Graphology

Designing a graph library for JavaScript

Speakers

Guillaume Plique (@Yomguithereal)

Developer at SciencesPo's médialab

~

Alexis Jacomy (@jacomyal)

CTO of Matlo, sigma.js developer

Observation

- In JavaScript, unlike most other languages, there is no obvious graph library to use.
- In python, for instance, you have networkx etc.
- In C++, you have the Boost library etc.
- Repeat with your favorite language...

JavaScript State of the art

- Cytoscape.js (tied to rendering)
- Sigma.js (tied to rendering)
- graphlib
- jsnetworkx
- graph
- Graph

What is the problem?

- Graph data structures are often tied to a rendering library.
- It is hard to use them on the server (hello, node.js)
- More generally, most libraries are not generic enough and targets really specific use cases.
- This means we are bound to implement popular SNA algorithms each time again, and again, and again...

SNA Algorithms

- "Standard" Gephi SNA workflow:
  1. Compute **metrics**, map to node sizes
  2. Search for **communities**, map to node colors
  3. Run some **layout algorithm**
  4. ...and here is a network map!

SNA Algorithms

- Metrics (Pagerank, HITS, centralities, ...)?
  - **No standard implementation** for quite standard algorithms

SNA Algorithms

- Community detection?
  - Some rogue implementations
  - Some graph rendering libs have their own

SNA Algorithms

- Force directed layouts?
  - Again, some rogue implementations
  - **Most** graph rendering libs have their own
  - Source algorithms are various

Are we doomed?

Well, we certainly hope not.

Graphology

`graphology` is a specification and reference implementation for a robust & multipurpose JavaScript `Graph` object.

It aims at supporting various kinds of graphs with the same unified interface.

A `graphology` graph can therefore be directed, undirected or mixed and can be simple or support parallel edges.

Along with those specifications, one will also find a standard library full of graph theory algorithms and common utilities such as graph generators, layouts etc.

Installation

To install the reference implementation:

```
npm install --save graphology
```

Quick Start

```
const Graph = require('graphology');

const graph = new Graph();
```

Graphology

An Open Source specification for a robust & multipurpose `Graph` object in JavaScript.

A reference implementation.

A standard library of common algorithms.

Multipurpose

- The graph can be **directed**, **undirected** or **mixed**.
- The graph can be **simple** or **multiple** (parallel edges).
- The graph will or will not accept **self-loops**.

Use cases

- Graph analysis (compute metrics & indices...)
- Graph handling (build graphs from data, modify an already existing graph...)
- Data model for graph rendering (interactive graph visualization in the browser...)
- ...

What we won't do

- Handle graph data that does not fit in RAM.

A specification not a library

```
import Graph from 'my-custom-graphology-implementation';
import {connectedComponents} from 'graphology-components';

const graph = new Graph(...);

// Still works!
const components = connectedComponents(graph);
```

Concepts

- A node is represented by a key and can be described by attributes.
- An edge is represented by a key (that may be provided or generated) and can also be represented by attributes.
- That's it. That'a graph, no?

```
import Graph from 'graphology';

const graph = new Graph();
graph.addNode('John');
graph.addNode('Suzy');
graph.addEdge('John', 'Suzy');

graph.setNodeAttribute('Suzy', {age: 34});

graph.order // >>> 2

graph.nodes(); // >>> ['John', 'Suzy']

graph.neighbors('John'); // >>> ['Suzy']
```

Current state of the standard library

- graphology-assertions
- graphology-centrality
- graphology-components
- graphology-generators
- graphology-hits
- graphology-layout
- graphology-operators
- graphology-utils

API Design

What were the issue we encountered when designing the specifications & what decisions were taken to solve them?

#notjava

No class for nodes & edges. Only the `Graph` is a class on its own.

```
// Nope
const node = new Node('John');

// Nope
const nodeInstance = graph.addNode('John');

// Node is just a key & some optional data
graph.addNode('John', {age: 34});
```

This is more idiomatic to JavaScript, saves up some memory and makes the graph the only object able to answer question about its structure.

Default graph type

By default, the graph is mixed, accept self-loops but does not accept parallel edges.

```
var graph = new Graph();
// Same as:
var graph = new Graph(null, {
  type: 'mixed',
  multi: false
});
```

Typed constructors

However, the user still remains free to indicate the graph's type as a kind of performance hint.

```
import {MultiDirectedGraph} from 'graphology';

// In this case, the implementation is able to optimize
// for this particular type of graph.
var graph = new MultiDirectedGraph();
```

Useful error messages & hints

```
var graph = new Graph();
graph.addNodesFrom(['John', 'Jack']);
graph.addEdge('John', 'Jack');

graph.addEdge('John', 'Jack');
// This will throw an error explaining to the user that
// this edge already exists in the graph but that he can use
// a `MultiGraph` if it's really what they intended to do.
```

Optional edge keys

```
// 1: key will be generated
graph.addEdge(source, target, [attributes]);

// 2: key is explicitly provided
graph.addEdgeWithKey(key, source, target, [attributes]);
```

On key generation

Fun fact: currently, the reference implementation generates v4 uuids for the edges (you can only go so far with incremental ids...).

With a twist: ids are encoded in `base62` so you can easily copy-paste them and save up some space.

```
# 110ec58a-a0f2-4ac4-8393-c866d813b8d1
# versus:
# 1vCowaraOzD5wzfJ9Avc0g
```

Adding & merging nodes

```
// Adding a node
graph.addNode('John', {age: 34});
// Adding the same node again, will throw
graph.addNode('John', {height: 172});
>>> Error

// Explicitly merge the node
graph.mergeNode('John', {height: 172});
```

What is a key?

What should we allow as a key?

Only strings?

Should we accept references like an ES6 `Map` does?

So we just dropped the idea of references as keys and went with JavaScript `Object`'s semantics.

We need events...

```
graph.on('nodeAttributesUpdated', data => {
  console.log(`Node ${data.key} was updated!`);
});
```

...so we need getters & setters for attributes...

#notjavabutalittlebitjavanevertheless

```
// Want an attribute or attributes?
graph.getNodeAttribute(node, name);
graph.getNodeAttributes(node);

// Same for the edge, surprise!
graph.getEdgeAttribute(edge, name);

// Or if you despise keys
graph.getEdgeAttribute(source, target, name);

// Want to set an attribute?
graph.setNodeAttribute(node, name, value);
```

...so we need getters & setters for attributes...

But this doesn't mean we have to be stupid about it

```
graph.addNode('John', {counter: 12});

// Nobody should have to write this to increment a counter
graph.setNodeAttribute(
  'John',
  'counter',
  graph.getNodeAttribute('John', 'counter') + 1
);

// #OOFP
graph.updateNodeAttribute('John', 'counter', x => x + 1);
```

...and this means simpler iteration semantics!

Iteration methods only provide keys.

```
graph.addNode('John', {age: 34});
graph.addNode('Suzy', {age: 35});

graph.nodes();
// Will output
['John', 'Suzy']
// And not something strange like
[{key: 'John', attributes: {age: 34}}, ...]
// nor
[['John', {age: 34}], ...]
```

Labels & weights & ... ?

No special treatment for labels & weights etc. They are just attributes like any other.

```
import hits from 'graphology-hits';

hits.assign(graph, {
  attributes: {
    weight: 'thisIsHowICallMyWeightsDontJudgeMe'
  }
});
```

The reference implementation

Constant time vs. memory

- We don't know the precise use cases.
- So we can't aggressively optimize.
- Most common read operations should therefore run in constant time.
- This obviously means some memory overhead.

The actual data structure

Two ES6 `Map` objects to store nodes & edges (faster on recent engines).

Lazy indexation of neighborhoods. #sparsematrix

Node map

Information stored about the nodes:

- Degrees
- Attribute data
- Lazy neighbors by type

```
{
  A: {
    out: {
      B: Set{A->B},
      C: ...
    }
  },
  B: {
    in: {
      A: Set{A->B} // Same reference as above
    }
  }
}
```

Edge map

Information stored about the edges:

- Source
- Target
- Directedness
- Attribute data

I am sure someone can find better. #halp

Last issue: the case of undirected edges

How to store undirected edges?

Implicit direction given.

Two equivalent graphs may have a different memory representation.

But is this really an issue?

Should we sort the source & target keys?

Should we hash the source & target keys?

Please do read the code for precisions, it is Open Source after all...

Future roadmap

Sigma.js

- Sigma as a rendering engine with *graphology* as a model
  - More specific functional scope (rendering + interactions only)
  - No more "We need a Pagerank for this rendering engine!" nonsense

Sigma.js (community note)

- Move from the "some guy's pet project" workflow:
  - More strict and efficient workflow (PRs, review, etc...)
  - An actual transparent roadmap
  - Move the project to a Github organization

Hypergraphs?

Immutable version?

Easy to write using `immutable-js` or `mori`.

TypeScript & friends

It would be nice to have `TypeScript` or/and `flow` definitions.

Thank you!

This is all but a *Work in Progress*.