Implement a New Measure

The BRAPH 2 Developers

August 14, 2024

This is the developer tutorial for implementing a new measure. In this tutorial, you will learn how to create the generator file `*.gen.m` for a new measure, which can the be compiled by `braph2genesis`. All measures are (direct or indirect) extensions of the element `Measure`. Here, you will use as examples the measures `Degree`, `DegreeAv`, `Distance`, and `Triangles`.

Contents

Implementation of Unilayer Measures

Nodal Unilayer Measure (Degree)

You 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 the generator code in `_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.
7
8  %%% ibuild!
9  1
```

(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 the generator code in `_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  %%%% isettings!
16  'Degree'
17
18  %%% iprop!
19  ID (data, string) is a few-letter code of the degree.
20  %%%% idefault!
21  'Degree ID'
22
23  %%% iprop!
24  LABEL (metadata, string) is an extended label of the degree.
25  %%%% idefault!
26  'Degree label'
27
28  %%% iprop!
29  NOTES (metadata, string) are some specific notes about the degree.
30  %%%% idefault!
```

```
31  'Degree notes'
32
33  %%% iprop! ①
34  SHAPE (constant, scalar) is the measure shape __Measure.NODAL__.
35  %%%% idefault!
36  Measure.NODAL
37
38  %%% iprop! ②
39  SCOPE (constant, scalar) is the measure scope __Measure.UNILAYER__.
40  %%%% idefault!
41  Measure.UNILAYER
42
43  %%% iprop! ③
44  PARAMETRICITY (constant, scalar) is the parametricity of the measure
            __Measure.NONPARAMETRIC__.
45  %%%% idefault!
46  Measure.NONPARAMETRIC
47
48  %%% iprop! ④
49  COMPATIBLE_GRAPHS (constant, classlist) is the list of compatible graphs.
50  %%%% idefault!
51  {'GraphWU' 'GraphBU' 'MultigraphBUD' 'MultigraphBUT' 'MultiplexWU' '
        MultiplexBU' 'MultiplexBUD' 'MultiplexBUT' 'OrdMxWU' 'OrdMxBU'}
52
53  %%% iprop! ⑤
54  M (result, cell) is the degree.
55  %%%% icalculate!
56  g = m.get('G'); ⑥
57  A = g.get('A'); ⑦
58
59  degree = cell(g.get('LAYERNUMBER'), 1); ⑧
60
61  parfor li = 1:1:g.get('LAYERNUMBER')
62      Aii = A{li, li};
63      Aii = binarize(Aii); ⑨
64      degree(li) = {sum(Aii, 2)}; ⑩
65  end
66
67  value = degree; ⑪
```

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).

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.

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!
```

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.

```
12  B = [
13      0   .6  1
14      .6  0   0
15      1   0   0
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  m_inside_g = g.get('MEASURE', 'Degree');  (8)
28  assert(isequal(m_inside_g.get('M'), known_degree), ...  (8)
29      [BRAPH2.STR ':Degree:' BRAPH2.FAIL_TEST], ...
30      [class(m_inside_g) ' is not being calculated correctly for ' class(g) '.
        '])
31
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 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.

Global Unilayer Measure (DegreeAv)

You 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 the generator code in `_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.
7
8  %%% ibuild!
9  1
```

(1) DegreeAv is a child of Degree.

Code 5: **DegreeAv element prop update.** The `props_update` section of the generator code in `_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  %%%% isettings!
16  'DegreeAv'
17
18  %%% iprop!
19  ID (data, string) is a few-letter code of the average degree.
20  %%%% idefault!
21  'DegreeAv ID'
22
23  %%% iprop!
24  LABEL (metadata, string) is an extended label of the average degree.
25  %%%% idefault!
26  'DegreeAv label'
27
28  %%% iprop!
29  NOTES (metadata, string) are some specific notes about the average degree.
30  %%%% idefault!
31  'DegreeAv notes'
```

```
32
33  %%% iprop!
34  SHAPE (constant, scalar) is the measure shape __Measure.GLOBAL__.
35  %%%% idefault!
36  Measure.GLOBAL
37
38  %%% iprop!
39  SCOPE (constant, scalar) is the measure scope __Measure.UNILAYER__.
40  %%%% idefault!
41  Measure.UNILAYER
42
43  %%% iprop!
44  PARAMETRICITY (constant, scalar) is the parametricity of the measure
          __Measure.NONPARAMETRIC__.
45  %%%% idefault!
46  Measure.NONPARAMETRIC
47
48  %%% iprop!
49  COMPATIBLE_GRAPHS (constant, classlist) is the list of compatible graphs.
50  %%%% idefault!
51  {'GraphWU' 'GraphBU' 'MultigraphBUD' 'MultigraphBUT' 'MultiplexWU' '
          MultiplexBU' 'MultiplexBUD' 'MultiplexBUT' 'OrdMxWU' 'OrdMxBU'}
52
53  %%% iprop!
54  M (result, cell) is the average degree.
55  %%%% icalculate!
56  degree = calculateValue@Degree(m, prop);  ( 1 )
57
58  g = m.get('G');
59
60  degree_av = cell(g.get('LAYERNUMBER'), 1);
61  parfor li = 1:1:g.get('LAYERNUMBER')
62      degree_av(li) = {mean(degree{li})};
63  end
64
65  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   %%%% iname!
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 you will implement the binodal measure `Distance`. Again the differences in the code will be highlighted.

Code 7: **Distance element header.** The `header` section of the generator code in `_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.
7
8  %%% ibuild!
9  1
```

Code 8: **Distance element prop update.** The `props_update` section of the generator code in `_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 %%%% isettings!
16 'Distance'
17
18 %%% iprop!
19 ID (data, string) is a few-letter code of the distance.
20 %%%% idefault!
21 'Distance ID'
22
23 %%% iprop!
24 LABEL (metadata, string) is an extended label of the distance.
25 %%%% idefault!
26 'Distance label'
27
28 %%% iprop!
29 NOTES (metadata, string) are some specific notes about the distance.
30 %%%% idefault!
31 'Distance notes'
```

```
32
33  %%% iprop!
34  SHAPE (constant, scalar) is the measure shape __Measure.BINODAL__.
35  %%%% idefault!
36  Measure.BINODAL
37
38  %%% iprop!
39  SCOPE (constant, scalar) is the measure scope __Measure.UNILAYER__.
40  %%%% idefault!
41  Measure.UNILAYER
42
43  %%% iprop!
44  PARAMETRICITY (constant, scalar) is the parametricity of the measure
         __Measure.NONPARAMETRIC__.
45  %%%% idefault!
46  Measure.NONPARAMETRIC
47
48  %%% iprop!
49  COMPATIBLE_GRAPHS (constant, classlist) is the list of compatible graphs.
50  %%%% idefault!
51  {'GraphBD' 'GraphBU' 'GraphWD' 'GraphWU' 'MultigraphBUD' 'MultigraphBUT' '
         MultiplexBD' 'MultiplexBU' 'MultiplexWD' 'MultiplexWU' 'MultiplexBUD' '
         MultiplexBUT' 'OrdMxBD' 'OrdMxBU' 'OrdMxWD' 'OrdMxWU'}
52
53  %%% iprop!
54  M (result, cell) is the distance.
55  %%%% icalculate!
56  g = m.get('G');
57  A = g.get('A');
58
59  distance = cell(g.get('LAYERNUMBER'), 1);
60  connectivity_type =  g.get('CONNECTIVITY_TYPE', g.get('LAYERNUMBER'));
61  connectivity_type = diag(connectivity_type);
62  Aii_tmp = {};
63  for li = 1:1:g.get('LAYERNUMBER')
64      Aii_tmp{li} = A{li, li};
65  end
66  for li = 1:1:g.get('LAYERNUMBER')
67      Aii = Aii_tmp{li};
68      connectivity_layer = connectivity_type(li);
69
70      if connectivity_layer == Graph.WEIGHTED  % weighted graphs
71          distance(li) = {getWeightedCalculation(Aii)};  ①
72      else  % binary (i.e., non-weighted) graphs
73          distance(li) = {getBinaryCalculation(Aii)};  ②
74      end
75  end
76
77  value = distance;
78  %%%% icalculate_callbacks!  ③
79  function weighted_distance = getWeightedCalculation(A)
80      ...
81  end
82  function binary_distance = getBinaryCalculation(A)
83      ...
84  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

```matlab
1  %% !tests!
2
3  %%% !excluded_props!
4  [Distance.PFM]
5
6  %%% !test!
7  %%%% !name!
8  GraphWU
9  %%%% !probability!
10 .01
11 %%%% !code!
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 m_inside_g = g.get('MEASURE', 'Distance');
36 assert(isequal(m_inside_g.get('M'), known_distance), ...
37     [BRAPH2.STR ':Distance:' BRAPH2.FAIL_TEST], ...
38     [class(m_inside_g) ' is not being calculated correctly for ' class(g)
       '.'])
39
40 ...
```

Implementation of Measure Parameters (Triangles)

Now, you will 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 the generator code in `_Triangles.gen.m` provides the general information about the `Triangles` element. ← Code 10

```
1  %% !header!
2  Triangles < Measure (m, triangles) is the triangles.
3
4  %%% !description!
5  The triangles are calculated as the number of neighbors of a node that are
        also neighbors of each other within a layer.
6  In weighted graphs, the triangles are calculated as geometric mean of the
        weights of the edges forming the triangle.
7
8  %%% !build!
9  1
```

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  %% !props_update!
3
4  %%% !prop!
5  NAME (constant, string) is the name of the triangles.
6  %%%% !default!
7  'Triangles'
8
9  %%% !prop!
10 DESCRIPTION (constant, string) is the description of the triangles.
11 %%%% !default!
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 %%% !prop!
17 M (result, cell) is the triangles.
18 %%% !calculate!
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 = 1:1: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) * ...
41               transpose(Aii).^(1/3) * Aii.^(1/3));
42       case 'in'  % in rule
43         triangles_layer = diag(transpose(Aii).^(1/3) * (Aii.^(1/3))^2);
44       case 'out'  % out rule
45         triangles_layer = diag((Aii.^(1/3))^2 * transpose(Aii).^(1/3));
46       otherwise  % {'cycle'}  % cycle rule
47         triangles_layer = diag((Aii.^(1/3))^3);
48     end
49     triangles_layer(isnan(triangles_layer)) = 0;  % Should return zeros, not
           NaN
50     triangles(li) = {triangles_layer};
51   end
52 end
53
54 value = triangles;
55
56 %% iprops!
57
58 %%% iprop!
59 RULE (parameter, option) is the rule to determine what is a triangle in a
       directed graph.  ②
60 %%%% isettings!
61 {'all' 'middleman' 'in' 'out' 'cycle'}  ③
62 %%%% idefault!
63 'cycle'  ④
```

① gets the rule to calculate the measure Triangle. See below how this rule is defined in ②, ③, and ④.

② defines a parameter to determine how the measure is calculated. This property must be of category parameter.

③ defines the available triangle rule options, which are case sensitive.

④ defines the default option, in this case 'cycle'.