

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 be compiled by `braph2genesis`, using the measures `Degree`, `DegreeAv`, `Distance`, and `Triangles` as examples.

Contents

<i>Implementation of Unilayer Measures</i>	2
<i>Nodal Unilayer Measure (Degree)</i>	2
<i>Global Unilayer Measure (DegreeAv)</i>	5
<i>Binodal Unilayer Measure (Distance)</i>	8
<i>Implementation of Measure Parameters (Triangles)</i>	11

Implementation of Unilayer Measures

Nodal Unilayer Measure (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.

```

1 %% iheader!
2 Degree < Measure (m, degree) is the graph degree. ①
3
4 %%% idescription!
5 The degree of a node is the number of edges connected to the node within a
   layer.
6 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.

```

1 %% iprops_update!
2
3 %%% iprop!
4 NAME (constant, string) is the name of the degree.
5 %%% idefault!
6 'Degree'
7
8 %%% iprop!
9 DESCRIPTION (constant, string) is the description of the degree.
10 %%% idefault!
11 'The degree of a node is the number of edges connected to the node within a
   layer. Connection weights are ignored in calculations.'
12
13 %%% iprop!
14 TEMPLATE (parameter, item) is the template of the degree.
15
16 %%% iprop!
17 ID (data, string) is a few-letter code of the degree.
18 %%% idefault!
19 'Degree ID'
20
21 %%% iprop!
22 LABEL (metadata, string) is an extended label of the degree.
23 %%% idefault!
24 'Degree label'
25
26 %%% iprop!
27 NOTES (metadata, string) are some specific notes about the degree.
28 %%% idefault!
29 'Degree notes'
30
31 %%% iprop! ①
32 SHAPE (constant, scalar) is the measure shape __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).

```

33 %%%% idefault!
34 Measure.NODAL
35
36 %%% iprop! (2)
37 SCOPE (constant, scalar) is the measure scope __Measure.UNILAYER__.
38 %%%% idefault!
39 Measure.UNILAYER
40
41 %%% iprop! (3)
42 PARAMETRICITY (constant, scalar) is the parametricity of the measure
   __Measure.NONPARAMETRIC__.
43 %%%% idefault!
44 Measure.NONPARAMETRIC
45
46 %%% iprop! (4)
47 COMPATIBLE_GRAPH (constant, classlist) is the list of compatible graphs.
48 %%%% idefault!
49 {'GraphWU' 'GraphBU' 'MultigraphBUD' 'MultigraphBUT' 'MultiplexWU' '
   MultiplexBU' 'MultiplexBUD' 'MultiplexBUT' 'OrdMxWU' 'OrdMxBU'}
50
51 %%% iprop! (5)
52 M (result, cell) is the degree.
53 %%%% icalculate!
54 g = m.get('G'); (6)
55 A = g.get('A'); (7)
56
57 degree = cell(g.get('LAYERNUMBER'), 1); (8)
58
59 parfor li = 1:1:g.get('LAYERNUMBER')
60     Aii = A{li, li};
61     Aii = binarize(Aii); (9)
62     degree(li) = {sum(Aii, 2)}; (10)
63 end
64
65 value = degree; (11)

```

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.

```

1 %%% itests!
2
3 %%% iexcluded_props! (1)
4 [Degree.PFM]
5
6 %%% itest! (2)
7 %%%% iname!
8 GraphWU
9 %%%% iprobability! (3)
10 .01
11 %%%% icode!
12 B = [
13     0   .6   1
14     .6   0   0
15     1   0   0

```

(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 calculated value of the measure degree assigning it to the output variable value.

(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) assigns a low test execution probability.

```

16     ];
17
18     known_degree = {[2 1 1]'}; ④
19
20     g = GraphWU('B', B); ⑤
21
22     m_outside_g = Degree('G', g); ⑥
23     assert(isequal(m_outside_g.get('M'), known_degree), ... ⑦
24         [BRAPH2.STR ':Degree:' BRAPH2.FAIL_TEST], ...
25         [class(m_outside_g) ' is not being calculated correctly for ' class(g) '
26         .'])
27
28     m_inside_g = g.get('MEASURE', 'Degree'); ⑧
29     assert(isequal(m_inside_g.get('M'), known_degree), ... ⑧
30         [BRAPH2.STR ':Degree:' BRAPH2.FAIL_TEST], ...
31         [class(m_inside_g) ' is not being calculated correctly for ' class(g) '
32         .'])

```

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

⑤ creates the graph.

⑥ creates the measure.

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

⑧ extracts the measure from the graph.

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

Global Unilayer Measure (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](#)

```

1 %% iheader!
2 DegreeAv < Degree (m, average degree) is the graph average degree. ①
3
4 %%% idescription!
5 The average degree of a graph is the average of all number of edges
  connected to a node within a layer.
6 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](#)

```

1 %% iprops_update!
2
3 %%% iprop!
4 NAME (constant, string) is the name of the average degree.
5 %%% idefault!
6 'DegreeAv'
7
8 %%% iprop!
9 DESCRIPTION (constant, string) is the description of the average degree.
10 %%% idefault!
11 '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.'
12
13 %%% iprop!
14 TEMPLATE (parameter, item) is the template of the average degree.
15
16 %%% iprop!
17 ID (data, string) is a few-letter code of the average degree.
18 %%% idefault!
19 'DegreeAv ID'
20
21 %%% iprop!
22 LABEL (metadata, string) is an extended label of the average degree.
23 %%% idefault!
24 'DegreeAv label'
25
26 %%% iprop!
27 NOTES (metadata, string) are some specific notes about the average degree.
28 %%% idefault!
29 'DegreeAv notes'
30
31 %%% iprop!
32 SHAPE (constant, scalar) is the measure shape __Measure.GLOBAL__.
33 %%% idefault!
34 Measure.GLOBAL
35
36 %%% iprop!

```

```

37 SCOPE (constant, scalar) is the measure scope __Measure.UNILAYER__.
38 %%% idefault!
39 Measure.UNILAYER
40
41 %%% iprop!
42 PARAMETRICITY (constant, scalar) is the parametricity of the measure
   __Measure.NONPARAMETRIC__.
43 %%% idefault!
44 Measure.NONPARAMETRIC
45
46 %%% iprop!
47 COMPATIBLE_GRAPHS (constant, classlist) is the list of compatible graphs.
48 %%% idefault!
49 {'GraphWU' 'GraphBU' 'MultigraphBUD' 'MultigraphBUT' 'MultiplexWU' '
   MultiplexBU' 'MultiplexBUD' 'MultiplexBUT' 'OrdMxWU' 'OrdMxBU'}
50
51 %%% iprop!
52 M (result, cell) is the average degree.
53 %%% icalculate!
54 degree = calculateValue@Degree(m, prop); ①
55
56 g = m.get('G');
57
58 degree_av = cell(g.get('LAYERNUMBER'), 1);
59 parfor li = 1:1:g.get('LAYERNUMBER')
60     degree_av(li) = {mean(degree{li})};
61 end
62
63 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](#)

```

1 %%% itests!
2
3 %%% iexcluded_props!
4 [DegreeAv.PFM]
5
6 %%% itest!
7 %%% iiname!
8 GraphWU
9 %%% iprobability!
10 .01
11 %%% icode!
12 B = [
13     0   .6  1
14     .6  0   0
15     1   0   0
16 ];
17
18 known_degree_av = {mean([2 1 1])};
19
20 g = GraphWU('B', B);
21
22 m_outside_g = DegreeAv('G', g);
23 assert(isequal(m_outside_g.get('M'), known_degree_av), ...
24     [BRAPH2.STR 'DegreeAv:' BRAPH2.FAIL_TEST], ...
25     [class(m_outside_g) ' is not being calculated correctly for ' class(g)
26     '.'])

```

```
27 m_inside_g = g.get('MEASURE', 'DegreeAv');
28 assert(isequal(m_inside_g.get('M'), known_degree_av), ...
29         [BRAPH2.STR 'DegreeAv:' BRAPH2.FAIL_TEST], ...
30         [class(m_inside_g) ' is not being calculated correctly for ' class(g)
31          '.'])
32 ...
```

Binodal Unilayer Measure (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](#)

```

1 %% iheader!
2 Distance < Measure (m, distance) is the distance.
3
4 %%% idescription!
5 The distance of a graph is the shortest path between all pairs of nodes
   within a layer of the graph.
6 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](#)

```

1 %% iprops_update!
2
3 %%% iprop!
4 NAME (constant, string) is the name of the distance.
5 %%% idefault!
6 'Distance'
7
8 %%% iprop!
9 DESCRIPTION (constant, string) is the description of the distance.
10 %%% idefault!
11 '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.'
12
13 %%% iprop!
14 TEMPLATE (parameter, item) is the template of the distance.
15
16 %%% iprop!
17 ID (data, string) is a few-letter code of the distance.
18 %%% idefault!
19 'Distance ID'
20
21 %%% iprop!
22 LABEL (metadata, string) is an extended label of the distance.
23 %%% idefault!
24 'Distance label'
25
26 %%% iprop!
27 NOTES (metadata, string) are some specific notes about the distance.
28 %%% idefault!
29 'Distance notes'
30
31 %%% iprop!
32 SHAPE (constant, scalar) is the measure shape __Measure.BINODAL__.
33 %%% idefault!
34 Measure.BINODAL

```

```

35
36 %% iprop!
37 SCOPE (constant, scalar) is the measure scope __Measure.UNILAYER__.
38 %%%% idefault!
39 Measure.UNILAYER
40
41 %% iprop!
42 PARAMETRICITY (constant, scalar) is the parametricity of the measure
   __Measure.NONPARAMETRIC__.
43 %%%% idefault!
44 Measure.NONPARAMETRIC
45
46 %% iprop!
47 COMPATIBLE_GRAPH (constant, classlist) is the list of compatible graphs.
48 %%%% idefault!
49 {'GraphBD' 'GraphBU' 'GraphWD' 'GraphWU' 'MultigraphBUD' 'MultigraphBUT' '
   MultiplexBD' 'MultiplexBU' 'MultiplexWD' 'MultiplexWU' 'MultiplexBUD' '
   MultiplexBUT' 'OrdMxBD' 'OrdMxBU' 'OrdMxWD' 'OrdMxWU'}
50
51 %% iprop!
52 M (result, cell) is the distance.
53 %%%% icalculate!
54 g = m.get('G');
55 A = g.get('A');
56
57 distance = cell(g.get('LAYERNUMBER'), 1);
58 connectivity_type = g.get('CONNECTIVITY_TYPE', g.get('LAYERNUMBER'));
59 connectivity_type = diag(connectivity_type);
60 Aii_tmp = {};
61 for li = 1:1:g.get('LAYERNUMBER')
62     Aii_tmp{li} = A{li, li};
63 end
64 for li = 1:1:g.get('LAYERNUMBER')
65     Aii = Aii_tmp{li};
66     connectivity_layer = connectivity_type(li);
67
68     if connectivity_layer == Graph.WEIGHTED % weighted graphs
69         distance(li) = {getWeightedCalculation(Aii)}; ①
70     else % binary (i.e., non-weighted) graphs
71         distance(li) = {getBinaryCalculation(Aii)}; ②
72     end
73 end
74
75 value = distance;
76 %%%% icalculate_callbacks! ③
77 function weighted_distance = getWeightedCalculation(A)
78     ...
79 end
80 function binary_distance = getBinaryCalculation(A)
81     ...
82 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](#)

```

1 %% itests!
2
3 %%%% iexcluded_props!
4 [Distance.PFM]
5

```

```

6  %%% itest!
7  %%%% iname!
8  GraphWU
9  %%%% iprobability!
10 .01
11 %%%% icode!
12 B = [
13     0     .1     .2     .25    0
14     .125  0     0     0     0
15     .2     .5     0     .25    0
16     .125  10     0     0     0
17     0     0     0     0     0
18 ];
19
20 known_distance = {[
21     0     5     5     4     Inf
22     5     0     2     1     Inf
23     5     2     0     3     Inf
24     4     1     3     0     Inf
25     Inf  Inf  Inf  Inf  0
26 ]};
27
28 g = GraphWU('B', B);
29
30 m_outside_g = Distance('G', g);
31 assert(isequal(m_outside_g.get('M'), known_distance), ...
32     [BRAPH2.STR ':Distance:' BRAPH2.FAIL_TEST], ...
33     [class(m_outside_g) ' is not being calculated correctly for ' class(g)
34     '.'])
35
36 m_inside_g = g.get('MEASURE', 'Distance');
37 assert(isequal(m_inside_g.get('M'), known_distance), ...
38     [BRAPH2.STR ':Distance:' BRAPH2.FAIL_TEST], ...
39     [class(m_inside_g) ' is not being calculated correctly for ' class(g)
40     '.'])
41 ...

```

Implementation of Measure Parameters (Triangles)

Now, we implement the (nodal unilayer) measure `Triangles`, which depends on a parameter `RULE`, which we add as a property of category parameter. 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](#)

```

1
2 %% iheader!
3 Triangles < Measure (m, triangles) is the triangles.
4
5 %%% idescription!
6 The triangles are calculated as the number of neighbors of a node that are
  also neighbors of each other within a layer.
7 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](#)

```

1
2 %% iprops_update!
3
4 %%% iprop!
5 NAME (constant, string) is the name of the triangles.
6 %%% idefault!
7 'Triangles'
8
9 %%% iprop!
10 DESCRIPTION (constant, string) is the description of the triangles.
11 %%% idefault!
12 '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.'
13
14 ...
15
16 %%% iprop!
17 M (result, cell) is the triangles.
18 %%% icalculate!
19 g = m.get('G'); % graph from measure class
20 A = g.get('A'); % cell with adjacency matrix (for graph) or 2D-cell array (
   for multigraph, multiplex, etc.)
21 L = g.get('LAYERNUMBER');
22
23 triangles = cell(L, 1);
24
25 directionality_type = g.get('DIRECTIONALITY_TYPE', L);
26 for li = 1:L
27   Aii = A{li, li};
28
29 if directionality_type == Graph.UNDIRECTED % undirected graphs

```

```

30 triangles_layer = diag((Aii.^(1/3))^3) / 2;
31 triangles_layer(isnan(triangles_layer)) = 0; % Should return zeros, not NaN
32 triangles(li) = {triangles_layer};
33
34 else % directed graphs
35 directed_triangles_rule = m.get('RULE'); ①
36 switch lower(directed_triangles_rule)
37 case 'all' % all rule
38 triangles_layer = diag((Aii.^(1/3) + transpose(Aii).^(1/3))^3) / 2;
39 case 'middleman' % middleman rule
40 triangles_layer = diag(Aii.^(1/3) * transpose(Aii).^(1/3) * Aii.^(1/3));
41 case 'in' % in rule
42 triangles_layer = diag(transpose(Aii).^(1/3) * (Aii.^(1/3))^2);
43 case 'out' % out rule
44 triangles_layer = diag((Aii.^(1/3))^2 * transpose(Aii).^(1/3));
45 otherwise % {'cycle'} % cycle rule
46 triangles_layer = diag((Aii.^(1/3))^3);
47 end
48 triangles_layer(isnan(triangles_layer)) = 0; % Should return zeros, not NaN
49 triangles(li) = {triangles_layer};
50 end
51 end
52 value = triangles;
53
54 %% iprops!
55 %% iprop!
56 rule (parameter, OPTION) is the rule to determine what is a triangle in
    directed networks. ②
57 %%% isettings!
58 {'all' 'middleman' 'in' 'out' 'cycle'}
59 %%% idefault!
60 'cycle'

```

① check for measure Triangle rules in

②.

② these are the available triangle rules.