Midpoint Review

Team Gamma: Graphical Web Crawler

Group Members:
Ashton Herrington
Greg Niemann
Jason Loomis

Midpoint Review Demonstration
The project may be accessed via the following URL:
http://web.engr.oregonstate.edu/~herrinas/webCrawler/webcrawler/

Please see the **Client-side Usage** information below for information about how to use the project.

IMPORTANT: This project has been developed and tested to work with Google Chrome browser on a desktop platforms. Other browsers do not fully support the functionality built into the project. Mobile platforms are similarly NOT well-supported (since they typically lack the capability to move the cursor to interact with the graph without "clicking" at the same time, and also lack a mouse scroll wheel for zooming).

Midpoint Summary
The project has gone exceedingly well so far. All three members of the project are communicating frequently, and have easily put in over 100 hours each, well before the halfway point of the quarter.

The primary requirements for the Graphical Web Crawler are:

1. Front-end client-side user interface that provides the user the ability to specify a starting URL and specify a depth-first or breadth-first crawl, as well as a numeric limit to terminate the crawl.
2. Back-end server-side crawler that performs the requested crawl.
3. Back-end transmits results to the front-end, which displays them graphically for the user to inspect.
4. The URLs of the crawled pages/nodes will be displayed, and the user may click them to navigate to them in a new tab or window.
5. The option to provide a keyword that the back-end crawler will use as a sentinel to end the crawl, i.e. prior to reaching the numeric limit.
6. The client-side user interface should use cookies to store the previous starting pages, if the user wishes to re-crawl them.

Additionally, we proposed the following features in our project plan:

7. UI will build and display the graph in real-time.
8. Graph will use a physics simulation to organize itself interactively, in real-time.
9. Nodes will display the site favicon, if available.

Nearly all of these major features/requirements of the project have been met at this point. Of the primary requirements, only item 6, the use of cookies to store previous crawls has not yet been implemented. Additionally, at present, clicking the crawled URL nodes opens in the same window as the crawl (so be careful!). And of the additional features, only 9, the use of the favicons, has yet to be implemented. Ashton will be adding these features in the near future.

Though most of the features have been implemented, there is still work to be done with validation and error checking in the code.

Ashton currently tentatively plans to also include the Javascript form validation library known as “Parsley” to validate the front-end form. Further work also needs to be done in regards to the placement of the “back” button to be more compatible with multiple screen sizes. Aside from these tasks, all initial specifications and stretch goals have been met, so Ashton’s remaining would would be to refactor the code into Javascript classes for a cleaner and more re-usable implementation. Jason is planning to further refine the graph-organization code, to avoid error conditions or numerical instabilities in the simulation (divide-by-zero errors, invalid or NaN values). Greg is improving the memory efficiency of the backend crawler engine, preventing zombie threads and limiting the size of the database.

Server-side Information
The back end for the project is being hosted on Google App Engine. None of the front-end is hosted on App Engine, only the crawler API which is called by the front-end system.

The front-end interacts with the crawler through two *routes*. The first route is a POST, which starts a new crawler job. The second is a GET, which returns all new results since the last set of returned results. Because these two routes are separate from the front end, they the crawler system could be used independently in a different project.

POST to https://gammacrawler.appspot.com/crawler with url-encoded form data:
start_page -> (required) a URL of the root (start) page
depth -> (optional) an integer, the depth of the crawl (number of pages away from the root to crawl). Defaults to 1
end_phrase -> (optional) a string of the termination phrase
search_type -> (optional) either 'DFS' or 'BFS', selects the crawl strategy. Defaults to 'BFS'

Returns a JSON object with the following fields:
status -> 'success' if the crawl started successfully, 'failure' if not
job_id -> if successful, contains the ID (an integer) of the crawl job
root -> if successful, contains a PageNode of the root URL
errors -> if unsuccessful, contains a list of strings of errors

On the initial POST, the server first validates the form fields. If they validate, it tries to retrieve the root URL. If successful, adds a record in the job database and spawns a separate worker thread which will execute the crawl.

The crawler thread executes the crawl strategy, either breadth or depth first. For depth first, it randomly selects a link from the page and attempts to follow it, repeating the process until either there are no more links or one works. Then it repeats for the next page until the desired depth is reached or the termination phrase is found.

A breadth first search is much more interesting. The crawler first sets up a pool of worker threads. These threads are used to retrieve links concurrently, allowing the crawler to process and return several hundred pages a minute.

As the crawler follows links and processes pages, it saves results to the database at regular intervals. The results can than be retrieved in using the second API call.

GET to https://gammacrawler.appspot.com/crawler/<job_id>
job_id -> the ID of the crawl job, as returned by the POST

This returns a JSON object with the following properties:
finished -> a boolean of whether the crawl has terminated

new_nodes -> a list of JSON PageNode objects, which each have the following properties
depth -> the page depth, or degree of separation from the root
favicon -> the URL for the page's favicon, or null if no favicon was found
id -> the page's unique ID number
parent -> the page's parent's ID
url -> the URL for the page

By calling this GET route at regular intervals, results from the crawl can be returned incrementally. The entire crawl might include thousands of nodes and take many minutes to execute, but the crawler provides data back at regular (roughly 2-5 second) intervals.

Known bugs
There is still a memory usage problem. If the thread consumes too much memory, App Engine kills and restarts it. This initially resulted in the entire crawl restarting. This has since been corrected, so that a restart thread will just die silently (but it will look like it was successful, so it won't be respawned). This behavior will be fixed to restart the thread where it left off.

Another issue is database management. Currently, the database only cleans up crawler jobs if the final result list is retrieved. This means that for jobs where the front end stops before retrieving all the results, or for jobs where the thread is killed unexpectedly, the results remain in the database indefinitely. This will be fixed, in part, by adding a start time to the database record and a cron job which will clean up stale jobs.

A third issue, or inefficiency rather, is how the results are stored in the database. Currently, each PageNode is stored with its internal list of links. This takes up a lot of unnecessary space, since the links are only required internally to the crawl. The final will have better model for storing page nodes without the links to limit database size.

Additionally, although it isn't a bug, we are going to improve the behavior of the depth first crawl. If the depth first crawl reaches a 'dead end', a page without any external links, prior to reaching the maximum depth or encountering the termination phrase, it currently just ends. We are going to change this behavior, so that it will backtrack and pick a different branch. This will produce a graph with possible false paths, but one which will be reasonably expected to end at the desired depth (or when the termination phrase is encountered).

Client-side Usage
The working project can be seen here:
http://web.engr.oregonstate.edu/~herrinas/webCrawler/webcrawler/

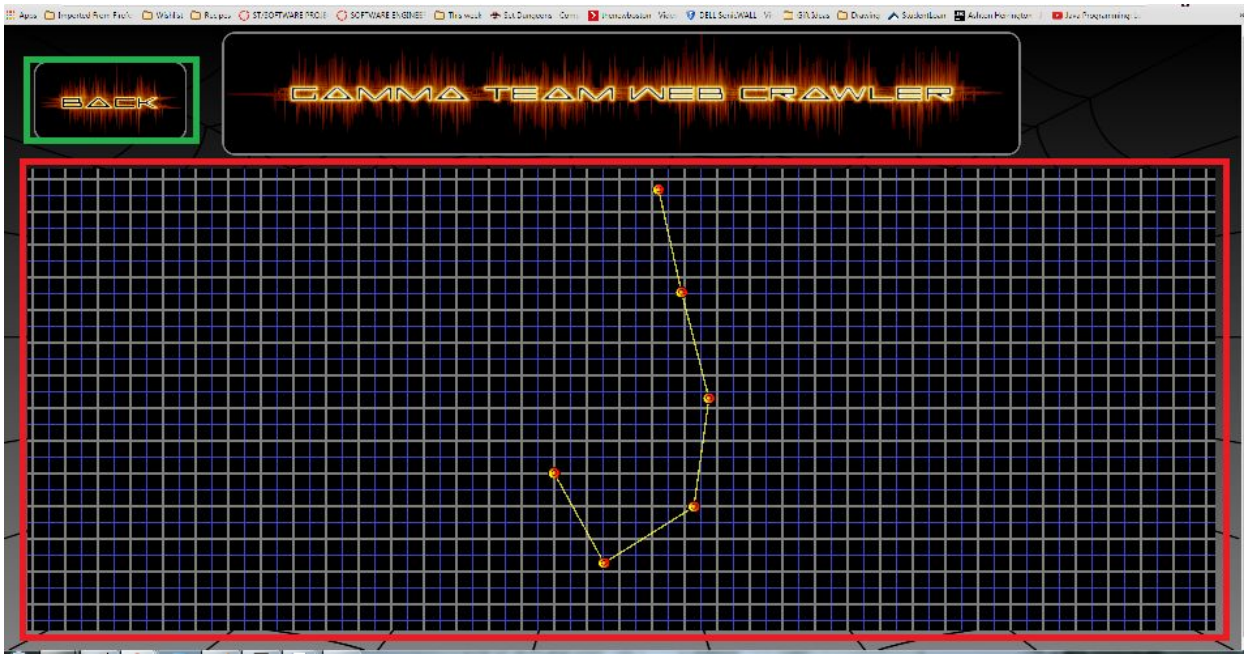
IMPORTANT: This project has been developed and tested to work with Google Chrome browser on a desktop platforms. Other browsers do not fully support the functionality built into the project. Mobile platforms are similarly NOT well-supported (since they typically lack the capability to move the cursor to interact with the graph without "clicking" at the same time, and also lack a mouse scroll wheel for zooming).

When a user first navigates to the page, they are entered with the landing screen shown below:

The page displays our team and project names, and a simple form that the user fills out to begin a graphical web crawl.

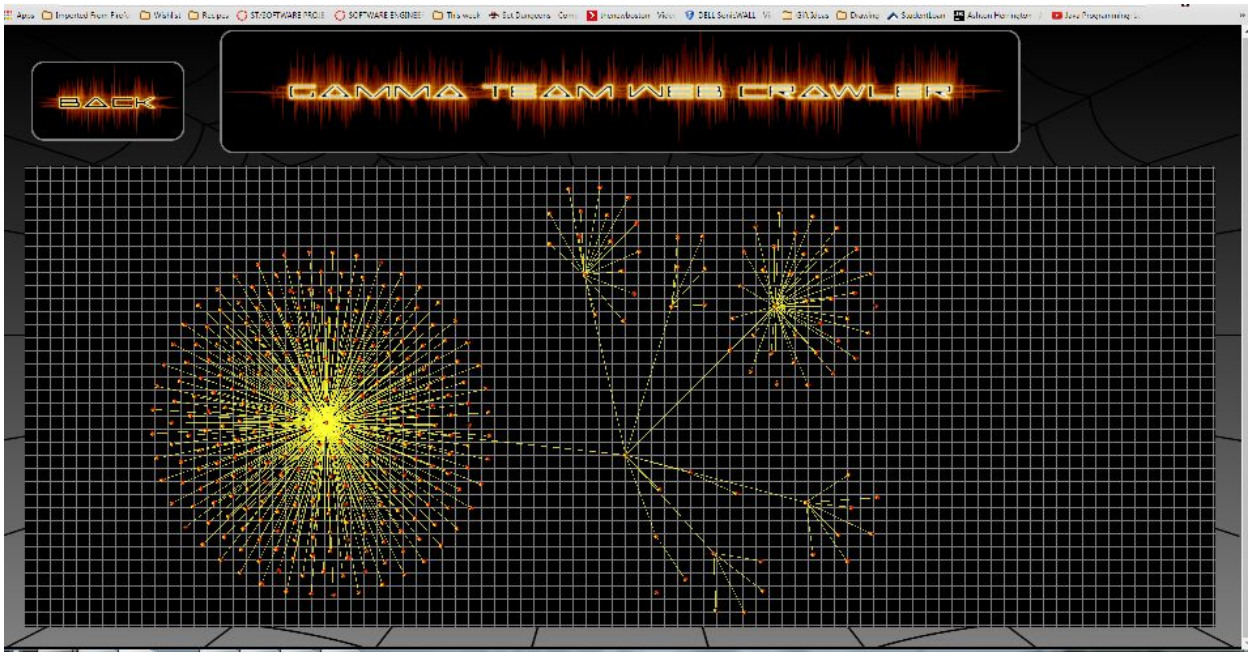
Shown in the red box above are the form fields the user must fill out prior to submitting the crawl for execution (sans the search term, which is an optional field, and the user is informed of this, shown above). A user enters a URL, and toggles between Search type of "Breadth First Search" and "Depth First Search". If Depth First Search is highlighted, such as in the above picture, the field to its right is prefaced with the term "Max Results", this is the maximum number of nodes a depth first search from the beginning URL will return. If instead the user chooses Breadth First Search in the dropdown box, the descriptor of the form field dynamically updates to now say "Search Depth". It is highly recommended at the moment that you choose either a depth of 3 or less if you choose this option. Lastly, the search term which will end the crawl can also be optionally entered by the user here. The spider picture, highlighted above in green, is the button that the user clicks in order to initialize the crawl. Of note, occasionally clicking the button here appears to not result in anything, usually this is due to slow loading of the results. Optimally this could be fixed as a stretch goal of the project by adding a spinner on click.

Shown below on the are the results of choosing the options: URL = ‘google.com’, Search type = ‘DFS’, Max results = 5, and no search term:

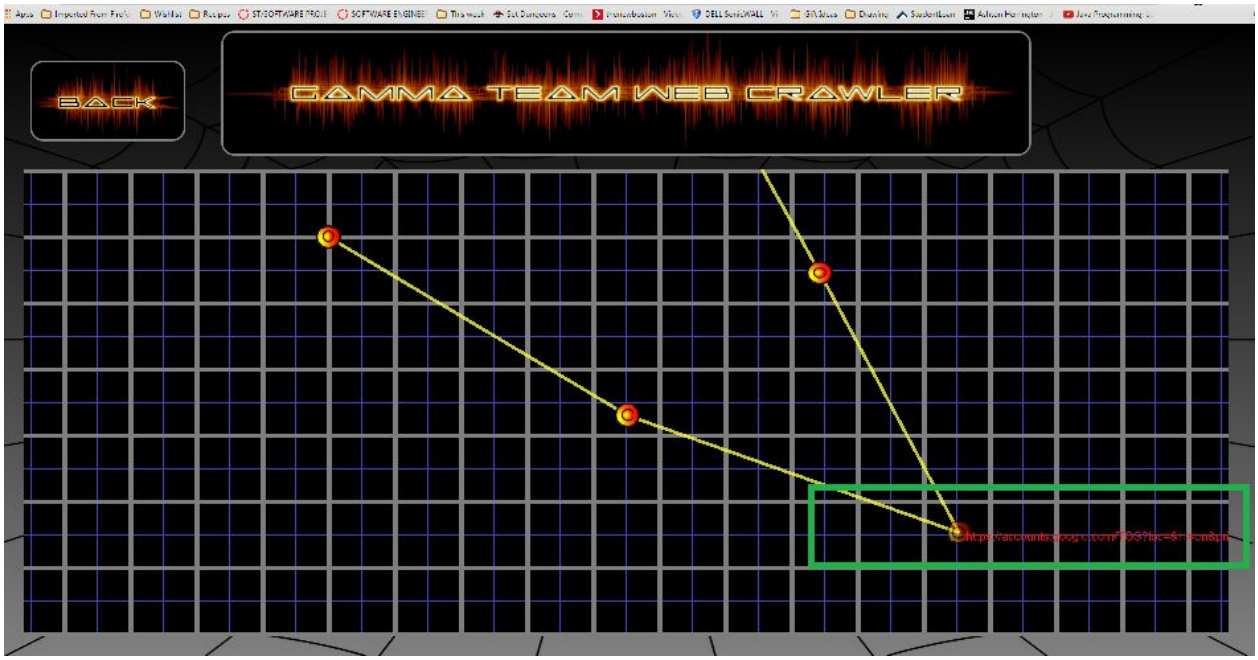


Of note, no page redirect occurs when a user initializes the crawl. Instead, the page’s form is hidden, and is replaced by a grid-map that expands to fill the remaining area of the screen. In this example, the DFS, the nodes initially appear “tangled” and randomly arranged, and move into equilibrium (a “smooth” curve) within a handful of seconds of initializing. The **back button, shown top left**, highlighted in green, **appears after the crawl has finished**. This button allows the user to show the input form and begin a new crawl. A known bug is that the “Gamma Team Web Crawler” title is a fixed width, and the back button is in fixed position, so certain window sizes will cause this button to appear on top of the title area.

Shown below is another example, if the user chooses the options: URL = ‘google.com’, Search type = ‘BFS’, Depth = 2, and no search term:



Both the grid itself and the nodes are fully interactive. Using the position of the mouse, the user can zoom in and out of the grid by rolling the mouse scroll wheel to see a larger frame of view (such is shown in the above picture of the breadth first search). The user can **left-click to grab and drag the grid itself to move the field of view (pan)**. The user can also **left-click to grab individual nodes**, shown below in green, **and move them around the map**. Connected nodes will be pulled along, and other nodes will be repelled by them, as the graph automatically re-orders itself per the physics simulation. Of note, the picture below shows how dragging the highlighted node is having an effect on the adjacent nodes by pulling the entire strand.



Any time the user **hovers the mouse over a node**, e.g. when dragging it, **the URL of that node will be displayed in red**. This **URL is clickable and will take you to that website**. Please note, at the moment, clicking the URL will navigate to the URL *in the same window/tab* of the browser--this behavior will be modified soon to open the URL in a new window/tab.

Graph Ordering

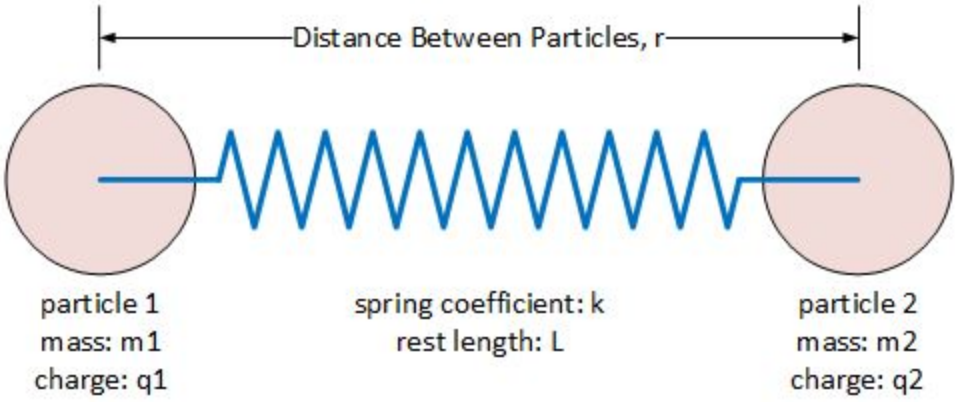
The interactive graph shown above is driven by a physics simulation. The simulation treats each URL/node as a particle with a charge and a mass, connected to other particles by elastic springs.

Particle charge applies a repulsive force, each particle repels every other. Springs between particles apply an elastic attractive force, balancing the charge repulsion. And the mass of the particles provides an inertial response, allowing the particles to resist instantaneous changes to their motion (and stabilizing the simulation).

In the project plan, the idea was to drive the simulation by solving the system of Ordinary Differential Equations (ODEs) provided by applying Newton's Second Law (F = ma) to the particles and springs. This approach was attempted initially, with excellent results for small graphs. However, as the graphs exceeded a few hundred nodes, the simulation would slow dramatically and become unusable.

A simpler, more efficient (though less-accurate) approach was implemented instead. The simpler approach easily provides a sufficient frame rate for usable interactions with a graph of several thousand nodes.

This simpler approach uses the same equations of motion detailed in the project plan:



Hooke's law: *Fspring = k(r-L)*
Coulomb's law : *Fcharge = q1*q2 / r2* (omitting the proportionality constant)
And linear damping: *Fdamping = -b / m1 * v1*

But instead of applying these equations to generate a system of ODEs to be solved, they are applied directly, with forces being applied for short time steps (1/60th of a second, to provide a "real time" simulation when the browser display refresh rate is 60Hz). This is a first-order

approximation of the particle system (where the initial ODE solution implementation provided higher-order accuracy).

This first-order approach is MUCH faster to compute (than the ODE solution), but due to its inaccuracy, can succumb to numerical instability when a large number of particles and springs are present in the simulation. This numerical instability results from VERY LARGE forces being applied (e.g. when two nodes are very near or on top of each other), and results in very large movements of the particles. These large movements resonate and the solution destabilizes when the large values are multiplied together or divided, and the particle motion becomes erratic and eventually overflows the limits of floating point accuracy (e.g. values become infinite or divide-by-zero NaN values).

Fortunately, some tuning and limiting methods were applied to stabilize the solution. Several options were tried, including logarithmic (instead of linear) spring forces and clamping various values. In the end, good results were achieved by clamping the acceleration of any particle at (an arbitrary value of) 100, effectively ignoring VERY LARGE forces. This had the immediate effect of preventing the simulation from running away. The particle positions were also confined to the bounds of the display area (indirectly clamping the force of any given spring).

The simulation still has a minor instability--when there are MANY child nodes--many springs--attached to a parent node, the sum of the forces on the parent node has a slight imbalance (likely due to the first-order simulation and the size of the timestep) that causes the parent node to vibrate in place and child nodes to wander slightly. In the ODE solution, this imbalance is not present to any meaningful degree because of the higher-order solution accuracy and the variable time-stepping applied by the solver.

To combat this imbalance, a simple infinite-impulse-response lowpass filter was applied to the node position. This filter (exponentially) smooths the motion of the by applying an attenuation factor (alpha) to cause the (filtered) output to be the sum of the attenuated input and the previous output value. Wikipedia provides further explanation here: https://en.wikipedia.org/wiki/Low-pass_filter#Simple_infinite_impulse_response_filter

The filter alpha is calculated based on the number of children each node has--the alpha is smaller for nodes with more children and so attenuates their motion more than those nodes with fewer children.

Graph Ordering API

As detailed above, the graph ordering physics simulation has some complexity. For the purpose of integrating that code (Jason's) with the front-end display (Ashton's code), an interface API was designed to handle the complexities and provide access to all of the necessary information that the front-end would need to display the ordered graph. This interface facilitated a quick and (relatively) painless integration of these two front-end components.

The interface also provides a solution to a compatibility issue between the back-end server and the front-end--the server provides a unique identification for each node, and uses this ID to associate parent and child nodes on the graph. This ID is communicated to the front-end, and there is a guarantee that parent nodes will be sent to the client before child nodes, but there is no guarantee of the ordering that the child nodes will be sent in. Unfortunately, the simulation (at least, the ODE solution) builds its representation of the graph sequentially, in the order that nodes are added. If a child node is received that must be inserted in the solution vector, the solution vector must be re-built and all indexing (of parents and children) updated--potentially a catastrophic failure for the front-end display, if something goes wrong.

This interface implements a simple node-lookup dictionary to translate between the server node indices and the graph-ordering engine node indices. No rebuilding or re-indexing required.

The interface exposes the following methods:

`setDisplayScale(width, height, physicalHeight)`
This method sets the physical size of the simulation, corresponding to the pixel-size of the display. The width and height parameters specify the pixel size of the drawing area, and the physical height specifies the equivalent simulation physical height. The function defines the simulation origin to be in the center of the display area, and to have a vertical scale ranging from +physicalHeight/2 at the top of the display area to -physicalHeight/2 at the bottom of the display area. The "physical width" is calculated from the aspect ratio of the display area and scaled per the physicaHeight parameter.

Other simulation parameters are fixed within the interface: the particles are given a mass of 1 and a charge of 0.5, the springs are given a resting length of 0.5, a spring constant of 10, and a damping ratio of 0.5. These parameters were determined empirically to "look good" for this display.

`runSimulation()` and `stopSimulation()`
This pair of methods start and stop the execution of the simulation.

The simulation is stepped, nominally every 1/60th of a second, using the JavaScript `setTimeout()` function to call a callback to calculate the solution. The callback then calls `setTimeout()` to run the calculation again for the next frame. This strategy allows the

simulation to run continuously “in the background” (not explicitly true, since JavaScript is typically single-threaded, but achieves the desired result), without blocking user interaction. It also allows the simulation to lag (e.g. when many nodes are being calculated, or the browser or client machine are otherwise occupied) without queuing additional executions.

Optionally, a callback may be specified that is called *after* each step is calculated. This callback may be specified using the provided `setSimulationStepCallback(function)` method. This callback can be used to, e.g., update graphics with each calculation result.

`addNode(id, parent)`
This method adds a new node to the simulation, connecting it as a child of the specified parent node. This method takes the *server-provided* ID of the node and its parent, and internally manages looking them up and translating their indexing for the simulation. The only pre-condition for this function is that the provided ID MUST be unique, or could corrupt the node lookup dictionary.

`provideCoordinates(id)`
This method calculates the pixel coordinates of the specified node and returns an object containing *px* and *py* fields that contain those coordinates. The returned coordinates may be used to directly position the node in the display area. The id parameter is the *server-provided* ID, and this method manages the lookup internally.

The id parameter is OPTIONAL. If it is omitted when this method is called (most typical usage), the pixel coordinates for ALL of the nodes will be calculated and returned as an array of objects containing *px* and *py* fields that may be used to directly position all of the nodes in the display area.

`nodeDragStart(id)`, `updateNodeCoordinates(id, x, y)`, and `nodeDragEnd(id)`
This set of methods is used to signal user interaction (drag) with the nodes. Like the methods above, the id parameter is the *server-provided* ID, and the methods internally manage the lookup and translation to the simulation indexing. The `nodeDragStart()` method signals to the simulation that the physics should not be applied to the specified node, i.e. when a drag is beginning. The `updateNodeCoordinates()` method signals the simulation that the indicated node should be moved to the specified coordinates. The x and y parameters are the *pixel coordinates*; this function internally manages scaling them to the simulation physical scale. The `nodeDragEnd()` method signals to the simulation that physics should be resumed for the specified node, i.e. when the drag is completed.

`getSimulation()`
This method returns the simulation object to provide unrestricted access to it, in case additional control or options are required that are not exposed by the interface. The caller must understand the complexities of the simulation if this method is to be used.