Graph API

Veselin Trakiyski edited this page on Dec 4, 2015 · 9 revisions

`sigma.classes.graph` implements sigma's graph model. It is also usable as a standalone object to deal with graph data in your applications. When the JavaScript file is loaded, if the `sigma` object is defined in the global scope, then the graph will be referenced at `sigma.classes.graph`. But if `sigma` is undefined, then the graph will simply be exported as `graph`.

Public methods

nodes() : *array* **nodes(*string*)** : *object* **nodes(*array*)** : *array*

- This method is used to retrieve nodes from the graph. If no argument is given, then an array containing references to every node will be returned. The method can also be called with the ID of a node to only retrieve the related node, or an array of IDs to obtain an array of the specified nodes.

edges() : *array* **edges(*string*)** : *object* **edges(*array*)** : *array*

- This method is used to retrieve edges from the graph. If no argument is given, then an array containing references to every edge will be returned. The method can also be called with the ID of an edge to only retrieve the related edge, or an array of IDs to obtain an array of the specified edges.

addNode(*object*) : *graph*

- This method adds a node to the graph. The node must be an object, with a string under the key `id`. Except for this, it is possible to add any other attribute that will be preserved along all manipulations. If the graph option `clone` has the value 'true', the node will be cloned when added to the graph. Also, if the graph option `immutable` has the value 'true', its `id` will be defined as immutable.
- The method returns the graph instance.

addEdge(*object*) : *graph*

- This method adds an edge to the graph. The edge must be an object, with a string under the key `id`, and strings under the keys `source` and `target` that are existing node IDs. Except for this, it is possible to add any other attribute that will be preserved along all manipulations. If the graph option `clone` has the value 'true', the edge will be cloned when added to the graph. Also, if the graph option `immutable` has the value 'true', its `id`, `source` and `target` will be defined as immutable.

- The method returns the graph instance.

dropNode(*string*) : *graph*

- This method takes an existing node `id` as argument and drops the related node from the graph. It also removes each edge that is bound to it, through the `dropEdge` method. An error is thrown if the node does not exist.
- The method returns the graph instance.

dropEdge(*string*) : *graph*

- This method takes an existing edge `id` as argument and drops the related edge from the graph. An error is thrown if the edge does not exist.
- The method returns the graph instance.

degree(*string*, ?*string*) : *number* **degree(*array*, ?*string*)** : *array*

- This methods is used to retrieve the degree of one or several nodes. If the ID of a node is given, then its degree will be returned. If an array of existing node IDs is given, then the array of the corresponding degrees will be returned instead.
- It is also possible to specify as second argument the type of degree to retrieve:
  - By default the disconnected degree will be used (the number of edges that are connected to the node).
  - If `"in"` is given as second argument, then the incoming degrees will be returned (the number of edges that "come from" the node)
  - If `"out"` is given as second argument, then the outcoming degrees will be returned (the number of edges that "go to" the node)

read(*object*) : *graph*

- This method reads an object and adds the nodes and edges, through the proper methods `addNode` and `addEdge`. It then returns the graph.
- Here is an example:

```
var myGraph = new graph();
myGraph.read({
  nodes: [
    { id: 'n0' },
    { id: 'n1' }
  ],
  edges: [
    {
      id: 'e0',
      source: 'n0',
      target: 'n1'
    }
  ]
});
```

```
console.log(
  myGraph.nodes().length,
  myGraph.edges().length
); // outputs 2 1
```

clear() : *graph*

- This method empties the nodes and edges arrays, as well as the different indexes, and then returns the graph. It is faster than removing by hand `dropNode` and `dropEdge`, since it basically just "reset" the graph.

kill(***object*** **)** : *void*

- This method kills the graph. It basically empties each index and methods attached to the graph, and destroys the references.

Static methods

For people who might want to implement graph algorithms (clustering, graph-specific metrics, etc...), `sigma.classes.graph` provides some static methods to help keeping custom indexes in the graph up to date through every method calls. Also, it is possible to add methods to help getting the indexes or just help manipulating the graphs.

addMethod(*string*, *function*)

- This static method adds a method that will be bound to the newly created graph instances. Since these methods will be bound to their scope when the instances are created, it does not use the prototype. The reason is to make those methods catchable to keep eventual custom indexes up to date with custom bindings (see the `attach` and `addIndex` methods). And because of that, **methods have to be added before instances are created to make them available**.
- Only the methods are accessible from outside the `sigma.classes.graph` instances, but every method is executed in a scope that has access to **the methods**, the **built-in indexes** (described later) and **the custom indexes**.
- The first parameter is the name of the function, and the second parameter is the function. Here is an example:

```
sigma.classes.graph.addMethod('getNodesCount', function() {
  return this.nodesArray.length;
});

var myGraph = new sigma.classes.graph();
console.log(myGraph.getNodesCount()); // outputs 0
```

attach(methodName : *string*, key : *string*, *function*)

- This static method binds a function to a method. Anytime the specified method is called, the bound function is called right after, with the same arguments and in the same scope. To attach a function to the graph constructor, use `"constructor"` as the method name (first argument). To ensure a function is not bound several times to the same method, a unique key (second argument) must identify the function. And if the key is already present an error will be thrown.
- The main idea is to have a clean way to keep custom indexes up to date, for instance:

```
var timesAddNodeCalled = 0;
sigma.classes.graph.attach('addNode', 'timesAddNodeCalledInc', function() {
  timesAddNodeCalled++;
});

var myGraph = new sigma.classes.graph();
console.log(timesAddNodeCalled); // outputs 0

myGraph.addNode({ id: '1' }).addNode({ id: '2' });
console.log(timesAddNodeCalled); // outputs 2
```

addIndex(*string*, *object*)

- This static method will add a custom index. It takes as arguments the name of the index and an object containing all the different functions to bind to the methods, these methods name used as the keys in the object. It is of course possible to bind functions to any method, added with `addMethod` or by yourself.
- Here is a basic example, that creates an index to keep the number of nodes in the current graph. It also adds a method to provide a getter on that new index:

```
sigma.classes.graph.addIndex('nodesCount', {
  constructor: function() {
    this.nodesCount = 0;
  },
  addNode: function() {
    this.nodesCount++;
  },
  dropNode: function() {
    this.nodesCount--;
  }
});

sigma.classes.graph.addMethod('getNodesCount', function() {
  return this.nodesCount;
});

var myGraph = new sigma.classes.graph();
console.log(myGraph.getNodesCount()); // outputs 0

myGraph.addNode({ id: '1' }).addNode({ id: '2' });
console.log(myGraph.getNodesCount()); // outputs 2
```

Private attributes

To ensure the indexes are never broken, they are only accessible in the scope of the methods. Here is an exhaustive list of the built-in indexes:

nodesArray : array

- The array of nodes, sorted by date of arrival (ie the first added node is the first node in the array).

edgesArray : array

- The array of edges, sorted by date of arrival (ie the first added edge is the first edge in the array).

nodesIndex : object

- The object of the nodes indexed by their IDs.

edgesIndex : object

- The object of the edges indexed by their IDs.

inNeighborsCount : object

- An object containing for each node (IDs are the object's keys) the number of incoming edges.

outNeighborsCount : object

- An object containing for each node (IDs are the object's keys) the number of outgoing edges.

allNeighborsCount : object

- An object containing for each node (IDs are the object's keys) the number of edges connected to the node.

inNeighborsIndex : object

- An object containing for each node (associated to its ID) an object containing for each of its incoming neighbors an object indexing the edges connecting the two nodes by their IDs.
- Basically, if an edge `e1` connects `n1` to `n2`, then `inNeighborsIndex[n2][n1][e1]` will reference the edge, but `inNeighborsIndex[n1][n2]` will be null. And if `n1` and `n2` are not connected, then `inNeighborsIndex[n1][n2]` and `inNeighborsIndex[n2][n1]` will both be null.

outNeighborsIndex : object

- An object containing for each node (associated to its ID) an object containing for each of its outgoing neighbors an object indexing the edges connecting the two nodes by their IDs.
- Basically, if an edge `e1` connects `n1` to `n2`, then `outNeighborsIndex[n1][n2][e1]` will reference the edge, but `outNeighborsIndex[n2][n1]` will be null. And if `n1` and `n2` are not connected, then `outNeighborsIndex[n1][n2]` and `outNeighborsIndex[n2][n1]` will both be null.

allNeighborsIndex : object

- An object containing for each node (associated to its ID) an object containing for each of its neighbors an object indexing the edges connecting the two nodes by their IDs.
- Basically, if an edge `e1` connects `n1` to `n2`, then `allNeighborsIndex[n1][n2][e1]` and `allNeighborsIndex[n2][n1][e1]` will reference the edge. But if `n1` and `n2` are not connected, then `allNeighborsIndex[n1][n2]` and `allNeighborsIndex[n2][n1]` will both be null.