Cytoscape.js

Factsheet

- A fully featured graph library written in pure JS
- Permissive open source license (MIT)
- Designed for users first, for both frontfacing app usercases *and* developer usecases
- Highly optimised
- Dependency free
- - All modern browsers
  - CommonJS/Node.js
  - AMD/Require.js
  - jQuery
  - npm
  - Bower
  - spm
  - Meteor/Atmosphere

- Has a full suite of unit tests that can be run in the browser or the terminal
- Documentation includes live code examples, doubling as an interactive requirements specification; example graphs may also be freely modified in your browser's JS console
- Fully serialisable and deserialisable via JSON
- Uses layouts for automatically or manually positioning nodes
- Supports selectors for terse filtering and graph querying
- Uses stylesheets to separate presentation from data in a rendering agnostic manner
- Abstracted and unified touch events on top of a familiar event model
- Builtin support for standard gestures on both desktop and touch
- Chainable for convenience
- Supports functional programming patterns
- Supports set theory operations
- Includes graph theory algorithms, from BFS to PageRank
- Animatable graph elements and viewport
- Fully extendable (and extensions can be autoscaffolded for you)
- Well maintained, with only a sliver of active bug time (i.e. minimised time to bugfix)

About

Cytoscape.js is an open-source graph theory (a.k.a. network) library written in JavaScript. You can use Cytoscape.js for graph analysis and visualisation.

Cytoscape.js allows you to easily display and manipulate rich, interactive graphs. Because

Cytoscape.js allows the user to interact with the graph and the library allows the client to hook into user events, Cytoscape.js is easily integrated into your app, especially since Cytoscape.js supports both desktop browsers, like Chrome, and mobile browsers, like on the iPad. Cytoscape.js includes all the gestures you would expect out-of-the-box, including pinch-to-zoom, box selection, panning, et cetera.

Cytoscape.js also has graph analysis in mind: The library contains many useful functions in graph theory. You can use Cytoscape.js headlessly on Node.js to do graph analysis in the terminal or on a web server.

Cytoscape.js is an open-source project, and anyone is free to contribute. For more information, refer to the GitHub README.

The library was developed at the Donnelly Centre at the University of Toronto. It is the successor of Cytoscape Web.

Citation

To cite Cytoscape.js in a paper, please cite the Oxford Bioinformatics issue:

Cytoscape.js: a graph theory library for visualisation and analysis
Franz M, Lopes CT, Huck G, Dong Y, Sumer O, Bader GD

Cytoscape.js & Cytoscape

Though Cytoscape.js shares its name with Cytoscape, Cytoscape.js is not exactly the same as Cytoscape desktop. Cytoscape.js is a JavaScript library for *programmers*. It is not an app for end-users, and developers need to write code around Cytoscape.js to build graphcentric apps.

Cytoscape.js is a JavaScript library: It gives you a reusable graph widget that you can integrate with the rest of your app with your own JavaScript code. The keen members of the audience will point out that this means that Cytoscape plugins/apps — written in Java — will obviously not work in Cytoscape.js — written in Java*Script*. However, Cytoscape.js supports its own ecosystem of extensions.

We are trying to make the two projects intercompatible as possible, and we do share philosophies with Cytoscape: Graph style and data should be separate, the library should provide core functionality with extensions adding functionality on top of the library, and so on.

Funding

Funding for Cytoscape.js and Cytoscape is provided by NRNB (U.S. National Institutes of Health, National Center for Research Resources grant numbers P41 RR031228 and GM103504) and by NIH grants 2R01GM070743 and 1U41HG006623. The following organizations help develop Cytoscape:

Notation

Graph model

Cytoscape.js supports many different graph theory usecases. It supports directed graphs, undirected graphs, mixed graphs, loops, multigraphs, compound graphs (a type of hypergraph), and so on.

We are regularly making additions and enhancements to the library, and we gladly accept feature requests and pull requests.

Architecture & API

There are two components in the architecture that a developer need concern himself in order to use Cytoscape.js, the core (i.e. a graph instance) and the collection. In Cytoscape.js, the core is a developer's main entry point into the library. From the core, a developer can run layouts, alter the viewport, and perform other operations on the graph as a whole.

The core provides several functions to access elements in the graph. Each of these functions returns a collection, a set of elements in the graph. Functions are available on collections that allow the developer to filter the collection, perform operations on the collection, traverse the graph about the collection, get data about elements in the collection, and so on.

Note that a collection is immutable by default, meaning that the set of elements within a collection can not be changed. The API returns a new collection with different elements when necessary, instead of mutating the existing collection. This allows the developer to safely use set theory operations on collections, use collections functionally, and so on. Note that because a collection is just a list of elements, it is inexpensive to create new collections.

Functions

There are several types that different functions can be executed on, and the variable names used to denote these types in the documentation are outlined below:

Shorthand	Works on
`cy`	the core
`eles`	a collection of one or more elements (nodes and edges)
`ele`	a collection of a single element (node or edge)
`nodes`	a collection of one or more nodes
`node`	a collection of a single node
`edges`	a collection of one or more edges
`edge`	a collection of a single edge

`layout`	a layout
`ani`	an animation

By default, a function returns a reference back to the calling object to allow for chaining (e.g. `obj.fn1().fn2().fn3()`). Unless otherwise indicated in this documentation, a function is chainable in this manner unless a different return value is specified. This applies both to the core and to collections.

For functions that return a value, note that calling a singular — `ele`, `node`, or `edge` — function on a collection of more than one element will return the expected value for only the first element.

Position

A node's position refers to the centre point of its bounding box.

There is an important distinction to make for position: A position may be a *model* position or a *rendered* position.

A model position — as its name suggests — is the position stored in the model for an element. An element's model position remains constant, despite changes to zoom and pan.

A rendered position is an on-screen location relative to the viewport. For example, a rendered position of `{ x: 100, y: 100 }` specifies a point 100 pixels to the right and 100 pixels down from the top-left corner of the viewport. An element's rendered position naturally changes as zoom and pan changes, because the element's on-screen position in the viewport changes as zooming and panning are applied. Panning is always measured in rendered coordinates.

In this documentation, "position" refers to model position unless otherwise stated.

Elements JSON

Examples are given that outline format of the elements JSON used to load elements into Cytoscape.js:

```
cytoscape({

  container: document.getElementById('cy'),

  elements: [
    { // node n1
      group: 'nodes', // 'nodes' for a node, 'edges' for an edge
      // NB the group field can be automatically inferred for you but
specifying it
      // gives you nice debug messages if you mis-init elements
```

```
      // NB: id fields must be strings or numbers
      data: { // element data (put dev data here)
        id: 'n1', // mandatory for each element, assigned automatically on
undefined
        parent: 'nparent', // indicates the compound node parent id; not
defined => no parent
      },

      // scratchpad data (usually temp or nonserialisable data)
      scratch: {
        foo: 'bar'
      },

      position: { // the model position of the node (optional on init,
mandatory after)
        x: 100,
        y: 100
      },

      selected: false, // whether the element is selected (default false)

      selectable: true, // whether the selection state is mutable (default
true)

      locked: false, // when locked a node's position is immutable
(default false)

      grabbable: true, // whether the node can be grabbed and moved by the
user

      classes: 'foo bar' // a space separated list of class names that the
element has
    },

    { // node n2
      data: { id: 'n2' },
      renderedPosition: { x: 200, y: 200 } // can alternatively specify
position in rendered on-screen pixels
    },

    { // node n3
      data: { id: 'n3', parent: 'nparent' },
      position: { x: 123, y: 234 }
```

```
    },

    { // node nparent
      data: { id: 'nparent', position: { x: 200, y: 100 } }
    },

    { // edge e1
      data: {
        id: 'e1',
        // inferred as an edge because `source` and `target` are
specified:
        source: 'n1', // the source node id (edge comes from this node)
        target: 'n2'  // the target node id (edge goes to this node)
      }
    }
  ],

  layout: {
    name: 'preset'
  },

  // so we can see the ids etc
  style: [
    {
      selector: 'node',
      style: {
        'content': 'data(id)'
      }
    },

    {
      selector: ':parent',
      style: {
        'background-opacity': 0.6
      }
    }
  ]

});
```

Compound nodes

Compound nodes are an addition to the traditional graph model. A compound node contains a number

of child nodes, similar to how a HTML DOM element can contain a number of child elements.

Compound nodes are specified via the `parent` field in an element's `data`. Similar to the `source` and `target` fields of edges, the `parent` field is immutable: A node's parent can be specified when the node is added to the graph, and after that point, this parent-child relationship is immutable. However, you can effectively move child nodes via `eles.move()`.

As far as the API is concerned, compound nodes are treated just like regular nodes — except in explicitly compound functions like `node.parent()`. This means that traditional graph theory functions like `eles.dijkstra()` and `eles.neighborhood()` do not make special allowances for compound nodes, so you may need to make different calls to the API depending on your usecase.

For instance:

```
var a = cy.$('#a'); // assume a compound node

// the neighbourhood of `a` contains directly connected elements
var directlyConnected = a.neighborhood();

// you may want everything connected to its descendants instead
// because the descendants "belong" to `a`
var indirectlyConnected = a.add( a.descendants() ).neighborhood();
```

Getting started

This section will familiarise you with the basic steps necessary to start using Cytoscape.js.

Including Cytoscape.js

If you are using a HTML environment, then include Cytoscape.js in a `<script>` tag, e.g.:

```
<script src="cytoscape.js"></script>
```

Note that Cytoscape.js uses the dimensions of your HTML DOM element container for layouts and rendering at initialisation. Thus, it is very important to place your CSS stylesheets in the `<head>` before any Cytoscape.js-related code. Otherwise, dimensions may be sporadically reported incorrectly, resulting in undesired behaviour.

Your stylesheet may include something like this (assuming a DOM element with ID `cy` is used as the container):

```
#cy {
  width: 300px;
```

```
  height: 300px;
  display: block;
}
```

Also note that you should call `cy.resize()` when your code resizes the viewport.

To install Cytoscape.js via npm:

```
npm install cytoscape
```

To use Cytoscape.js in a CommonJS environment like Node.js:

```
var cytoscape = require('cytoscape');
```

To use Cytoscape.js with AMD/Require.js:

```
require(['cytoscape'], function(cytoscape){
  // ...
});
```

To install Cytoscape.js via Bower (in the terminal):

```
bower install cytoscape
```

To install Cytoscape.js via spm (in the terminal):

```
spm install cytoscape
```

To install Cytoscape.js via Meteor/Atmosphere (in the terminal):

```
meteor add cytoscape:cytoscape
```

Initialisation

An instance of Cytoscape.js corresponds to a graph. You can create an instance as follows:

```
var cy = cytoscape({
  container: document.getElementById('cy') // container to render in
});
```

You can pass a jQuery instance as the `container` for convenience:

```
var cy = cytoscape({
  container: $('#cy')
});
```

If you are running Cytoscape.js in Node.js or otherwise running it headlessly, you will not specify the `container` option. When running Cytoscape.js headlessly in the browser, you should specify `options.renderer.name` as `'null'` so that the default canvas renderer is not used to draw the graph. Outside of the browser (e.g. in Node.js) or if the convenience option `options.headless` is `true`, the null renderer is used by default.

Specifying basic options

For visualisation, the `container`, `elements`, `style`, and `layout` options usually should be set:

```
var cy = cytoscape({

  container: document.getElementById('cy'), // container to render in

  elements: [ // list of graph elements to start with
    { // node a
      data: { id: 'a' }
    },
    { // node b
      data: { id: 'b' }
    },
    { // edge ab
      data: { id: 'ab', source: 'a', target: 'b' }
    }
  ],

  style: [ // the stylesheet for the graph
    {
      selector: 'node',
      style: {
        'background-color': '#666',
        'label': 'data(id)'
      }
    },

    {
      selector: 'edge',
      style: {
        'width': 3,
```

```
        'line-color': '#ccc',
        'target-arrow-color': '#ccc',
        'target-arrow-shape': 'triangle'
      }
    }
  ],

  layout: {
    name: 'grid',
    rows: 1
  }

});
```

Next steps

Now that you have a core (graph) instance with basic options, explore the core API. It's your entry point to all the features in Cytoscape.js.

If you have code questions about Cytoscape.js, please feel free to post your question to Stackoverflow.

Core

The core object is your interface to a graph. It is your entry point to Cytoscape.js: All of the library's features are accessed through this object.

Initialisation

Initialisation

A graph can be created as follows:

```
var cy = cytoscape({ /* options */ });
```

You can initialise the core without any options. If you want to use Cytoscape as a visualisation, then a `container` DOM element is required, e.g.:

```
var cy = cytoscape({
  container: document.getElementById('cy')
});
```

The following sections go over the options in more detail.

Initialisation options

An instance of Cytoscape.js has a number of options that can be set on initialisation. They are outlined below with their default values.

Note that everything is optional. By default, you get an empty graph with the default stylesheet. Environments outside the browser (e.g. Node.js) are automatically set as headless for convenience.

```
var cy = cytoscape({
  // very commonly used options:
  container: undefined,
  elements: [ /* ... */ ],
  style: [ /* ... */ ],
  layout: { name: 'grid' /* , ... */ },

  // initial viewport state:
  zoom: 1,
  pan: { x: 0, y: 0 },

  // interaction options:
  minZoom: 1e-50,
  maxZoom: 1e50,
  zoomingEnabled: true,
  userZoomingEnabled: true,
  panningEnabled: true,
  userPanningEnabled: true,
  boxSelectionEnabled: false,
  selectionType: 'single',
  touchTapThreshold: 8,
  desktopTapThreshold: 4,
  autolock: false,
  autoungrabify: false,
  autounselectify: false,

  // rendering options:
  headless: false,
  styleEnabled: true,
  hideEdgesOnViewport: false,
  hideLabelsOnViewport: false,
  textureOnViewport: false,
  motionBlur: false,
  motionBlurOpacity: 0.2,
  wheelSensitivity: 1,
  pixelRatio: 'auto',
```

```
  renderer: { /* ... */ }
});
```

Very commonly used options

`container` : A HTML DOM element in which the graph should be rendered. This is optional if Cytoscape.js is run headlessly or if you initialise using jQuery (in which case your jQuery object already has an associated DOM element).

`elements` : An array of elements specified as plain objects. For convenience, this option can alternatively be specified as a promise that resolves to the elements JSON.

`style` : The stylesheet used to style the graph. For convenience, this option can alternatively be specified as a promise that resolves to the stylesheet.

`layout` : A plain object that specifies layout options. Which layout is initially run is specified by the `name` field. Refer to a layout's documentation for the options it supports. If you want to specify your node positions yourself in your elements JSON, you can use the `preset` layout — by default it does not set any positions, leaving your nodes in their current positions (e.g. specified in `options.elements` at initialisation time).

Initial viewport state

`zoom` : The initial zoom level of the graph. Make sure to disable viewport manipulation options, such as `fit`, in your layout so that it is not overridden when the layout is applied. You can set **`options.minZoom`** and **`options.maxZoom`** to set restrictions on the zoom level.

`pan` : The initial panning position of the graph. Make sure to disable viewport manipulation options, such as `fit`, in your layout so that it is not overridden when the layout is applied.

Interaction options

`minZoom` : A minimum bound on the zoom level of the graph. The viewport can not be scaled smaller than this zoom level.

`maxZoom` : A maximum bound on the zoom level of the graph. The viewport can not be scaled larger than this zoom level.

`zoomingEnabled` : Whether zooming the graph is enabled, both by user events and programmatically.

`userZoomingEnabled` : Whether user events (e.g. mouse wheel, pinch-to-zoom) are allowed to zoom the graph. Programmatic changes to zoom are unaffected by this option.

`panningEnabled` : Whether panning the graph is enabled, both by user events and programmatically.

`userPanningEnabled` : Whether user events (e.g. dragging the graph background) are allowed to pan the graph. Programmatic changes to pan are unaffected by this option.

`boxSelectionEnabled` : Whether box selection (i.e. drag a box overlay around, and release it to select) is enabled. If enabled, the user must taphold to pan the graph.

`selectionType` : A string indicating the selection behaviour from user input. For `'additive'`, a new selection made by the user adds to the set of currently selected elements. For `'single'`, a new selection made by the user becomes the entire set of currently selected elements (i.e. the previous elements are unselected).

`touchTapThreshold` & **`desktopTapThreshold`** : A nonnegative integer that indicates the maximum allowable distance that a user may move during a tap gesture, on touch devices and desktop devices respectively. This makes tapping easier for users. These values have sane defaults, so it is not advised to change these options unless you have very good reason for doing so. Larger values will almost certainly have undesirable consequences.

`autoungrabify` : Whether nodes should be ungrabified (not grabbable by user) by default (if `true`, overrides individual node state).

`autolock` : Whether nodes should be locked (not draggable at all) by default (if `true`, overrides individual node state).

`autounselectify` : Whether nodes should be unselectified (immutable selection state) by default (if `true`, overrides individual element state).

Rendering options

`headless` : A convenience option that initialises the instance to run headlessly. You do not need to set this in environments that are implicitly headless (e.g. Node.js). However, it is handy to set `headless: true` if you want a headless instance in a browser.

`styleEnabled` : A boolean that indicates whether styling should be used. For headless (i.e. outside the browser) environments, display is not necessary and so neither is styling necessary — thereby speeding up your code. You can manually enable styling in headless environments if you require it for a special case. Note that it does not make sense to disable style if you plan on rendering the graph.

`hideEdgesOnViewport` : When set to `true`, the renderer does not render edges while the viewport is being manipulated. This makes panning, zooming, dragging, et cetera more responsive for large graphs.

`hideLabelsOnViewport` : When set to `true`, the renderer does not render labels while the viewport is being manipulated. This makes panning, zooming, dragging, et cetera more responsive for large graphs.

`textureOnViewport` : When set to `true`, the renderer uses a texture (if supported) during panning

and zooming instead of drawing the elements, making large graphs more responsive.

`motionBlur` : When set to `true`, the renderer will use a motion blur effect to make the transition between frames seem smoother. This can significantly increase the perceived performance for a large graphs.

`motionBlurOpacity` : When `motionBlur: true`, this value controls the opacity of motion blur frames. Higher values make the motion blur effect more pronounced.

`wheelSensitivity` : Changes the scroll wheel sensitivity when zooming. This is a multiplicative modifier. So, a value between 0 and 1 reduces the sensitivity (zooms slower), and a value greater than 1 increases the sensitivity (zooms faster).

`pixelRatio` : Overrides the screen pixel ratio with a manually set value (`1.0` recommended, if set). This can be used to increase performance on high density displays by reducing the effective area that needs to be rendered, though this is much less necessary on more recent browser releases. If you want to use the hardware's actual pixel ratio, you can set `pixelRatio: 'auto'` (default).

`renderer` : A plain object containing options for the renderer to be used. The `options.renderer.name` field specifies which renderer is used. You need not specify anything for the `renderer` option, unless you want to specify one of the rendering options below:

- **`renderer.name`** : The name of the renderer to use. By default, the `'canvas'` renderer is used. If you build and register your own renderer, then you can specify its name here.

cy.add(eleObj)
Add a specified element to the graph.

- eleObj
  A plain object that specifies the element.

cy.add(eleObjs)
Add the specified elements to the graph.

- eleObjs
  An array of elements specified by plain objects.

cy.add(eles)
Add the specified elements to the graph.

- eles
  A collection of elements.

Details

If plain element objects are used, then the same format used at initialisation must be followed.

If a collection of existing elements is specified to a different core instance, then copies of those elements are added, which allows for elements to be effectively transferred between instances of Cytoscape.js.

Examples

Add a node from a plain object.

```
cy.add({
    group: "nodes",
    data: { weight: 75 },
    position: { x: 200, y: 200 }
});
```

Add nodes and edges to the graph as plain objects:

```
// can use reference to eles later
var eles = cy.add([
  { group: "nodes", data: { id: "n0" }, position: { x: 100, y: 100 } },
  { group: "nodes", data: { id: "n1" }, position: { x: 200, y: 200 } },
  { group: "edges", data: { id: "e0", source: "n0", target: "n1" } }
]);
```

cy.remove()
Remove elements from the graph and return them.

cy.remove(eles)
Remove the specified elements.

cy.remove(selector)
Remove elements in the graph matching the specified selector.

- selector
  Elements matching this selector are removed.

Details

Note that removing a node necessarily removes its connected edges.

Though the elements specified to this function are removed from the graph, they may still exist in memory. However, almost all functions will not work on removed elements. For example, the `eles.neighborhood()` function will fail for a removed element: An element outside of the context of the graph can not have a neighbourhood defined. In effect, removed elements just exist so you can restore them back to the originating core instance or to a new instance.

Examples

Remove an element:

```
var j = cy.$("#j");
cy.remove( j );
```

Remove a collection:

```
var collection = cy.elements("node[weight > 50]");
cy.remove( collection );
```

Remove elements matching a selector:

```
cy.remove("node[weight > 50]"); // remove nodes with weight greater than
50
```

cy.collection()
Return a new collection.

cy.collection(selector)
Get a collection from elements in the graph matching the specified selector.

- selector
  Elements matching this selector are in the returned collection.

cy.collection(elesArray)
Get a collection from an array of elements.

- elesArray
  The elements in this array are in the returned collection.

Details

This function is useful for building up collections.

Examples

Keep a collection of nodes that have been clicked:

```
var collection = cy.collection();
cy.nodes().on("click", function(){
  collection = collection.add(this);
});
```

cy.getElementById()
Get an element from its ID in a very performant way.

cy.getElementById(id)

- id
  The ID of the element to get.

Examples

```
cy.getElementById('j');
```

cy.$(selector)
Get elements in the graph matching the specified selector.

- selector
  The selector the elements should match.

cy.elements(selector)
Get elements in the graph matching the specified selector.

- selector
  The selector the elements should match.

cy.nodes(selector)
Get nodes in the graph matching the specified selector.

- selector
  The selector the nodes should match.

cy.edges(selector)
Get edges in the graph matching the specified selector.

- selector
  The selector the edges should match.

cy.filter(selector)
Get elements in the graph matching the specified selector.

- selector
  The selector the elements should match.

cy.filter(function(i, ele))
Get elements in the graph matching the specified filter function.

- function(i, ele)
  The filter function that returns true for elements that should be returned.

- i
  The counter used for iteration over the elements in the graph.

- ele
  The current element under consideration for filtering (also accessible as `this`).

Details

If no elements in the graph match the selector, an empty collection is returned.

The function `cy.$()` acts as an alias to `cy.filter()`: It's just convenient to save you typing. It is analogous to the jQuery `$` alias used to search the document

Examples

Get nodes with weight greater than 50:

```
cy.nodes("[weight>50]");
```

Get edges with source node `n0`:

```
cy.edges("[source='j']");
```

Get all nodes and edges with weight greater than 50:

```
cy.elements("[weight>50]");
cy.filter("[weight>50]"); // works the same as the above line
```

Get nodes with weight greater than 50 with a filter function:

```
cy.filter(function(i, element){
  if( element.isNode() && element.data("weight") > 50 ){
    return true;
  }
  return false;
});
```

cy.batch() et al
Allow for manipulation of elements without triggering multiple style calculations or multiple redraws.

cy.batch(function())

- function()
  A callback within which you can make batch updates to elements.

cy.startBatch()

Starts batching manually (useful for asynchronous cases).

cy.endBatch()
Ends batching manually (useful for asynchronous cases).

Details

Normally, when you modify elements, each modification can trigger a style calculation and a redraw — depending on timing for a redraw. For example, the following will cause two style calculations and at least one draw:

```
cy.$('#j')
  .data('weight', '70')   // style update
  .addClass('funny')      // style update AGAIN
  .removeClass('serious') // style update YET AGAIN

  // at least 1 redraw here
  // possibly 3 total depending on speed of above operations
  // (for one ele almost certainly 1 redraw, but consider many eles)
;
```

This is not a problem for a handful of operations on a handful of elements, but for many operations on many elements you end up with redundant style calculations and probably redundant redraws. In the worst case, you have `eles.length * numOps` style updates and redraws — and both style updates and redraws can be expensive. In the worst case when using `cy.batch()`, you limit the style updates to `eles.length` and you limit the redraws to just one.

Thus, this function is useful for making many changes to elements at once. When the specified callback function is complete, only elements that require it have their style updated and the renderer makes at most a single redraw.

This makes for very efficient modifications to elements, but it has some caveats. While inside the batch callback,

- you can not reliably read element style or dimensions (it may have changed, or computed values may be out of date),
- you probably do not want to use `eles.style()` et cetera because they force a style bypass rather than a recalculation.

Examples

Synchronous style:

```
cy.batch(function(){
  cy.$('#j')
```

```
    .data('weight', '70')
    .addClass('funny')
    .removeClass('serious')
  ;
});
```

Asynchronous style:

```
cy.startBatch();

cy.$('#j')
  .data('weight', '70')
  .addClass('funny')
  .removeClass('serious')
;

cy.endBatch();
```

cy.destroy()
A convenience function to explicitly destroy the instance.

Details

The `cy.destroy()` function is not necessary but can be convenient in some cases. It cleans up references and rendering loops such that the memory used by an instance can be garbage collected.

If you remove the container DOM element from the page, then the instance is cleaned up automatically. Similarly, calling `cy.destroy()` does this cleanup and removes all the container's children from the page.

When running Cytoscape.js headlessly, using `cy.destroy()` is necessary only if you've explicitly enabled style functionality.

To drop the memory used by an instance, it is necessary to drop all of your own references to that instance so it can be garbage collected.

cy.scratch()
Extension function: This function is intended for use in extensions.
Set or get scratchpad data, where temporary or non-JSON data can be stored. App-level scratchpad data should use namespaces prefixed with underscore, like `'_foo'`. This is analogous to the more common `ele.scratch()` but for graph global data.

cy.scratch()
Get the entire scratchpad object for the core.

cy.scratch(namespace)
Get the scratchpad at a particular namespace.

- namespace
  A namespace string.

- namespace
  A namespace string.

- value
  The value to set at the specified namespace.

cy.removeScratch()
Extension function: This function is intended for use in extensions.
Remove scratchpad data. You should remove scratchpad data only at your own namespaces. This is analogous to the more common `ele.removeScratch()` but for graph global data.

cy.removeScratch(namespace)
Remove the scratchpad data at a particular namespace.

- namespace
  A namespace string.

cy.on(events [, selector] [, data], function(event))
- events
  A space separated list of event names.

- selector [optional]
  A selector to specify elements for which the handler is triggered.

- data [optional]
  A plain object which is passed to the handler in the event object argument.

- function(event)
  The handler function that is called when one of the specified events occurs.

  - event
    The event object.

Examples

Bind to events that bubble up from elements matching the specified `node` selector:

```
cy.on('tap', 'node', { foo: 'bar' }, function(evt){
  console.log( evt.data.foo ); // 'bar'
```

```
  var node = evt.cyTarget;
  console.log( 'tapped ' + node.id() );
});
```

Bind to all tap events that the core receives:

```
cy.on('tap', function(event){
  // cyTarget holds a reference to the originator
  // of the event (core or element)
  var evtTarget = event.cyTarget;

  if( evtTarget === cy ){
      console.log('tap on background');
  } else {
    console.log('tap on some element');
  }
});
```

cy.promiseOn()
Aliases: `cy.pon()`
Get a promise that is resolved with the first of any of the specified events triggered on the graph.

cy.promiseOn(events [, selector])

- events
  A space separated list of event names.

- selector [optional]
  A selector to specify elements for which the handler is triggered.

Examples

```
cy.pon('tap').then(function( event ){
  console.log('tap promise fulfilled');
});
```

cy.one()
Bind to events that occur in the graph, and trigger the handler only once.

cy.one(events [, selector] [, data], function(event))

- events
  A space separated list of event names.

- selector [optional]
  A selector to specify elements for which the handler is triggered.

- data [optional]
  A plain object which is passed to the handler in the event object argument.

- function(event)
  The handler function that is called when one of the specified events occurs.

  - event
    The event object.

Examples

```
cy.one('tap', 'node', function(){
  console.log('tap!');
});

cy.$('node').eq(0).trigger('tap'); // tap!
cy.$('node').eq(1).trigger('tap'); // nothing b/c already tapped
```

cy.off()
Aliases: `cy.unbind()`, `cy.unlisten()`, `cy.removeListener()`
Remove event handlers.

- events
  A space separated list of event names.

- selector [optional]
  The same selector used to bind to the events.

- handler [optional]
  A reference to the handler function to remove.

Examples

For all handlers:

```
cy.on("tap", function(){ /* ... */ });

// unbind all tap handlers, including the one above
cy.off("tap");
```

For a particular handler:

```
var handler = function(){
  console.log("called handler");
};
```

```
cy.on("tap", handler);

var otherHandler = function(){
  console.log("called other handler");
};
cy.on("tap", otherHandler);

// just unbind handler
cy.off("tap", handler);
```

cy.trigger()
Aliases: `cy.emit()`
Trigger one or more events.

cy.trigger(events [, extraParams])

- events
  A space separated list of event names to trigger.

- extraParams [optional]
  An array of additional parameters to pass to the handler.

Examples

```
cy.bind('tap', function(evt, f, b){
  console.log('tap', f, b);
});

cy.trigger('tap', ['foo', 'bar']);
```

cy.initrender()
Extension function: This function is intended for use in extensions.
Get whether the initial render event has occurred (useful for extensions etc).

Details

This function returns whether the `initrender` event has occurred on the graph, meaning that the renderer has drawn the graph at least once. This is useful when you need to grab image data from the core, as this function will let you know whether that data is available yet: You can not grab the graph scene if it has not yet been rendered.

cy.onRender()
Run a handler function every time a frame is rendered.

cy.onRender(function())

- function()

The handler function to call on each frame.

Examples

```
cy.onRender(function(){
  console.log('frame rendered');
});
```

cy.offRender()
Remove handlers function bound via `cy.onRender()`.

cy.offRender([handler])

- handler [optional]
  A reference to the handler function to remove. All handlers are removed if this is unspecified.

Examples

```
var handler;
cy.onRender(handler = function(){
  console.log('frame rendered');
});

cy.offRender( handler );
```

cy.ready()
Run a callback as soon as the graph becomes ready (i.e. data loaded and initial layout completed). If the graph is already ready, then the callback is called immediately. If data is loaded synchronously and the layout used is discrete/synchronous/unanimated/unspecified, then you don't need `cy.ready()`.

cy.ready(function(event))

- function(event)
  The callback run as soon as the graph is ready, inside which `event.cy` refers to the core (`cy`).
  - event
    The `ready` event.

cy.center()
Centre on all elements in the graph.

cy.center(eles)
Centre on the specified elements.

- eles
  The collection to centre upon.

Details

If an empty collection or no collection is specified, then the graph is centred on all nodes and edges in the graph.

Examples

Centre the graph on node `j`:

```
var j = cy.$("#j");
cy.center( j );
```

cy.fit()
Pan and zooms the graph to fit to a collection.

cy.fit()
Fit to all elements in the graph.

cy.fit([eles] [, padding])
Fit to the specified elements.

- eles [optional]
  The collection to fit to.

- padding [optional]
  An amount of padding (in pixels) to have around the graph

Details

If an empty collection or no collection is specified, then the graph is fit to all nodes and edges in the graph.

Examples

Fit the graph on nodes `j` and `e`:

```
cy.fit( cy.$('#j, #e') );
```

cy.reset()
Resets the zoom and pan.

Details

This resets the viewport to the origin (0, 0) at zoom level 1.

Examples

```
setTimeout( function(){
    cy.pan({ x: 50, y: -100 });
}, 1000 );

setTimeout( function(){
    cy.zoom( 2 );
}, 2000 );

setTimeout( function(){
    cy.reset();
}, 3000 );
```

cy.pan()
Get or set the panning position of the graph.

cy.pan()
Get the current panning position.

cy.pan(renderedPosition)
Set the current panning position.

- renderedPosition
  The rendered position to pan the graph to.

Details

This function pans the graph viewport origin to the specified rendered pixel position.

Examples

Pan the graph to (100, 100) rendered pixels.

```
cy.pan({
  x: 100,
  y: 100
});

console.log( cy.pan() ); // prints { x: 100, y: 100 }
```

cy.panBy()
Relatively pan the graph by a specified rendered position vector.

cy.panBy(renderedPosition)

Details

This function shifts the viewport relatively by the specified position in rendered pixels. That is, specifying a shift of 100 to the right means a translation of 100 on-screen pixels to the right.

Examples

Pan the graph 100 pixels to the right.

```
cy.panBy({
  x: 100,
  y: 0
});
```

cy.panningEnabled()
Get or set whether panning is enabled.

cy.panningEnabled()
Get whether panning is enabled.

cy.panningEnabled(bool)
Set whether panning is enabled.

- bool
  A truthy value enables panning; a falsey value disables it.

Examples

Enable:

```
cy.panningEnabled( true );
```

Disable:

```
cy.panningEnabled( false );
```

cy.userPanningEnabled()
Get whether user panning is enabled.

cy.userPanningEnabled(bool)
Set whether user panning is enabled.

- bool
  A truthy value enables user panning; a falsey value disables it.

Examples

Enable:

```
cy.userPanningEnabled( true );
```

Disable:

```
cy.userPanningEnabled( false );
```

cy.zoom()
Get the zoom level.

cy.zoom(options)
Set the zoom level.

- level
  The zoom level to set.
- position
  The position about which to zoom.
- renderedPosition
  The rendered position about which to zoom.

Details

The zoom level must be a positive number. Zoom levels that are not numbers are ignored; zoom levels that are numbers but outside of the range of valid zoom levels are considered to be the closest, valid zoom level.

When zooming about a point via `cy.zoom( options )`, the options are defined as follows.

For zooming about a rendered position (i.e. a position on-screen):

```
cy.zoom({
  level: 2.0, // the zoom level
  renderedPosition: { x: 100, y: 100 }
});
```

For zooming about a model position:

```
cy.zoom({
  level: 2.0, // the zoom level
  position: { x: 0, y: 0 }
});
```

For obvious reasons, you can zoom about a position or a rendered position but not both. You should specify only one of `options.position` or `options.renderedPosition`.

Examples

Zoom in to factor 2

```
cy.zoom(2);
```

Zoom in to the minimum zoom factor

```
cy.zoom(0); // 0 is outside of the valid range and
            // its closest valid level is the min
```

Zoom in to the maximum zoom factor

```
cy.zoom(1/0); // infinity is outside of the valid range and
              // its closest valid level is the max
```

Zoom about a node

```
var pos = cy.nodes("#j").position();
cy.zoom({
  level: 1.5,
  position: pos
});
```

cy.zoomingEnabled()
Get or set whether zooming is enabled.

cy.zoomingEnabled()
Get whether zooming is enabled.

cy.zoomingEnabled(bool)
Set whether zooming is enabled.

- bool
  A truthy value enables zooming; a falsey value disables it.

Examples

Enable:

```
cy.zoomingEnabled( true );
```

Disable:

```
cy.zoomingEnabled( false );
```

cy.userZoomingEnabled()
Get whether user zooming is enabled.

cy.userZoomingEnabled(bool)
Set whether user zooming is enabled.

- bool
  A truthy value enables user zooming; a falsey value disables it.

Examples

Enable:

```
cy.userZoomingEnabled( true );
```

Disable:

```
cy.userZoomingEnabled( false );
```

cy.minZoom()
Get the minimum zoom level.

cy.minZoom(zoom)
Set the minimum zoom level.

- zoom
  The new minimum zoom level to use.

cy.maxZoom()
Get or set the maximum zoom level.

cy.maxZoom()
Get the maximum zoom level.

cy.maxZoom(zoom)
Set the maximum zoom level.

- zoom
  The new maximum zoom level to use.

cy.viewport()

Set the viewport state (pan & zoom) in one call.

cy.viewport(zoom, pan)

- zoom
  The zoom level to set.

- pan
  The pan to set (a rendered position).

Examples

```
cy.viewport({
  zoom: 2,
  pan: { x: 100, y: 100 }
});
```

cy.boxSelectionEnabled()
Get or set whether box selection is enabled. If enabled along with panning, the user must hold one of shift, control, meta, or alt down to initiate box selection.

cy.boxSelectionEnabled()
Get whether box selection is enabled.

cy.boxSelectionEnabled(bool)
Set whether box selection is enabled.

- bool
  A truthy value enables box selection; a falsey value disables it.

Examples

Enable:

```
cy.boxSelectionEnabled( true );
```

Disable:

```
cy.boxSelectionEnabled( false );
```

cy.height()
Get the on-screen height of the viewport in pixels.

cy.extent()
Get the extent of the viewport, a bounding box in model coordinates that lets you know what model positions are visible in the viewport.

Details

This function returns a plain object bounding box with format `{ x1, y1, x2, y2, w, h }`.

cy.autolock()
Get or set whether nodes are automatically locked (i.e. if `true`, nodes are locked despite their individual state).

cy.autolock()
Get whether autolocking is enabled.

cy.autolock(bool)
Set whether autolocking is enabled.

- bool
  A truthy value enables autolocking; a falsey value disables it.

Examples

Enable:

```
cy.autolock( true );
```

Disable:

```
cy.autolock( false );
```

cy.autoungrabify()
Get or set whether nodes are automatically ungrabified (i.e. if `true`, nodes are ungrabbale despite their individual state).

cy.autoungrabify()
Get whether autoungrabifying is enabled.

cy.autoungrabify(bool)
Set whether autoungrabifying is enabled.

- bool
  A truthy value enables autoungrabbifying; a falsey value disables it.

Examples

Enable:

```
cy.autoungrabify( true );
```

Disable:

```
cy.autoungrabify( false );
```

cy.autounselectify()
Get or set whether nodes are automatically unselectified (i.e. if `true`, nodes are unselectable despite their individual state).

cy.autounselectify()
Get whether autounselectifying is enabled.

cy.autounselectify(bool)
Set whether autounselectifying is enabled.

- bool
  A truthy value enables autounselectifying; a falsey value disables it.

Examples

Enable:

```
cy.autounselectify( true );
```

Disable:

```
cy.autounselectify( false );
```

Details

This function forces the renderer to draw a new frame. It is useful for very specific edgecases, such as in certain UI extensions, but it should not be needed for most developers.

cy.resize()
Aliases: `cy.invalidateDimensions()`
Force the renderer to recalculate the viewport bounds.

Details

If your code resizes the graph's dimensions or position (i.e. by changing the style of the HTML DOM element that holds the graph, or by changing the DOM element's position in the DOM tree), you will want to call `cy.resize()` to have the graph resize and redraw itself.

If tapping in the graph is offset rather than at the correct position, then a call to `cy.resize()` is necessary.

Cytoscape.js can not automatically monitor the bounding box of the viewport, as querying the DOM for those dimensions can be expensive. Although `cy.resize()` is automatically called for you on the `window`'s `resize` event, there is no `resize` or `style` event for arbitrary DOM elements.

cy.animate()
Animate the viewport.

- zoom
  A zoom level to which the graph will be animated.
- pan
  A panning position to which the graph will be animated.
- panBy
  A relative panning position to which the graph will be animated.
- fit
  An object containing fitting options from which the graph will be animated.
  - eles
    Elements or a selector to which the viewport will be fitted.
  - padding
    Padding to use with the fitting.
- center
  An object containing centring options from which the graph will be animated.
  - eles
    Elements or a selector to which the viewport will be centred.
- duration
  The duration of the animation in milliseconds.
- queue
  A boolean indicating whether to queue the animation.
- complete
  A function to call when the animation is done.
- step
  A function to call each time the animation steps.
- easing
  A `transition-timing-function` easing style string that shapes the animation progress curve.

Examples

Manual pan and zoom:

```
cy.animate({
  pan: { x: 100, y: 100 },
  zoom: 2
}, {
  duration: 1000
});
```

Fit to elements:

```
var j = cy.$('#j');

cy.animate({
  fit: {
    eles: j,
    padding: 20
  }
}, {
  duration: 1000
});
```

cy.animation()
Get an animation of the viewport.

cy.animation(options)

- zoom
  A zoom level to which the graph will be animated.
- pan
  A panning position to which the graph will be animated.
- panBy
  A relative panning position to which the graph will be animated.
- fit
  An object containing fitting options from which the graph will be animated.
  - eles
    Elements or a selector to which the viewport will be fitted.
  - padding
    Padding to use with the fitting.

- center
  An object containing centring options from which the graph will be animated.

  - eles
    Elements or a selector to which the viewport will be centred.

- duration
  The duration of the animation in milliseconds.

- easing
  A `transition-timing-function` easing style string that shapes the animation progress curve.

cy.delay()
Add a delay between animations for the viewport.

cy.delay(duration, complete)

- duration
  How long the delay should be in milliseconds.

- complete
  A function to call when the delay is complete.

Examples

```
cy
  .animate({
    fit: { eles: '#j' }
  })

  .delay(1000)

  .animate({
    fit: { eles: '#e' }
  })
;
```

cy.delayAnimation()
Get a delay animation of the viewport.

cy.delayAnimation(duration)

- duration
  How long the delay should be in milliseconds.

cy.stop()

Stop all viewport animations that are currently running.

cy.stop(clearQueue, jumpToEnd)

- clearQueue
  A boolean, indicating whether the queue of animations should be emptied.

- jumpToEnd
  A boolean, indicating whether the currently-running animations should jump to their ends rather than just stopping midway.

Examples

```
cy.animate({
  fit: { eles: '#j' }
}, { duration: 2000 });

// stop in the middle
setTimeout(function(){
  cy.stop();
}, 1000);
```

cy.clearQueue()
Remove all queued animations for the viewport.

cy.layout(options)

- options
  The layout options.

Details

For layouts included with Cytoscape.js, you can find their options documented in the Layouts section.
For external layouts, please refer to their accompanying documentation.

An analogue to run a layout on a subset of the graph exists as `eles.layout()`.

Examples

Run the grid layout:

```
cy.layout({ name: 'grid' });
```

cy.makeLayout()
Aliases: `cy.createLayout()`
Get a new layout, which can be used to algorithmically position the nodes in the graph.

cy.makeLayout(options)

- options
  The layout options.

You must specify `options.name` with the name of the layout you wish to use.

This function creates and returns a layout object. You may want to keep a reference to the layout for more advanced usecases, such as running multiple layouts simultaneously.

Note that you must call `layout.run()` in order for it to affect the graph.

An analogue to make a layout on a subset of the graph exists as `eles.makeLayout()`.

Examples

```
var layout = cy.makeLayout({
  name: 'random'
});

layout.run();
```

cy.style()
Get the current style object.

cy.style(stylesheet)
Assign a new stylesheet to replace the existing one.

- stylesheet
  Either a `cytoscape.stylesheet()` object, a string stylesheet, or a JSON stylesheet (the same formats accepted for `options.style` at initialisation).

Details

You can use this function to gain access to the visual style (stylesheet) after initialisation. This is useful if you need to change the entire stylesheet at runtime.

Sets a new style by reference:

```
// here a string stylesheet is used, but you could also use json or a
cytoscape.stylesheet() object
var stringStylesheet = 'node { background-color: cyan; }';
cy.style( stringStylesheet );
```

Set an entirely new style to the graph, specifying selectors and style properties via function calls:

```
cy.style()
  .resetToDefault() // start a fresh default stylesheet

  // and then define new styles
  .selector('node')
      .style('background-color', 'magenta')

  // ...

  .update() // update the elements in the graph with the new style
;
```

You can also add to the existing stylesheet:

```
cy.style()
  .selector('node')
    .style({
      'background-color': 'yellow'
    })

  .update() // update the elements in the graph with the new style
;
```

You can also set the style from plain JSON:

```
cy.style()
  .fromJson([
    {
      selector: 'node',
      style: {
        'background-color': 'red'
      }
    }

    // , ...
  ])

  .update() // update the elements in the graph with the new style
;
```

You can also set the style from a style string (that you would probably pull from a file on your server):

```
cy.style()
```

```
  .fromString('node { background-color: blue; }')

  .update() // update the elements in the graph with the new style
;
```

You can also get the current style as JSON:

```
var styleJson = cy.style().json();
var serializedJson = JSON.stringify( styleJson );
```

cy.png(options)

- bg
  The background colour of the image (transparent by default).
- full
  Whether to export the current viewport view (`false`, default) or the entire graph (`true`).
- scale
  This value specifies a positive number that scales the size of the resultant image.
- maxWidth
  Specifies the scale automatically in combination with `maxHeight` such that the resultant image is no wider than `maxWidth`.
- maxHeight
  Specifies the scale automatically in combination with `maxWidth` such that the resultant image is no taller than `maxHeight`.

Examples

```
var png64 = cy.png();

// put the png data in an img tag
$('#png-eg').attr('src', png64);
```

Example image tag:

cy.jpg(options)

- bg
  The background colour of the image (white by default).
- full
  Whether to export the current viewport view (`false`, default) or the entire graph (`true`).
- scale

This value specifies a positive number that scales the size of the resultant image.

- maxWidth
  Specifies the scale automatically in combination with `maxHeight` such that the resultant image is no wider than `maxWidth`.

- maxHeight
  Specifies the scale automatically in combination with `maxWidth` such that the resultant image is no taller than `maxHeight`.

Details

The JPEG format is lossy, whereas PNG is not. This means that `cy.jpg()` is useful for cases where filesize is more important than pixel-perfect images. JPEG compression will make your images (especially edge lines) blurry and distorted.

Examples

```
var jpg64 = cy.jpg();

// put the png data in an img tag
$('#jpg-eg').attr('src', jpg64);
```

Example image tag:

cy.json()
Export the graph as JSON.

cy.json(cyJson)
Import the graph as JSON, updating only the fields specified.

- cyJson
  The object with the fields corresponding to the states that should be changed.

Details

This function returns the same object that is used for initialisation. You will find this function useful if you would like to save the entire state of the graph, either for your own purposes or for future restoration of that graph state.

This function can also be used to set graph state as in `cy.json( cyJson )`, where each field in `cyJson` is to be mutated in the graph. For each field defined in `cyJson`, `cy` is diffed and updated to match with the corresponding events emitted. This allows for declarative changes on the graph to be made.

For `cy.json( cyJson )`, all mutable initialisation options are supported.

When setting `cy.json({ elements: ... })`

- the included elements are mutated as specified (i.e. as they would be by `ele.json( eleJson )`),
- the included elements not in the graph are added, and
- the not included elements are removed from the graph.

Examples

```
console.log( cy.json() );
```

```
cy.json({
  zoom: 2
});
```

Collection

A collection contains a set of nodes and edges. Calling a function applies the function to all elements in the collection. When reading values from a collection, `eles.data()` for example, the value of the first element in the collection is returned. This follows the jQuery convention. For example:

```
var weight = cy.nodes().data("weight");

console.log( cy.nodes()[0].data("weight") + ' == ' + weight ); // weight
is the first ele's weight
```

You can insure that you're reading from the element you want by using a selector to narrow down the collection to one element (i.e. `eles.size() === 1`) or the `eles.eq()` function.

Details

This function removes the calling elements from the graph. The elements are not deleted — they still exist in memory — but they are no longer in the graph.

A removed element just exists to be added back to its originating core instance or some other core instance. A removed element is not functional, because it is no longer a part of the graph: Nothing really makes sense for it anymore outside of the context of a graph. It merely exists in this limbo state so you can later add it back to some core instance.

Examples

Remove selected elements:

```
cy.$(':selected').remove();
```

ele.inside()
Get whether the element is inside the graph (i.e. not removed).

eles.restore()
Put removed elements back into the graph.

Details

This function puts back elements in the graph that have been removed. It will do nothing if the elements are already in the graph.

An element can not be restored if its ID is the same as an element already in the graph. You should specify an alternative ID for the element you want to add in that case.

Examples

```
// remove selected elements
var eles = cy.$(':selected').remove();

// ... then some time later put them back
eles.restore();
```

eles.clone()
Aliases: `eles.copy()`
Get a new collection containing clones (i.e. copies) of the elements in the calling collection.

eles.move() et al
Effectively move edges to different nodes or move nodes to different parent node. The modified (actually new) elements are returned.

edges.move(location)
Move edges to different nodes.

- location
  Where the edges are moved. You can specify a new source, a new target, or both.
  - source
    The ID of the new source node.
  - target
    The ID of the new target node.

nodes.move(location)
Move nodes to different parent node.

- location

Where the nodes are moved.

- parent
  The ID of the new parent node (use `null` for no parent).

Details

Note that this function does not really move the elements. That's not possible in the semantics of a graph. Instead, this function

- gets JSON copies of the elements,
- removes the original elements,
- modifies the JSON copies as specified, and
- adds new elements from the JSON copies and restores relationships (in the case of compound node descendants and connected edges).

This creates the same effect as though the elements have been moved while maintaining the correct semantics for a graph.

Examples

Move an edge:

```
cy.$('#ej').move({
  target: 'g'
})
```

eles.on(events [, selector] [, data], function(event))

- events
  A space separated list of event names.

- selector [optional]
  A delegate selector to specify child elements for which the handler is triggered.

- data [optional]
  A plain object which is passed to the handler in the event object argument.

- function(event)
  The handler function that is called when one of the specified events occurs.

  - event
    The event object.

Defails

In the handler function, `this` references the originally bound object, and `evt.cyTarget` references

the target of the event.

Examples

```
cy.on('tap', function(evt){
  console.log( 'tap ' + evt.cyTarget.id() );
});
```

eles.promiseOn()
Aliases: `eles.pon()`
Get a promise that is resolved with the first of any of the specified events triggered on any of the elements in the collection.

eles.promiseOn(events [, selector])

- events
  A space separated list of event names.

- selector [optional]
  A selector to specify elements for which the handler is triggered.

Examples

```
cy.$('#j').pon('tap').then(function( event ){
  console.log('tap promise fulfilled');
});
```

eles.one()
Bind a callback function that is triggered once per event per element.

eles.one(events [, selector] [, data], function(event))

- events
  A space separated list of event names.

- selector [optional]
  A delegate selector to specify child elements for which the handler is triggered.

- data [optional]
  A plain object which is passed to the handler in the event object argument.

- function(event)
  The handler function that is called when one of the specified events occurs.

  - event
    The event object.

Details

For each event specified to this function, the handler function is triggered once per element. This is useful for one-off events that occur on each element in the calling collection once.

The semantics is a bit more complicated for compound nodes where a delegate selector has been specified: Note that the handler is called once per element in the *calling collection*, and the handler is triggered by matching descendant elements.

Examples

```
cy.$('node').one('tap', function(e){
  var ele = e.cyTarget;
  console.log('tapped ' + ele.id());
});
```

eles.once()
Bind a callback function that is triggered once per event per collection.

eles.once(events [, selector] [, data], function(event))

- events
  A space separated list of event names.

- selector [optional]
  A delegate selector to specify child elements for which the handler is triggered.

- data [optional]
  A plain object which is passed to the handler in the event object argument.

- function(event)
  The handler function that is called when one of the specified events occurs.

  - event
    The event object.

Details

For each event specified to this function, the handler function is triggered once. This is useful for one-off events that occur on just one element in the calling collection.

Examples

```
cy.$('node').once('click', function(e){
  var ele = e.cyTarget;
  console.log('clicked ' + ele.id());
});
```

eles.off()
Aliases: `eles.unbind()`, `eles.unlisten()`, `eles.removeListener()`
Unbind one or more callback functions on the elements.

eles.off(events [, selector] [, handler])

- events
  A space separated list of event names.

- selector [optional]
  The same delegate selector used to bind to the events.

- handler [optional]
  A reference to the handler function to remove.

Examples

```
var j = cy.$('#j');
var handler = function(){ console.log('tap') };

// bind
j.on('tap', handler);

// bind some other handler
j.on('tap', function(){
  console.log('some other handler');
});

j.trigger('tap'); // 'tap' & 'some other handler'

// unbind the renferenced handler
j.off('tap', handler);

j.trigger('tap'); // some other handler

// unbind all tap handlers (including unnamed handler)
j.off('tap');
```

eles.trigger()
Aliases: `eles.emit()`
Trigger events on the elements.

eles.trigger(events [, extraParams])

- events
  A space separated list of event names to trigger.

- extraParams [optional]
  An array of additional parameters to pass to the handler.

Examples

```
var j = cy.$('#j');

j.on('tap', function(){
  console.log('tap!!');
});

j.trigger('tap'); // tap!!
```

ele.data()
Get all data for the element.

ele.data(name)
Get a particular data field for the element.

- name
  The name of the field to get.

ele.data(name, value)
Set a particular data field for the element.

- name
  The name of the field to set.

- value
  The value to set for the field.

ele.data(obj)
Update multiple data fields at once via an object.

- obj
  The object containing name-value pairs to update data fields.

Details

The following fields are immutable:

- `id` : The `id` field is used to uniquely identify an element in the graph.
- `source` & `target` : These fields define an edge's relationship to nodes, and this relationship can not be changed after creation.
- `parent` : The `parent` field defines the parent (compound) node.

Examples

```
var j = cy.$('#j');

// set the weight field in data
j.data('weight', 60);

// set several fields at once
j.data({
  name: 'Jerry Jerry Dingleberry',
  height: 176
});

var weight = j.data('weight');
```

eles.removeData()
Aliases: `eles.removeAttr()`
Remove developer-defined data associated with the elements.

eles.removeData()
Removes all mutable data fields for the elements.

eles.removeData(names)
Removes the specified mutable data fields for the elements.

- names
  A space-separated list of fields to delete.

Details

The following data fields are immutable, and so they can not be removed:

- `id` : The `id` field is used to uniquely identify an element in the graph.
- `source` & `target` : These fields define an edge's relationship to nodes, and this relationship can not be changed after creation.
- `parent` : The `parent` field defines the parent (compound) node.

ele.scratch()
Extension function: This function is intended for use in extensions.
Set or get scratchpad data, where temporary or non-JSON data can be stored. App-level scratchpad data should use namespaces prefixed with underscore, like `'_foo'`.

ele.scratch()
Get the entire scratchpad object for the element.

ele.scratch(namespace)
Get the scratchpad at a particular namespace.

- namespace
  A namespace string.

- namespace
  A namespace string.

- value
  The value to set at the specified namespace.

Details

This function is useful for storing temporary, possibly non-JSON data. Extensions — like layouts, renderers, and so on — use `ele.scratch()` namespaced on their registered name. For example, an extension named `foo` would use the namespace `'foo'`.

If you want to use this function for your own app-level data, you can prefix the namespaces you use by underscore to avoid collisions with extensions. For example, using `ele.scratch('_foo')` in your app will avoid collisions with an extension named `foo`.

This function is useful for associating non-JSON data to an element. Whereas data stored via `ele.data()` is included by `ele.json()`, data stored by `ele.scratch()` is not. This makes it easy to temporarily store unserialisable data.

Examples

```
var j = cy.$('#j');

// entire scratchpad:
// be careful, since you could clobber over someone else's namespace or
forget to use one at all!
var fooScratch = j.scratch()._foo = {};
// ... now you can modify fooScratch all you want

// set namespaced scratchpad to ele:
// safer, recommended
var fooScratch = j.scratch('_foo', {});
// ... now you can modify fooScratch all you want

// get namespaced scratchpad from ele (assumes set before)
var fooScratch = j.scratch('_foo');
// ... now you can modify fooScratch all you want
```

ele.removeScratch()
Extension function: This function is intended for use in extensions.
Remove scratchpad data. You should remove scratchpad data only at your own namespaces.

ele.removeScratch(namespace)
Remove the scratchpad data at a particular namespace.

- namespace
  A namespace string.

ele.json()
Get or mutate the element's plain JavaScript object representation.

ele.json()
Get the element's JSON.

ele.json(eleJson)
Mutate the element's state as specified.

- eleJson
  For each field in the object, the element's state is mutated as specified.

Details

This function returns the plain JSON representation of the element, the same format which is used at initialisation, in `cy.load()`, in `cy.add()` etc.

This function can also be used to set the element's state using the plain JSON representation of the element. Each field specified in `ele.json( eleJson )` is diffed against the element's current state, the element is mutated accordingly, and the appropriate events are emitted. This can be used to declaratively modify elements.

Examples

```
console.log( cy.$('#j').json() );
```

```
cy.$('#j').json({ selected: true });
```

eles.jsons()
Get an array of the plain JavaScript object representation of all elements in the collection.

Details

This function returns the plain JSON representation of all elements in the collection, the same format which is used at initialisation, in `cy.load()`, in `cy.add()` etc.

Examples

```
console.log( cy.elements().jsons() );
```

Details

The group strings are `'nodes'` for nodes and `'edges'` for edges. In general, you should be using `ele.isEdge()` and `ele.isNode()` instead of `ele.group()`.

ele.isNode()
Get whether the element is a node.

ele.isEdge()
Get whether the element is an edge.

edge.isLoop()
Get whether the edge is a loop (i.e. source same as target).

edge.isSimple()
Get whether the edge is simple (i.e. source different than target).

node.degree(includeLoops)
Get the degree of a node.

- includeLoops
  A boolean, indicating whether loops are to be included in degree calculations.

node.indegree(includeLoops)
Get the indegree of a node.

- includeLoops
  A boolean, indicating whether loops are to be included in degree calculations.

node.outdegree(includeLoops)
Get the outdegree of a node.

- includeLoops
  A boolean, indicating whether loops are to be included in degree calculations.

nodes.totalDegree(includeLoops)
Get the total degree of a collection of nodes.

- includeLoops
  A boolean, indicating whether loops are to be included in degree calculations.

nodes.minDegree(includeLoops)

Get the minimum degree of the nodes in the collection.

- includeLoops
  A boolean, indicating whether loops are to be included in degree calculations.

nodes.maxDegree(includeLoops)
Get the maximum degree of the nodes in the collection.

- includeLoops
  A boolean, indicating whether loops are to be included in degree calculations.

nodes.minIndegree(includeLoops)
Get the minimum indegree of the nodes in the collection.

- includeLoops
  A boolean, indicating whether loops are to be included in degree calculations.

nodes.maxIndegree(includeLoops)
Get the maximum indegree of the nodes in the collection.

- includeLoops
  A boolean, indicating whether loops are to be included in degree calculations.

nodes.minOutdegree(includeLoops)
Get the minimum outdegree of the nodes in the collection.

- includeLoops
  A boolean, indicating whether loops are to be included in degree calculations.

nodes.maxOutdegree(includeLoops)
Get the maximum outdegree of the nodes in the collection.

- includeLoops
  A boolean, indicating whether loops are to be included in degree calculations.

Details

Degree : For a node, the degree is the number of edge connections it has. Each time a node is referenced as `source` or `target` of an edge in the graph, that counts as an edge connection.

Indegree : For a node, the indegree is the number of incoming edge connections it has. Each time a node is referred to as `target` of an edge in the graph, that counts as an incoming edge connection.

Outdegree : For a node, the outdegree is the number of outgoing edge connections it has. Each time a node is referred to as `source` of an edge in the graph, that counts as an outgoing edge connection.

Total degree : For a set of nodes, the total degree is the total number of edge connections to nodes in

the set.

node.position()
Get the entire position object.

node.position(dimension)
Get the value of a specified position dimension.

- dimension
  The position dimension to get.

node.position(dimension, value)
Set the value of a specified position dimension.

- dimension
  The position dimension to set.

- value
  The value to set to the dimension.

node.position(pos)
Set the position using name-value pairs in the specified object.

- pos
  An object specifying name-value pairs representing dimensions to set.

Details

A position has two fields, `x` and `y`, that can take on numerical values.

Examples

```
// get x for j
var x = cy.$('#j').position('x');

// get the whole position for e
var pos = cy.$('#e').position();

// set y for j
cy.$('#j').position('y', 100);

// set multiple
cy.$('#e').position({
  x: 123,
  y: 200
});
```

nodes.positions()
Aliases: `nodes.modelPositions()`, `nodes.points()`
Set the (model) positions of several nodes with a function.

nodes.positions(function(i, ele))
Set the positions functionally.

- i
  The index of the element when iterating over the elements in the collection.

- ele
  The element being iterated over for which the function should return a position to set.

nodes.positions(pos)
Set positions for all nodes based on a single position object.

- pos
  An object specifying name-value pairs representing dimensions to set.

Examples

```
cy.nodes().positions(function( i, node ){
  return {
    x: i * 100,
    y: 100
  };
});
```

node.renderedPosition()
Aliases: `node.renderedPoint()`
Get or set the rendered (on-screen) position of a node.

node.renderedPosition()
Get the entire rendered position object.

node.renderedPosition(dimension)
Get the value of a specified rendered position dimension.

- dimension
  The position dimension to get.

node.renderedPosition(dimension, value)
Set the value of a specified rendered position dimension.

- dimension
  The position dimension to set.

- value
  The value to set to the dimension.

node.renderedPosition(pos)
Set the rendered position using name-value pairs in the specified object.

- pos
  An object specifying name-value pairs representing dimensions to set.

node.relativePosition()
Aliases: `node.relativePoint()`
Get or set the position of a node, relative to its compound parent.

node.relativePosition()
Get the entire relative position object.

node.relativePosition(dimension)
Get the value of a specified relative position dimension.

- dimension
  The position dimension to get.

node.relativePosition(dimension, value)
Set the value of a specified relative position dimension.

- dimension
  The position dimension to set.

- value
  The value to set to the dimension.

node.relativePosition(pos)
Set the relative position using name-value pairs in the specified object.

- pos
  An object specifying name-value pairs representing dimensions to set.

ele.width() et al
Get the width of the element.

ele.width()
Get the width of the element.

ele.outerWidth()
Get the outer width of the element (includes width & border).

ele.renderedWidth()

Get the width of the element in rendered dimensions.

ele.renderedOuterWidth()
Get the outer width of the element (includes width & border) in rendered dimensions.

ele.height() et al
Get the height of the element.

ele.height()
Get the height of the element.

ele.outerHeight()
Get the outer height of the element (includes height & border).

ele.renderedHeight()
Get the height of the element in rendered dimensions.

ele.renderedOuterHeight()
Get the outer height of the element (includes height & border) in rendered dimensions.

eles.boundingBox()
Aliases: `eles.boundingbox()`
Get the bounding box of the elements.

eles.boundingBox(options)
Get the bounding box of the elements in model coordinates.

- options
  An object containing options for the function.
  - includeNodes
    A boolean indicating whether to include nodes in the bounding box.
  - includeEdges
    A boolean indicating whether to include edges in the bounding box.
  - includeLabels
    A boolean indicating whether to include labels in the bounding box.

Details

This function returns a plain object with the fields `x1`, `x2`, `y1`, `y2`, `w`, and `h` defined.

eles.renderedBoundingBox()
Aliases: `eles.renderedBoundingbox()`
Get the rendered bounding box of the elements.

eles.renderedBoundingBox(options)
Get the bounding box of the elements in rendered coordinates.

- options
  An object containing options for the function.
  - includeNodes
    A boolean indicating whether to include nodes in the bounding box.
  - includeEdges
    A boolean indicating whether to include edges in the bounding box.
  - includeLabels
    A boolean indicating whether to include labels in the bounding box.

Details

This function returns a plain object with the fields `x1`, `x2`, `y1`, `y2`, `w`, and `h` defined.

node.grabbed()
Get whether a node is currently grabbed, meaning the user has hold of the node.

node.grabbable()
Get whether the user can grab a node.

nodes.grabify()
Allow the user to grab the nodes.

Examples

```
cy.$('#j').grabify();
```

Examples

```
cy.$('#j').ungrabify();
```

nodes.lock()
Lock the nodes such that their positions can not be changed.

Examples

```
cy.$('#j').lock();
```

Examples

```
cy.$('#j').unlock();
```

eles.layout(options)

- options
  The layout options.

Details

This function is useful for running a layout on a subset of the elements in the graph.

For layouts included with Cytoscape.js, you can find their options documented in the Layouts section. For external layouts, please refer to their accompanying documentation.

Examples

Run the grid layout:

```
cy.elements().layout({ name: 'grid' });
```

eles.makeLayout()
Aliases: `eles.createLayout()`
Get a new layout, which can be used to algorithmically position the nodes in the collection.

eles.makeLayout(options)

- options
  The layout options.

This function is useful for running a layout on a subset of the elements in the graph, perhaps in parallel to other layouts.

You must specify `options.name` with the name of the layout you wish to use.

This function creates and returns a layout object. You may want to keep a reference to the layout for more advanced usecases, such as running multiple layouts simultaneously.

Note that you must call `layout.run()` in order for it to affect the graph.

Examples

```
var layout = cy.elements().makeLayout({
  name: 'random'
});

layout.run();
```

nodes.layoutPositions()
Extension function: This function is intended for use in extensions.
Position the nodes for a discrete/synchronous layout.

nodes.layoutPositions(layout, options, function(i, ele))

- layout
  The layout.

- options
  The layout options object.

- function(i, ele)
  A function that returns the new position for the specified node.

  - i
    The index of the current node while iterating over the nodes in the layout.

  - ele
    The node being iterated over for which the function should return a position to set.

Details

This function is useful for running a layout on a subset of the elements in the graph.

For layouts included with Cytoscape.js, you can find their options documented in the Layouts section.
For external layouts, please refer to their accompanying documentation.

Examples

Run the grid layout:

```
cy.elements().layout({ name: 'grid' });
```

eles.select()
Make the elements selected (NB other elements outside the collection are not affected).

Examples

```
cy.$('#j').select();
```

Examples

```
cy.$('#j').unselect();
```

eles.selectify()

Make the selection states of the elements mutable.

Examples

```
cy.$('#j').unselectify();
```

Examples

```
cy.$('#j').unselectify();
```

Examples

```
cy.$('#j, #e').addClass('foo');
```

eles.removeClass()
Remove classes from elements.

Examples

```
cy.$('#j, #e').removeClass('foo');
```

eles.toggleClass()
Toggle whether the elements have the specified classes.

eles.toggleClass(classes [, toggle])

- classes
  A space-separated list of class names to toggle on the elements.

- toggle [optional]
  Instead of automatically toggling, adds the classes on truthy values or removes them on falsey values.

Examples

Toggle:

```
cy.$('#j, #e').toggleClass('foo');
```

Toggle on:

```
cy.$('#j, #e').toggleClass('foo', true);
```

Toggle off:

```
cy.$('#j, #e').toggleClass('foo', false);
```

eles.classes()
Replace the current list of classes on the elements with the specified list.

eles.classes(classes)

- classes
  A space-separated list of class names that replaces the current class list.

Examples

Remove all classes:

```
cy.nodes().classes(); // no classes
```

Replace classes:

```
cy.nodes().classes('foo');
```

eles.flashClass(classes [, duration])

- classes
  A space-separated list of class names to flash on the elements.

- duration [optional]
  The duration in milliseconds that the classes should be added on the elements. After the duration, the classes are removed.

Examples

```
cy.$('#j, #e').flashClass('foo', 1000);
```

ele.hasClass()
Get whether an element has a particular class.

ele.hasClass(className)

- className
  The name of the class to test for.

Examples

```
console.log( 'j has class `foo` : '  + cy.$('#j').hasClass('foo') );
```

eles.style() et al
Aliases: `eles.css()`

Get the style of the element.

ele.style()
Get a name-value pair object containing visual style properties and their values for the element.

ele.style(name)
Get a particular style property value.

- name
  The name of the visual style property to get.

Details

You should use this function very sparingly, because it *overrides* the style of an element, despite the state and classes that it has. In general, it's much better to specify a better stylesheet at initialisation that reflects your application state rather than programmatically modifying style.

If you would like to remove a particular overridden style property, set `null` or `''` (the empty string) to it.

ele.renderedStyle()
Aliases: `eles.renderedCss()`
Get the style of the element in rendered dimensions.

ele.renderedStyle()
Get a name-value pair object containing rendered visual style properties and their values for the element.

ele.renderedStyle(name)
Get a particular rendered style property value.

- name
  The name of the visual style property to get.

ele.visible() et al
Get whether the element is visible (i.e. `display: element` and `visibility: visible`).

ele.visible()
Get whether the element is visible.

ele.hidden()
Get whether the element is hidden.

ele.effectiveOpacity()
Get the effective opacity of the element (i.e. on-screen opacity), which takes into consideration parent node opacity.

ele.transparent()
Get whether the element's effective opacity is completely transparent, which takes into consideration parent node opacity.

eles.animate()
Animate the elements.

- position
  A position to which the elements will be animated.
- renderedPosition
  A rendered position to which the elements will be animated.
- style
  An object containing name-value pairs of style properties to animate.
- duration
  The duration of the animation in milliseconds.
- queue
  A boolean indicating whether to queue the animation.
- complete
  A function to call when the animation is done.
- step
  A function to call each time the animation steps.
- easing
  A `transition-timing-function` easing style string that shapes the animation progress curve.

Details

Note that you can specify only one of `position` and `renderedPosition`: You can not animate to two positions at once.

Examples

```
cy.nodes().animate({
  position: { x: 100, y: 100 },
  style: { backgroundColor: 'red' }
}, {
  duration: 1000
});
```

```
console.log('Animating nodes...');
```

ele.animation()
Get an animation for the element.

ele.animation(options)

- position
  A position to which the elements will be animated.
- renderedPosition
  A rendered position to which the elements will be animated.
- style
  An object containing name-value pairs of style properties to animate.
- duration
  The duration of the animation in milliseconds.
- easing
  A `transition-timing-function` easing style string that shapes the animation progress curve.

eles.delay()
Add a delay between animations for the elements.

eles.delay(duration, complete)

- duration
  How long the delay should be in milliseconds.
- complete
  A function to call when the delay is complete.

Examples

```
cy.nodes()
  .animate({
      style: { 'background-color': 'blue' }
    }, {
      duration: 1000
    })

  .delay( 1000 )

  .animate({
    style: { 'background-color': 'yellow' }
```

```
  })
;

console.log('Animating nodes...');
```

ele.delayAnimation()
Get a delay animation for the element.

ele.delayAnimation(duration)

- duration
  How long the delay should be in milliseconds.

eles.stop()
Stop all animations that are currently running.

eles.stop(clearQueue, jumpToEnd)

- clearQueue
  A boolean, indicating whether the queue of animations should be emptied.

- jumpToEnd
  A boolean, indicating whether the currently-running animations should jump to their ends rather than just stopping midway.

Examples

```
cy.nodes().animate({
  style: { 'background-color': 'cyan' }
}, {
  duration: 5000,
  complete: function(){
    console.log('Animation complete');
  }
});

console.log('Animating nodes...');

setTimeout(function(){
  console.log('Stopping nodes animation');
  cy.nodes().stop();
}, 2500);
```

eles.clearQueue()
Remove all queued animations for the elements.

eles.same(eles)

Examples

```
var heavies = cy.$('node[weight > 60]');
var guys = cy.$('#j, #g, #k');

console.log( 'same ? ' + heavies.same(guys) );
```

eles.anySame()
Determine whether this collection contains any of the same elements as another collection.

eles.anySame(eles)

Examples

```
var j = cy.$('#j');
var guys = cy.$('#j, #g, #k');

console.log( 'any same ? ' + j.anySame(guys) );
```

eles.allAreNeighbors()
Aliases: `eles.allAreNeighbours()`
Determine whether all elements in the specified collection are in the neighbourhood of the calling collection.

eles.allAreNeighbors(eles)

Examples

```
var j = cy.$('#j');
var gAndK = cy.$('#g, #k');

console.log( 'all neighbours ? ' + j.allAreNeighbors(gAndK) );
```

eles.is()
Determine whether any element in this collection matches a selector.

eles.is(selector)

Examples

```
var j = cy.$('#j');

console.log( 'j has weight > 50 ? ' + j.is('[weight > 50]') );
```

eles.allAre()
Determine whether all elements in the collection match a selector.

eles.allAre(selector)

Examples

```
var jAndE = cy.$('#j, #e');

console.log( 'j and e all have weight > 50 ? ' + jAndE.allAre('[weight > 50]') );
```

eles.some()
Determine whether any element in this collection satisfies the specified test function.

eles.some(function(ele, i, eles) [, thisArg])

- function(ele, i, eles)
  The test function that returns truthy values for elements that satisfy the test and falsey values for elements that do not satisfy the test.
  - ele
    The current element.
  - i
    The index of the current element.
  - eles
    The collection of elements being tested.
- thisArg [optional]
  The value for `this` within the test function.

Examples

```
var jAndE = cy.$('#j, #e');
var someHeavierThan50 = jAndE.some(function( ele ){
  return ele.data('weight') > 50;
});

console.log( 'some heavier than 50 ? ' + someHeavierThan50 );
```

eles.every()
Determine whether all elements in this collection satisfy the specified test function.

eles.every(function(ele, i, eles) [, thisArg])

- function(ele, i, eles)
  The test function that returns truthy values for elements that satisfy the test and falsey values for elements that do not satisfy the test.
  - ele
    The current element.
  - i
    The index of the current element.
  - eles
    The collection of elements being tested.
- thisArg [optional]
  The value for `this` within the test function.

Examples

```
var jAndE = cy.$('#j, #e');
var everyHeavierThan50 = jAndE.every(function( ele ){
  return ele.data('weight') > 50;
});

console.log( 'every heavier than 50 ? ' + everyHeavierThan50 );
```

Details

Note that as an alternative, you may read `eles.length` instead of `eles.size()`. The two are interchangeable.

eles.empty() et al
Get whether the collection is empty, meaning it has no elements.

eles.empty()
Get whether the collection is empty.

eles.nonempty()
Get whether the collection is nonempty.

eles.each()
Iterate over the elements in the collection.

eles.each(function(i, ele))
- i
  The index of the element in the collection.

- ele
  The element at the current index.

Details

This function behaves like the jQuery `.each()` function. For a more standard implementation, you may want to use `eles.forEach()`.

Note that although this function is convenient in some cases, it is less efficient than making your own loop:

```
var eles = cy.elements();
for( var i = 0; i < eles.length; i++ ){
    var ele = ele[i];

    console.log( ele.id() + ' is ' + ( ele.selected() ? 'selected' : 'not
selected' ) );
}
```

Examples

```
cy.elements().each(function(i, ele){
  console.log( ele.id() + ' is ' + ( ele.selected() ? 'selected' : 'not
selected' ) );
});
```

eles.forEach()
Iterate over the elements in the collection using an implementation like the native array function namesake.

eles.forEach(function(ele, i, eles) [, thisArg])

- function(ele, i, eles)
  The function executed each iteration.
  - ele
    The current element.
  - i
    The index of the current element.
  - eles
    The collection of elements being iterated.
- thisArg [optional]
  The value for `this` within the iterating function.

Details

This function behaves like `Array.prototype.forEach()` with minor changes for convenience:

- You can exit the iteration early by returning `false` in the iterating function. The `Array.prototype.forEach()` implementation does not support this, but it is included anyway on account of its utility.

Examples

```
// print all the ids of the nodes in the graph
cy.nodes().forEach(function( ele ){
  console.log( ele.id() );
});
```

eles.eq() et al
Get an element at a particular index in the collection.

eles.eq(index)

- index
  The index of the element to get.

eles.first()
Get the first element in the collection.

eles.last()
Get the last element in the collection.

Details

You may use `eles[i]` in place of `eles.eq(i)` as a more performant alternative.

eles.slice()
Get a subset of the elements in the collection based on specified indices.

eles.slice([start] [, end])

- start [optional]
  An integer that specifies where to start the selection. The first element has an index of 0. Use negative numbers to select from the end of an array.

- end [optional]
  An integer that specifies where to end the selection. If omitted, all elements from the start position and to the end of the array will be selected. Use negative numbers to select from the end of an array.

Building & filtering

eles.union()
Aliases: `eles.add()`, `eles.or()`, `eles['u']()`, `eles['+']()`, `eles['|']()`
Get a new collection, resulting from adding the collection with another one

eles.union(eles)
eles.union(selector)

Examples

With a collection:

```
var j = cy.$('#j');
var e = cy.$('#e');

j.union(e);
```

With a selector:

```
cy.$('#j').union('#e');
```

eles.difference()
Aliases: `eles.not()`, `eles.subtract()`, `eles.relativeComplement()`, `eles['\\']()`, `eles['!']()`, `eles['-']()`
Get a new collection, resulting from the collection without some specified elements.

eles.difference(eles)
eles.difference(selector)

Examples

With a collection:

```
var j = cy.$('#j');
var nodes = cy.nodes();

nodes.difference(j);
```

With a selector:

```
cy.nodes().difference('#j');
```

Examples

```
cy.$('#j').absoluteComplement();
```

eles.intersection()
Aliases: `eles.intersect()`, `eles.and()`, `eles['n']()`, `eles['&']()`, `eles['.']()`
Get the elements in both this collection and another specified collection.

eles.intersection(eles)
eles.intersection(selector)

- selector
  A selector representing the elements to intersect with. All elements in the graph matching the selector are used as the passed collection.

Examples

```
var jNhd = cy.$('#j').neighborhood();
var eNhd = cy.$('#e').neighborhood();

jNhd.intersection( eNhd );
```

eles.symmetricDifference()
Aliases: `eles.symdiff()`, `eles.xor()`, `eles['^']()`, `eles['(+)']()`, `eles['(-)']()`
Get the elements that are in the calling collection or the passed collection but not in both.

eles.symmetricDifference(eles)
eles.symmetricDifference(selector)

- selector
  A selector representing the elements to apply the symmetric difference with. All elements in the graph matching the selector are used as the passed collection.

Examples

```
cy.$('#j, #e, #k').symdiff('#j, #g');
```

eles.diff()
Perform a traditional left/right diff on the two collections.

eles.diff(eles)
eles.diff(selector)

- selector
  A selector representing the elements on the right side of the diff. All elements in the graph matching the selector are used as the passed collection.

Details

This function returns a plain object of the form `{ left, right, both }` where

- `left` is the set of elements only in the calling (i.e. left) collection,
- `right` is the set of elements only in the passed (i.e. right) collection, and
- `both` is the set of elements in both collections.

Examples

```
var diff = cy.$('#j, #e, #k').diff('#j, #g');
var getNodeId = function( n ){ return n.id() };

console.log( 'left: ' + diff.left.map( getNodeId ).join(', ') );
console.log( 'right: ' + diff.right.map( getNodeId ).join(', ') );
console.log( 'both: ' + diff.both.map( getNodeId ).join(', ') );
```

eles.filter() et al
Get a new collection containing elements that are accepted by the specified filter.

eles.filter(selector)
Get the elements that match the specified selector.

- selector
  The selector to match against.

eles.filter(function(i, ele))
Get the elements that match the specified filter function.

- function(i, ele)
  The filter function that returns true for elements to include.
  - i
    The index of the current element being considered.
  - ele
    The element being considered.

eles.nodes(selector)
Get the nodes that match the specified selector.

- selector
  The selector to match against.

eles.edges(selector)
Get the edges that match the specified selector.

- selector

The selector to match against.

Examples

```
cy.nodes().filter('[weight > 50]');
```

eles.filterFn()
Aliases: `eles.fnFilter()`, `eles.stdFilter()`
Get a new collection containing elements that are accepted by the specified filter, using an implementation like the standard array namesake.

eles.filterFn(function(ele, i, eles) [, thisArg])

- function(ele, i, eles)
  The filter function that returns truthy values for elements to include and falsey values for elements to exclude.
  - ele
    The current element.
  - i
    The index of the current element.
  - eles
    The collection of elements being filtered.
- thisArg [optional]
  The value for `this` within the iterating function.

Examples

```
cy.nodes().filterFn(function( ele ){
  return ele.data('weight') > 50;
});
```

eles.sort()
Get a new collection containing the elements sorted by the specified comparison function.

eles.sort(function(ele1, ele2))

- function(ele1, ele2)
  The sorting comparison function that returns a negative number for `ele1` before `ele2`, 0 for `ele1` same as `ele2`, or a positive number for `ele1` after `ele2`.

Examples

Get collection of nodes in order of increasing weight:

```
var nodes = cy.nodes().sort(function( a, b ){
  return a.data('weight') - b.data('weight');
});

// show order via animations
var duration = 1000;
nodes.removeStyle().forEach(function( node, i ){
  node.delay( i * duration ).animate({
    style: {
      'border-width': 4,
      'border-color': 'green'
    }
  }, { duration: duration });
});

console.log('Animating nodes to show sorted order');
```

eles.map()
Get an array containing values mapped from the collection.

eles.map(function(ele, i, eles) [, thisArg])

- function(ele, i, eles)
  The function that returns the mapped value for each element.
  - ele
    The current element.
  - i
    The index of the current element.
  - eles
    The collection of elements being mapped.
- thisArg [optional]
  The value for `this` within the iterating function.

Examples

Get an array of node weights:

```
var weights = cy.nodes().map(function( ele ){
  return ele.data('weight');
});

console.log(weights);
```

eles.min()
Find a minimum value in a collection.

eles.min(function(ele, i, eles) [, thisArg])

- function(ele, i, eles)
  The function that returns the value to compare for each element.
  - ele
    The current element.
  - i
    The index of the current element.
  - eles
    The collection of elements being searched.
- thisArg [optional]
  The value for `this` within the iterating function.

Details

This function returns an object with the following fields:

- `value` : The minimum value found.
- `ele` : The element that corresponds to the minimum value.

Examples

Find the node with the minimum weight:

```
var min = cy.nodes().min(function(){
  return this.data('weight');
});

console.log( 'min val: ' + min.value + ' for element ' + min.ele.id() );
```

eles.max()
Find a maximum value and the corresponding element.

eles.max(function(ele, i, eles) [, thisArg])

- function(ele, i, eles)
  The function that returns the value to compare for each element.
  - ele
    The current element.

- i
  The index of the current element.

- eles
  The collection of elements being searched.

- thisArg [optional]
  The value for `this` within the iterating function.

Details

This function returns an object with the following fields:

- `value` : The maximum value found.
- `ele` : The element that corresponds to the maximum value.

Examples

Find the node with the maximum weight:

```
var max = cy.nodes().max(function(){
  return this.data('weight');
});

console.log( 'max val: ' + max.value + ' for element ' + max.ele.id() );
```

eles.neighborhood([selector])
Aliases: eles.neighbourhood()
Get the open neighbourhood of the elements.

- selector [optional]
  An optional selector that is used to filter the resultant collection.

eles.openNeighborhood([selector])
Aliases: eles.openNeighbourhood()
Get the open neighbourhood of the elements.

- selector [optional]
  An optional selector that is used to filter the resultant collection.

eles.closedNeighborhood([selector])
Aliases: eles.closedNeighbourhood()
Get the closed neighbourhood of the elements.

- selector [optional]
  An optional selector that is used to filter the resultant collection.

Details

The neighbourhood returned by this function is a bit different than the traditional definition of a "neighbourhood": This returned neighbourhood includes the edges connecting the collection to the neighbourhood. This gives you more flexibility.

An **open neighbourhood** is one that **does not** include the original set of elements. If unspecified, a neighbourhood is open by default.

A **closed neighbourhood** is one that **does** include the original set of elements.

Examples

```
cy.$('#j').neighborhood();
```

eles.components()
Get the connected components, considering only the elements in the calling collection. An array of collections is returned, with each collection representing a component.

nodes.edgesWith()
Get the edges connecting the collection to another collection. Direction of the edges does not matter.

nodes.edgesWith(eles)
nodes.edgesWith(selector)

- selector
  The other collection, specified as a selector which is matched against all elements in the graph.

Examples

```
var j = cy.$('#j');
var e = cy.$('#e');

j.edgesWith(e);
```

nodes.edgesTo()
Get the edges coming from the collection (i.e. the source) going to another collection (i.e. the target).

nodes.edgesTo(eles)
nodes.edgesTo(selector)

- selector
  The other collection, specified as a selector which is matched against all elements in the graph.

Examples

```
var j = cy.$('#j');
```

```
var e = cy.$('#e');

j.edgesTo(e);
```

edges.connectedNodes()
Get the nodes connected to the edges in the collection.

edges.connectedNodes([selector])

Examples

```
var je = cy.$('#je');

je.connectedNodes();
```

nodes.connectedEdges()
Get the edges connected to the nodes in the collection.

nodes.connectedEdges([selector])

Examples

```
var j = cy.$('#j');

j.connectedEdges();
```

edge.source()
Get source node of this edge.

edge.source([selector])

Examples

```
var je = cy.$('#je');

je.source();
```

edges.sources()
Get source nodes connected to the edges in the collection.

edges.sources([selector])

Examples

```
var edges = cy.$('#je, #kg');
```

```
edges.sources();
```

edge.target()
Get target node of this edge.

edge.target([selector])

Examples

```
var je = cy.$('#je');

je.target();
```

edges.targets()
Get target nodes connected to the edges in the collection.

edges.targets([selector])

Examples

```
var edges = cy.$('#je, #kg');

edges.targets();
```

edges.parallelEdges()
Get edges parallel to those in the collection.

edges.parallelEdges([selector])

Details

Two edges are said to be parallel if they connect the same two nodes. Any two parallel edges may connect nodes in the same direction, in which case the edges share the same source and target. They may alternatively connect nodes in the opposite direction, in which case the source and target are reversed in the second edge.

Examples

```
cy.$('#je').parallelEdges();
```

edges.codirectedEdges([selector])

Details

Two edges are said to be codirected if they connect the same two nodes in the same direction: The edges have the same source and target.

Examples

```
cy.$('#je').codirectedEdges(); // only self in this case
```

nodes.roots()
From the set of calling nodes, get the nodes which are roots (i.e. no incoming edges, as in a directed acyclic graph).

nodes.roots([selector])
nodes.leaves()
From the set of calling nodes, get the nodes which are leaves (i.e. no outgoing edges, as in a directed acyclic graph).

nodes.leaves([selector])
nodes.outgoers()
Get edges (and their targets) coming out of the nodes in the collection.

nodes.outgoers([selector])

Examples

Get outgoers of `j`:

```
cy.$('#j').outgoers();
```

nodes.successors()
Recursively get edges (and their targets) coming out of the nodes in the collection (i.e. the outgoers, the outgoers' outgoers, ...).

nodes.successors([selector])

Examples

Get successors of `j`:

```
cy.$('#j').successors();
```

nodes.incomers([selector])

Examples

Get incomers of `j`:

```
cy.$('#j').incomers();
```

nodes.predecessors()
Recursively get edges (and their sources) coming into the nodes in the collection (i.e. the incomers, the incomers' incomers, ...).

nodes.predecessors([selector])

Examples

Get predecessors of `j`:

```
cy.$('#j').predecessors();
```

eles.breadthFirstSearch(options)

- root
  The root nodes (selector or collection) to start the search from.
  - visit: function(i, depth, v, e, u) [optional]
    A handler function that is called when a node is visited in the search. The handler returns `true` when it finds the desired node, and it returns `false` to cancel the search.
    - i
      The index indicating this node is the ith visited node.
    - depth
      How many edge hops away this node is from the root nodes.
    - v
      The current node.
    - e
      The edge connecting the previous node to the current node.
    - u
      The previous node.
  - directed [optional]
    A boolean indicating whether the algorithm should only go along edges from source to target (default `false`).

Details

Note that this function performs a breadth-first search on only the subset of the graph in the calling collection.

This function returns an object that contains two collections (`{ path: eles, found: node }`), the node found by the search and the path of the search:

- If no node was found, then `found` is empty.
- If your handler function returns `false`, then the only the path up to that point is returned.
- The path returned includes edges such that if `path[i]` is a node, then `path[i - 1]` is the edge used to get to that node.

Examples

```
var bfs = cy.elements().bfs({
  roots: '#e',
  visit: function(i, depth){
    console.log( 'visit ' + this.id() );

    // example of finding desired node
    if( this.data('weight') > 70 ){
      return true;
    }

    // example of exiting search early
    if( this.data('weight') < 0 ){
      return false;
    }
  },
  directed: false
});

var path = bfs.path; // path to found node
var found = bfs.found; // found node

// select the path
path.select();
```

eles.depthFirstSearch()
Aliases: `eles.dfs()`
Perform a depth-first search within the elements in the collection.

eles.depthFirstSearch(options)

- root
  The root nodes (selector or collection) to start the search from.

- visit: function(i, depth, v, e, u) [optional]
  A handler function that is called when a node is visited in the search. The handler returns `true`

when it finds the desired node, and it returns `false` to cancel the search.

- i
  The index indicating this node is the ith visited node.
- depth
  How many edge hops away this node is from the root nodes.
- v
  The current node.
- e
  The edge connecting the previous node to the current node.
- u
  The previous node.

- directed [optional]
  A boolean indicating whether the algorithm should only go along edges from source to target (default `false`).

Details

Note that this function performs a depth-first search on only the subset of the graph in the calling collection.

This function returns an object that contains two collections (`{ path: eles, found: node }`), the node found by the search and the path of the search:

- If no node was found, then `found` is empty.
- If your handler function returns `false`, then the only the path up to that point is returned.
- The path returned includes edges such that if `path[i]` is a node, then `path[i - 1]` is the edge used to get to that node.

Examples

```
var dfs = cy.elements().dfs({
  roots: '#e',
  visit: function(i, depth){
    console.log( 'visit ' + this.id() );

    // example of finding desired node
    if( this.data('weight') > 70 ){
      return true;
    }
```

```
    // example of exiting search early
    if( this.data('weight') < 0 ){
      return false;
    }
  },
  directed: false
});

var path = dfs.path; // path to found node
var found = dfs.found; // found node

// select the path
path.select();
```

eles.dijkstra()
Perform Dijkstra's algorithm on the elements in the collection. This finds the shortest paths to all other nodes in the collection from the root node.

eles.dijkstra(options)

- root
  The root node (selector or collection) where the algorithm starts.

- weight: function(edge) [optional]
  A function that returns the positive numeric weight for `this` edge.

- directed [optional]
  A boolean indicating whether the algorithm should only go along edges from source to target (default `false`).

Details

Note that this function performs Dijkstra's algorithm on only the subset of the graph in the calling collection.

This function returns an object of the following form:

```
{
  distanceTo: function( node ){ /* impl */ }
  pathTo: function( node ){ /* impl */ }
}
```

`distanceTo(node)` returns the distance from the source node to `node`, and `pathTo(node)` returns a collection containing the shortest path from the source node to `node`. The path starts with the source node and includes the edges between the nodes in the path such that if `pathTo(node)[i]` is an edge, then `pathTo(node)[i-1]` is the previous node in the path and `pathTo(node)[i+1]` is the

next node in the path.

If no weight function is defined, a constant weight of 1 is used for each edge.

Examples

```
var dijkstra = cy.elements().dijkstra('#e', function(){
  return this.data('weight');
});

var pathToJ = dijkstra.pathTo( cy.$('#j') );
var distToJ = dijkstra.distanceTo( cy.$('#j') );
```

eles.aStar()
Perform the A* search algorithm on the elements in the collection. This finds the shortest path from the root node to the goal node.

eles.aStar(options)

- root
  The root node (selector or collection) where the search starts.
- goal
  The goal node (selector or collection) where the search ends.
- weight: function(edge) [optional]
  A function that returns the positive numeric weight for `this` edge.
- heuristic: function(node) [optional]
  A function that returns an estimation (cannot be overestimation) on the shortest distance from the current (`this`) node to the goal.
- directed [optional]
  A boolean indicating whether the algorithm should only go along edges from source to target (default `false`).

Details

Note that this function performs A* search on only the subset of the graph in the calling collection.

This function returns an object of the following form:

```
{
  found, /* true or false */
  distance, /* Distance of the shortest path, if found */
  path /* Ordered collection of elements in the shortest path, if found */
}
```

Regarding optional options:

- If no weight function is defined, a constant weight of 1 is used for each edge.
- If no heuristic function is provided, a constant null function will be used, turning this into the same behaviour as Dijkstra's algorithm. The heuristic should be monotonic (also called consistent) in addition to being 'admissible'.

Examples

```
var aStar = cy.elements().aStar({ root: "#j", goal: "#e" });

aStar.path.select();
```

eles.floydWarshall()
Perform the Floyd Warshall search algorithm on the elements in the collection. This finds the shortest path between all pairs of nodes.

eles.floydWarshall(options)

- weight: function(edge) [optional]
  A function that returns the positive numeric weight for `this` edge.

- directed [optional]
  A boolean indicating whether the algorithm should only go along edges from source to target (default `false`).

Details

This function returns an object of the following form:

```
{
  /* function that computes the shortest path between 2 nodes
  (either objects or selector strings) */
  path: function( fromNode, toNode ){ /* impl */ },

  /* function that computes the shortest distance between 2 nodes
  (either objects or selector strings) */
  distance: function( fromNode, toNode ){ /* impl */ }
}
```

If no weight function is defined, a constant weight of 1 is used for each edge.

Examples

```
var fw = cy.elements().floydWarshall();
```

```
fw.path('#k', '#g').select();
```

eles.bellmanFord()
Perform the Bellman-Ford search algorithm on the elements in the collection. This finds the shortest path from the starting node to all other nodes in the collection.

eles.bellmanFord(options)

- root
  The root node (selector or collection) where the search starts.

- weight: function(edge) [optional]
  A function that returns the positive numeric weight for `this` edge.

- directed [optional]
  A boolean indicating whether the algorithm should only go along edges from source to target (default `false`).

Details

This function returns an object of the following form:

```
{
  /* function that computes the shortest path from root node to the
argument node
  (either objects or selector string) */
  pathTo: function(node){ /* impl */ },

  /* function that computes the shortest distance from root node to
argument node
  (either objects or selector string) */
  distanceTo: function(node){ /* impl */ },

  /* true/false. If true, pathTo and distanceTo will be undefined */
  hasNegativeWeightCycle
}
```

If no weight function is defined, a constant weight of 1 is used for each edge.

The Bellman-Ford algorithm is good at detecting negative weight cycles, but it can not return path or distance results if it finds them.

Examples

```
var bf = cy.elements().bellmanFord({ root: "#j" });
```

```
bf.pathTo('#g').select();
```

eles.kruskal()
Perform Kruskal's algorithm on the elements in the collection, returning the minimum spanning tree, assuming undirected edges.

eles.kruskal([function(edge)])

- function(edge) [optional]
  A function that returns the positive numeric weight for `this` edge.

Details

Note that this function runs Kruskal's algorithm on the subset of the graph in the calling collection.

Examples

```
cy.elements().kruskal();
```

eles.kargerStein()
Finds the minimum cut in a graph using the Karger-Stein algorithm. The optimal result is found with a high probability, but without guarantee.

Details

This function returns an object of the following form:

```
{
  cut, /* Collection of edges that are in the cut */
  partition1, /* Collection of nodes that are in the first partition */
  partition2 /* Collection of nodes that are in the second partition */
}
```

Examples

```
var ks = cy.elements().kargerStein();

ks.cut.select();
```

eles.pageRank(options)

- dampingFactor [optional]
  Numeric parameter for the algorithm.

- precision [optional]

Numeric parameter that represents the required precision.

- iterations [optional]
  Maximum number of iterations to perform.

Details

This function returns an object of the following form:

```
{
  /* function that computes the rank of a given node (either object or
selector string) */
  rank: function( node ){ /* impl */ }
}
```

Examples

```
var pr = cy.elements().pageRank();

console.log('g rank: ' + pr.rank('#g'));
```

eles.degreeCentrality()
Aliases: `eles.dc()`
Considering only the elements in the calling collection, calculate the degree centrality of the specified root node.

eles.degreeCentrality(options)

- root
  The root node (selector or collection) for which the centrality calculation is made.

- weight: function(edge) [optional]
  A function that returns the weight for `this` edge.

- alpha [optional]
  The alpha value for the centrality calculation, ranging on [0, 1]. With value 0 (default), disregards edge weights and solely uses number of edges in the centrality calculation. With value 1, disregards number of edges and solely uses the edge weights in the centrality calculation.

- directed [optional]
  A boolean indicating whether the directed indegree and outdegree centrality is calculated (`true`) or whether the undirected centrality is calculated (`false`, default).

Details

For `options.directed: false`, this function returns an object of the following form:

```
{
  degree /* the degree centrality of the root node */
}
```

For `options.directed: true`, this function returns an object of the following form:

```
{
  indegree, /* the indegree centrality of the root node */
  outdegree /* the outdegree centrality of the root node */
}
```

Examples

```
console.log( 'dc of j: ' + cy.$().dc({ root: '#j' }).degree );
```

eles.degreeCentralityNormalized()
Aliases: `eles.dcn()`, `eles.degreeCentralityNormalised()`
Considering only the elements in the calling collection, calculate the normalised degree centrality of the nodes.

eles.degreeCentralityNormalized(options)

- weight: function(edge) [optional]
  A function that returns the weight for `this` edge.

- alpha [optional]
  The alpha value for the centrality calculation, ranging on [0, 1]. With value 0 (default), disregards edge weights and solely uses number of edges in the centrality calculation. With value 1, disregards number of edges and solely uses the edge weights in the centrality calculation.

- directed [optional]
  A boolean indicating whether the directed indegree and outdegree centrality is calculated (`true`) or whether the undirected centrality is calculated (`false`, default).

Details

For `options.directed: false`, this function returns an object of the following form:

```
{
  /* the normalised degree centrality of the specified node */
  degree: function( node ){ /* impl */ }
}
```

For `options.directed: true`, this function returns an object of the following form:

```
{
  /* the normalised indegree centrality of the specified node */
  indegree: function( node ){ /* impl */ },

  /* the normalised outdegree centrality of the specified node */
  outdegree: function( node ){ /* impl */ }
}
```

Examples

```
var dcn = cy.$().dcn();
console.log( 'dcn of j: ' + dcn.degree('#j') );
```

eles.closenessCentrality()
Aliases: `eles.cc()`
Considering only the elements in the calling collection, calculate the closeness centrality of the specified root node.

eles.closenessCentrality(options)

- root
  The root node (selector or collection) for which the centrality calculation is made.

- weight: function(edge) [optional]
  A function that returns the weight for `this` edge.

- directed [optional]
  A boolean indicating whether the algorithm operates on edges in a directed manner from source to target (`true`) or whether the algorithm operates in an undirected manner (`false`, default).

- harmonic [optional]
  A boolean indicating whether the algorithm calculates the harmonic mean (`true`, default) or the arithmetic mean (`false`) of distances. The harmonic mean is very useful for graphs that are not strongly connected.

Details

This function directly returns the numerical closeness centrality value for the specified root node.

Examples

```
console.log( 'cc of j: ' + cy.$().cc({ root: '#j' }) );
```

eles.closenessCentralityNormalized()
Aliases: `eles.ccn()`, `eles.closenessCentralityNormalised()`

Considering only the elements in the calling collection, calculate the closeness centrality of the nodes.

eles.closenessCentralityNormalized(options)

- weight: function(edge) [optional]
  A function that returns the weight for `this` edge.

- directed [optional]
  A boolean indicating whether the algorithm operates on edges in a directed manner from source to target (`true`) or whether the algorithm operates in an undirected manner (`false`, default).

- harmonic [optional]
  A boolean indicating whether the algorithm calculates the harmonic mean (`true`, default) or the arithmetic mean (`false`) of distances. The harmonic mean is very useful for graphs that are not strongly connected.

Details

This function returns an object of the form:

```
{
  /* returns the normalised closeness centrality of the specified node */
  closeness: function( node ){ /* impl */ }
}
```

Examples

```
var ccn = cy.$().ccn();
console.log( 'ccn of j: ' + ccn.closeness('#j') );
```

eles.betweennessCentrality()
Aliases: `eles.bc()`
Considering only the elements in the calling collection, calculate the betweenness centrality of the nodes.

eles.betweennessCentrality(options)

- weight: function(edge) [optional]
  A function that returns the weight for `this` edge.

- directed [optional]
  A boolean indicating whether the algorithm operates on edges in a directed manner from source to target (`true`) or whether the algorithm operates in an undirected manner (`false`, default).

Details

This function returns an object of the form:

```
{
  /* returns the betweenness centrality of the specified node */
  betweenness: function( node ){ /* impl */ },

  /* returns the normalised betweenness centrality of the specified node
*/
  betweennessNormalized: function( node ){ /* impl */ }
  /* alias : betweennessNormalised() */
}
```

Examples

```
var bc = cy.$().bc();
console.log( 'bc of j: ' + bc.betweenness('#j') );
```

node.isParent()
Get whether the node is a compound parent (i.e. a node containing one or more child nodes)

node.isChild()
Get whether the node is a compound child (i.e. contained within a node)

nodes.parent()
Get the compound parent node of each node in the collection.

nodes.parent([selector])
nodes.parents()
Aliases: `nodes.ancestors()`
Get all compound ancestor nodes (i.e. parents, parents' parents, etc.) of each node in the collection.

nodes.parents([selector])
nodes.commonAncestors()
Get all compound ancestors common to all the nodes in the collection, starting with the closest and getting progressively farther.

nodes.commonAncestors([selector])

Details

You can get the closest common ancestor via `nodes.commonAncestors().first()` and the farthest via `nodes.commonAncestors().last()`, because the common ancestors are in descending order of closeness.

nodes.orphans()
Get all orphan (i.e. has no compound parent) nodes in the calling collection.

nodes.orphans([selector])
nodes.nonorphans()
Get all nonorphan (i.e. has a compound parent) nodes in the calling collection.

nodes.nonorphans([selector])
nodes.children()
Get all compound child (i.e. direct descendant) nodes of each node in the collection.

nodes.children([selector])
nodes.descendants()
Get all compound descendant (i.e. children, children's children, etc.) nodes of each node in the collection.

nodes.descendants([selector])
nodes.siblings()
Get all sibling (i.e. same compound parent) nodes of each node in the collection.

nodes.siblings([selector])

Selectors

Notes & caveats

A selector functions similar to a CSS selector on DOM elements, but selectors in Cytoscape.js instead work on collections of graph elements. Note that wherever a selector may be specified as the argument to a function, a `eles.filter()`-style filter function may be used in place of the selector. For example:

```
cy.$('#j').neighborhood(function(){
  return this.isEdge();
});
```

The selectors can be combined together to make powerful queries in Cytoscape.js, for example:

```
// get all nodes with weight more than 50 and height strictly less than
180
cy.elements("node[weight >= 50][height < 180]");
```

Selectors can be joined together (effectively creating a logical OR) with commas:

```
// get node j and the edges coming out from it
cy.elements('node#j, edge[source = "j"]');
```

It is important to note that strings need to be enclosed by quotation marks:

```
//cy.filter('node[name = Jerry]'); // this doesn't work
cy.filter('node[name = "Jerry"]'); // but this does
```

Note that metacharacters (^ $ \ / ( ) | ? + * [ ] { } , .) need to be escaped:

```
cy.filter('#some\\$funky\\@id');
```

Group, class, & ID

`node`, `edge`, or `*` (group selector) Matches elements based on group (`node` for nodes, `edge` for edges, `*` for all).

`.className` Matches elements that have the specified class (e.g. use `.foo` for a class named "foo").

`#id` Matches element with the matching ID (e.g. `#foo` is the same as `[id = 'foo']`)

Data

`[name]`
Matches elements if they have the specified data attribute defined, i.e. not `undefined` (e.g. `[foo]` for an attribute named "foo"). Here, `null` is considered a defined value.

`[^name]`
Matches elements if the specified data attribute is not defined, i.e. `undefined` (e.g `[^foo]`). Here, `null` is considered a defined value.

`[?name]`
Matches elements if the specified data attribute is a truthy value (e.g. `[?foo]`).

`[!name]`
Matches elements if the specified data attribute is a falsey value (e.g. `[!foo]`).

`[name = value]`
Matches elements if their data attribute matches a specified value (e.g. `[foo = 'bar']` or `[num = 2]`).

`[name != value]`
Matches elements if their data attribute doesn't match a specified value (e.g. `[foo != 'bar']` or `[num != 2]`).

`[name > value]`
Matches elements if their data attribute is greater than a specified value (e.g. `[foo > 'bar']` or `[num > 2]`).

`[name >= value]`
Matches elements if their data attribute is greater than or equal to a specified value (e.g. `[foo >= 'bar']` or `[num >= 2]`).

`[name < value]`
Matches elements if their data attribute is less than a specified value (e.g. `[foo < 'bar']` or `[num < 2]`).

`[name <= value]`
Matches elements if their data attribute is less than or equal to a specified value (e.g. `[foo <= 'bar']` or `[num <= 2]`).

`[name *= value]`
Matches elements if their data attribute contains the specified value as a substring (e.g. `[foo *= 'bar']`).

`[name ^= value]`
Matches elements if their data attribute starts with the specified value (e.g. `[foo ^= 'bar']`).

`[name $= value]`
Matches elements if their data attribute ends with the specified value (e.g. `[foo $= 'bar']`).

`@` **(data attribute operator modifier)**
Prepended to an operator so that is case insensitive (e.g. `[foo @$= 'ar']`, `[foo @>= 'a']`, `[foo @= 'bar']`)

`!` **(data attribute operator modifier)**
Prepended to an operator so that it is negated (e.g. `[foo !$= 'ar']`, `[foo !>= 'a']`)

`[[]]` **(metadata brackets)**
Use double square brackets in place of square ones to match against metadata instead of data (e.g. `[[degree > 2]]` matches elements of degree greater than 2). The properties that are supported include `degree`, `indegree`, and `outdegree`.

Compound nodes

`>` **(child selector)**
Matches direct children of the parent node (e.g. `node > node`).

` ` **(descendant selector)**
Matches descendants of the parent node (e.g. `node node`).

`$` **(subject selector)**
Sets the subject of the selector (e.g. `$node > node` to select the parent nodes instead of the children).

State

Animation

- `:animated` : Matches elements that are currently being animated.
- `:unanimated` : Matches elements that are not currently being animated.

Selection

- `:selected` : Matches selected elements.
- `:unselected` : Matches elements that aren't selected.
- `:selectable` : Matches elements that are selectable.
- `:unselectable` : Matches elements that aren't selectable.

LockingStyle

- `:visible` : Matches elements that are visible (i.e. `display: element` and `visibility: visible`).
- `:hidden` : Matches elements that are hidden (i.e. `display: none` or `visibility: hidden`).
- `:transparent` : Matches elements that are transparent (i.e. `opacity: 0` for self or parents).
- `:backgrounding` : Matches an element if its background image is currently loading.
- `:nonbackgrounding` : Matches an element if its background image not currently loading; i.e. there is no image or the image is already loaded).

User interaction:

- `:grabbed` : Matches elements that are being grabbed by the user.
- `:free` : Matches elements that are not currently being grabbed by the user.
- `:grabbable` : Matches elements that are grabbable by the user.
- `:ungrabbable` : Matches elements that are not grabbable by the user.
- `:active` : Matches elements that are active (i.e. user interaction, similar to `:active` in CSS).
- `:inactive` : Matches elements that are inactive (i.e. no user interaction).
- `:touch` : Matches elements when displayed in a touch-based enviroment (e.g. on a tablet).

In or out of graph

- `:removed` : Matches elements that have been removed from the graph.
- `:inside` : Matches elements that have are in the graph (they are not removed).

Compound nodes

- `:parent` : Matches parent nodes (they have one or more child nodes).
- `:child` or `:nonorphan`: Matches child nodes (they each have a parent).
- `:orphan` : Matches orphan nodes (they each have no parent).

Edges

- `:loop` : Matches loop edges (same source as target).
- `:simple` : Matches simple edges (i.e. as would be in a *simple* graph, different source as target).

Style

Style in Cytoscape.js follows CSS conventions as closely as possible. In most cases, a property has the same name and behaviour as its corresponding CSS namesake. However, the properties in CSS are not sufficient to specify the style of some parts of the graph. In that case, additional properties are introduced that are unique to Cytoscape.js.

For simplicity and ease of use, specificity rules are completely ignored in stylesheets. For a given style property for a given element, the last matching selector wins.

Format

The style specified at initialisation can be in a function format, in a plain JSON format, or in a string format — the plain JSON format and string formats being more useful if you want to pull down the style from the server.

String format

Note that the trailing semicolons for each property are mandatory. Parsing will certainly fail without them.

An example style file:

```
/* comments may be entered like this */
node {
  background-color: green;
}
```

At initialisation:

```
cytoscape({
  container: document.getElementById('cy'),

  // ...

  style: 'node { background-color: green; }' // probably previously loaded
via ajax rather than hardcoded

  // , ...
});
```

Plain JSON format

```
cytoscape({
  container: document.getElementById('cy'),
```

```
  // ...

  style: [
    {
      selector: 'node',
      style: {
        'background-color': 'red'
      }
    }

    // , ...
  ]

  // , ...
});
```

Function format

```
cytoscape({
  container: document.getElementById('cy'),

  // ...

  style: cytoscape.stylesheet()
    .selector('node')
      .style({
        'background-color': 'blue'
      })

      // ...


  // , ...
});
```

You may alternatively use `css` in place of `style`, e.g. `.selector( ... ).css( ... )` or `{ selector: ..., css: ... }`.

Function values

In the JSON or function stylesheet formats, it is possible to specify a function as the value for a style property. In this manner, the style value can be specified via a function on a per-element basis.

Note that if using functions as style values, it will not be possible to serialise and deserialise your

stylesheet to JSON proper.

Example:

```
cytoscape({
  container: document.getElementById('cy'),

  // ...

  style: cytoscape.stylesheet()
    .selector('node')
      .style({
        'background-color': function( ele ){ return ele.data('bg') }

        // which works the same as

        // 'background-color': 'data(bg)'
      })

      // ...


  // , ...
});
```

Using a function as a style property value may be convenient in certain cases. However, it may not be a performant option. Thus, it may be worthwhile to use caching if possible, such as by using the lodash `_.memoize()` function.

Property types

- Colours may be specified by name (e.g. `red`), hex (e.g. `#ff0000` or `#f00`), RGB (e.g. `rgb(255, 0, 0)`), or HSL (e.g. `hsl(0, 100%, 50%)`).
- Values requiring a number, such as a length, can be specified in pixel values (e.g. `24px`), unitless values that are implicitly in pixels (`24`), or em values (e.g. `2em`).
- Opacity values are specified as numbers ranging on `0 <= opacity <= 1`.
- Time is measured in units of ms or s.

Mappers

In addition to specifying the value of a property outright, the developer may also use a mapper to dynamically specify the property value.

If a mapping is defined, either define the mapped data for all elements or use selectors to limit the mapping to elements that have the mapped data defined. For example, the selector `[foo]` will apply only to elements with the data field `foo` defined.

- **`data()`** specifies a direct mapping to an element's data field. For example, `data(descr)` would map a property to the value in an element's `descr` field in its data (i.e. `ele.data("descr")`). This is useful for mapping to properties like label text content (the `content` property).
- **`mapData()`** specifies a linear mapping to an element's data field. For example, `mapData(weight, 0, 100, blue, red)` maps an element's weight to gradients between blue and red for weights between 0 and 100. An element with `ele.data("weight") === 0` would be mapped to blue, for instance. Elements whose values fall outside of the specified range are mapped to the extremity values. In the previous example, an element with `ele.data("weight") === -1` would be mapped to blue.
- **`function( ele ){ ... }`** A function may be passed as the value of a style property. The function has a single `ele` argument which specifies the element for which the style property value is being calculated. The function must specify a valid value for the corresponding style property for all elements that its corresponding selector block applies. Note that while convenient, these functions ought to be inexpensive to execute, ideally cached with something like lodash's `_.memoize()`.

Node body

Shape:

- **`width`** : The width of the node's body. This property can take on the special value `label` so the width is automatically based on the node's label.
- **`height`** : The height of the node's body. This property can take on the special value `label` so the height is automatically based on the node's label.
- **`shape`** : The shape of the node's body; may be `rectangle`, `roundrectangle`, `ellipse`, `triangle`, `pentagon`, `hexagon`, `heptagon`, `octagon`, `star`, `diamond`, `vee`, `rhomboid`, or `polygon` (custom polygon specified via `shape-polygon-points`). Note that each shape fits within the specified `width` and `height`, and so you may have to adjust `width` and `height` if you desire an equilateral shape (i.e. `width !== height` for several equilateral shapes).
- **`shape-polygon-points`** : A space-separated list of numbers ranging on [-1, 1], representing alternating x and y values (i.e. `x1 y1 x2 y2, x3 y3 ...`). This represents the points in the polygon for the node's shape. The bounding box of the node is given by (-1, -1), (1, -1), (1, 1), (-1, 1).

Background:

- **`background-color`** : The colour of the node's body.
- **`background-blacken`** : Blackens the node's body for values from 0 to 1; whitens the node's body for values from 0 to -1.
- **`background-opacity`** : The opacity level of the node's background colour.

Border:

- `border-width` : The size of the node's border.
- `border-style` : The style of the node's border; may be `solid`, `dotted`, `dashed`, or `double`.
- `border-color` : The colour of the node's border.
- `border-opacity` : The opacity of the node's border.

Padding:

A padding defines an addition to a node's dimension. For example, `padding-left` adds to a node's outer (i.e. total) width. This can be used to add spacing around the label of `width: label; height: label;` nodes, or it can be used to add spacing between a compound node parent and its children.

- `padding-left` : The amount of left padding.
- `padding-right` : The amount of right padding.
- `padding-top` : The amount of top padding.
- `padding-bottom` : The amount of bottom padding.

Compound parent:

- `compound-sizing-wrt-labels` : Whether to include labels of descendants in sizing a compound node; may be `include` or `exclude`.

Background image

A background image may be applied to a node's body:

- `background-image` : The URL that points to the image that should be used as the node's background. PNG, JPG, and SVG are supported formats. You may use a data URI to use embedded images, thereby saving a HTTP request.
- `background-image-opacity` : The opacity of the background image.
- `background-width` : Specifies the width of the image. A percent value (e.g. `50%`) may be used to set the image width relative to the node width. If used in combination with `background-fit`, then this value overrides the width of the image in calculating the fitting — thereby overriding the aspect ratio. The `auto` value is used by default, which uses the width of the image.
- `background-height` : Specifies the height of the image. A percent value (e.g. `50%`) may be used to set the image height relative to the node height. If used in combination with `background-fit`, then this value overrides the height of the image in calculating the fitting — thereby overriding the aspect ratio. The `auto` value is used by default, which uses the height of the image.
- `background-fit` : How the background image is fit to the node; may be `none` for original size, `contain` to fit inside node, or `cover` to cover the node.
- `background-repeat` : Whether to repeat the background image; may be `no-repeat`, `repeat-x`, `repeat-y`, or `repeat`.

- **`background-position-x`** : The x position of the background image, measured in percent (e.g. `50%`) or pixels (e.g. `10px`).
- **`background-position-y`** : The y position of the background image, measured in percent (e.g. `50%`) or pixels (e.g. `10px`).
- **`background-clip`** : How background image clipping is handled; may be `node` for clipped to node shape or `none` for no clipping.

Pie chart background

These properties allow you to create pie chart backgrounds on nodes. Note that 16 slices maximum are supported per node, so in the properties `1 <= i <= 16`. Of course, you must specify a numerical value for each property in place of `i`. Each nonzero sized slice is placed in order of `i`, starting from the 12 o'clock position and working clockwise.

You may find it useful to reserve a number to a particular colour for all nodes in your stylesheet. Then you can specify values for `pie-i-background-size` accordingly for each node via a mapper. This would allow you to create consistently coloured pie charts in each node of the graph based on element data.

- **`pie-size`** : The diameter of the pie, measured as a percent of node size (e.g. `100%`) or an absolute length (e.g. `25px`).
- **`pie-i-background-color`** : The colour of the node's ith pie chart slice.
- **`pie-i-background-size`** : The size of the node's ith pie chart slice, measured in percent (e.g. `25%` or `25`).
- **`pie-i-background-opacity`** : The opacity of the node's ith pie chart slice.

Edge line

These properties affect the styling of an edge's line:

- **`width`** : The width of an edge's line.
- **`curve-style`** : The curving method used to separate two or more edges between two nodes; may be `bezier` (default, bundled curved edges), `unbundled-bezier` (curved edges for use with manual control points), `haystack` (very fast, bundled straight edges for which loops and compounds are unsupported), or `segments` (a series of straight lines). Note that `haystack` edges work best with `ellipse`, `rectangle`, or similar nodes. Smaller node shapes, like `triangle`, will not be as aesthetically pleasing. Also note that edge arrows are unsupported for `haystack` edges.
- **`line-color`** : The colour of the edge's line.
- **`line-style`** : The style of the edge's line; may be `solid`, `dotted`, or `dashed`.

Bezier edges

For automatic, bundled bezier edges (`curve-style: bezier`):

- **`control-point-step-size`** : From the line perpendicular from source to target, this value specifies the distance between successive bezier edges.
- **`control-point-distance`** : A single value that overrides `control-point-step-size` with a manual value. Because it overrides the step size, bezier edges with the same value will overlap. Thus, it's best to use this as a one-off value for particular edges if need be.
- **`control-point-weight`** : A single value that weights control points along the line from source to target. The value usually ranges on [0, 1], with 0 towards the source node and 1 towards the target node — but larger or smaller values can also be used.

Unbundled bezier edges

For bezier edges with manual control points (`curve-style: unbundled-bezier`):

- **`control-point-distances`** : A series of values that specify for each control point the distance perpendicular to a line formed from source to target, e.g. `-20 20 -20`.
- **`control-point-weights`** : A series of values that weights control points along a line from source to target, e.g. `0.25 0.5 0.75`. A value usually ranges on [0, 1], with 0 towards the source node and 1 towards the target node — but larger or smaller values can also be used.

Haystack edges

Loop edges and compound parent nodes are not supported by haystack edges. Haystack edges are a more performant replacement for plain, straight line edges.

For fast, straight line edges (`curve-style: haystack`):

- **`haystack-radius`** : A value between 0 and 1 inclusive that indicates the relative radius used to position haystack edges on their connected nodes. The outside of the node is at 1, and the centre of the node is at 0.

Segments edges

For edges made of several straight lines (`curve-style: segments`):

- **`segment-distances`** : A series of values that specify for each segment point the distance perpendicular to a line formed from source to target, e.g. `-20 20 -20`.
- **`segment-weights`** : A series of values that weights segment points along a line from source to target, e.g. `0.25 0.5 0.75`. A value usually ranges on [0, 1], with 0 towards the source node and 1 towards the target node — but larger or smaller values can also be used.

Edge arrow

- **`<pos>-arrow-color`** : The colour of the edge's source arrow.

- `<pos>-arrow-shape` : The shape of the edge's source arrow; may be `tee`, `triangle`, `triangle-tee`, `triangle-backcurve`, `square`, `circle`, `diamond`, or `none`.
- `<pos>-arrow-fill` : The fill state of the edge's source arrow; may be `filled` or `hollow`.

For each edge arrow property above, replace `<pos>` with one of

- `source` : Pointing towards the source node, at the end of the edge.
- `mid-source` : Pointing towards the source node, at the middle of the edge.
- `target` : Pointing towards the target node, at the end of the edge.
- `mid-target`: Pointing towards the target node, at the middle of the edge.

Only mid arrows are supported on haystack edges.

Visibility

- `display` : Whether to display the element; may be `element` for displayed or `none` for not displayed. Note that a `display: none` bezier edge does not take up space in its bundle.
- `visibility` : Whether the element is visible; may be `visible` or `hidden`. Note that a `visibility: hidden` bezier edge still takes up space in its bundle.
- `opacity` : The opacity of the element, ranging from 0 to 1. Note that the opacity of a compound node parent affects the effective opacity of its children.
- `z-index` : An integer value that affects the relative draw order of elements. In general, an element with a higher `z-index` will be drawn on top of an element with a lower `z-index`. Note that edges are under nodes despite `z-index`, except when necessary for compound nodes.

Labels

Basic font styling:

- `color` : The colour of the element's label.
- `label` : The text to display for an element's label.
- `text-opacity` : The opacity of the label text, including its outline.
- `font-family` : A comma-separated list of font names to use on the label text.
- `font-size` : The size of the label text.
- `font-style` : A CSS font style to be applied to the label text.
- `font-weight` : A CSS font weight to be applied to the label text.
- `text-transform` : A transformation to apply to the label text; may be `none`, `uppercase`, or `lowercase`.

Wrapping text:

- `text-wrap` : A wrapping style to apply to the label text; may be `none` for no wrapping (including manual newlines: `\n`) or `wrap` for manual and/or autowrapping.
- `text-max-width` : The maximum width for wrapped text, applied when `text-wrap` is set to

`wrap`. For only manual newlines (i.e. `\n`), set a very large value like `1000px` such that only your newline characters would apply.

Node label alignment:

- **`text-halign`** : The vertical alignment of a node's label; may have value `left`, `center`, or `right`.
- **`text-valign`** : The vertical alignment of a node's label; may have value `top`, `center`, or `bottom`.

Rotating text:

- **`edge-text-rotation`** : Whether to rotate edge labels as the relative angle of an edge changes; may be `none` for page-aligned labels or `autorotate` for edge-aligned labels. This works best with left-to-right text.

Outline:

- **`text-outline-color`** : The colour of the outline around the element's label text.
- **`text-outline-opacity`** : The opacity of the outline on label text.
- **`text-outline-width`** : The size of the outline on label text.

Shadow:

- **`text-shadow-blur`** : The shadow blur distance.
- **`text-shadow-color`** : The colour of the shadow.
- **`text-shadow-offset-x`** : The x offset relative to the text where the shadow will be displayed, can be negative. If you set blur to 0, add an offset to view your shadow.
- **`text-shadow-offset-y`** : The y offset relative to the text where the shadow will be displayed, can be negative. If you set blur to 0, add an offset to view your shadow.
- **`text-shadow-opacity`** : The opacity of the shadow on the text; the shadow is disabled for `0` (default value).

Background:

- **`text-background-color`** : A colour to apply on the text background.
- **`text-background-opacity`** : The opacity of the label background; the background is disabled for `0` (default value).
- **`text-background-shape`** : The shape to use for the label background, can be `rectangle` or `roundrectangle`.

Border:

- **`text-border-opacity`** : The width of the border around the label; the border is disabled for `0` (default value).
- **`text-border-width`** : The width of the border around the label.

- `text-border-style` : The style of the border around the label; may be `solid`, `dotted`, `dashed`, or `double`.
- `text-border-color` : The colour of the border around the label.

Interactivity:

- `min-zoomed-font-size` : If zooming makes the effective font size of the label smaller than this, then no label is shown.
- `text-events` : Whether events should occur on an element if the label receives an event; may be `yes` or `no`. You may want a style applied to the text on `:active` so you know the text is activatable.

Events

- `events` : Whether events should occur on an element (e.g. `tap`, `mouseover`, etc.); may be `yes` or `no`. For `no`, the element receives no events and events simply pass through to the core/viewport.
- `text-events` : Whether events should occur on an element if the label receives an event; may be `yes` or `no`. You may want a style applied to the text on `:active` so you know the text is activatable.

Overlay

These properties allow for the creation of overlays on top of nodes or edges, and are often used in the `:active` state.

- `overlay-color` : The colour of the overlay.
- `overlay-padding` : The area outside of the element within which the overlay is shown.
- `overlay-opacity` : The opacity of the overlay.

Shadow

These properties allow for the creation of shadows on nodes or edges. Note that shadow-blur could seriously impact performance on large graph.

- `shadow-blur` :The shadow blur, note that if greater than 0, this could impact performance.
- `shadow-color` : The colour of the shadow.
- `shadow-offset-x` : The x offset relative to the node/edge where the shadow will be displayed, can be negative. If you set blur to 0, add an offset to view your shadow.
- `shadow-offset-y` : The y offset relative to the node/edge where the shadow will be displayed, can be negative. If you set blur to 0, add an offset to view your shadow.
- `shadow-opacity` : The opacity of the shadow.

Transition animation

- `transition-property` : A comma separated list of style properties to animate in this state.
- `transition-duration` : The length of the transition in seconds (e.g. `0.5s`).
- `transition-delay` : The length of the delay in seconds before the transition occurs (e.g. `250ms`).
- `transition-timing-function` : An easing function that controls the animation progress curve; may be `linear` (default), `spring( tension, friction )` (the demo has details for parameter values), `cubic-bezier( x1, y1, x2, y2 )` (the demo has details for parameter values), `ease`, `ease-in`, `ease-out`, `ease-in-out`, `ease-in-sine`, `ease-out-sine`, `ease-in-out-sine`, `ease-in-quad`, `ease-out-quad`, `ease-in-out-quad`, `ease-in-cubic`, `ease-out-cubic`, `ease-in-out-cubic`, `ease-in-quart`, `ease-out-quart`, `ease-in-out-quart`, `ease-in-quint`, `ease-out-quint`, `ease-in-out-quint`, `ease-in-expo`, `ease-out-expo`, `ease-in-out-expo`, `ease-in-circ`, `ease-out-circ`, `ease-in-out-circ` (a visualisation of easings serves as a reference).

Core

These properties affect UI global to the graph, and apply only to the core. You can use the special `core` selector string to set these properties.

Indicator:

- `active-bg-color` : The colour of the indicator shown when the background is grabbed by the user.
- `active-bg-opacity` : The opacity of the active background indicator.
- `active-bg-size` : The size of the active background indicator.

Selection box:

- `selection-box-color` : The background colour of the selection box used for drag selection.
- `selection-box-border-color` : The colour of the border on the selection box.
- `selection-box-border-width` : The size of the border on the selection box.
- `selection-box-opacity` : The opacity of the selection box.

Texture during viewport gestures:

- `outside-texture-bg-color` : The colour of the area outside the viewport texture when `initOptions.textureOnViewport === true`.
- `outside-texture-bg-opacity` : The opacity of the area outside the viewport texture.

Events

Event object

Events passed to handler callbacks are similar to jQuery event objects in that they wrap native event objects, mimicking their API.

Fields:

- `cy` : a reference to the corresponding core instance
- `cyTarget` : indicates the element or core that first caused the event
- `type` : the event type string (e.g. `"tap"`)
- `namespace` : the event namespace string (e.g. `"foo"` for `"foo.tap"`)
- `data` : additional data object passed by `.trigger()`
- `timeStamp` : Unix epoch time of event

Fields for only user input device events:

- `cyPosition` : indicates the model position of the event
- `cyRenderedPosition` : indicates the rendered position of the event
- `originalEvent` : the original user input device event object

Fields for only layout events:

- `layout` : indicates the corresponding layout that triggered the event (useful if running multiple layouts simultaneously)

Event bubbling

All events that occur on elements get bubbled up to the core. You must take this into consideration when binding to the core so you can differentiate between events that happened on the background and ones that happened on elements. Use the `eventObj.cyTarget` field, which indicates the originator of the event (i.e. `eventObj.cyTarget === cy || eventObj.cyTarget === someEle`).

User input device events

These are normal browser events that you can bind to via Cytoscape.js. You can bind these events to the core and to collections.

- `mousedown` : when the mouse button is pressed
- `mouseup` : when the mouse button is released
- `click` : after `mousedown` then `mouseup`
- `mouseover` : when the cursor is put on top of the target
- `mouseout` : when the cursor is moved off of the target
- `mousemove` : when the cursor is moved somewhere on top of the target
- `touchstart` : when one or more fingers starts to touch the screen
- `touchmove` : when one or more fingers are moved on the screen
- `touchend` : when one or more fingers are removed from the screen

There are also some higher level events that you can use so you don't have to bind to different events

for mouse-input devices and for touch devices.

- `tapstart` or `vmousedown` : normalised tap start event (either `mousedown` or `touchstart`)
- `tapdrag` or `vmousemove` : normalised move event (either `touchmove` or `mousemove`)
- `tapdragover` : normalised over element event (either `touchmove` or `mousemove`/`mouseover`)
- `tapdragout` : normalised off of element event (either `touchmove` or `mousemove`/`mouseout`)
- `tapend` or `vmouseup` : normalised tap end event (either `mouseup` or `touchend`)
- `tap` or `vclick` : normalised tap event (either `click`, or `touchstart` followed by `touchend` without `touchmove`)
- `taphold` : normalised tap hold event
- `cxttapstart` : normalised right-click mousedown or two-finger `tapstart`
- `cxttapend` : normalised right-click `mouseup` or two-finger `tapend`
- `cxttap` : normalised right-click or two-finger `tap`
- `cxtdrag` : normalised mousemove or two-finger drag after `cxttapstart` but before `cxttapend`
- `cxtdragover` : when going over a node via `cxtdrag`
- `cxtdragout` : when going off a node via `cxtdrag`

Collection events

These events are custom to Cytoscape.js. You can bind to these events for collections.

- `add` : when an element is added to the graph
- `remove` : when an element is removed from the graph
- `select` : when an element is selected
- `unselect` : when an element is unselected
- `lock` : when an element is locked
- `unlock` : when an element is unlocked
- `grab` : when an element is grabbed by the mouse cursor or a finger on a touch input
- `drag` : when an element is grabbed and then moved
- `free` : when an element is freed (i.e. let go from being grabbed)
- `position` : when an element changes position
- `data` : when an element's data is changed
- `scratch` : when an element's scratchpad data is changed
- `style` : when an element's style is changed

Graph events

These events are custom to Cytoscape.js, and they occur on the core.

- `layoutstart` : when a layout starts running
- `layoutready` : when a layout has set initial positions for all the nodes (but perhaps not final positions)
- `layoutstop` : when a layout has finished running completely or otherwise stopped running

- `load` : when a new graph is loaded via initialisation or `cy.load()`
- `ready` : when a new instance of Cytoscape.js is ready to be interacted with
- `initrender` : when the first frame is drawn by the renderer (useful for synchronising with image exports etc)
- `done` : when a new instance of Cytoscape.js is ready to be interacted with and its initial layout has finished running
- `pan` : when the viewport is panned
- `zoom` : when the viewport is zoomed

Layouts

The function of a layout is to set the positions on the nodes in the graph. Layouts are extensions of Cytoscape.js such that it is possible for anyone to write a layout without modifying the library itself.

Several layouts are included with Cytoscape.js by default, and their options are described in the sections that follow with the default values specified. Note that you must set `options.name` to the name of the layout to specify which one you want to run.

null

The `null` layout puts all nodes at (0, 0). It's useful for debugging purposes.

Options

```
var options = {
  name: 'null',

  ready: function(){}, // on layoutready
  stop: function(){} // on layoutstop
};

cy.layout( options );
```

random

The `random` layout puts nodes in random positions within the viewport.

Options

```
var options = {
  name: 'random',

  fit: true, // whether to fit to viewport
```

```
  padding: 30, // fit padding
  boundingBox: undefined, // constrain layout bounds; { x1, y1, x2, y2 }
or { x1, y1, w, h }
  animate: false, // whether to transition the node positions
  animationDuration: 500, // duration of animation in ms if enabled
  animationEasing: undefined, // easing of animation if enabled
  ready: undefined, // callback on layoutready
  stop: undefined // callback on layoutstop
};

cy.layout( options );
```

preset

The `preset` layout puts nodes in the positions you specify manually.

Options

```
var options = {
  name: 'preset',

  positions: undefined, // map of (node id) => (position obj); or
function(node){ return somPos; }
  zoom: undefined, // the zoom level to set (prob want fit = false if set)
  pan: undefined, // the pan level to set (prob want fit = false if set)
  fit: true, // whether to fit to viewport
  padding: 30, // padding on fit
  animate: false, // whether to transition the node positions
  animationDuration: 500, // duration of animation in ms if enabled
  animationEasing: undefined, // easing of animation if enabled
  ready: undefined, // callback on layoutready
  stop: undefined // callback on layoutstop
};

cy.layout( options );
```

grid

The `grid` layout puts nodes in a well-spaced grid.

Options

```
var options = {
```

```
  name: 'grid',

  fit: true, // whether to fit the viewport to the graph
  padding: 30, // padding used on fit
  boundingBox: undefined, // constrain layout bounds; { x1, y1, x2, y2 }
or { x1, y1, w, h }
  avoidOverlap: true, // prevents node overlap, may overflow boundingBox
if not enough space
  avoidOverlapPadding: 10, // extra spacing around nodes when
avoidOverlap: true
  condense: false, // uses all available space on false, uses minimal
space on true
  rows: undefined, // force num of rows in the grid
  cols: undefined, // force num of columns in the grid
  position: function( node ){}, // returns { row, col } for element
  sort: undefined, // a sorting function to order the nodes; e.g.
function(a, b){ return a.data('weight') - b.data('weight') }
  animate: false, // whether to transition the node positions
  animationDuration: 500, // duration of animation in ms if enabled
  animationEasing: undefined, // easing of animation if enabled
  ready: undefined, // callback on layoutready
  stop: undefined // callback on layoutstop
};

cy.layout( options );
```

circle

The `circle` layout puts nodes in a circle.

Options

```
var options = {
  name: 'circle',

  fit: true, // whether to fit the viewport to the graph
  padding: 30, // the padding on fit
  boundingBox: undefined, // constrain layout bounds; { x1, y1, x2, y2 }
or { x1, y1, w, h }
  avoidOverlap: true, // prevents node overlap, may overflow boundingBox
and radius if not enough space
  radius: undefined, // the radius of the circle
  startAngle: 3/2 * Math.PI, // where nodes start in radians
  sweep: undefined, // how many radians should be between the first and
```

```
last node (defaults to full circle)
  clockwise: true, // whether the layout should go clockwise (true) or
counterclockwise/anticlockwise (false)
  sort: undefined, // a sorting function to order the nodes; e.g.
function(a, b){ return a.data('weight') - b.data('weight') }
  animate: false, // whether to transition the node positions
  animationDuration: 500, // duration of animation in ms if enabled
  animationEasing: undefined, // easing of animation if enabled
  ready: undefined, // callback on layoutready
  stop: undefined // callback on layoutstop
};

cy.layout( options );
```

concentric

The `concentric` layout positions nodes in concentric circles, based on a metric that you specify to segregate the nodes into levels. This layout sets the `concentric` value in `ele.scratch()`.

Options

```
var options = {
  name: 'concentric',

  fit: true, // whether to fit the viewport to the graph
  padding: 30, // the padding on fit
  startAngle: 3/2 * Math.PI, // where nodes start in radians
  sweep: undefined, // how many radians should be between the first and
last node (defaults to full circle)
  clockwise: true, // whether the layout should go clockwise (true) or
counterclockwise/anticlockwise (false)
  equidistant: false, // whether levels have an equal radial distance
betwen them, may cause bounding box overflow
  minNodeSpacing: 10, // min spacing between outside of nodes (used for
radius adjustment)
  boundingBox: undefined, // constrain layout bounds; { x1, y1, x2, y2 }
or { x1, y1, w, h }
  avoidOverlap: true, // prevents node overlap, may overflow boundingBox
if not enough space
  height: undefined, // height of layout area (overrides container height)
  width: undefined, // width of layout area (overrides container width)
  concentric: function(node){ // returns numeric value for each node,
placing higher nodes in levels towards the centre
  return node.degree();
```

```
  },
  levelWidth: function(nodes){ // the variation of concentric values in
each level
  return nodes.maxDegree() / 4;
  },
  animate: false, // whether to transition the node positions
  animationDuration: 500, // duration of animation in ms if enabled
  animationEasing: undefined, // easing of animation if enabled
  ready: undefined, // callback on layoutready
  stop: undefined // callback on layoutstop
};

cy.layout( options );
```

breadthfirst

The `breadthfirst` layout puts nodes in a hierarchy, based on a breadthfirst traversal of the graph.

Options

```
var options = {
  name: 'breadthfirst',

  fit: true, // whether to fit the viewport to the graph
  directed: false, // whether the tree is directed downwards (or edges can
point in any direction if false)
  padding: 30, // padding on fit
  circle: false, // put depths in concentric circles if true, put depths
top down if false
  spacingFactor: 1.75, // positive spacing factor, larger => more space
between nodes (N.B. n/a if causes overlap)
  boundingBox: undefined, // constrain layout bounds; { x1, y1, x2, y2 }
or { x1, y1, w, h }
  avoidOverlap: true, // prevents node overlap, may overflow boundingBox
if not enough space
  roots: undefined, // the roots of the trees
  maximalAdjustments: 0, // how many times to try to position the nodes in
a maximal way (i.e. no backtracking)
  animate: false, // whether to transition the node positions
  animationDuration: 500, // duration of animation in ms if enabled
  animationEasing: undefined, // easing of animation if enabled
  ready: undefined, // callback on layoutready
  stop: undefined // callback on layoutstop
};
```

```
cy.layout( options );
```

cose

The `cose` (Compound Spring Embedder) layout uses a physics simulation to lay out graphs. It works well with noncompound graphs and it has additional logic to support compound graphs well.

It was implemented by Gerardo Huck as part of Google Summer of Code 2013 (Mentors: Max Franz, Christian Lopes, Anders Riutta, Ugur Dogrusoz).

Based on the article "A layout algorithm for undirected compound graphs" by Ugur Dogrusoz, Erhan Giral, Ahmet Cetintas, Ali Civril and Emek Demir.

The `cose` layout is very fast and produces good results. The `cose-bilkent` extension is an evolution of the algorithm that is more computationally expensive but produces near-perfect results.

Options

```
var options = {
  name: 'cose',

  // Called on `layoutready`
  ready               : function() {},

  // Called on `layoutstop`
  stop                : function() {},

  // Whether to animate while running the layout
  animate             : true,

  // The layout animates only after this many milliseconds
  // (prevents flashing on fast runs)
  animationThreshold  : 250,

  // Number of iterations between consecutive screen positions update
  // (0 -> only updated on the end)
  refresh             : 20,

  // Whether to fit the network view after when done
  fit                 : true,

  // Padding on fit
  padding             : 30,
```

```
  // Constrain layout bounds; { x1, y1, x2, y2 } or { x1, y1, w, h }
  boundingBox         : undefined,

  // Extra spacing between components in non-compound graphs
  componentSpacing    : 100,

  // Node repulsion (non overlapping) multiplier
  nodeRepulsion       : function( node ){ return 400000; },

  // Node repulsion (overlapping) multiplier
  nodeOverlap         : 10,

  // Ideal edge (non nested) length
  idealEdgeLength     : function( edge ){ return 10; },

  // Divisor to compute edge forces
  edgeElasticity      : function( edge ){ return 100; },

  // Nesting factor (multiplier) to compute ideal edge length for nested
edges
  nestingFactor       : 5,

  // Gravity force (constant)
  gravity             : 80,

  // Maximum number of iterations to perform
  numIter             : 1000,

  // Initial temperature (maximum node displacement)
  initialTemp         : 200,

  // Cooling factor (how the temperature is reduced between consecutive
iterations
  coolingFactor       : 0.95,

  // Lower temperature threshold (below this point the layout will end)
  minTemp             : 1.0,

  // Whether to use threading to speed up the layout
  useMultitasking     : true
};

cy.layout( options );
```

Layout manipulation

Layouts have a set of functions available to them, which allow for more complex behaviour than the primary run-one-layout-at-a-time usecase. A new, developer accessible layout can be made via `cy.makeLayout()`.

layout.run()
Aliases: `layout.start()`
Start running the layout.

Details

If the layout is asynchronous (i.e. continuous), then calling `layout.run()` simply starts the layout. Synchronous (i.e. discrete) layouts finish before `layout.run()` returns. Whenever the layout is started, the `layoutstart` event is emitted.

The layout will emit the `layoutstop` event when it has finished or has been otherwise stopped (e.g. by calling `layout.stop()`). The developer can bind to `layoutstop` using `layout.on()` or setting the layout options appropriately with a callback.

Examples

```
var layout = cy.makeLayout({ name: 'random' });

layout.run();
```

layout.stop()
Stop running the (asynchronous/discrete) layout.

Details

Calling `layout.stop()` stops an asynchronous (continuous) layout. It's useful if you want to prematurely stop a running layout.

Examples

```
var layout = cy.makeLayout({ name: 'cose' });

layout.run();

// some time later...
setTimeout(function(){
  layout.stop();
}, 100);
```

layout.off(events [, handler])

- events
  A space separated list of event names.

- handler [optional]
  A reference to the handler function to remove.

layout.trigger()
Aliases: `layout.emit()`
Trigger one or more events on the layout.

layout.trigger(events [, extraParams])

- events
  A space separated list of event names to trigger.

- extraParams [optional]
  An array of additional parameters to pass to the handler.

Animations

An animation represents a visible change in state over a duration of time for a single element. Animations can be generated via `cy.animation()` (for animations on the viewport) and `ele.animation()` (for animations on graph elements).

Animation manipulation

ani.play()
Requests that the animation be played, starting on the next frame. If the animation is complete, it restarts from the beginning.

Examples

```
var jAni = cy.$('#j').animation({
  style: {
    'background-color': 'red',
    'width': 75
  },
  duration: 1000
});

jAni.play();
```

ani.playing()
Get whether the animation is currently playing.

ani.progress() et al
Get or set how far along the animation has progressed.

ani.progress()
Get the progress of the animation in percent.

ani.progress(progress)
Set the progress of the animation in percent.

- progress
  The progress in percent (i.e. between 0 and 1 inclusive) to set to the animation.

ani.time()
Get the progress of the animation in milliseconds.

ani.time(time)
Set the progress of the animation in milliseconds.

- time
  The progress in milliseconds (i.e. between 0 and the duration inclusive) to set to the animation.

ani.rewind()
Rewind the animation to the beginning.

ani.fastforward()
Fastforward the animation to the end.

Examples

```
var jAni = cy.$('#j').animation({
  style: {
    'background-color': 'red',
    'width': 75
  },
  duration: 1000
});

// set animation to 50% and then play
jAni.progress(0.5).play();
```

ani.pause()
Pause the animation, maintaining the current progress.

Examples

```
var j = cy.$('#j');
```

```
var jAni = j.animation({
  style: {
    'background-color': 'red',
    'width': 75
  },
  duration: 1000
});

jAni.play();

// pause about midway
setTimeout(function(){
  jAni.pause();
}, 500);
```

ani.stop()
Stop the animation, maintaining the current progress and removing the animation from any associated queues.

Details

This function is useful in situations where you don't want to run an animation any more. Calling `ani.stop()` is analogous to calling `ele.stop()` in that the animation is no longer queued.

Calling `ani.stop()` makes animation frames faster by reducing the number of animations to check per element per frame. You should call `ani.stop()` when you want to clean up an animation, especially in situations with many animations. You can still reuse a stopped animation, but an animation that has not been stopped can not be garbage collected unless its associated target (i.e. element or core instance) is garbage collected as well.

Examples

```
var j = cy.$('#j');
var jAni = j.animation({
  style: {
    'background-color': 'red',
    'width': 75
  },
  duration: 1000
});

jAni.play();

// stop about midway
```

```
setTimeout(function(){
  jAni.stop();
}, 500);
```

ani.completed()
Aliases: `ani.complete()`
Get whether the animation has progressed to the end.

ani.apply()
Apply the animation at its current progress.

Details

This function allows you to step directly to a particular progress of the animation while it's paused.

Examples

```
var jAni = cy.$('#j').animation({
  style: {
    'background-color': 'red',
    'width': 75
  },
  duration: 1000
});

jAni.progress(0.5).apply();
```

ani.applying()
Get whether the animation is currently applying.

ani.reverse()
Reverse the animation such that its starting conditions and ending conditions are reversed.

Examples

```
var jAni = cy.$('#j').animation({
  style: {
    'background-color': 'red',
    'width': 75
  },
  duration: 1000
});

jAni
  .play() // start
```

```
  .promise('completed').then(function(){ // on next completed
    jAni
      .reverse() // switch animation direction
      .rewind() // optional but makes intent clear
      .play() // start again
    ;
  })
;
```

ani.promise()
Get a promise that is fulfilled with the specified animation event.

ani.promise()
Get a promise that is fulfilled with the next `completed` event.

ani.promise(animationEvent)
Get a promise that is fulfilled with the specified animation event.

- animationEvent
  A string for the event name; `completed` or `complete` for completing the animation or `frame` for the next frame of the animation.

Examples

When `ani.apply()` has updated the element style:

```
var jAni = cy.$('#j').animation({
  style: {
    'background-color': 'red',
    'width': 75
  },
  duration: 1000
});

jAni.progress(0.5).apply().promise('frame').then(function(){
  console.log('j has now has its style at 50% of the animation');
});
```

When `ani.play()` is done:

```
var jAni = cy.$('#j').animation({
  style: {
    height: 60
  },
  duration: 1000
```

```
});

jAni.play().promise().then(function(){
  console.log('animation done');
});
```

Architecture

Cytoscape.js uses an event-driven model with a core API. The core has several extensions, each of which is notified of events by the core, as needed. Extensions modify the elements in the graph and notify the core of any changes.

The client application accesses Cytoscape.js solely through the core. Clients do not access extensions directly, apart from the case where a client wishes to write their own custom extension.

The following diagramme summarises the extensions of Cytoscape.js, which are discussed in further detail elsewhere in this documentation.

A diagramme of the architecture of Cytoscape.js, made using Cytoscape.js

Extensions

The extensions below are a curated list. To add your extension, please submit a request that includes your extension's URL and a one line description.

UI extensions

- `cerebralweb` : Enable fast and interactive visualisation of molecular interaction networks stratified based on subcellular localisation or other custom annotation.
- `cxtmenu` : A circular context menu that allows for one-swipe commands on the graph.
- `edge-editation` : Adds handles to nodes and allows creation of different types of edges
- `edgehandles` : UI for connecting nodes with edges.
- `navigator` : A bird's eye view widget of the graph.
- `noderesize` : A node resize control.
- `panzoom` : A panzoom UI widget.
- `qtip` : A wrapper that lets you use qTips on graph elements or the graph background.
- `supportimages` : Support images on Cytoscape.js.
- `toolbar` : Allow a user to create a custom toolbar to add next to a Cytoscape core instance.

Layout extensions

- `arbor` : The Arbor physics simulation layout. It's a basic physics layout.
- `cola` : The Cola.js physics simulation layout. Cola makes beautiful layout results, it animates very

smoothly, and it has great options for controlling the layout.
- `cose-bilkent` : The CoSE layout by Bilkent with enhanced compound node placement. CoSE Bilkent gives near-perfect end results. However, it's more expensive than the version of CoSE directly included with Cytoscape.js.
- `dagre` : The Dagre layout for DAGs and trees.
- `spread` : The speedy Spread physics simulation layout. It tries to use all the viewport space, but it can be configured to produce a tighter result.
- `springy` : The Springy physics simulation layout. It's a basic physics layout.

API

To register an extension, make the following call: `cytoscape( type, name, extension );`

The value of `type` can take on the following values:

- `'core'` : The extension adds a core function.
- `'collection'` : The extension adds a collection function.
- `'layout'` : The extension registers a layout prototype.
- `'renderer'` : The extension registers a renderer prototype.

The `name` argument indicates the name of the extension. For example, `cytoscape( 'collection', 'fooBar', function(){ return 'baz'; } )` registers `eles.fooBar()`.

Autoscaffolding

There exists a Slush project for Cytoscape.js such that the full project scaffolding for a new extension is automatically generated for you. By following the included instructions, you can easily create Cytoscape.js extensions that are well organised, easily maintained, and published to npm, bower, spm, and meteor.

Multitasking

Multitasking APIs are built into Cytoscape.js for extensions like layouts — making layout much faster, for example. The APIs are pulled in from the Weaver library and put on the `cytoscape` object instead of `weaver`. For example, you can make a thread via `cytoscape.thread()` instead of the usual `weaver.thread()`.

Performance

Background

You may notice that performance starts to degrade on graphs with large numbers of elements. This happens for several reasons:

- Performance is a function of graph size, so performance decreases as the number of elements increases.
- The rich visual styles that Cytoscape.js supports can be very expensive. Only drawing circles and straight lines is cheap, but drawing complex graphs is not.
- Edges are particularly expensive to render. Multigraphs become even more expensive with the need for bezier curve edges.
- Interactivity is expensive. Being able to pan, pinch-to-zoom, drag nodes around, et cetera is expensive — especially when having to rerender edges.
- The performance of rendering a (bitmap) canvas is a function of the area that it needs to render. As such, an increased pixel ratio (as in high density displays, like on iPad) can significantly decrease rendering performance.

Optimisations

You can get much better performance out of Cytoscape.js by tuning your options, in descending order of significance:

- **Haystacks make fast edges** : Set your edges `curve-style` to `haystack` in your stylesheet. Haystack edges are straight lines, which are much less expensive to render than `bezier` edges.
- **Batch element modifications** : Use `cy.batch()` to modify many elements at once.
- **Use textured zoom & pan** : Set `textureOnViewport` to `true` in your initialisation options. Rather than rerendering the entire scene, this makes a texture cache of the viewport at the start of pan and zoom operations, and manipulates that instead. Makes panning and zooming much smoother for large graphs.
- **Labels** : Drawing labels is expensive.
  - If you can go without them or show them on tap/mouseover, you'll get better performance.
  - Consider not having labels for edges.
  - Consider setting `min-zoomed-font-size` in your style so that when labels are small — and hard to read anyway — they are not rendered. When the labels are at least the size you set (i.e. the user zooms in), they will be visible.
  - Adding background color and borders to your labels makes more shape to draw on the canvas so you might want to remove them

- **Animations** : You will get better performance without animations. If using animations anyway:
  - `eles.flashClass()` is a cheaper alternative than a smooth animation.
  - Try to limit the number of concurrent animating elements.
  - When using transition animations in the style, make sure `transition-property` is defined only for states that you want to animate. If you have `transition-property` defined in a default state, the animation will try to run more often than if you limit it to particular states you actually want to animate.

- **Function style property values** : While convenient, function style property values can be expensive. Thus, it may be worthwhile to use caching if possible, such as by using the lodash

`_.memoize()` function. If your style property value is a simple passthrough or linear mapping, consider using `data()` or `mapData()` instead.

- **Simplify edge style** : Use solid edges. Dotted and dashed edges are much more expensive to draw, so you will get increased performance by not using them. Edge arrows are also expensive to render, so consider not using them if they do not have any semantic meaning in your graph. Opaque edges with arrows are more than twice as fast as semitransparent edges with arrows.
- **Simplify node style** : Keep your node styles simple to improve performance.
  - Background images are very expensive in certain cases. The most performant background images are non-repeating (`background-repeat: no-repeat`) and non-clipped (`background-clip: none`). For simple node shapes like squares or circles, you can use `background-fit` for scaling and preclip your images to simulate software clipping (e.g. with Gulp so it's automated). In lieu of preclipping, you could make clever use of PNGs with transparent backgrounds.
  - Node borders can be slightly expensive, so you can experiment with removing them to see if it makes a noticeable difference for your use case.

- **Set shadow-blur to 0** if you want shadow and offset them to view them since shadow-blur can rapidely decrease performance.
- **Hide edges during interactivity** : Set `hideEdgesOnViewport` to `true` in your initialisation options. This makes interactivity a lot less expensive by hiding edges during pan, mouse wheel zoom, pinch-to-zoom, and node drag actions.
- **Hide labels during interactivity** : Set `hideLabelsOnViewport` to `true` in your initialisation options. This works similarly to hiding edges on viewport operations.

By making these optimisations, you can increase the performance of Cytoscape.js such that you can have high performance graphs several orders of magnitude greater in size.