

# **Final Report**

## **Team Gamma: Graphical Web Crawler**

Group Members:  
Ashton Herrington  
Greg Niemann  
Jason Loomis

August 12, 2016

<https://gammacrawler.appspot.com>

# Introduction

We have created a graphical web crawler that allows users to create an interactive map to explore the interconnections (hyperlinks) among web pages. Users interact with this service by entering a URL from which to start exploring and other parameters in a form. Users can explore via either a breadth first or depth first search from their URL of interest. Upon submitting the form information, the web crawler generates a map, and it is dynamically constructed within a grid map area for the user to view. The grid itself allows users to zoom in and out, drag the grid around, and drag individual map nodes. When a user highlights one of the map nodes the URL the node represents appears. Users can click on these nodes to open these URLs in a separate window.

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.

As will be detailed in the remainder of this report, all of these requirements have been met and the additional features implemented.

## Final Report Demonstration

The project may be accessed via the following URL: <https://gammacrawler.appspot.com>  
Please see the **Client-side Usage** information below for specific, detailed information about how to use the project.

# Client Side Usage

**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).

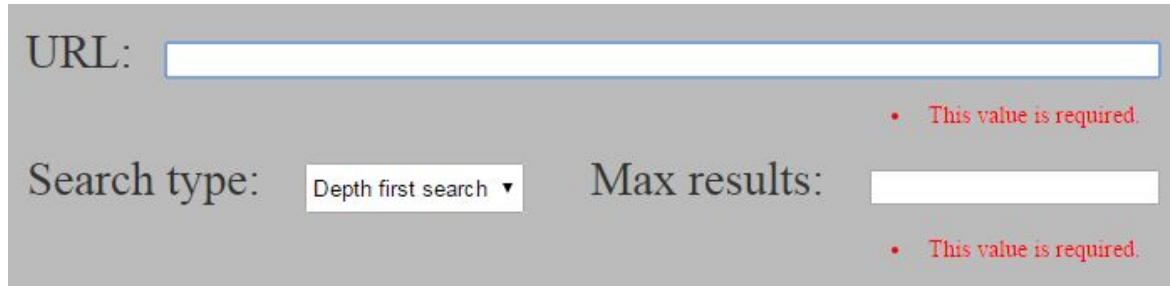
When a user first navigates to the URL of the crawler service (<https://gammacrawler.appspot.com>) they are greeted by the landing screen shown below:



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 that you choose a depth of 3 or less if you choose this option. Lastly, a 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.

When providing form information to the crawler, the front-end form validation prevents the user from leaving critical fields blank. As in shown below, if the user attempts to submit the form with missing fields the form validation fails and they are prompted to enter this information.



URL:

Search type:  Max results:

- This value is required.
- This value is required.

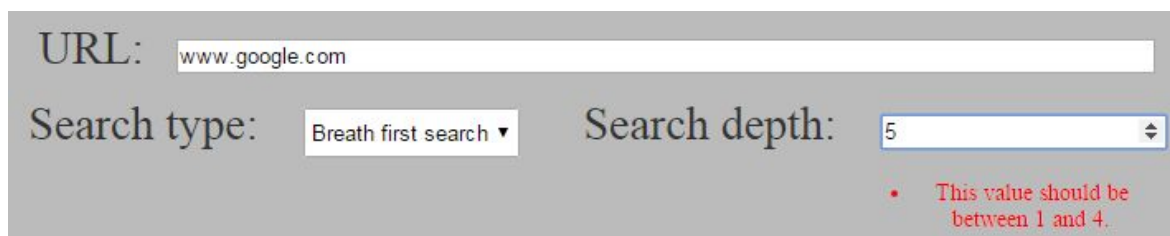
Further, there are limits in place to the number of nodes a user can specify in a breadth first search or the number of levels a user can specify in a breadth first search. Entering numbers outside these ranges causes a second type of validation to prevent users from entering values that would result in an extremely poor viewing experience.



URL:

Search type:  Max results:

- This value should be between 1 and 1000.



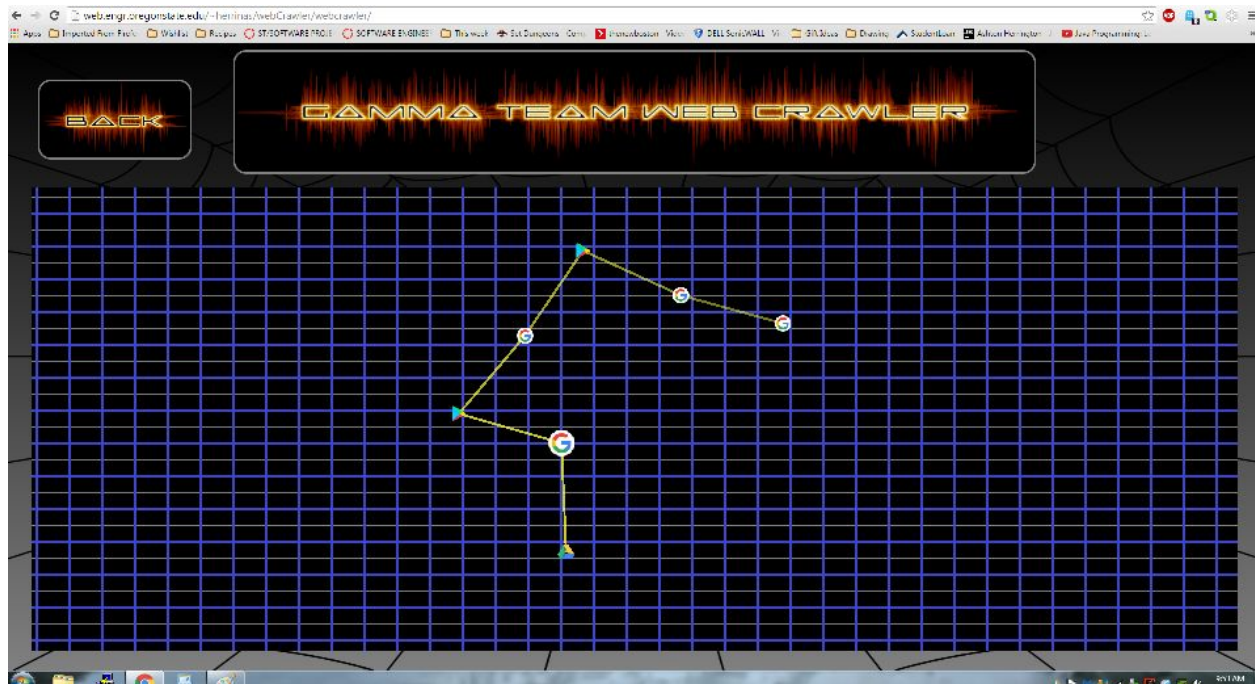
URL:

Search type:  Search depth:

- This value should be between 1 and 4.

Providing valid values for these fields results in the disappearance of these error messages.

Upon providing valid search parameters and submitting the form, the form itself disappears and is replaced by a graphic map, shown below.



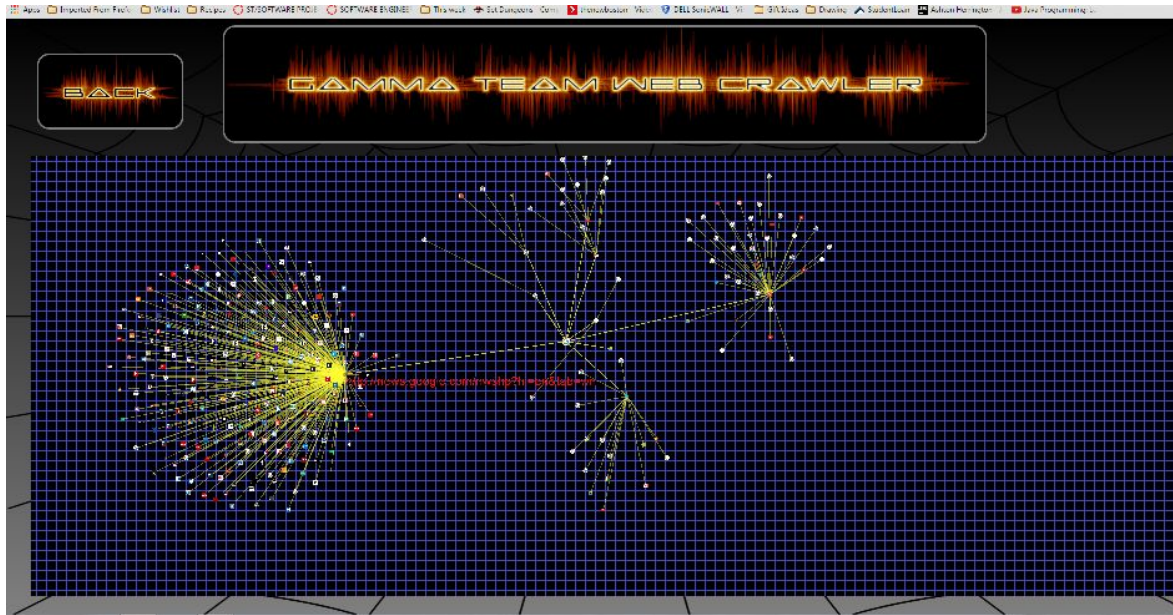
The graphical map of inter-connected websites is now generated on the grid. The node the user specified is approximately 50% larger than the rest of the nodes--allowing the user a point of reference as to where the search began. Further, the tethers that connect the nodes become more opaque as they become farther away from the root node.

The back button, shown top left, allows the user to go back to the form to re-enter a new graphical crawl. This button only appears once all of the nodes have been gathered for the crawl they are currently on--thus preventing users from generating multiple crawls without viewing the information and thereby putting unnecessary strain on the server.

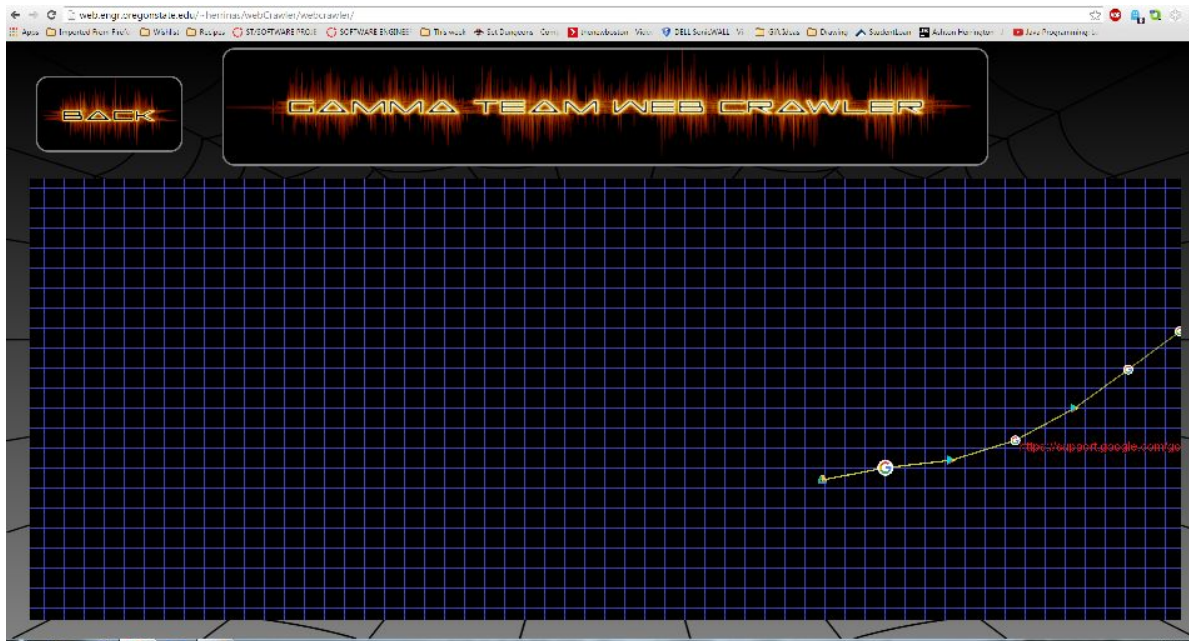
The nodes themselves are denoted by the favicons of the sites themselves (when available). Otherwise a default icon (orange dot) is provided if a favicon does not exist.

The screenshot shows a web browser window with the title "GAMMA TEAM WEB CRAWLER". The main content area is a large blue grid. A yellow path of dots and lines is visible on the grid. A "BACK" button is located in the top left corner. The browser's address bar shows a URL starting with "https://support.google.com/googleplay/feedback\_". The Windows taskbar is visible at the bottom.





The user can also drag the map around (pan) to re-center their field of view. As is shown below, the user has now re-adjusted the view to be to the left of where it previously was:

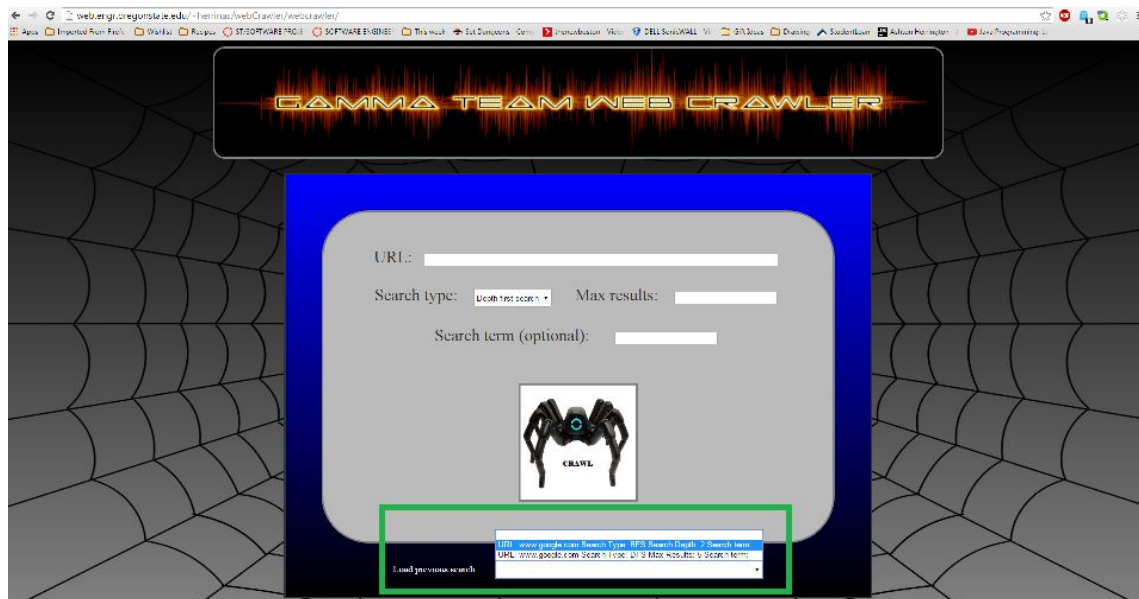


The user can zoom in and out on the map using the mouse scroll wheel. The center of the map after the zoom event will be the location of the mouse pointer when the scroll wheel was activated. Shown below is the result of the user zooming in to focus more on the area containing the nodes of interest.

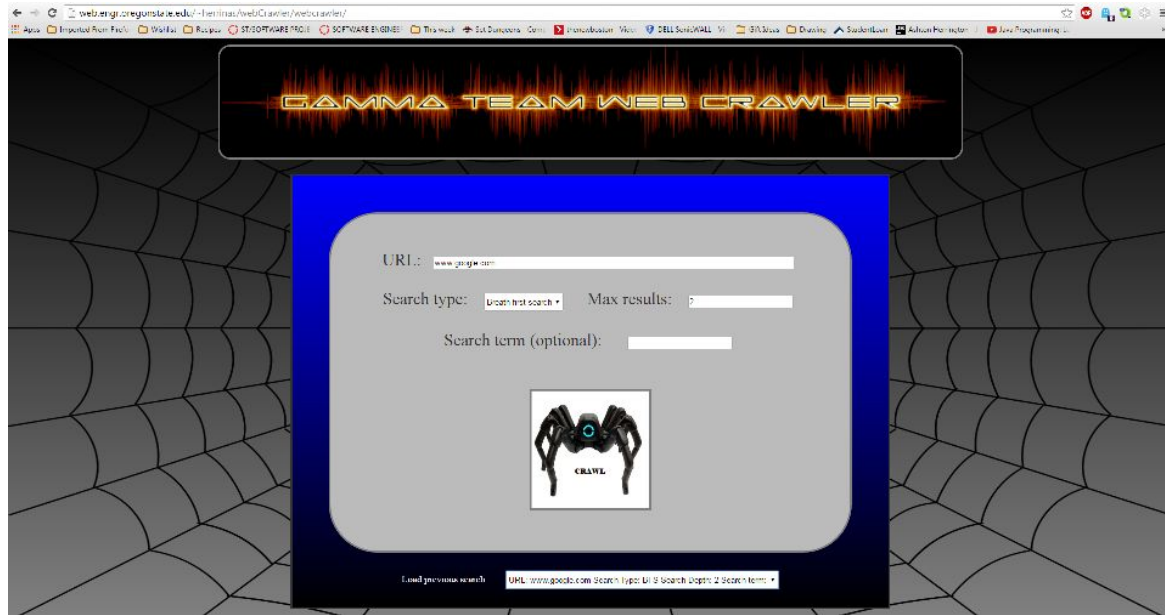




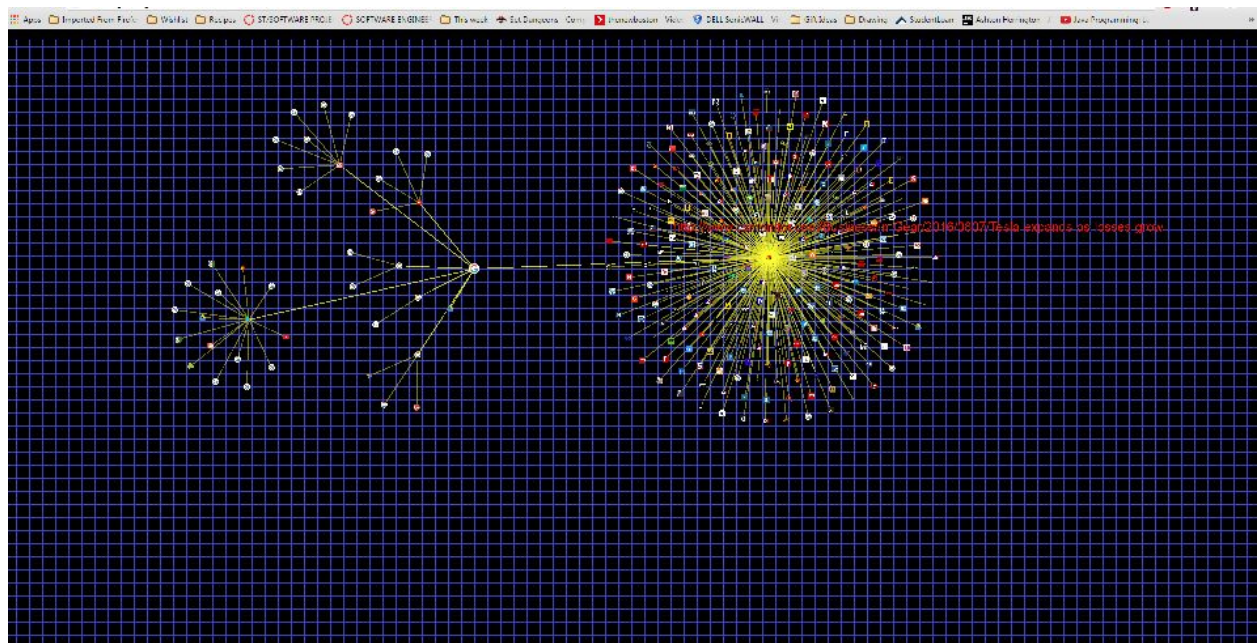
Each time a user performs a search using this service, a cookie is saved to their computer that contains the parameters specified by each search. As is shown below highlighted in green, the user can specify from a dropdown list any of the previous searches they have done. Note: this dropdown does not appear until the user has made at least one previous search.



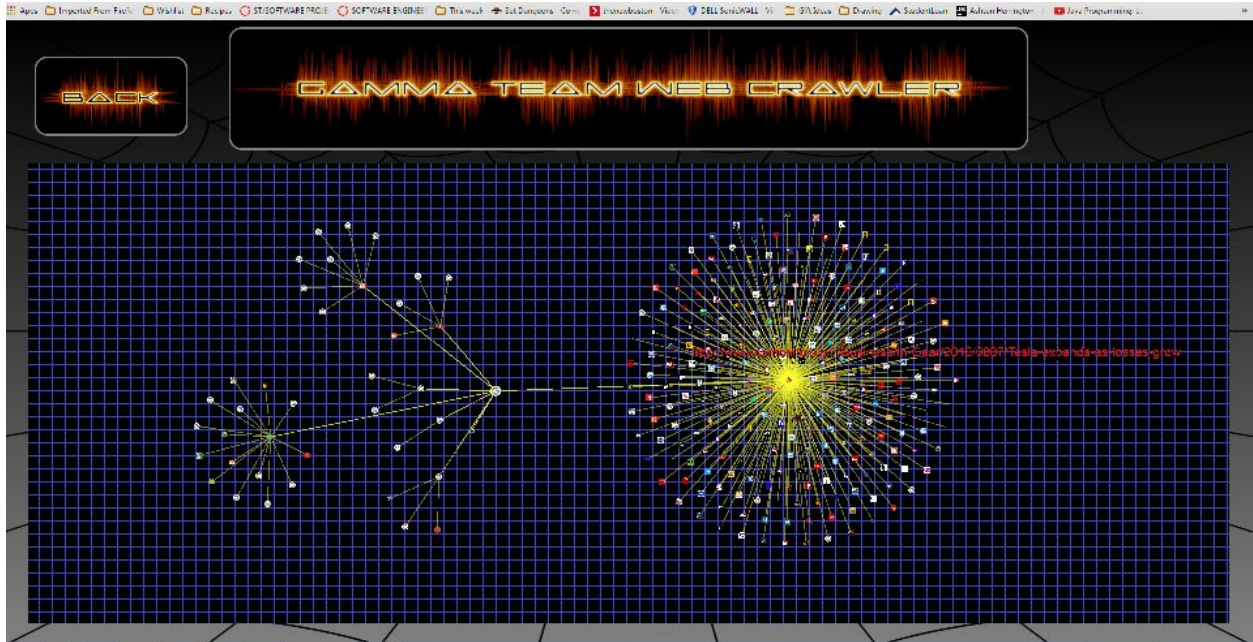
Selecting any of these options will automatically fill out the form fields with these search parameters; the user may modify any of the parameters or simply re-run the crawl.



The user may display the graph full screen by pressing the “Enter” key. The title is then hidden and the graph display fills the browser window:



If the user wishes to exit full screen mode, pressing the “Escape” key causes the window to resize to normal dimensions.



Users can also specify a search term that will end the crawl when the crawler finds a page that contains the word specified in this search term.

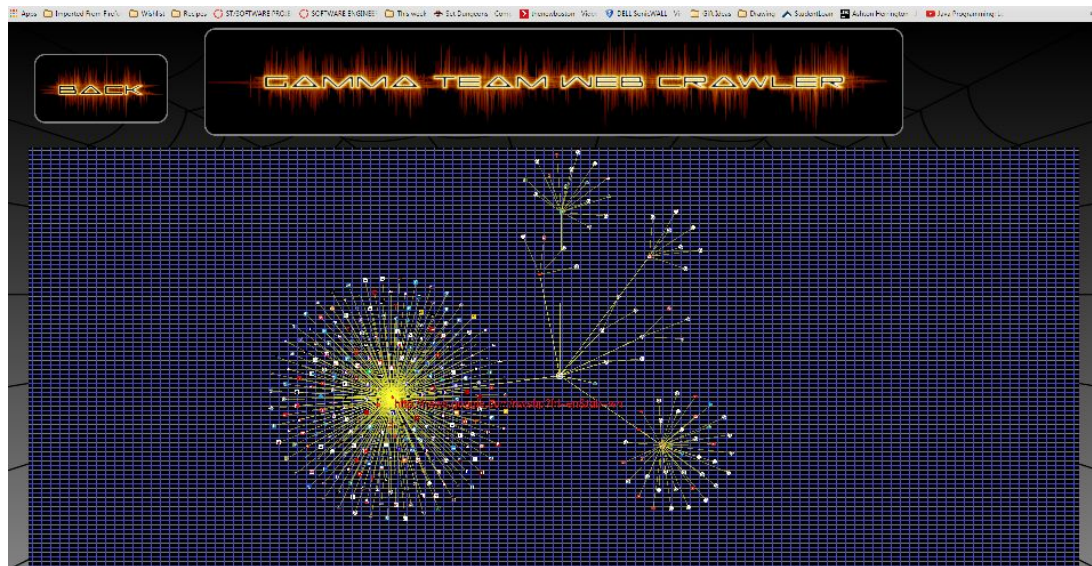
URL:

Search type:  Search depth:

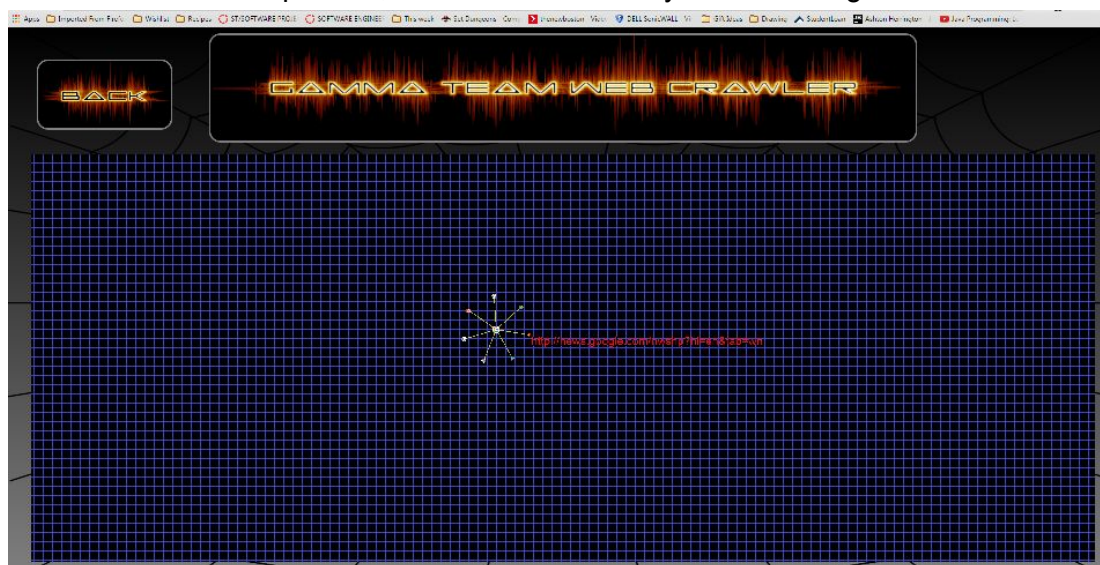
Search term (optional):

Shown below is the same search of [www.google.com](http://www.google.com) breath first search depth 2 with and then without the search term. Of note, the gigantic cluster shown in the first picture is Google News which at the time of writing this contains the word “waterslide” on the site:





With the search term present, the crawl immediately ends at Google News:



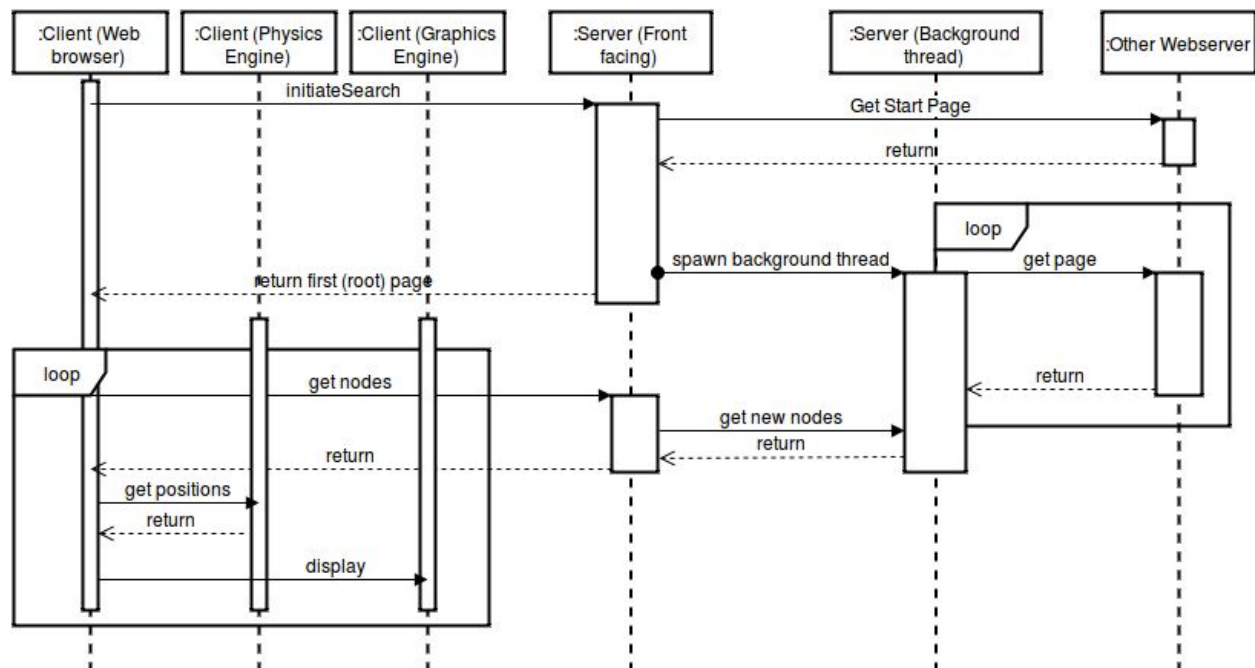
## Overview of the System

The web crawler is split into three major subsystems, each of which is relatively independent of each other. This means that each sub-system is capable of standing on it's own, or being swapped out with another system which provides the same interface. The team specifically wanted to limit the coupling between the subsystems, and designed the interactions of the components to achieve this using design by interface and contract. This allowed the team to work on each component at different speeds, which allowed the project to quickly come together. It also helps prevent breaking one component by changing another.

The crawler is initiated by a request from a client (in our system, using a web browser). The client makes a request to the backend server with the specifics of the crawl, including the start page, the type of crawl, the desired depth of the crawl, and an optional termination phrase. This triggers the server to start a crawl job, and return the job ID and root node info to the client. The client then begins requesting updates from the server (which we refer to as polling) at regular intervals.

On the client side, the resulting page nodes are added to the physics engine. The job of the physics engine is to determine the positioning of all nodes, making the graph self-ordering. The physics engine does this by treating each node as a charged particle, which is attached to its parent by a spring. Using a simulation of the physics of this charged-particle system, the engine calculates how each node repels other nodes (charge force) and is attracted to (or repelled from) its parent node (spring force). This allows for nodes to self-order.

The pages and their positioning are then passed to the graphics engine. The job of the graphics engine is to display the nodes on the screen (in our case, the browser window). The graphics engine is also responsible for user interactions with the simulation. For instance, it is the graphics engine which allows the user to zoom in and out and pan left, right, up and down. It is also the graphics engine which allows the user to drag pages around on the simulation. Finally, the graphics engine allows users to interact with the page itself, by displaying and clicking on hyperlinks.



Graphical Webcrawler Message Sequence Diagram



# System Components

In the following sections, each of the three major components are discussed in detail.

## Backend Server

The backend system is responsible for conducting the actual crawl according using the desired search strategy (either depth first or breadth first), storing and returning results, and retrieving and serving site favicons. Greg Niemann took the lead in the development of the Backend Server component.

The backend is served from Google's App Engine (GAE). App Engine was chosen due to a number of factors, including cost, ease of set-up and monitoring, familiarity with the system and features. GAE provides automatic scaling, an in-depth system monitoring dashboard system, and is generally free for smaller projects (GAE uses a quota system--beyond the daily free quotas, cost apply). Scaling is important, as it allows the 'instance' (our virtual server) to shut down when not in use and respawn as necessary - which reduces costs.

## Public API

The backend system exposes a public API. Clients 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.

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

GET to [https://gammacrawler.appspot.com/crawler/<job\\_id>](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 local 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.

## Implementation

The backend is itself built from a number of sub-components. These are the front-facing framework, the crawler, the page node and the favicon. The chart below shows the interactions of each of these components, which are discussed below.

### Front Facing Framework

The public-facing components (the API and the page server) use the Flask Python web framework (<http://flask.pocoo.org/>). Flask is a Web Service Gateway Interface (WSGI) ([https://en.wikipedia.org/wiki/Web\\_Server\\_Gateway\\_Interface](https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface)) compliant framework, which provides a link between the web server (run by GAE) and the Python code which generates web responses. Flask is the framework which sets up the various routes used by the backend to interact with clients. The public-facing components rely on the other sub-components to actually perform the crawl.

### Crawler

The Crawler object is responsible for actual execution of the crawl. The Crawler is implemented as an abstract base class, with subclasses responsible for defining the actual crawl strategy. Currently, two strategies are defined--depth first and breadth first. Crawler objects are created using the `start_crawler` factory function:

```

def start_crawler(url, search_type, max_depth=3, end_phrase=None):
    """
    Starts a crawler job in the background. Returns the root page (or None if the root is
    inaccessible), and the
    crawler job_id
    :param url: The URL of the root page
    :param search_type: The type of crawl, either BFS or DFS
    :param max_depth: the maximum page depth of the crawl, defaults to 3
    :param end_phrase: The termination phrase, defaults to None
    :return: a 2-tuple of the root page (PageNode object) and the job_id (integer)
    """
    # First get the root page. This validates that the URL is valid, so we can start and
    return something useful
    root = PageNode.make_pagenode(0, url, end_phrase=end_phrase)

    if not root:
        return None, None

    job = JobModel(root=root.url, type=search_type, depth=max_depth, end_phrase=end_phrase)
    job.put()

    # set up the correct crawler, schedule it with defer and return the root and ID
    if search_type == 'DFS':
        crawler = DepthFirstCrawler(job.key, crawler_output_to_datastore, max_depth,
        end_phrase)
    else:
        crawler = BreadthFirstCrawl(job.key, crawler_output_to_datastore, max_depth,
        end_phrase)

    start_thread(crawler, root)

    return root, job.key.id()

```

This function first retrieves the requested start page, ensuring that it is a valid page. It then creates a record in the jobs database (JobModel is the job database model), under which all results will be stored. Afterwards, it creates the correct type of Crawler object for the search and starts it in a new thread. The constructor for Crawler objects takes 4 parameters: a key object, a function for outputting results (in this case, it outputs to the Datastore database), and the max depth and end phrase. The root node is passed when the crawl begins. Finally, it returns both the root page info and the job ID.

The Crawler object is a *callable*--which means that the object can act as a function--which allows it to be started in it's own thread with the `start_thread` function<sup>1</sup>.

```

def __call__(self, root):
    """
    Begins the crawl, terminating either when the end phrase is found or when max depth is
    achieved
    Behind the scenes, uses the abstract crawl method to utilize whatever crawl strategy
    derived classes implement
    """

```

<sup>1</sup> `start_thread` is a simple wrapper around GAE's `deferred.defer` method, which is GAE's method of spawning persisting threads (threads which can outlive the HTTP request handler). Care was made to abstract away as much GAE implementation specific code to as few modules as possible, to allow the project to more easily be ported to another provider if needed.

```

:param root: The root of the search, as either a URL or a PageNode
:return: returns when the crawl has terminated. No return value
"""
# First see if this job has already started. If so, get the start list, set the ID
generator to the next
# ID. If not, set up the start list and ID generator for a new job
if JobModel.get_from_key(self.job_key).has_results():
    logging.warning("Deferred job restarted...loading unfinished nodes")
    start_list = self._get_unfinished_nodes()

    # if the job is done, return
    if TerminationSentinal in start_list:
        return

    last_id = max(n.id for n in start_list) if start_list else 0

    self.id_gen = IDGenerator(last_id + 1)
else:
    if not isinstance(root, PageNode):
        root = PageNode(0, root)

    self.id_gen = IDGenerator()
    start_list = [root]

output_buffer = []
timer_start = time.time()

try:
    for node in self._crawl(start_list):
        output_buffer.append(node)

        # check for the end phrase, if present, break out which triggers finally block
        if node.phrase_found:
            break

        # write every 2 seconds
        if time.time() - timer_start >= 1.5:
            output_buffer.sort(key=lambda n: (n.parent, n.id))
            self.output_func(self.job_key, output_buffer)
            output_buffer = []
            timer_start = time.time()
            unreachable = gc.collect()
            logging.debug("Running garbage collection, {} unreachable
objects".format(unreachable))
            logging.debug("{} total objects".format(len(gc.get_objects())))
        except Exception:
            logging.error(traceback.print_exc(5))

    finally:
        # we're done with the crawl. Append the termination sentinal to the results before
        pushing the last batch
        output_buffer.sort(key=lambda n: (n.parent, n.id))
        output_buffer.append(TerminationSentinal())
        self.output_func(self.job_key, output_buffer)

```

First, the Crawler checks to see whether this is a new job or a restarted job. This is important for two reasons: first, restarted jobs should obviously be restarted where they left off (providing minimal inconvenience to the client). Second, GAE will continually restart deferred work (i.e.

background threads) until they successfully complete. Without this check, the crawler could enter a loop where it starts the crawl from the beginning, terminates unexpectedly (in GAE, this is frequently due to running out of memory, as each instance is capped at 256MB), and is restarted. This would result in the crawl never finishing and, worse, using resources indefinitely. Also, although this check is done regardless of the strategy used, it is up to the strategy subclass to determine how jobs should be restarted. The `_get_unfinished_nodes` method is implemented in each sub-class to accomplish this.

The for loop is the main portion of the `__call__` method. This loop is fed `PageNode` objects from the `_crawl` method, which is defined in each sub-class and implements the crawl strategy. The `_crawl` method is a *generator function*, which means it can generate and yield results back to the calling function, acting as an iterator. This allows results to be fully processed and output before the entire job is done. Further, this allows the super-class to define a generic way to handle results, while delegating to sub-classes how to generate those results.

The body of the for loop appends results to an output buffer, checks to see if the termination phrase is present, and periodically outputs the buffer to the output function. It also periodically runs the garbage collector<sup>2</sup>.

The entire for loop is encased in a try - except - finally structure. The crawler will fail gracefully by design. The finally block, which is executed regardless of whether the try block succeeds or an exception is raised, adds the Termination Sentinel<sup>3</sup>, the signal that the crawl is over, to the results and returns (which signals to GAE that the job is done and does not require a restart).

The two strategies currently implemented are a Depth First Crawl and a Breadth First Crawl. The Depth First is the simpler of the two.

```
def _crawl(self, nodes):
    node_list = nodes

    cur_node = node_list[-1]

    while cur_node and cur_node.depth < self.max_depth:
        # first attempt a random link from this page. If none of the links are valid, we will
        # backtrack up to the
        # parent

        new_node = None
        while not new_node and len(cur_node.links) > 0:
            link = random.choice(cur_node.links)
            cur_node.links.remove(link)
            new_node = PageNode.make_pagenode(self.id_gen, link, cur_node, self.end_phrase)
```

---

<sup>2</sup> While this doesn't have anything to do with the crawl, it does limit memory usage. The crawl produces lots of objects, such as strings, with short life spans. Forcing the garbage collector to run more frequently reduces the amount of new memory which must be allocated.

<sup>3</sup> This is stripped off before being returned, and instead the 'finished' flag is set on the JSON object returned by the API



```

        # if we could not get a working link, backtrack to the parent
        if not new_node:
            cur_node = node_list[cur_node.parent]
        else:
            # otherwise, yield this node, check the phrase, reset the links, increment the
            IDs
            yield new_node

            node_list.append(new_node)
            cur_node = new_node

```

The function is passed a list of nodes. If it is a restarted job, the Depth First implementation of `_get_unfinished_nodes` will pass it the complete list of all previously visited nodes. If it is a new job, the list will contain only the root node. The current node is then set to the last node in the list (for a new job, this is of course the root as it is the only node in the list).

The loop is the main part of this function, and continues until we reach the correct depth, or exhaust all possible paths (in which case, the current node becomes None, the parent of the root). At each step, the function picks a random link from the links on that page, removes it from the link list, and attempts to load it. If the link loads, that page becomes the current node, is yielded and appended to the node list. If no link on the page works, the crawl backs up to the parent node, and resumes trying links. This continues until a path of the desired depth is found, or the parent of the root (which is None) is reached.

The Breadth First crawl is much more complicated, both conceptually and in implementation. The breadth first retrieves all pages at a given depth before moving to the next depth. This has two important implications. First, a very large number of pages can end up being retrieved, and second, if executed serially (one after the other), this search will take a very long time. Therefore, the Breadth First crawl is multi-threaded, allowing for several pages to be retrieved at once. This is possible because retrieving the page is I/O bound, so each thread can wait for the page to be retrieved independently of the other threads.

```

def _crawl(self, nodes):
    current_nodes = nodes

    with futures.ThreadPoolExecutor(max_workers=self.NUM_WORKERS) as executor:
        for depth in range(1, self.max_depth + 1):
            pending_futures = set()
            next_nodes = []

            # for each node on this level, retrieve every link in the node and process
            while len(current_nodes) > 0:
                current_node = current_nodes.pop()

                logging.info("Processing links for parent {}".format(current_node.id))
                for link in current_node:
                    pending_futures.add(executor.submit(PageNode.make_pagenode, self.id_gen,
                                                            link, current_node, self.end_phrase))

            # this ensures that we never have more than twice the number of workers
            while len(pending_futures) > self.PENDING_FUTURE_LIMIT:

```

```

        completed_futures, pending_futures = futures.wait(pending_futures,
                                                            timeout=.25)

        logging.info("CHECKING FUTURES: {} completed, {}
pending".format(len(completed_futures),
len(pending_futures)))

        if len(completed_futures) < 1:
            time.sleep(.5)

        # process and yield the finished futures
        for future in completed_futures:
            try:
                new_node = future.result()
            except:
                new_node = None
                logging.error(traceback.print_exc(5))

            if not new_node:
                continue

            yield new_node

            if depth < self.max_depth:
                next_nodes.append(new_node)

# clean out the rest of the pending futures at this level
logging.info("End level: {} pending futures".format(len(pending_futures)))

# finish off this level before moving to the next
for future in futures.as_completed(pending_futures):
    try:
        new_node = future.result()
    except:
        new_node = None
        logging.error(traceback.print_exc(5))

    if not new_node:
        continue

    yield new_node

    if depth < self.max_depth:
        next_nodes.append(new_node)

current_nodes = next_nodes

logging.info("End level: {} new nodes".format(len(next_nodes)))

```

The first line sets up a *thread pool* (from the concurrent library, which was added to the standard library in Python 3), which is a pool of worker threads. Work can be submitted to the thread pool, which will assign it to worker threads internally, making it much easier to implement this multi-threaded solution. When submitted, a Future object is generated. This object can be checked to see if the work is complete or still pending.

The second line sets up the depth of the search. A for loop is used, since each level is completely processed before the next level is reached. The two implications of this is that at the start of each level, there is no pending work (pending\_futures is an empty set) and all future pages must be stored for the next level.

The while loop is the main part of this function. This loop iterates over all the pages at the current depth. For each page, it further iterates over all the links in the page (the PageNode is an *iterator*, which iterates over its links). For each link, a job is added to the thread pool to retrieve the link and make a PageNode:

```
pending_futures.add(executor.submit(PageNode.make_pagenode, self.id_gen, link,
current_node, self.end_phrase))
```

Here, executor.submit adds the function PageNode.make\_pagenode (with its arguments) to the work that the thread pool must do. It returns a Future object, which is added to the set of pending futures. This set will later be polled for completed futures.

Next, the function checks if the maximum number of pending futures has been reached. If so, execution is paused briefly to wait for any futures to complete, using the futures.wait function. This returns two sets, a set of completed futures and a set of still pending futures. If no futures have completed, we will pause a little longer, than repeat the check, continuing in this fashion until there are less than the maximum number of pending (incomplete) futures.

For each of the futures that did complete, we will get the resulting PageNode by calling the future's result() method. This is encased in a try block, as result() will either return the result of the function call, or re-raise any exception made by the call. Since we aim to fail gracefully, we will treat any exception raised when making the page as an unreachable page (but log it for debugging purposes). If the result is a retrieved page (that is, result returned non-None), this PageNode is yielded, and if the depth is less than the desired search depth, added to the set of next level pages.

Since the Breadth First crawl processes all pages at the current level first, when it exhausts the list of pages at the particular level, it must pause and wait for all the futures to complete. The futures.as\_completed function yields future objects as they complete, ending when all futures have completed. Again, the futures are processed as before. As a note, this near identical code repeat is unfortunate. Although it would make sense to refactor into a separate function, both because GAE does not support Python 3 (which has increased functionality for working with generators and co-routines) and because of the different ways in which the two parts dealt with futures (the first has only a list of completed future, while the second uses as\_completed), there was not an easy way to refactor this block into a separate function.

As shown, the Crawler system only executes the crawl strategy. A separate function returns the results, and the Crawler does not do any page retrieval or parsing. For that, the PageNode class is used.

## PageNode

The PageNode represents a single web page and is responsible for retrieving the site, parsing the links, locating the Favicon (through the Favicon Singleton, discussed below), and checking for the termination phrase. The PageNode also implements methods to serialize itself in two formats: JSON (for transmitting over the web to clients), and Python's pickle format, for internal communication. The serialization allows the PageNode data to be stored or transmitted, without storing unnecessary info such as the page contents or list of links.

An important note is that a PageNode represents a retrievable website, not just a hypothetical one. The PageNode constructor will fail (throw an exception) if the page is not retrievable. PageNodes are meant to be created through the make\_pagenode factory method. This method creates the PageNode in a try block, and returns either a successful PageNode or None if the page is not reachable.

The heavy lifting of PageNode is done in the load method:

```
def load(self, end_phrase=None):
    """
    Loads the page, extracts the links, gets the favicon and looks for the end phrase
    :param end_phrase: end_phrase to search for. Only useful when called from __init__
    :return: returns the content of the page
    throws a TypeError when page retrieval fails
    """
    logging.debug("Retrieving {}".format(self.url))

    res = retrieve_url(self.url)

    # if we could not retrieve a page, raise an exception to ensure that this page is not created
    if res is None or res.status_code != 200:
        raise RuntimeError("Page is not retrievable")

    host = get_host(self.url)
    self._links = list(set(link for link in extract_links(res.content) if not link.startswith(host)))

    if end_phrase and phrase_in_page(res.content, end_phrase):
        self.phrase_found = True
    else:
        self.phrase_found = False

    return res.content
```

This method first attempts to retrieve the page (retrieve\_url is a wrapper around any provider-specific URL function - in this case, GAE's urlfetch module). If the site is not retrievable, an exception is thrown, as expected. Otherwise, two helper functions are called to extract the links and search for the end phrase. An important note about the links - links to the same host are discarded. This limits the links to pages external to the current page (although it still allows the same domain - for instance, [www.google.com](http://www.google.com) would still allow links to news.google.com).

The `extract_links` function is a wrapper around the links regular expression:

```
<a [^>]*href=[""]?(?P<link>(https?:/)?([a-z0-9\-\.\+]{1,2}[a-z0-9]+(?<!\.html)((?/)[^"]*)?)[""] ]
```

This regular expression (complex as it is), does the following. First, it starts with an '`<a`', followed by any number of other characters except the closing `>` until `href=` is encountered. From here, there may be either single or double quotes (or no quotes at all, although that is not standard), followed by the actual URL (which is referred to as 'link'). The URL may start with `http://` or `https://`, but doesn't have to. This is followed by 1 - 2 sections of characters (alphanumeric plus dash), separated by a dot. These sections are followed by a section of alphanumeric characters (this is the TLD, such as `.com` or `.info`). Next, we ensure that this last section is not `.html`, as that would be a local link, which we are filtering out. Following the host, there may be a path or query string. The URL ends with either quotes or a space

The phrase finding function utilizes three regular expressions. The first finds all `<script>` tags within the document, which are then removed from the content. The second does the same with all `<style>` tags. Finally, the third finds all characters which are not inside of tags. This is generally the page content that will be searched for the phrase:

```
scripts_regex = re.compile(r"<script.*?</script>", re.IGNORECASE | re.DOTALL)
styles_regex = re.compile(r"<style.*?</style>", re.IGNORECASE | re.DOTALL)
words_regex = re.compile(r">(P<words>[^\<]+?)<", re.IGNORECASE | re.DOTALL)

def phrase_in_page(page, phrase):
    content = page
    for script in scripts_regex.finditer(content):
        content = content.replace(script.group(), "")

    for style in styles_regex.finditer(content):
        content = content.replace(style.group(), "")

    for word_match in words_regex.finditer(content):
        words = to_utf8(word_match.group())
        if phrase in words:
            return True

    return False
```

## Favicon Singleton

The Favicon Singleton class has a very important job. It is responsible for finding, retrieving and storing a host's favicon.

Originally, the Favicon singleton was only responsible for finding the location of the favicon, and returning that URL. However, during testing it was discovered that favicons were generally inaccessible to client side AJAX requests due to the Cross Origin Resource Sharing policy. Therefore, the backend had to begin retrieving, storing and serving all the favicons needed.



Since this could be resource intensive, the system was designed to be caching and to use as little space as possible.

The Favicon class implements the Singleton design pattern. It does this in a Pythonic way by neither defining a constructor or any instance methods. All methods and data of the Favicon class are at class level. This has the effect of making one object (the class itself).

The only 'public' method of Favicon is the `get_favicon` class method:

```
@classmethod
def get_favicon(cls, url, page=None):
    """
    Retrieves and stores the site's favicon. Returns a local (on this server) URL to the stored favicon
    :param url: site for which we want a favicon
    :return: if a favicon is found, returns a URL to our locally served favicon.
    If no favicon is found, returns None
    """
    with cls.lock:
        cls.total_requests += 1.0
        host, host_key = cls._get_host_key(url)

        if host_key in cls.host_to_hash:
            cls.cache_hits += 1.0
            logging.info("Favicon cache hit! Hit percentage {:.2%}".format(cls.cache_hits / cls.total_requests))
            icon_hash = cls.host_to_hash[host_key]
            if icon_hash:
                filename = icon_hash + '.ico'
                return cls.BASE + filename
            else:
                return None

        logging.info("Favicon cache miss. Hit percentage {:.2%}".format(cls.cache_hits / cls.total_requests))

        if page:
            # attempt to extract from the page first
            icon = cls._extract_favicon_from_page(page, url)

        # if either we didn't get passed the page, or no icon could be extracted, attempt the default route
        if not page or not icon:
            icon = cls._download_favicon(host + '/favicon.ico')

        if not icon:
            cls.host_to_hash[host_key] = None
            return None

        icon_hash = hashlib.md5(icon).hexdigest()
        cls.host_to_hash[host_key] = icon_hash

        if icon_hash not in cls.hash_set:
            save_file(icon, icon_hash + '.ico')
            cls.hash_set.add(icon_hash)

        return cls.BASE + icon_hash + '.ico'
```

This method is passed a URL and optional page content. Since this singleton will be called by multiple threads (via PageNode, when executed through a breadth first crawl, for example), it

begins by acquiring a lock to prevent race conditions with its internal data structures. Although this does slow down multithreaded code somewhat as threads may have to wait for the lock, the gains made through caching more than make up for any loss.

The function first extracts the hostname and host key (the hostname without the leading http:// or https://), then checks the `host_to_hash` dictionary to see if there is already an entry for this `host_key`. The hash refers to the md5 hash of the site's favicon (the actual icon, not a filename). Internally, the favicons are stored on the server with the md5 hash as the first part of the filename (with the extension `.ico`). Therefore, if the `host_key` is in the dictionary, the filename can be built and returned from the stored hash without ever going to disk.

If the `host_key` is not in the dictionary, and a page content has been passed in, the function attempts to extract the favicon from the page. Favicons can be listed in the `<head>` section, using the tags `<link rel='icon' href='URL'>` or `<link rel='shortcut icon' href='URL'>`.

`_extract_favicon_from_page` uses a regular expression to try to extract a URL from that tag, then download and return its contents.

If extraction fails (or no page data was passed), the function attempts to download the favicon directly from the standard location, `{host}/favicon.ico`. If this is unsuccessful, then there is no favicon and so `None` is entered into the dictionary for the `host_key`. If it is successful, the icon is hashed with the md5 algorithm. This hash is then associated with the `host_key` in the dictionary. Next, the hash is checked against the set of all favicon hashes. If it is not present, it is saved to the disk. If it is present, there is no need to save it. The function then returns the filename of the locally saved favicon.

The Favicon singleton possesses a persistent caching ability. That is, on creation it loads its cache from disk, and periodically saves it as newer entries are added. Two member variables, the `host_to_hash` dictionary and the `hash_set` set provide this capability. These are both subclasses of the `FileCache` class:

```
class FileCache(object):
    def __init__(self, data_cls, filename):
        self.new_count = 0
        self.filename = filename
        self.data = data_cls()

        self._load()

    def _load(self):
        logging.info('loading from {}'.format(self.filename))
        raw_data = read_file(self.filename)
        if raw_data is None:
            logging.info('Nothing loaded')
            return

        logging.info('Loaded from file')
        self.data = pickle.load(io.BytesIO(raw_data))

    def _save(self):
```

```

logging.info('Saving to file {}'.format(self.filename))
data = pickle.dumps(self.data)
try:
    save_file(data, self.filename)
except:
    pass
else:
    self.new_count = 0

```

The FileCache constructor is designed to be called from a derived class constructor. It takes 2 arguments. The first is a type and is for the internal data structure to be used. This structure must be picklable, but otherwise can be any type. The second is the filename of the cache file. The `_load` and `_save` functions load and save pickled cache information to the disc.

The subclasses can implement any data structure desired. The HashSet emulates a set object:

```

class HashSet(FileCache):
    def __init__(self):
        super(type(self), self).__init__(set, 'hash_set')

    def __contains__(self, item):
        return item in self.data

    def add(self, item):
        if item not in self.data:
            self.data.add(item)
            self.new_count += 1

            if self.new_count > 5:
                self._save()

```

The `new_count` variable is used to track how many new entries are made in the cache. Periodically, the object can then call `_save` to write itself to the file system. In this manner, the cache can be loaded again when a new instance is spawned, and doesn't have to be rebuilt. This cache can allow for a cache hit percentage of much greater than 70%.

## Front-end and system framework

Ashton's work was split between setting up the front end graphics engine, creating a framework that A. connected the graphics engine and physics engine; and B. creating calls to the API and providing them to the respective engines from part A. Each of these systems will be discussed separately in the following section.

The beginning of the front end development involved creating a simple HTML form for the user to enter information about what type of crawl they wished to view. This process was mostly straightforward, and the primary work here involved CSS styling and beautification of the site. The spiderweb effect shown on the front page was designed by Ashton using Sketchpad Pro.

Small amounts of Javascript were applied to this form. For one, when the user switches between the drop down of depth first search and breadth first search it activates a function that changes the contents of field descriptor to either "Search depth" or "Max results", the following code is how this is achieved:

Dropdown modifier in the html file:

```
<select name="search_type" onchange="switchType(this.value);" style="margin-left: 20px; margin-right: 32px" required= "">
```

Corresponding Javascript:

```
function switchType(value) {
    if (value == 'BFS') {
        $('#search_type').html('Search depth:');
        $('#type_data').attr("data-parsley-range", "[1,4]")
    } else {
        $('#search_type').html('Max results:');
        $('#type_data').attr("data-parsley-range", "[1,1000]")
    }
}
```

Further Javascript modifications to the html were required to prevent the previous searches being shown if there were no cookies containing previous searches present. This was achieved by spanning these sections with an ID that was hidden if no cookies were found, shown below:

Span that hides the previous searches if no cookie exists for previous searches:

```
<span id="loadPreviousSearch">Load previous search</span> <select id="previous_search" onchange="fillForm(this.value);">
```

Lastly, the form must not cause a submit, so additional Javascript prevents the form from being submitted, but instead routes it to a custom AJAX action, shown below:

```
//When the submit "crawl" from the form is pressed, this in combination with stopSubmit
prevents redirect
$(document).ready(function () {
    var submitButton = document.getElementById("submitpic");
    submitButton.addEventListener('click', stopSubmit, false);
});

//Stops form submission, checks form validity, and alters view as needed
function stopSubmit(evt) {
    evt.preventDefault();
    var isValid = $("#crawl").parsley().validate();
    if (isValid) {
        process();
    } else {
        if (!$('#type_data').parsley().isValid()) {
            $("#spacer1").html('');
        }
        if (!$('#url_data').parsley().isValid()) {
            $("#spacer2").html('');
        }
    }
}
```

The logic itself for interacting with the web crawler API is stored within the file `xmlrequests.js`. The primary logic of this file is to generate a `XMLHttpRequest`, and use it for a typical AJAX call. Because the form is not actually submitted, the form acquires the information from the form field by acquiring the values directly out of the form itself and constructing a URL out of them, shown below.

```
//grab the variables of interest from the form
var url = document.forms["crawl"]["url"].value;
var searchType = document.forms["crawl"]["search_type"].value;
var maxResults = document.forms["crawl"]["max_results"].value;
var searchTerm = document.forms["crawl"]["search_term"].value;

//dynamically create a params string from these variables
var params = "start_page=" + url + "&";
params += "search_type=" + searchType + "&";
params += "depth=" + maxResults + "&";
params += "end_phrase=" + searchTerm;
```



Once the server responds to the initial AJAX call, xmlhttprequests.js first saves the information provided by the crawl into the root node, it adds this node to the graphics and physics engines, causes the form to disappear and be replaced by the grid map, and begins long polling for the rest of the nodes from the crawler, shown below.

```
//information from crawler is received in return: jobId, and rootNode stats
jobId = postResponse['job_id'];
var rootId = postResponse['root']['id'];
var rootNode = postResponse['root'];
nodeMap[rootId] = rootNode;

//rootNode is added to the physics engine
physicsEngine.addNode(rootNode['id'], null);
//at this point we start the simulation
if (!started) {
    // physicsEngine.runSimulation();
    started = true;
}
addNode(
    physicsEngine.provideCoordinates(rootNode['id']).px,
    physicsEngine.provideCoordinates(rootNode['id']).py,
    rootNode['url'],
    rootNode['id'],
    Null,
    rootNode['favicon'], 1.6
);
//remove the form, replace it with the interactive map
$('#form').css("visibility", "hidden");
$('#demo').css("visibility", "visible");
//begin polling for futher nodes acquired by the crawl
pollCrawlResults();
```

The end step of the initial AJAX request calls “pollCrawlResults”, this function makes constant calls to the crawler API via setting a timeout to call itself every 2 seconds as long as the response returned from the API does not have the flag “finished” set to true. It parses the contents received on each call and adds all the new nodes to both the graphics and physics engines. Once the flag is set to finished, it no longer makes additional calls, and the back button is presented to the user.

```
var theResponse = JSON.parse(xmlHttp2.responseText);
crawlNodes = theResponse;
//cycle through each of the new nodes found
theResponse['new_nodes'].forEach(function (node) {

//add the node to the nodemap and physics engine
```

```

nodeMap[node['id']] = node;
physicsEngine.addNode(node['id'], node['parent']);
addNode(
    physicsEngine.provideCoordinates(node['id']).px,
    physicsEngine.provideCoordinates(node['id']).py,
    Node['url'],
    Node['id'],
    Node['parent'],
    node['favicon'], 1
);
});

//if the API declares the crawl isn't finished, poll again in 2 seconds
if (!theResponse['finished']) {
    setTimeout('pollCrawlResults()', 2000);
} else {
    //otherwise, display the back button allowing the user to start a new crawl
    $('#backButton').css("visibility", "visible");
}

```

The actual logic to graphically render the results is a combination of both the contents of main.js and tilemap.js. I will discuss main.js here, and discuss tilemap.js in the libraries used section following this one.

Ultimately, main.js is a complete failure in term of OOP style programming. It is effectively just a global file with a series of functions allowing it to easily connect all the other parts of the program. The original goal was to turn this into class, but as more and more functionality was layered into it, it became harder and harder to accomplish.

When the window loads, main.js runs a function to determine the size of the screen. It allows for an offset such that the graphical map will not cover the title at the top of the screen. It further determines if any cookie is saved under the cookie name “gammacrawler”. If this cookie is not null it JSON parses it and adds it to an array of cookies - these are the previous searches the user has done using the crawler. Further, it then goes through the contents of the cookie array and dynamically creates a dropdown list that the user sees at the beginning screen, shown below:

Of note, the getCookie and setCookie functions were taken unaltered from W3 schools.

```

var offset = $('#demo').offset();
ySize = $(window).height() - offset.top - 30;
xSize = $(window).width() - 80;

graphicsEngine = new Main('', xSize, ySize);

```

```

$('#demo').append(graphicsEngine);

// get cookies
var savedCookies = getCookie('gammacrawler');
if (savedCookies != null) {
    cookieArray = JSON.parse(savedCookies);
    cookieArray.sort(
        function(a, b) {
            // sort in descending order by date
            if(a[4] == undefined) return 1;
            if(b[4] == undefined) return -1;
            return b[4] - a[4];
        });

    // remove duplicate cookies
    var cookieSet = {}
    var uniqueCookies = []
    for (var i = 0; i < cookieArray.length; i++) {
        var key = JSON.stringify(cookieArray[i].slice(0, 4));
        if (!cookieSet[key]) {
            uniqueCookies.push(cookieArray[i]);
            cookieSet[key] = true;
        }
    }
    cookieArray = uniqueCookies;
    cookieSet = undefined;

    // store save cleaned cookie array
    var jsonCookie = JSON.stringify(cookieArray);
    setCookie('gammacrawler', jsonCookie, 14);
}
var oldSearches = document.getElementById("previous_search");
if (cookieArray.length == 0){
    oldSearches.style.visibility = 'hidden';
    document.getElementById("loadPreviousSearch").style.visibility = 'hidden';
}

var defaultOption = document.createElement("option");
defaultOption.textContent = ' ';
defaultOption.value = -1;
oldSearches.appendChild(defaultOption);
for(var i = 0; i < cookieArray.length; i++) {
    var option = cookieArray[i];
    var newOption = document.createElement("option");
    var maxResults = option[1] == 'DFS' ? 'Max Results' : 'Search Depth';
    newOption.textContent = "URL: " + option[0] + ' Search Type: ' + option[1] + ' ' +
maxResults + ": " +
    option[2] + ' Search term: ' + option[3];
}

```

```

    newOption.value = i;
    oldSearches.appendChild(newOption);
}

```

To allow the window to automatically be resized, the new x and y sizes of the window need to be recalculated when the window resize event occurs. Further, this information needs to be provided to the renderer so it correctly changes its dimensions. This can occur both on window resize, as well as if the user wishes to toggle fullscreen mode using “Enter” and “Escape” buttons, the code that allows for this functionality is detailed below:

```

window.onresize = function () {
    var offset = $('#demo').offset();
    ySize = $(window).height() - offset.top - 30;
    xSize = $(window).width() - 80;

    //resize map
    renderer.view.style.width = xSize + "px";
    renderer.view.style.height = ySize + "px";
    renderer.resize(xSize, ySize);

    //resize these for tilemap usage
    renderWidth = xSize;
    renderHeight = ySize;
}

var savedOffset = null;

//This allows the user to exit full screen when they press escape button
$(document).keyup(function(e) {
    if (e.keyCode == 27) {
        var offset = $('#demo').offset();
        ySize = $(window).height() - savedOffset - 30;
        xSize = $(window).width() - 80;

        $('#demo').css({top:0, left:0, position:'static'});
        $('#title').show();
        renderer.view.style.width = xSize + "px";
        renderer.view.style.height = ySize + "px";
        renderer.resize(xSize, ySize);

        //resize these for tilemap usage
        renderWidth = xSize;
        renderHeight = ySize;
    }
    if (e.keyCode == 13) {
        var offset = $('#demo').offset();
        savedOffset = offset.top;
    }
}

```

```

//resize map
$('#demo').css({top:0, left:0, position:'absolute'});
$('#title').hide();
renderer.view.style.width = $(window).width() + "px";
renderer.view.style.height = $(window).height() + "px";
renderer.resize($(window).width(), $(window).height());

//resize these for tilemap usage
renderWidth = $(window).width();
renderHeight = $(window).height();
}
});

```

main.js provides a method allowing for nodes to be input into the graphics engine, which allows them to be rendered. Further, it also provides the functionality for the physics engine to send batch information about nodes such that it can provide updated coordinates to the graphics engine for each render cycle. The graphical representation was done using the PIXI.js library, which be discussed in further detail in the below libraries section. The main point of convergence between the physics and graphics engines is the graphicsMap. The receiveCoordinates function alters the position of the nodes within this map which is registered by the graphics map and updates their coordinates each time the map is rendered.

```

//Wrapper that interfaces with graphics engine to update node coordinates after calculations
function receiveCoordinates(nodeArray) {

    nodeArray.forEach(addOrUpdateNode);

    function addOrUpdateNode(item, index, array) {
        //this means that the node is not currently tracked
        if (typeof graphicsMap[item.id] === 'undefined') {
            addNode(item.px, item.py, nodeMap[index].url, item.id, nodeMap[index].parent,
nodeMap[index].favicon, 1);
        } else {
            graphicsMap[item.id][0].position.x = item.px;
            graphicsMap[item.id][0].position.y = item.py;
        }
    }
}

//Add a node to be tracked (by the graphics engine, not physics engine)
function addNode(x, y, url, id, parentId, favicon, faviconscale) {

    var texture;

    if (favicon) {
        texture = PIXI.Texture.fromImage(favicon);
    }
}

```

```

    } else {
        texture = PIXI.Texture.fromImage("images/sunburst.png");
    }

    //originally the sprite were bunnies, kept this for kicks :)
    var bunny = new PIXI.Sprite(texture);
    bunny.anchor.x = 0.5;
    bunny.anchor.y = 0.5;
    bunny.position.x = x;
    bunny.position.y = y;
    bunny.height = 100 * faviconscale;
    bunny.width = 100 * faviconscale;
    bunny.interactive = true;
    bunny.buttonMode = true;
    bunny
        // events for drag start
        .on('mousedown', onDragStart)
        .on('touchstart', onDragStart)
        // events for drag end
        .on('mouseup', onDragEnd)
        .on('mouseupoutside', onDragEnd)
        .on('touchend', onDragEnd)
        .on('touchendoutside', onDragEnd)
        // events for drag move
        .on('mousemove', onDragMove)
        .on('touchmove', onDragMove)
        .on('mouseover', onMouseover);

    graphicsMap[id] = [bunny, url, parentId];
    var hiddenId = new PIXI.Text(id);
    hiddenId.visible = false;
    bunny.addChild(hiddenId);
    currentParentId = parentId;
    tilemap.addChild(bunny);
}

```

The Main class defined within main.js was originally supposed to be the the “graphics engine” object that contained all of these as its public facing methods; however, in practice this method only ended up being something that created and returned the renderer and provided the actual animation loop.

The animation loop itself first requests updated coordinates from the physics engine, it then calls receiveCoordinates to update them for the graphics engine, next it updates the tethers that exist between each of the nodes, and finally calls render on the stage itself. Update tethers is a very important step here (in regards to the graphics rendering) as the new lines need to be calculated between a child and its parent node each time they change locations.



```

//main animation loop, updates tethers/URL/nodes, and repositions "top" elements each
iteration
function animate() {

if (started) {
    physicsEngine.stepSimulation(1/60); // step simulation by 1/60th of a second
    var updates = physicsEngine.provideCoordinates();
    receiveCoordinates(updates);
}
requestAnimationFrame(animate);
updateTethers();

renderer.render(stage);
}

```

Updating the tethers is achieved by looping through the contents of the graphicsMap. Within the forEach loop of the graphicsMap, the item[0] signifies the node itself, and item[2] signifies the parent node (which is saved within the node). Lines are drawn that start at the position of the child node and end at the position of the parent node and then the graphics object is added to the tile map.

Because PIXI.JS gives layer priority to children added last, all the nodes also have to be removed and re-added to the map such that the tethers do not appear on top of the nodes.

Lastly, the URL needs to also update such that if either the user or the physics engine causes it related node to move it moves alongside it, these steps are all highlighted in the code snippet shown below:

```

//new tethers are redrawn every animation loop to correspond to updated location of the
nodes
function updateTethers() {
    tilemap.removeChild(graphics);
    graphics.clear();
    //graphics.lineStyle(10, 0xffff33, 0.8);
    graphicsMap.forEach(function (item, index) {
        var depth = nodeMap[index].depth;
        if (depth > maxDepth) maxDepth = depth;
        if (typeof graphicsMap[index][2] != 'undefined' && graphicsMap[index][2] !=
null) {

            //// choose color
            //var color = maxColor;
            //if (maxDepth > 1) color = Math.round(lerp(maxColor, minColor, (depth -
1) / (maxDepth - 1)));
            //color = (color << 16) ^ (color << 8) ^ 0x33; // build RGB color

```

```

        //graphics.lineStyle(15, color, 0.8);

        // choose transparency
        var alpha = maxAlpha;
        if (maxDepth > 1) alpha = lerp(maxAlpha, minAlpha, (depth - 1) /
(maxDepth - 1));

        graphics.lineStyle(15, 0xffff33, alpha);

        var startX = item[0]['position']['x'];
        var startY = item[0]['position']['y'];
        graphics.moveTo(startX, startY);
        var endX = graphicsMap[item[2]][0]['position']['x'];
        var endY = graphicsMap[item[2]][0]['position']['y'];
        graphics.lineTo(endX, endY);
    }
});
graphics.endFill();
tilemap.addChild(graphics);

//nodes are removed and then replaced to be on top of tethers as pixijs uses a
linked list with
//sprite priority going to last on list
graphicsMap.forEach(function (item, index) {
    tilemap.removeChild(item[0]);
    tilemap.addChild(item[0]);
});

if (popupText.height !== 20/tilemap.zoom * 1.2) {
    popupText.height = 20/tilemap.zoom * 1.2;
    popupText.scale.x = popupText.scale.y;
}

//to have the URL follow the sprite it is attached to, this occurs
if (popupText && currentSprite) {
    popupText.position.x = currentSprite.position.x + 40;
    popupText.position.y = currentSprite.position.y - 10;
}
tilemap.addChild(popupText);
}

```

Lastly, main.js provides a series of event based functions such as onMouseOver, onDragStart, onDragEnd, onDragMove, turnParentDragOn/Off, and onWheelZoom, detailed below:

On mouse over event sets the global currentSprite to the event's target. This helps in subsequent rendering by allowing the URL to always know which node causes the URL to appear. Then, the URL is searched for using the contents of the graphics map, and this is set

equal to the contents of currentURL. Because PIXI.JS gives layer priority to the last added child the text is removed from the map, reconstituted, given the mousedown event to create a hyperlink effect, some styling, and added back to the map.

```
//callback that occurs when a node is moused over
function onMouseover(event) {

    //currentSprite is set as global, and the node is matched to the graphics map
    currentSprite = event['target'];
    graphicsMap.forEach(function (item, index) {
        if (item[0] === currentSprite) {
            currentUrl = item[1];
        }
    });

    //a new URL pops up slightly offset from the location of the node it was spawned from
    tilemap.removeChild(popupText);
    popupText = null;
    popupText = new PIXI.Text(currentUrl, { font: '36px Arial', fill: 0xff1010, dropShadow:
true, dropShadowColor: 0x000000, dropShadowDistance: 3, align: 'center' });
    popupText.on('mousedown', takeHyperlink);
    popupText.interactive = true;
    popupText.buttonMode = true;
    popupText.height = 120/tilemap.zoom * 1.2;
    popupText.scale.x = popupText.scale.y;

    popupText.position.x = event['target']['position']['x'] + 25;
    popupText.position.y = event['target']['position']['y'];
    tilemap.addChild(popupText);
}
```

OnDragStart and onDragEnd are the methods that tell the physics engine that the user is currently dragging a node, so please release (and regain respectively) control over the node during this time. They do this by using the physicsEngine's nodeDragStart and nodeDragEnd methods to inform these engines of these events occurring. They also importantly call turnParentdragOn and turnParentDragOff respectively. These set a flag within the tilemap itself (which is also draggable) such that it turns off the map dragging while a node is being dragged.

```
//callback that occurs on node drag start
function onDragStart(event) {
    tilemap.removeChild(popupText);
    //prevent the map beneath from simultaneous drag, and turn off physics control of node
    turnParentDragOff();
    physicsEngine.nodeDragStart(event['target'].children[0].text);
    this.data = event.data;
    this.alpha = 0.5;
}
```

```

        this.dragging = true;
    }

    //callback that occurs on node drag end
    function onDragEnd(event) {
        //return ability to both drag the canvas as well as physics control of the node
        turnParentDragOn();
        physicsEngine.nodeDragEnd(event['target'].children[0].text);
        this.alpha = 1;
        this.dragging = false;
        this.data = null;
    }

```

When a user drags a node across the screen, it not only needs to be updated for the graphics engine, but it also has to inform the physics engine of how the user has changed the coordinates that it has saved in its system. This is done via the onDragMove function. This function updates the position of the node itself, and updates the coordinates via the physicsEngine updateNodeCoordinates method which is constantly called while the node is being dragged. Further, it also needs to update the popupText position to match that of the node.

```

//on drag move event
function onDragMove(event) {

    //if the node is actively being dragged
    if (this.dragging) {
        //update position based on the position provided by the event
        var newPosition = this.data.getLocalPosition(this.parent);
        this.position.x = newPosition.x;
        this.position.y = newPosition.y;
        //inform the physics engine and related popup text of the new position
        physicsEngine.updateNodeCoordinates(event['target'].children[0].text,
        this.position.x, this.position.y);
        popupText.position.x = newPosition.x + 20;
        popupText.position.y = newPosition.y - 10;
    }
}

```

## Libraries used by the front-end system

To provide form validation for the front-end form. The Parsley.js library was included in the project. To have parsley validate the form, data-parsley-validate="" was added to the form definition. Validating individual fields such as required="" and data-parsley-range="XXXX" to the inputs of the form itself, shown below:

```

<div id="form">
  <form id="crawl" method="POST" data-parsley-validate="">

    URL:<input id="url_data" type="text" name="url" style="margin-left: 20px"
width="600px" size="80" required="">

    <span id="spacer2"><br><br></span>

    Search type:
    <select name="search_type" onchange="switchType(this.value);" style="margin-left:
20px; margin-right: 32px" required="">
      <option value="DFS">Depth first search</option>
      <option value="BFS">Breath first search</option>
    </select>

    <span id="search_type">Max results:</span>
    <input id="type_data" type="number" name="max_results" style="margin-left: 20px"
required="" data-parsley-range="[1,1000]">

    <span id="spacer1"><br><br></span>

    Search term (optional):<input id="st_data" type="text" name="search_term"
style="margin-left: 42px">
    <br><span id="spacer2"><br><br></span>
    <input id="submitpic" type="image" src="images/newCrawl.png" alt="Submit"
width="200" height="200">

  </form>

```

To provide graphics rendering capability, the PIXI.js library was used for our project. A large section of time during the earlier stages of the project was devoted to my learning how to interact with this library as well as its small caveats of usage.

<http://www.pixijs.com/>

The last “library” used was code provided in the below link:

<http://www.bhopkins.net/2014/10/08/draggable-zoomable-tile-map-with-pixi-js/>

This was used as the skeleton to the current version of the tilemap, but was very heavily modified from the original version. The above article does a good job of describing how the system works, so instead of re-stating this I will discuss how I changed this library to suit the needs of our project.



The major change here is how zooming is handled. In the original version you had to select a tile, and then use the menu bar on the right to zoom in and out of the selected tile. I have heavily altered this behavior such that A. you can now use the mouse wheel to zoom, and B. you do not need to select a tile ahead of time, it will dynamically determine the tile that is focused on at runtime and zoom to this tile.

Shown below is the end of main.js where the zoom event is called in the first place, scroll event of up or down is calculated to determine which zoom is called.

```
//On zoom event checks if the mousewheel was up or down and calls the tilemaps zoom in or out respectively
function onWheelZoom(event) {
    event.preventDefault();
    if (event.deltaY < 0) {
        tilemap.zoomIn();
    } else {
        tilemap.zoomOut();
    }
}
```

Further, now that objects within the map are allowed to be moved throughout the map, additional overrides needed to be in place such that the map would not drag while the user was attempting to drag a node. In main.js I added functions that set a flag within the tilemap to prevent it from being dragged while a child node was being dragged, shown below:

```
//these functions allow node dragging without dragging map underneath
function turnParentDragOff() {
    tilemap.setChildDragging(true);
}
function turnParentDragOn() {
    tilemap.setChildDragging(false);
}
```

In response to this, the drag events within the tilemap are now altered to check this flag, and if it is set they ignore the drag events.

The logic of centerOnSelectedTile and constrainTilemap had to be refactored to remove the accounting for the menu bar on the far left of the original version.

Lastly, the size of the tiles (and the tilemap) itself was drastically increased. The original tiles were 16 x 16. To create a map large enough for our purposes with this map, it required ~65k tiles in the map which was a drastic performance hit. After experimentation I finally settled on tiles that were 200px squared. The entire map is 160 of these tiles wide and 80 of these tiles tall.

## Known limitations of the Front End System:

Due to the nature of the tilemap, size becomes a critical limiter. If you wish a tilemap to grow in size you must add child tiles to this. At 65,000 tiles this produces enough overhead to significantly slow down the system. The alternative is to make much bigger tiles, which I attempted. Unfortunately, as the specificity of the zoom is directly tied to tile size, bigger tiles made zoom awkward. As such I settled on a tile size of 200 x 200, which does force a large, but real limit on the size of the map before slowdown begins to occur.

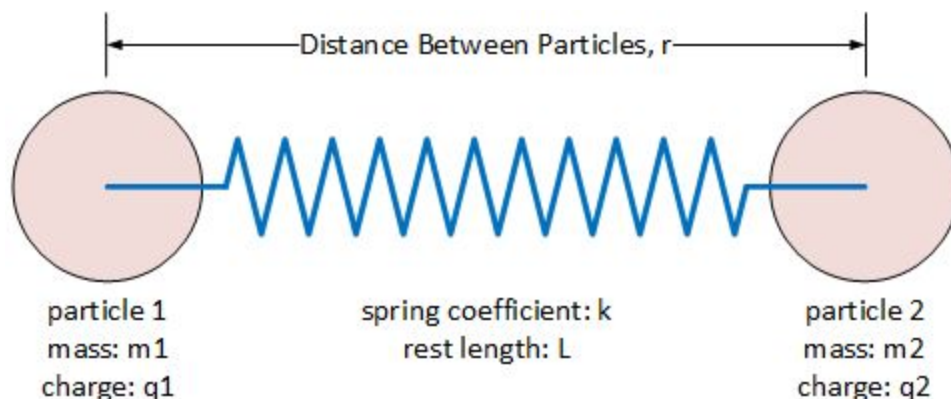
## Physics Engine

This system is responsible for determining the coordinates of all nodes on the graph in such a way that the graph is near-optimally organized. That is, so that it is well spaced and readable. The goal of this system is to continually organize the graph as new data comes in, so that the display responds dynamically to new data and reorganizes as necessary; the system also allows for user-interaction with the simulation, and thereby allows the user to affect the graph display. Jason Loomis took the lead and provided the development of the Physics Engine and its associated objects (particles, springs, simulation and solver, filtering, scaling).

The engine of the graph ordering system is a physics simulation, which applies physical properties to the graph nodes and edges. Nodes are modeled as charged particles (with mass), and edges as springs, connecting the particles. The simulation exists in a 2D space, and positions the particles (nodes of the graph) based on the simulated forces (also 2D) among the particles. Particle charge applies a repulsive force among all the particles, forcing them apart. Springs between particles apply an elastic attractive force, balancing the charge repulsion. And mass provides an inertial response, allowing the particles to resist instantaneous changes to their motion (and stabilizing the simulation). Ultimately, the simulation reaches a (local) equilibrium, a condition wherein all the forces on all the particles are balanced. This configuration is an optimal arrangement of the particles, where the energy of the system is (at least locally) minimized. And as such, generally, is a near-optimal arrangement of the graph being displayed.

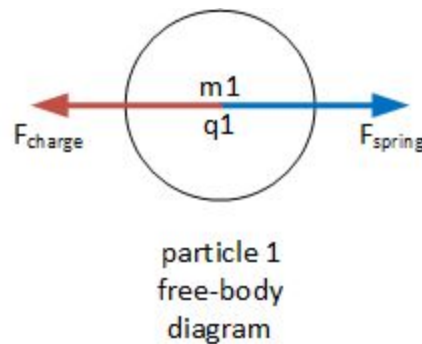
## Physics Simulation

Observe the following physical model and free-body diagrams for a simulation containing two spring-connected particles. For the simulation driving the graph position, the equations shall be generalized to many such particles in two-dimensional space.



For this simulation, motion for the particles are governed by Newton's second law<sup>4</sup>,  $F = m \cdot a$ . For our simulation, we are interested to solve for the position of the particles, which we can derive from their acceleration (from physics, acceleration is the second time-derivative of position). We rearrange Newton's law as  $a = F / m$  to calculate the accelerations of the particles due to the forces acting on them. Note:  $F$ , force and  $a$ , acceleration are generalized as (2D) vector quantities for the simulation.

For particle 1, we have the following free-body diagram. Other particles in the system are similar.



The forces on particle 1 are the sum of the forces due to the repulsion from *each* of the other charged particles, and the forces due to the strain (change in length) of the springs connecting it to other particles.

Hooke's law<sup>5</sup>:

$$F_{spring} = k(r-L)$$

where  $k$  is the spring constant and  $r$  and  $L$  are the stretched and resting lengths of the spring, respectively.

Coulomb's law<sup>6</sup>:

$$F_{charge} = q1 \cdot q2 / r^2$$

where  $q1$  and  $q2$  are the charges of particles 1 and 2, respectively, and  $r$  is the distance between the particles; note, we omit the Coulomb proportionality constant,  $k_e$ , for simplicity.

<sup>4</sup> See: [https://en.wikipedia.org/wiki/Newton%27s\\_laws\\_of\\_motion#Newton.27s\\_2nd\\_Law](https://en.wikipedia.org/wiki/Newton%27s_laws_of_motion#Newton.27s_2nd_Law)

<sup>5</sup> See: [https://en.wikipedia.org/wiki/Hooke%27s\\_law](https://en.wikipedia.org/wiki/Hooke%27s_law)

<sup>6</sup> See: [https://en.wikipedia.org/wiki/Coulomb%27s\\_law](https://en.wikipedia.org/wiki/Coulomb%27s_law)

Damping,  $b$ , may be included to modify the acceleration. Damping of the particle motion is due to a frictional force, proportional to and directly opposing the particle's velocity,  $v1$ .

$$F_{damping} = -b / m1 * v1$$

The resulting acceleration of the particle is the sum of the accelerations due to the forces acting on the particle:

$$a_{particle} = m1 * F_{charge} + m1 * F_{spring} - m1 * F_{damping} = m1 * (F_{charge} + F_{spring} + F_{damping})$$

## Changes from the Project Plan

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.

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 (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 (IIR) lowpass filter<sup>7</sup> was applied to the node position. This filter (exponentially) smooths the motion of the nodes by applying an attenuation factor (alpha) to cause the (filtered) output to be the sum of the attenuated input and the previous output value. The filter alpha is calculated dynamically, 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.

## Implementation

The physics engine is wholly contained in the `simulation.js` JavaScript code file. The implementation of the physics engine includes 3 objects: *Particle*, *Spring*, and *Simulation*.

### *Particle*

Particle provides a representation of a simulated particle with simulated physical properties of mass, charge, position, velocity, acceleration. Each Particle object maintains a list of the Particles it is attached to by Springs (its children). The Particle object also maintains flags indicating whether its position should be controlled by the simulation or by user-interaction.

With each timestep of the simulation, forces are applied to each Particle object, based on its interaction with other particles. At the end of each simulation iteration, once all forces have been applied, the (change in) position of the particle is calculated. This calculation is performed by the `Particle.updateAcceleration()` method:

```
// update the acceleration on the specified particle after
// forces have been applied for the specified time interval, dt
this.updateAcceleration = function (dt) {
  if (this.mass > 0) {
```

---

<sup>7</sup> See the Wikipedia article for more information:

[https://en.wikipedia.org/wiki/Low-pass\\_filter#Simple\\_infinite\\_impulse\\_response\\_filter](https://en.wikipedia.org/wiki/Low-pass_filter#Simple_infinite_impulse_response_filter)



```

    // update acceleration
    this.ax += this.fx / this.mass;
    this.ay += this.fy / this.mass;

    // clamp acceleration
    var a2 = this.ax * this.ax + this.ay * this.ay;
    var a = Math.sqrt(a2);

    if (a > 100) {
        this.ax = this.ax / a * 100;
        this.ay = this.ay / a * 100;
    }

    // update velocity
    this.vx += this.ax * dt;
    this.vy += this.ay * dt;

    // update displacement
    this.x += this.vx * dt;
    this.y += this.vy * dt;

    this.xf.alpha = 1 / Math.pow(this.children.length + 1, 1);
    this.yf.alpha = 1 / Math.pow(this.children.length + 1, 1);

    this.x = this.xf.update(this.x + this.vx * dt);
    this.y = this.yf.update(this.y + this.vy * dt);
}
};

```

The basic steps performed by this are to calculate the change in acceleration of the particle (based on the force applied and the mass of the particle, i.e. Newton's second law,  $F = m \cdot a$ ). Then applies this acceleration for the specified timestep,  $dt$ , to change the velocity of the particle. The updated velocity is then applied for the specified timestep to change the position of the particle.

These basic steps are modified to improve the stability and behavior of the simulation. First, the acceleration is clamped to a maximum magnitude of 100. This prevents the simulation from “running away”--applying excessive or infinite changes in position--when forces are very large or masses are very small. Second, a lowpass filter is applied to the particle position ( $xf$  and  $xy$  are Lowpass objects, see below). These filters' alpha values are updated to have a dependence on the number of children a Particle has--Particles with more children will have stronger filtering applied to avoid the “vibration” instability, discussed in the section above. The alpha value for the filters is calculated as  $1 / [\text{number of children}]$ . This value was found empirically to provide a reasonable filter behavior for this system.

### Spring

Spring provides a representation of a simulated spring with simulated physical properties of elasticity (spring constant), rest length, and damping (ratio). Most importantly, the Spring object also stores the index of the child node that it attaches to its parent node.

In this simulation, damping is used to bleed the energy stored in the particle system and allow the system to stabilize in a stationary configuration. If damping was not applied, the particle interactions would be perfectly elastic and they would vibrate indefinitely (well, until floating-point rounding errors either damped the system or caused it to destabilize by increasing the energy). Not a terribly user-friendly display, if the nodes were continually bouncing around.

### *Simulation*

Simulation encapsulates the calculation of the physical simulation.

In the initial implementation of this graph-ordering system, the “simulation” consisted of two halves: an object encapsulating the system of differential equations modeling the particle system, and the differential equation solver. The differential equation object contained the particle system and provided the methods for defining the particle system and the spring interconnections, and was passed to the solver to be updated for each time step of the solution. The solver implemented the Dormand-Prince<sup>8</sup> ODE solution method (related to Runge-Kutta) with adaptive stepsize control (that is, the overall timestep would be broken into smaller sub-steps, and the size of the sub-steps would be chosen as to stay within a specified error tolerance as the solution was progressed). The implementation was based on the C++ code provided in *Numerical Recipes*<sup>9</sup>. This code provides an excellent general-purpose differential equation solver, but the algorithm requires many evaluations of the differential equations per timestep. And unfortunately, evaluation of the differential equations is a calculation with  $O(V^2 + E)$  complexity—that is, quadratic in the number of nodes, since the forces on each particle include the force due to the charge of every other particle ( $V^2$ ), plus the force applied by every spring ( $E$ ). Clearly, the number of calculations goes up very quickly as more nodes are added to the system, and the adaptive step size control means that these calculations were iterated many times.

When testing revealed that the differential equation solution was not fast enough, the alternative was to accept a less-accurate solution. In this case, since we needed the solution only for the purpose of displaying a graphical interface to the user (and not, for example, for calculating orbital trajectories), the less-accurate solution was acceptable.

The less-accurate solution, the Simulation object, grew from the differential equation object that had been initially developed. The heart of the Simulation is the `step()` method:

```
// apply the specified timestep to the force model
// this function is the heart of the simulation
this.step = function (dt) {
```

<sup>8</sup> See: [https://en.wikipedia.org/wiki/Dormand%E2%80%93Prince\\_method](https://en.wikipedia.org/wiki/Dormand%E2%80%93Prince_method)

<sup>9</sup> Press, William H.; Teukolsky, Saul A.; Vetterling, William T.; Flannery, Brian P. (2007). “Integration of Ordinary Differential Equations”. *Numerical Recipes: The Art of Scientific Computing*. Third Edition. New York: Cambridge University Press. p. 899

```

var particle;
this.t += dt;
// reset acceleration
for (var i = 0; i < this.particles.length; i++) {

    particle = this.particles[i];
    // reset force
    particle.fx = 0;
    particle.fy = 0;

    // reset acceleration
    particle.ax = 0;
    particle.ay = 0;

    // particle fixed in place, e.g. under user control
    if (particle.fixity || particle.hold) {
        // no velocity
        particle.vx = 0;
        particle.vy = 0;
    }
}

// evaluate physics for each particle and spring
for (var i = 0; i < this.particles.length; i++) {
    evaluateCharge(this.particles, i);
    evaluateSprings(this.particles, i);
}

// apply accelerations
for (var i = 0; i < this.particles.length; i++) {
    if (!this.particles[i].hold && !this.particles[i].fixity) {
        this.particles[i].updateAcceleration(dt);
    }
    else {
        this.particles[i].xf.y = this.particles[i].x;
        this.particles[i].yf.y = this.particles[i].y;
    }
}

// constrain children to be within bounds, with filter
for (var i = 0; i < this.particles.length; i++) {
    particle = this.particles[i];

    if (particle.x < this.xmin) {
        particle.x = this.xmin;
        particle.xf.y = particle.x;
    }
    else if (particle.x > this.xmax) {
        particle.x = this.xmax;
        particle.xf.y = particle.x;
    }

    if (particle.y < this.ymin) {
        particle.y = this.ymin;
        particle.yf.y = particle.y;
    }
    else if (particle.y > this.ymax) {
        particle.y = this.ymax;
        particle.yf.y = particle.y;
    }
}

```

```

    }
  };
};

```

This method evaluates the physics (charge and spring) and applies the forces to each Particle. Once all the forces have been applied, it calls the `Particle.updateAcceleration()` method for each particle to update its acceleration, velocity, and position in the simulation.

In order to improve the stability of the solution, it further constrains the particles to stay within user-specified bounds. This prevents particles with excessive acceleration from “running away”.

Within the `step()` method, the methods that apply the forces to each particle are the `evaluateCharge()` and `evaluateSprings()` methods.

`evaluateCharge()` calculates the force contribution from each particle onto a specific particle. The distance between the specified particle and each other particle in the simulation is calculated and the charge force determined. This force is inversely proportional to the square of the distance between particles ( $F = q_1 * q_2 / r^2$ , Coulomb's law, see above).

If the distance between particles is zero (something that can't really happen in the universe as we know it, but is perfectly possible in this simulation), the resulting force should be infinite. Instead of an infinitely large force, a force based on some minimum radius, `rmin`, is determined. This force is applied at a randomly chosen angle.

```

// evaluate the charge physics among all particles
var evaluateCharge = function (particles, index) {
  var dx, dy;
  var r2, r;
  var cos, sin;
  var theta;
  var f;
  var particle, other;

  particle = particles[index];
  if (particle.hold || particle.fixity) return;

  //accelerations due to charges from (all) other particles
  for (var i = 0; i < particles.length; i++) {
    if (index == i) continue; //don't evaluate a particle against itself
    other = particles[i];
    dx = particle.x - other.x; //positive when index to the right of i
    dy = particle.y - other.y; //positive when index above i

    r2 = dx * dx + dy * dy; //distance^2 between particles

    r = Math.sqrt(r2); //distance between particles
    if (r > 0.0) {
      cos = dx / r;
      sin = dy / r;
    }
  }
}

```

```

    else {
        //avoid angular singularity; use a random angle
        theta = 2.0 * Math.PI * Math.random();
        cos = Math.cos(theta);
        sin = Math.sin(theta);
    }
    if (r < rmin) r2 = rmin * rmin; //limit the minimum distance between particles
    (hence the maximum force)

    f = particles[index].charge * particles[i].charge / r2;
    if (like_attract) f = -f;
    particles[index].applyForce(f * cos, f * sin);
}
};

```

evaluateSprings () calculates the force applied by each spring attached to a specific particle. The force applied by each spring is proportional to the strain of the spring ( $F = -kx$ , Hooke's law, see above). It is worth noting that the force applied by each spring is simultaneously applied to *both* particles at each end of the spring, with equal and opposite vectors. It is also worth noting that the spring force is typically attractive (countering the charge force), but can also be repulsive, if the spring is compressed.

Similarly to the charge force calculation, if the distance between particles is zero, the spring force is applied at a randomly chosen angle.

```

// evaluate the spring physics
var evaluateSprings = function (particles, index) {
    var dx, dy;
    var r2, r;
    var cos, sin;
    var theta;
    var f;
    var s;
    var c;
    var parent;
    var spring;
    var child;

    parent = particles[index];

    for (var i = 0; i < parent.children.length; i++) {
        spring = parent.children[i];
        if (index == spring.childIndex) continue; //ignore particles connected to themselves
        child = particles[spring.childIndex];

        // distance between particles
        dx = parent.x - child.x;
        dy = parent.y - child.y;
        r2 = dx * dx + dy * dy;
        r = Math.sqrt(r2); //distance between particles
        if (r > 0.0) {
            // calculate component trig ratio components
            cos = dx / r;
            sin = dy / r;

```

```

    }
    else {
        //calculate the force components using a random angle
        theta = 2.0 * Math.PI * Math.random();
        cos = Math.cos(theta);
        sin = Math.sin(theta);
    }

    // spring strain (stretched length)
    s = r - spring.restLength;

    //parent is free...
    if (!parent.hold && !parent.fixity) {
        f = -spring.springConstant * s;
        parent.applyForce(f * cos, f * sin);

        // calculate damping from damping ratio
        // zeta = b / (2 * sqrt(m*k)) --> b = 2 * sqrt(m*k) / zeta
        // calculate critical damping...
        c = 2.0 * Math.sqrt(parent.mass * spring.springConstant);
        c *= spring.dampingRatio; // ...then adjust for damping ratio
        parent.applyForce(-c * parent.vx, -c * parent.vy);
    }

    //child is free...
    if (!child.hold && !child.fixity) {
        //force has opposite sign for p2
        f = spring.springConstant * s;
        child.applyForce(f * cos, f * sin);

        // calculate damping from damping ratio
        // zeta = b / (2 * sqrt(m*k)) --> b = 2 * sqrt(m*k) / zeta
        // critical damping
        c = 2.0 * Math.sqrt(child.mass * spring.springConstant);
        c *= spring.dampingRatio; // ...then adjust for damping ratio
        child.applyForce(-c * child.vx, -c * child.vy);
    }
}
}
}

```

In both calculations, the “hold” and “fixity” parameters are used to control whether forces should be applied to each particle during the timestep. If the user is interacting with a particle (e.g. dragging it to a new location), the physics simulation must not change the position of the particle (the user is changing its position!), otherwise it would “slip away” from the user’s cursor due to the forces applied by the simulation. Additionally, the “fixity” parameter is used to anchor the simulation at the root node—that is, the root node is not ever moved by the physics simulation—this prevents the graph system from “drifting away”.

## Filtering

The simplified simulation code was found to have an 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 (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. To combat this imbalance, a simple infinite-impulse-response (IIR) lowpass filter<sup>10</sup> 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. The filter alpha is calculated dynamically, 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.

The code for a simple IIR lowpass filter is wonderfully simple and useful in this and many other scenarios (e.g. noisy user input, e.g. from a touch screen). This code may be found in the `lowpass.js` file:

```
var Lowpass = function (alpha) {
  this.firstrun = true;
  this.y;
  this.alpha = alpha;

  // Calculate alpha filter parameter from sample interval and cutoff frequency.
  // dt: Sample interval (in seconds)
  // fc: Cutoff frequency (in Hz)
  // returns: normalized frequency cutoff alpha parameter
  this.calculateAlpha = function (dt, fc) {
    return dt / ((1.0 / fc) + dt);
  }

  // apply filter to an input sample.
  // x: an input value
  // returns: the current filtered output value
  this.update = function(x)
  {
    if (this.firstrun) {
      this.y = x;
      this.firstrun = false;
    }
    else this.y = this.y + this.alpha * (x - this.y);
    return this.y;
  }
}
```

This Lowpass object retains its current (filtered) output value, `y`. With each new input/sample applied (via the `Lowpass.update()` method), the output value is recalculated. The output value is the sum of  $(1 - \alpha)$  percent of the previous output value and  $(\alpha)$  percent of the input value. Each subsequent input then affects the output similarly. The alpha value is in the range from 0 to 1 (inclusive)--values of alpha closer to 1 cause the output to react more quickly to changes in the input (higher cutoff frequency); values of alpha closer to 0 cause the output to react more slowly to changes in the input (lower cutoff frequency).

---

<sup>10</sup> See the Wikipedia article for more information:  
[https://en.wikipedia.org/wiki/Low-pass\\_filter#Simple\\_infinite\\_impulse\\_response\\_filter](https://en.wikipedia.org/wiki/Low-pass_filter#Simple_infinite_impulse_response_filter)

This Lowpass object is applied to the position of each particle, and the alpha value is adjusted to be inversely proportional to the number of children a node has--a node with no children has an alpha value of 1 (unfiltered), while a node with many children has an alpha value much closer to 0 (low cutoff frequency)--and hence is not allowed to change position quickly. This strategy proved very effective in reducing the “vibration” instability described above.

## Scaling

The physical parameters applied to the simulation--charge, spring constant, rest length, damping, mass, distance among particles--are strongly interrelated, and all affect the stability and behavior of the particle system. As such, changing a parameter like the size of the simulation (which affects the distance between particles, which affects the charge forces, which affects...), e.g. to adjust for a different screen size, requires careful modification of each of the parameters of the system in order to retain comparable behavior of the simulation.

Instead of trying to determine parameters for the simulation that would allow it to fit a variety of screen sizes, a scaling system was implemented to simply allow a single set of parameters for the the physical simulation to be used, and the physical simulation world *coordinates* to be scaled to fit any desired screen size.

This scaling is encapsulated in the LinearScale object, found in the linearScale.js file. The LinearScale object maintains bi-directional linear transforms between “input” and “output” coordinate systems. These linear transforms are of the form  $y = m * x + b$ , where y and x are input and output values, transformed via the slope m and offset b.

The `LinearScale.calculate()` method (re)calculates the input scale and output scale slope and offset values based on two points on each of the input and output scales. These values are then used to transform coordinates between the systems.

```
// recalculate the scaling transformation values
// note: this function is automatically called by all of the functions
// that assign input or output values
this.calculate = function () {
    slope_tooutput = (outputfar - outputnear) / (inputfar - inputnear);
    offset_tooutput = outputnear - slope_tooutput * inputnear;
    slope_toinput = (inputfar - inputnear) / (outputfar - outputnear);
    offset_toinput = inputnear - slope_toinput * outputnear;
}
```

Using the LinearScale object, the choice of input and output scale is arbitrary. In the case of scaling the physical simulation for the screen, the physical world scale was chosen to be the input scale, and the screen pixel scale was chosen as the output scale.

The LinearScale object provides several methods for transforming coordinates. To draw the positions of the nodes on the screen, the `LinearScale.toOutputScaleOffset()` method was used.

```
// transform an absolute value in the input scale to the output scale
this.toOutputScaleOffset = function (input) { return input * slope_tooutput +
offset_tooutput; }
```

This method simply scales an (absolute) input value--physical coordinates from the simulation--to an (absolute) output value--pixels on the screen.

Similarly, to transform user mouse movements into movements of the nodes, the `LinearScale.toInputScaleOffset()` method was used to transform the scale in the opposite direction.

```
// transform an absolute value in the output scale to the input scale
this.toInputScaleOffset = function (output) { return output * slope_toinput +
offset_toinput; }
```

This method simply scales an (absolute) output value--mouse coordinate pixels--to an (absolute) input value--physical coordinates for a node in the simulation.

## Simulation Interface

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 can be found in the JavaScript file `simulationInterface.js`. This interface facilitated a reasonably simple 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  $+\text{physicalHeight}/2$  at the top of the display area to  $-\text{physicalHeight}/2$  at the bottom of the display area. The “physical width” is calculated from the aspect ratio of the display area and scaled per the `physicalHeight` 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.

```
stepSimulation(dt)
```

This method advances the simulation by the specified timestamp, `dt` (in seconds).

This method was introduced to solve a “hang” that was found to occur when the simulation was being automatically stepped by the `setTimeout()` function (`runSimulation()` and `stopSimulation()`, below).

JavaScript provides a single thread of execution for client-side code (with the notable exception of WebWorkers). In order to provide a responsive UI in a single-threaded context, the UI needs to be able to frequently handle user events (e.g. mouse input) and, importantly, function calls cannot run continuously or block that thread, or else user events will not be processed and the UI will hang. In this case, I had assumed that the `setTimeout` function, which executes a function call after a user-specified period of time, would be an ideal solution to step the physics simulation. `setTimeout` would step the simulation and then schedule another step for a handful of milliseconds later (nominally 1/60th of a second, to match the screen frame rate).

This method worked well in preliminary testing; however, `setTimeout` does not actually spawn a new thread to execute a function call, rather, the function call gets queued up (after approximately the specified amount of time) and processed with all the other function calls that are being handled (e.g. the event handlers for mouse movement or scrolling). In this case, the hang was being caused by the queue being flooded by mouse events (mouse wheel scrolls or mouse movements) that--once there were a significant number of nodes being displayed--required a non-trivial amount of time to process. The enqueued `setTimeout` function call (i.e. the physics simulation step) would end up being delayed (or in the worst case, dropped entirely), waiting for the mouse events to be processed, thereby hanging (or stopping entirely) the simulation.

Fortunately, the solution was a simple one: forgo the use of `setTimeout` and instead introduce the `stepSimulation` method and call it within the `requestAnimationFrame` method (which is called at the browser's refresh rate, to update the graphics display) to update the simulation just prior to rendering the graphics. `requestAnimationFrame` appears to be prioritized differently, and doesn't get starved out by mouse events, so the simulation continues to be updated at the browser's frame rate and no longer hangs when the system is processing a flood of mouse events (zooming and dragging).

`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.

# Conclusion

We consider this project to be extremely successful. We first met our primary goals--that is, to satisfy the which were effectively the goals set forth to us within the assignment bounds. Further, we also met our secondary goals such as increasing performance across all three realms of the system. We took this further still to even meet our tertiary goals of including additional features such as favicons and Parsley form validation.

The team as a whole worked extremely well. We mostly communicated via Google Hangouts, and in nearly all occasions team members responded to each other within the hour when they were asked a question. This resulted in virtually zero downtime where any of our work was blocked by the other. The collaboration efforts were very impressive as well. Each of us came into this project with differing ideas of how the project should work in our minds, and we were able to quickly come up with a solid and cohesive plan which both used each other's ideas and allowed the systems to easily interact.

Due to these and other factors, this project transcended being a "chore" and instead became a labor of love for our group, who voraciously set out to constantly improve it to be the best that it could be. We are all very proud of what we have done here, were very moved by your compliments regarding our midpoint review, and hope you have enjoyed this final project.

Regards,  
Gamma Team