⑧

Code 7: **MultilayerWD element tests.** The `tests` section from the element generator `_MultilayerWD.gen.m`.

```
%% ¡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.')
```

Multiplex ..... Graph ( )

Ordinal Multilayer ..... Graph ( )

Ordinal Multiplex ... Graph ( )