Implement a new Measure

The BRAPH 2 Developers

June 1, 2023

This is the developer tutorial for implementing a new measure. In this Tutorial, we will explain how to create the generator file `*.gen.m` for a new measure which can the be compiled by `braph2genesis`, using the measures `Degree`, `DegreeAv`, `Distance`, and `Triangles` as examples.

Contents

Implementation of Degree

We will start by implementing in detail the measure `Degree`, which applies to most graphs and is a direct extension of the element `Measure`.

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

```
%% ¡header!
Degree < Measure (m, degree) is the graph degree. (1)

%%% ¡description!
The degree of a node is the number of edges connected to the node within a
    layer.
Connection weights are ignored in calculations.
```

(1) The element `Degree` is defined as a subclass of `Measure`. The moniker will be `m`.

Code 2: **Degree element prop update.** The `props_update` section of generator code for `_Degree.gen.m` updates the properties of the `Measure` element. This defines the core properties of the measure.

```
%% ¡props_update!

%%% ¡prop!
NAME (constant, string) is the name of the degree.
%%%% ¡default!
'Degree'

%%% ¡prop!
DESCRIPTION (constant, string) is the description of the degree.
%%%% ¡default!
'The degree of a node is the number of edges connected to the node within a
    layer. Connection weights are ignored in calculations.'

%%% ¡prop!
TEMPLATE (parameter, item) is the template of the degree.

%%% ¡prop!
ID (data, string) is a few-letter code of the degree.
%%%% ¡default!
'Degree ID'

%%% ¡prop!
LABEL (metadata, string) is an extended label of the degree.
%%%% ¡default!
'Degree label'

%%% ¡prop!
NOTES (metadata, string) are some specific notes about the degree.
%%%% ¡default!
'Degree notes'

%%% ¡prop! (1)
SHAPE (constant, scalar) is the measure shape __Measure.NODAL__.
%%%% ¡default!
Measure.NODAL
```

(1) Measures have a *shape*: `Measure.GLOBAL` (a value for the whole brain graph, e.g., average degree), `Measure.NODAL` (a value for each brain region, e.g., degree, or `Measure.BINODAL` (a value for each couple of brain regions, e.g., distance between couples of nodes).

```
%%% ¡prop! (2)
SCOPE (constant, scalar) is the measure scope __Measure.UNILAYER__.
%%%% ¡default!
Measure.UNILAYER

%%% ¡prop! (3)
PARAMETRICITY (constant, scalar) is the parametricity of the measure
    __Measure.NONPARAMETRIC__.
%%%% ¡default!
Measure.NONPARAMETRIC

%%% ¡prop! (4)
COMPATIBLE_GRAPHS (constant, classlist) is the list of compatible graphs.
%%%% ¡default!
{'GraphWU' 'GraphBU' 'MultigraphBUD' 'MultigraphBUT' 'MultiplexWU' '
    MultiplexBU' 'MultiplexBUD' 'MultiplexBUT' 'OrdMxWU' 'OrdMxBU'}

%%% ¡prop! (5)
M (result, cell) is the degree.
%%%% ¡calculate!
g = m.get('G'); (6)
A = g.get('A'); (7)

degree = cell(g.get('LAYERNUMBER'), 1); (8)

parfor li = 1:1:g.get('LAYERNUMBER')
    Aii = A{li, li};
    Aii = binarize(Aii); (9)
    degree(li) = {sum(Aii, 2)}; (10)
end

value = degree; (11)
```

(2) Measures have a *scope*: `Measure.SUPERGLOBAL` (a result for the whole multi-layer graph, e.g., overlapping strength), `Measure.UNILAYER` (a result for each layer, e.g., average strength), or `Measure.BILAYER` (a result for each couple of layers).

(3) Measures are either `Measure.NONPARAMETRIC` (the usual case) or `Measure.PARAMETRIC` (depending on some parameter).

(4) Each measure has a list of compatible graphs for which the measure can be used.

(5) The property `M` contains the code to be executed to calculate the measure. Here is where most of the action happens.

(6) retrieves the graph from the property `G` of the measure `m`.

(7) retrieves the cell with the adjacency matrix (for graph) or 2D-cell array (for multigraph, multiplex, etc.).

(8) preallocates the variable to contain the result of the measure calculation.

(9) binarizes the adjacency matrix (removing diagonal).

(10) calculates the degree of the node for layer `li`.

(11) returns the calcualted value of the measure `degree` assigning it to the output variable `value`.

Code 3: **Degree element tests.** The `tests` section from the element generator `_Degree.gen.m`. A test should be prepared for each graph with which the measure is compatible. The test should at least verify in some simple cases that the value of the measure is correct.

```
%% ¡tests!

%%% ¡excluded_props! (1)
[Degree.PFM]

%%% ¡test! (2)
%%%% ¡name!
GraphWU
%%%% ¡probability! (3)
.01
%%%% ¡code!
B = [
    0   .6  1
    .6  0   0
    1   0   0
    ];

known_degree = {[2 1 1]'}; (4)
```

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

(2) Test for `GraphWU`. Similar tests should be implemented for each graph compatible with the measure.

(3) assings a low test execution probability.

(4) is the expected value of the measure calculated by external means.

```
g = GraphWU('B', B); 5

m_outside_g = Degree('G', g); 6
assert(isequal(m_outside_g.get('M'), known_degree), ... 7
    [BRAPH2.STR ':Degree:' BRAPH2.FAIL_TEST], ...
    [class(m_outside_g) ' is not being calculated correctly for ' class(g) '.'])

m_inside_g = g.get('MEASURE', 'Degree'); 8
assert(isequal(m_inside_g.get('M'), known_degree), ... 8
    [BRAPH2.STR ':Degree:' BRAPH2.FAIL_TEST], ...
    [class(m_inside_g) ' is not being calculated correctly for ' class(g) '.'])

...
```

5 creates the graph.

6 creates the measure.

7 tests that the value of the measure coicides with its expected value.

8 extracts the measure from the graph.

8 tests that the value of the measure extracted from the graph coicides with its expected value.

Implementation of DegreeAv

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

Code 4: **DegreeAv element header.** The `header` section of generator code for `_DegreeAv.gen.m` provides the general information about the `DegreeAv` element. ← Code 1

```
%% ¡header!
DegreeAv < Degree (m, average degree) is the graph average degree. 1

%%% ¡description!
The average degree of a graph is the average of all number of edges
    connected to a node within a layer.
Connection weights are ignored in calculations.
```

1 `DegreeAv` is a child of `Degree`.

Code 5: **DegreeAv element prop update.** The `props_update` section of generator code for `_DegreeAv.gen.m` updates the properties of the `Degree` element. ← Code 2

```
%% ¡props_update!

%%% ¡prop!
NAME (constant, string) is the name of the average degree.
%%%% ¡default!
'DegreeAv'

%%% ¡prop!
DESCRIPTION (constant, string) is the description of the average degree.
%%%% ¡default!
'The average degree of a graph is the average of all number of edges
    connected to a node within a layer. Connection weights are ignored in
    calculations.'

%%% ¡prop!
TEMPLATE (parameter, item) is the template of the average degree.

%%% ¡prop!
ID (data, string) is a few-letter code of the average degree.
%%%% ¡default!
'DegreeAv ID'

%%% ¡prop!
LABEL (metadata, string) is an extended label of the average degree.
%%%% ¡default!
'DegreeAv label'

%%% ¡prop!
NOTES (metadata, string) are some specific notes about the average degree.
%%%% ¡default!
'DegreeAv notes'

%%% ¡prop!
SHAPE (constant, scalar) is the measure shape __Measure.GLOBAL__.
%%%% ¡default!
Measure.GLOBAL

```

```
%%% ¡prop!
SCOPE (constant, scalar) is the measure scope __Measure.UNILAYER__.
%%%% ¡default!
Measure.UNILAYER

%%% ¡prop!
PARAMETRICITY (constant, scalar) is the parametricity of the measure __Measure.NONPARAMETRIC__.
%%%% ¡default!
Measure.NONPARAMETRIC

%%% ¡prop!
COMPATIBLE_GRAPHS (constant, classlist) is the list of compatible graphs.
%%%% ¡default!
{'GraphWU' 'GraphBU' 'MultigraphBUD' 'MultigraphBUT' 'MultiplexWU' 'MultiplexBU' 'MultiplexBUD' 'MultiplexBUT' 'OrdMxWU' 'OrdMxBU'}

%%% ¡prop!
M (result, cell) is the average degree.
%%%% ¡calculate!
degree = calculateValue@Degree(m, prop); 1

g = m.get('G');

degree_av = cell(g.get('LAYERNUMBER'), 1);
parfor li = 1:1:g.get('LAYERNUMBER')
    degree_av(li) = {mean(degree{li})};
end

value = degree_av;
```

1 calculates the value of the degree calling its parent `Degree`.

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

```
%% ¡tests!

%%% ¡excluded_props!
[DegreeAv.PFM]

%%% ¡test!
%%%% ¡name!
GraphWU
%%%% ¡probability!
.01
%%%% ¡code!
B = [
    0   .6  1
    .6  0   0
    1   0   0
    ];

known_degree_av = {mean([2 1 1])};

g = GraphWU('B', B);

m_outside_g = DegreeAv('G', g);
assert(isequal(m_outside_g.get('M'), known_degree_av), ...
    [BRAPH2.STR ':DegreeAv:' BRAPH2.FAIL_TEST], ...
    [class(m_outside_g) ' is not being calculated correctly for ' class(g) '.'])
```

```
m_inside_g = g.get('MEASURE', 'DegreeAv');
assert(isequal(m_inside_g.get('M'), known_degree_av), ...
    [BRAPH2.STR ':DegreeAv:' BRAPH2.FAIL_TEST], ...
    [class(m_inside_g) ' is not being calculated correctly for ' class(g)
     '.'])

...
```

Implementation of Distance

Now we implement the binodal measure `Distance`, again highlighting the differences.

Code 7: **Distance element header.** The `header` section of generator code for `_Distance.gen.m` provides the general information about the `Distance` element. ← Code 1

```
%% ¡header!
Distance < Measure (m, distance) is the distance.

%%% ¡description!
The distance of a graph is the shortest path between all pairs of nodes
     within a layer of the graph.
For weighted graphs, the distance is calculated with the Dijkstra algorithm
     using the inverse weight as the distance associated to the edge.
```

Code 8: **Distance element prop update.** The `props_update` section of generator code for `_Distance.gen.m` updates the properties of the `Measure` element. ← Code 2

```
%% ¡props_update!

%%% ¡prop!
NAME (constant, string) is the name of the distance.
%%%% ¡default!
'Distance'

%%% ¡prop!
DESCRIPTION (constant, string) is the description of the distance.
%%%% ¡default!
'The distance of a graph is the shortest path between all pairs of nodes
     within a layer of the graph. For weighted graphs, the distance is
     calculated with the Dijkstra algorithm using the inverse weight as the
     distance associated to the edge.'

%%% ¡prop!
TEMPLATE (parameter, item) is the template of the distance.

%%% ¡prop!
ID (data, string) is a few-letter code of the distance.
%%%% ¡default!
'Distance ID'

%%% ¡prop!
LABEL (metadata, string) is an extended label of the distance.
%%%% ¡default!
'Distance label'

%%% ¡prop!
NOTES (metadata, string) are some specific notes about the distance.
%%%% ¡default!
'Distance notes'

%%% ¡prop!
SHAPE (constant, scalar) is the measure shape __Measure.BINODAL__.
%%%% ¡default!
Measure.BINODAL
```

```
%%% ¡prop!
SCOPE (constant, scalar) is the measure scope __Measure.UNILAYER__.
%%%% ¡default!
Measure.UNILAYER

%%% ¡prop!
PARAMETRICITY (constant, scalar) is the parametricity of the measure __Measure.NONPARAMETRIC__.
%%%% ¡default!
Measure.NONPARAMETRIC

%%% ¡prop!
COMPATIBLE_GRAPHS (constant, classlist) is the list of compatible graphs.
%%%% ¡default!
{'GraphBD' 'GraphBU' 'GraphWD' 'GraphWU' 'MultigraphBUD' 'MultigraphBUT' 'MultiplexBD' 'MultiplexBU' 'MultiplexWD' 'MultiplexWU' 'MultiplexBUD' 'MultiplexBUT' 'OrdMxBD' 'OrdMxBU' 'OrdMxWD' 'OrdMxWU'}

%%% ¡prop!
M (result, cell) is the distance.
%%%% ¡calculate!
g = m.get('G');
A = g.get('A');

distance = cell(g.get('LAYERNUMBER'), 1);
connectivity_type =  g.get('CONNECTIVITY_TYPE', g.get('LAYERNUMBER'));
connectivity_type = diag(connectivity_type);
Aii_tmp = {};
for li = 1:1:g.get('LAYERNUMBER')
    Aii_tmp{li} = A{li, li};
end
for li = 1:1:g.get('LAYERNUMBER')
    Aii = Aii_tmp{li};
    connectivity_layer = connectivity_type(li);

    if connectivity_layer == Graph.WEIGHTED  % weighted graphs
        distance(li) = {getWeightedCalculation(Aii)}; 1
    else  % binary (i.e., non-weighted) graphs
        distance(li) = {getBinaryCalculation(Aii)}; 2
    end
end

value = distance;
%%%% ¡calculate_callbacks! 3
function weighted_distance = getWeightedCalculation(A)
    ...
end
function binary_distance = getBinaryCalculation(A)
    ...
end
```

1 and 2 call some callback functions that are provided below in 3.

3 This section contains the callback functions for the calculation of the measure.

Code 9: **Distance element tests.** The `tests` section from the element generator `_Distance.gen.m`. ← Code 3

```
%% ¡tests!

%%% ¡excluded_props!
[Distance.PFM]

```

```
%%% ¡test!
%%%% ¡name!
GraphWU
%%%% ¡probability!
.01
%%%% ¡code!
B = [
    0     .1    .2  .25  0
    .125  0     0   0    0
    .2    .5    0   .25  0
    .125  10    0   0    0
    0     0     0   0    0
    ];

known_distance = {[
    0   5   5   4   Inf
    5   0   2   1   Inf
    5   2   0   3   Inf
    4   1   3   0   Inf
    Inf Inf Inf Inf 0
    ]};

g = GraphWU('B', B);

m_outside_g = Distance('G', g);
assert(isequal(m_outside_g.get('M'), known_distance), ...
    [BRAPH2.STR ':Distance:' BRAPH2.FAIL_TEST], ...
    [class(m_outside_g) ' is not being calculated correctly for ' class(g)
     '.'])

m_inside_g = g.get('MEASURE', 'Distance');
assert(isequal(m_inside_g.get('M'), known_distance), ...
    [BRAPH2.STR ':Distance:' BRAPH2.FAIL_TEST], ...
    [class(m_inside_g) ' is not being calculated correctly for ' class(g)
     '.'])

...
```

Implementation of Triangles

Now we implement the nodal measure `Triangles`, again highlighting the differences. The parts of the code that are modified are highlighted.

Code 10: **Triangles element header.** The `header` section of generator code for `_Triangles.gen.m` provides the general information about the `Triangles` element. ← Code 10

```

%% ¡header!
Triangles < Measure (m, triangls) is the triangles.

%%% ¡description!
The triangles are calculated as the number of neighbors of a node that are also neighbors of each other within a layer.
In weighted graphs, the triangles are calculated as geometric mean of the weights of the edges forming the triangle.
```

Code 11: **Triangles element calculate.** The `calculate` section of `_Triangles.gen.m` utilizes the rule property to select which algorithm it will use to calculate the `Triangles` element. ← Code 11

```

%% ¡props_update!

%%% ¡prop!
NAME (constant, string) is the name of the triangles.
%%%% ¡default!
'Triangles'

%%% ¡prop!
DESCRIPTION (constant, string) is the description of the triangles.
%%%% ¡default!
'The triangles are calculated as the number of neighbors of a node that are also neighbors of each other within a layer. In weighted graphs, the triangles are calculated as geometric mean of the weights of the edges forming the triangle.'

...

%%% ¡prop!
M (result, cell) is the triangles.
%%% ¡calculate!
g = m.get('G');  % graph from measure class
A = g.get('A');  % cell with adjacency matrix (for graph) or 2D-cell array (for multigraph, multiplex, etc.)
L = g.get('LAYERNUMBER');

triangles = cell(L, 1);

directionality\_type =   g.get('DIRECTIONALITY\_TYPE', L);
for li = 1:1:L
Aii = A{li, li};

if directionality_type == Graph.UNDIRECTED  % undirected graphs
triangles_layer = diag((Aii.^(1/3))^3) / 2;
```

```
triangles_layer(isnan(triangles_layer)) = 0;  % Should return zeros, not NaN
triangles(li) = {triangles_layer};

else  % directed graphs
directed_triangles_rule = m.get('RULE'); 1
switch lower(directed_triangles_rule)
case 'all'  % all rule
triangles_layer = diag((Aii.^(1/3) + transpose(Aii).^(1/3))^3) / 2;
case 'middleman'  % middleman rule
triangles_layer = diag(Aii.^(1/3) * transpose(Aii).^(1/3) * Aii.^(1/3));
case 'in'  % in rule
triangles_layer = diag(transpose(Aii).^(1/3) * (Aii.^(1/3))^2);
case 'out'  % out rule
triangles_layer = diag((Aii.^(1/3))^2 * transpose(Aii).^(1/3));
otherwise  % {'cycle'}  % cycle rule
triangles_layer = diag((Aii.^(1/3))^3);
end
triangles_layer(isnan(triangles_layer)) = 0;  % Should return zeros, not NaN
triangles(li) = {triangles_layer};
end
end
value = triangles;

%% ¡props!
%%% ¡prop!
rule (parameter, OPTION) is the rule to determine what is a triangle in
    directed networks. 2
%%%% ¡settings!
{'all' 'middleman' 'in' 'out' 'cycle'}
%%%% ¡default!
'cycle'
```

1 check for measure Triangle rules in 2.

2 these are the available triangle rules.