Cetus Project Final Report

Team: Christiano Vannelli, Christopher Kirchner, and Himal Patel

Introduction

The Cetus Web Project graphically visualizes depth-first-search (DFS) and breadth-first-search (BFS) behavior on an origin web page by traversing through a graph of connected web page links. Two major graph types are used to visualize the searches - a dynamic revolving collapsible force layout and a pack hierarchical layout. In the force layout, or simulation, web pages are treated as nodes connected to each other through edge links. In the pack hierarchical layout, web pages are also treated as nodes, but the edges are perpendicular to the user's perspective, directed inward along the viewport's z axis. Three different web scrapers were implemented to bring varying degrees of DFS and BFS search behavior. The user is able to input search fields via a compact navigation bar at the top of the page, with deletable user search history saved in a dropdown to allow the quick re-entry of search information via cookies.

After the journey that is this web project, one team member gained experience in web scraping; another refreshed in python scraping, delved into Scrapy scraping, learned new data transfer techniques and a data visualization library; while the third concentrated on data visualization, frontend presentation, and data transfer as well.

User Perspective

The user begins by inputting a starting url along with an optional search keyword, and decides on the type of search algorithm, how many layers deep the algorithm traverses from the starting website, the type of graphical representation of the search, as well as the type of crawler. After the user hits "Crawl", a visual representation will be generated based upon the path(s) taken by the web scraper as web pages are discovered. The scraper will finish when the keyword is found, or all paths and levels traversed. After the visualization finishes, if the user refreshes the page, a history of their inputs will be stored in the form of a dropdown. This dropdown will populate with the user's search history after every crawl. The user can use this dropdown to quickly redo previous searches. Subsequently, any number of additional searches can be run with new user-specified inputs.

User Options

1) URL - starting url
2) Search Term: search keyword that halts crawler when found and highlights the webpage it is found on (red in force layout, grey in pack layout)
3) Levels: search depth or layers of web pages the scraper traverses before halting (in single-path DFS) or reversing back pages to find unsearched paths (DFS and BFS)
4) Search Type
    a) Depth-first: searches web pages by traversing to the search depth level and reversing back to unsearched paths from the parents until all nodes within the depth level have been searched (uses a stack data structure in traditional DFS algorithm). This search type goes beyond client requirements, but looks so cool, the client should reconsider.
    b) Breadth-first: searches web pages by traversing all children, followed by all children's children, and so forth, until all nodes have been searched within the depth level (uses a queue data structure in traditional BFS algorithm and meets client requirements)
    c) Single-Path DFS: similar to Depth-First search, but stops when the depth level is first encountered, if possible (meets client requirements)
5) Graph Type
    a) Force Layout: represents a search algorithm by treating web pages as nodes connected to each other through links
    b) Pack Layout: translates search algorithm by packing nodes of web pages into parent web pages
6) Scraper Type
    a) Html (Lxml) - headless multithreaded lxml scraper
    b) Js (Casper) - single threaded javascript loading scraper
    c) Html (Scrapy) - uses production level python scraper library known as Scrapy (https://scrapy.org/)

On entering options, the user is presented with a dynamic construction of nodes connected to parents as the scraper builds these relationships while looking for the optional keyword.

Graphs

Each graph treats web pages as nodes with child nodes connected to parents through edges. Each graph is built dynamically, per node, after the keyword is searched in it's corresponding scraped web page. Hovering over each node reveals the title of the web page (if available), the URL of the web page, and additional information depending on the graph type. A node's web page can be opened in another tab by double clicking with the left mouse button (pack hierarchical layout) or right clicking (force layout). The node containing the given keyword

search term is highlighted when found (red in force layout, grey in pack layout), halting the search effort.

Force Layout Graph

In the dynamic revolving collapsible force layout graph, web pages appear as nodes while links are represented as edges. Various forces like gravitational centering, repulsive walls, ionic charges, atmospheric friction, and springs are combined to give a readable graph representation of a search algorithm. Force settings are changed depending on whether the graph is growing to allow faster simulation, or weaned, to allow better graph manipulation. The user is able to interact with the graph in order to control visibility and reveal partially hidden nodes. The graph is built dynamically node-by-node as the scraper searches each web page. To maintain performance, a maximum node occupancy is mandated whereby child nodes begin to collapse onto their parent nodes in order to maintain a maximum number of nodes visible at one time. Collapsed nodes, or “supernodes”, are hollow, with a size that is intended to be proportional to the number of nodes that are hidden in that parent node. Collapsed nodes can be expanded by clicking on the parent node. Recently collapsed nodes are highlighted along with recently expanded nodes in order to clarify interaction due to fast shifts in the graph layout. Expanding the parent node reveals children that have been hidden, allowing nodes to revolve so that, regardless of the size of the graph and the maximum visible node constraint, hidden nodes can be revealed.

Settings can be provided in the future that allow the user, or browser, to control the maximum number of nodes based on the browser, or computers, hardware capabilities. This layout is aptly named “dynamic” since the search is portrayed real-time, “revolving” since node visibility prefers the longest hidden node, “collapsible” as child nodes can be grouped by their parents, and “force layout” since the display of nodes and edges are controlled by forces inspired by classical physics.

The graphics were mainly inspired by the v4 force layout implementation from static data in a fiddle (https://jsfiddle.net/t4vzg650/6/) with a lot of customization to make the data dynamic, and the force layout revolving. The traditional approach of using d3.hierarchy object to seed the force simulation with nodes and links led to poor and glitchy graphics. A new dynamic tree ADT was implemented instead. The data is also buffered before being uploaded to graphics since the multithreaded python scraper tends to work much faster than the d3 graphics can handle.

Pack Hierarchical Layout Graph

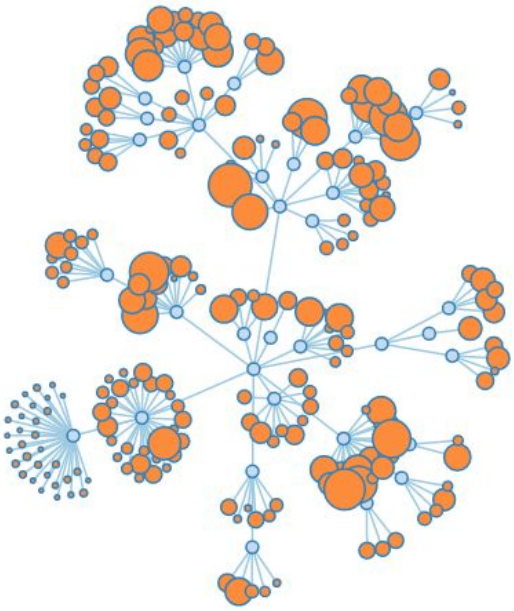
In the Pack Hierarchical Layout, first a large circle (graph node) is spawned which represents the user input web page. As the scraper discovers web pages, additional circles are generated inside this original circle which represent a new level in the hierarchy of nodes. Depending on the number of levels pre-set by the user, each circle will contain circles of their own (children of

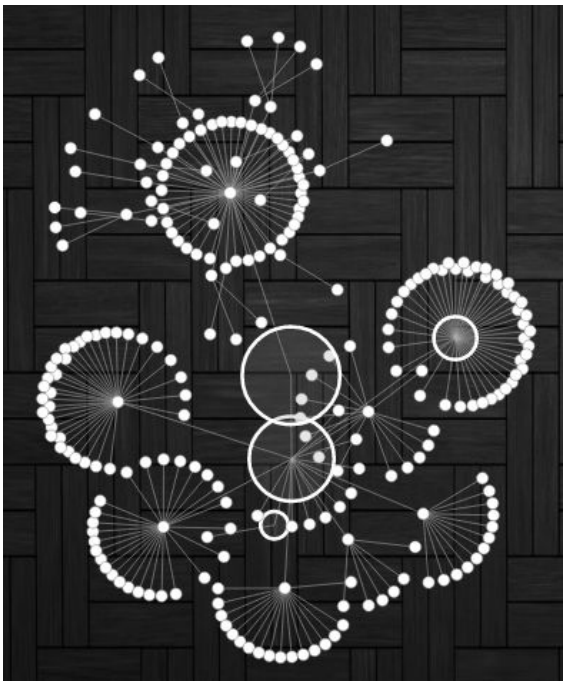
the parent node); continuing as many layers deep as needed. When clicked, the application will zoom into the clicked circle, giving the user a better view of the layers which may be contained inside. Tooltips are generated on hover which display the url's and titles associated with each node. On a double-click, the web page represented by the circle will open. If the user specifies a keyword and it is discovered during circle packing, the graph will halt and the circle containing the phrase will change color.

Graphical Representations

Collapsible Force Layout:

This collapsible force layout in D3 allows a natural representation of search algorithm by spanning multiple nodes per parent node in a repeated fashion. The graph is fun to play with after analysis. The first graph is the planned graph (http://bl.ocks.org/mbostock/1062288). The second graph is what was implemented.

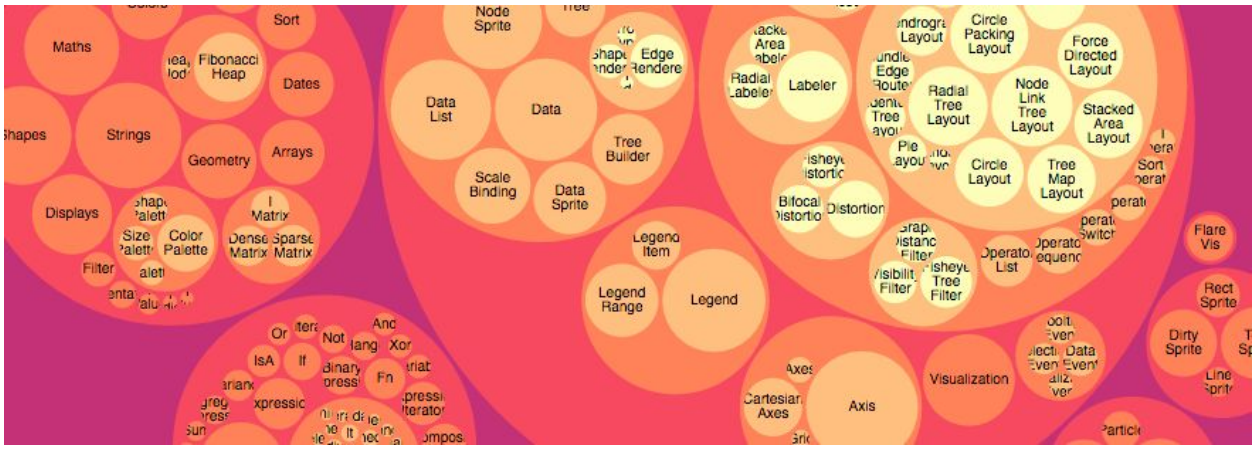


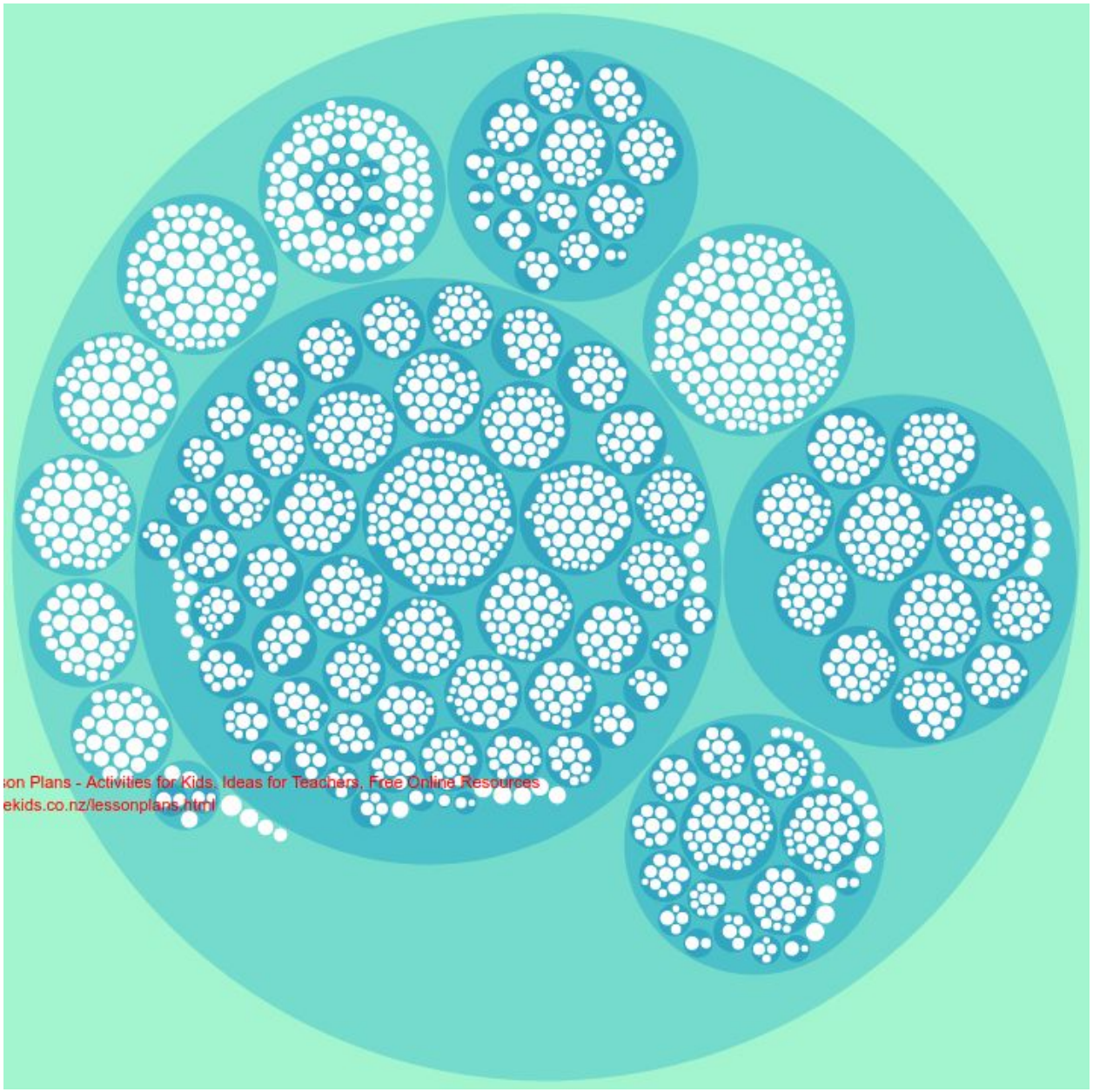


Pack Hierarchy Layout ():

A DFS graphical search may be fairly boring if the nodes grow in one direction, or chaotic if random positions are chosen. A fractal like pack hierarchical layout would allow a different

perspective of search algorithms where parent nodes essentially swallow their children, or web pages they link to. If a starting web page leads to another page without active links, it will show up as a small bubble next to a bigger sibling bubble that gives many deep links, both inside their parent bubble of course. The pack graph below would show many web pages with shallow links, or search level limit. Zooming would also allow mesmerizing fractal like trances as smaller nodes emerge into sight. The first design was what was envisioned from a d3 layout (https://github.com/d3/d3-hierarchy), the second version was what was implemented from a similar d3 layout (https://bl.ocks.org/mbostock/7607535)





User Instructions

NOTE: The application fits all of required inputs into the navigation bar at the top of the web page. Additional scraper type options can be found on the top right of the navbar. **The BFS and Single-Path search types, in conjunction with the "html" Python scraper (default), meet all user requirements. The casperjs ("js") option has been removed from options (see Discussion of Casper Scraper).**

User Input:

1. Go to "http://138.68.29.97:8080" or "http://138.197.208.219:8200/"

2. Input the starting URL. (e.g, http://www.sciencekids.co.nz/sciencefacts/animals/cat.html)
3. <OPTIONAL> Input the desired keyword (the scraper will halt on discovery).
4. Input the number of levels to traverse (must be greater than 0).
5. Select a search type (BFS, DFS, or Single-Path DFS)
6. <OPTIONAL> Specify the desired scraper (HTML, JS, or Scrapy)
    a. We suggest testing on HTML or Scrapy first since JS has had some memory issues
7. Click “Crawl” (Graph will generate).

Post-Scraping (Force Layout)

1. Hover on node for tooltip (displays title and url).
2. Single-click on parent nodes that are collapsed (helps condense large graphs)
3. Right click on nodes to open associated web pages.
4. Use scroll-wheel to zoom in and out of visualization.
5. Click on an empty space outside of nodes or edges and hold to drag, or pan the graph side to side

Post-Scraping (Pack Hierarchy Layout)

1. Hover on node for tooltip (displays title and url).
2. Single-click on parent node for zoom to next level.
3. Single-click to side of parent node for zoom back to previous level.
4. Double-click on nodes to open associated web pages.

Search History

1. Refresh page (this generates the user’s search history)
2. Select option from dropdown (this automatically populates user inputs with past info)
3. Click “Clear History” (this deletes the search history corresponding to the user’s cookie)

Setup on Ubuntu 16.04 LTS

`./setup`

`node index.js`

Note: a second npm install` command may be needed again in case it crashes.

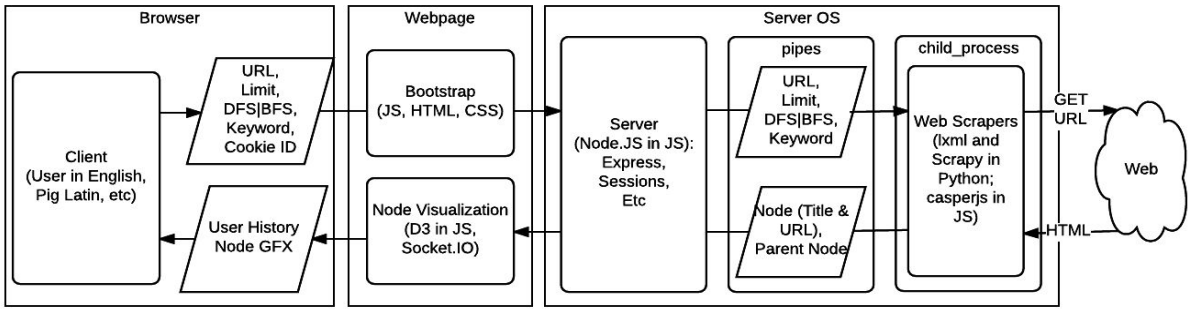
Software Structure

The software structure comprises of three architectural components as shown in the flow diagram below - client browser, web page, and server. The browser passes user input as the URL of the starting web page, the depth or breadth limit, the depth-first or breadth-first search algorithm type, the scraper type, and the halting keyword. The Bootstrap styled front-end and

user interface of the web page collects this information, passing it on to the hosting Node.JS server via socket.io. User input is added to sessions with the user's cookie's key as an identifier for their memory store on the server via express-sessions. Ajax is used to reset user history so that the browser does not have to rely on a persistent connection with socket.io after scraping, when the connection has been terminated. User history is loaded on the initial GET request sent to the web page via the cookie's key as a unique identifier. The identifier pulls the client's data and responds with their history.

The scrapers, written in javascript or python, are started through child_processes in Node.js. Python-shell, a convenience wrapper for child_processes, is utilized for python based scrapers. The casperjs scraper is controlled directly by child_processes, using spawning to create the child, and JSONStream to capture the child's piped JSON output through stdout before routing it to the front-end.

The three web scrapers work in the following general manner. The web scraper first sends a GET request to the initial URL. The server of the URL responds with the HTML of the target webpage. The HTML is parsed by the web scraper into a list of additional URLs that will be targeted based on the DFS or BFS algorithm until the limit is reached, or the user keyword is found in the contents of the target web page. As target URLs make successful connections with the web scrapper, they are sent back to the Node.JS server with their title and parent node. Each node is used to form a graphical representation of nodes under the visualization library of D3 at the client-side using a persistent connection with socket.io to upload data. Once the graph scraper is terminated, the persistent connection is severed as well.



Discussion of Python Scrapers

Two python scrapers were eventually developed, with one being more of a production level scraper called Scrapy, and the other being more of a "i'm in computer science" scraper by crawling with lower-level libraries. The lower level scraper utilizes lxml c-bindings to parse html quickly, relying on Beautiful Soup's UnicodeDammit to convert html into unicode, where lxml would otherwise hiccup. Multithreading is also utilized to create concurrent connections, grabbing and parsing many links at once for both BFS and DFS algorithms. Only the Single-Path DFS algorithm uses a single-thread to create a single DFS path per the client's

requirement. Unfortunately, the python threading module appears to contain memory leaks that become predominant after a few thousand web pages are scraped in a matter of minutes. Significant effort was administered to find the source of the memory leaks, including memory_profiler to find the point the in the program that memory was accumulating, as well as Pympler and objgraph to track memory usage for different data structures over time. Additionally, a persistent queue (queuelib) was used to determine if memory leaks were caused by the scrapers link queues. Lxml, having c-bindings, was sub-processed to track it's memory usage without luck while the curl subprocesses were used to exclude the requests package as the culprit. After nothing worked, and realizing that the issue may be embedded in the Python library itself after reading another scraper's admittance of the issue (https://doc.scrapy.org/en/latest/topics/leaks.html), that scraper, called Scrapy, was implemented.

Discussion of Casper Scraper

The casperjs scraper utilizes a single thread of phantomjs. It is available with its own documentation and functions. The team decided to write multiple scrapers so that they can be compared with each other and utilize one as an additional backup in case we ran into errors later into development.
This scraper has some limitations due to it being single threaded, it also consumed roughly 20% of the RAM on the server. This issue had gone unnoticed until further testing was performed on the server. Although, the casperjs scraper was slower than the python scrapers, most of the functionality required was working properly. The main issue with the casperjs scraper was that it could not load every single page using BFS to obtain all the titles because it would take a substantial amount of time due to single thread limitations. However, there was a nifty work-around implemented to get titles using casperjs, but that would not yield 100% results, some titles would be empty.
Another issue that we ran into with casperjs is that anytime a process is terminated or interrupted, the casperjs scraper would not run again. So we were unable to have the server run indefinitely using a function named casper.exit().

Used Technologies

Languages

- Javascript/Node.js
- HTML & CSS
- Python

Software Libraries

- Node.js

    - Express (web framework)
    - Express-handlebars (html templating)
    - Express-session
    - Body-parser (parses bodies, e.g. POST requests)
    - Http (server)
    - JSONStream
    - Socket.IO (used to create persistent connection for uploading links to client real-time, http://socket.io/)
    - Socket.io-client (used to receive links real-time on client)
    - CaspjerJS (browser-based js-loading web scraper, https://casperjs.org/)
    - Python-shell (used to integrate child-processes into nodejs)
- Bootstrap (https://getbootstrap.com)
    - Bootstrap-Select
- jQuery
- Python
    - Lxml (fast c-based html and xml parsing library)
    - Bs4 (used for their Unicode conversion library from bad html)
    - Requests (http connection library)
    - Scrapy (awesome production level scraping module)
    - Cssselect (selecting elements by css)
- D3.js (https://d3js.org/)

Shared Tools

- GitHub

Server

- DigitalOcean
    - Ubuntu 16.04.1
    - Uname -a
        - Linux crawlr-ubuntu-512mb-sfo2-01 4.4.0-63-generic #84-Ubuntu SMP Wed Feb 1 17:20:32 UTC 2017 x86_64 x86_64 x86_64 GNU/Linux

Team Member Completed Tasks

Himal Patel - Casper Crawler Developer

Time	Tasks	Time Estimate (hrs)
Week 3	**Learned** casperjs web scraper **Learned** casperjs integration with Node.js **Created** initial code to start crawling websites	**15**

	Learned scraper/ piping method for log files	
Week 4	**Designed** keyword algorithm (halts if found) **Ensured** proper integration for scraper/server **Tested** functionality of web scraper **Created** and submit week 4 update video	**15**
Week 5	**Designed** algorithms for BFS and DFS **Continued** working on casperjs(web scraper) code **Verified** required scraping is performed	**15**
Week 6	**Designed** URL recording algorithm **Completed** casperjs(web scraper) code **Tested** integration between pipe and scraper **Reviewed** Mid-Point Check	**15**
Week 7	**Reviewed** visualization portion(D3) and website **Tested** scraper performance versus requirements **Refactored** crawler/scraper code **Created** and submit week 7 update video	**10**
Week 8	**Adjusted** any necessary casperjs code **Integrated** casperjs with website options **Ensured** scraper finds all links **Tested** crawler integrity(find bugs)	**10**
Week 9	**Fixed** bugs(if any) **Verified** crawler is ready for deployment **Performed** additional testing prior to finalizing	**10**
Week 10	**Drafted and finalized** report	**10**
	Total	**100**

Christiano Vannelli - Data Visualizer

Time	**Tasks**	**Time Estimate (hrs)**
Week 3	**Setup/Designed** front-end website	**15**
Week 4	**Setup** server **Reviewed/ Learned** D3.js **Started** D3 in client-side code with ajax-sent json	**15**
Week 5	**Continued** D3 visualization development **Started** Cookie handling **Submitted** Week 5 Update video	**15**

Week 6	**Continued** D3 visualization development **Tested** D3 functionality **Review** Mid-Point Check	**15**
Week 7	**Adjusted** Pack Hierarchy Visualization **Ensured** appropriate functionality	**10**
Week 8	**Finished** Pack Hierarchy Visualization **Submit** Week 8 Update video	**10**
Week 9	**Tweaked** UI and styling **Polished** website appearance	**10**
Week 10	**Fixed** Cookie handling **Draft and finalize** report	**10**
	Total	**100**

Christopher Kirchner - Data Transfer Developer

Time	Tasks	Time Estimate (hrs)
Week 3	**Setuped** prototype scraper to integrate with Node.JS using scrapman **Tested** web scraping with nightmare and other libraries **Learnt** how to spawn and pipe with scraper **Learnt** basic D3 development with dashingd3js.com **Gave** week 3 update	**25**
Week 4	**Setup** BFS output data integration with front-end for more realistic GFX development using Socket.IO instead of Ajax **Started** GFX development with BFS representation using d3 **Implemented** first stable d3 version using recursive scrapman scraper	**25**
Week 5	**Built** multithreaded python scraper with lxml parsing **Integrated** multithreaded python scraper with Node.js using python shell (wrapper for child processes) **Added** node tooltip functionality **Added** keyword search functionality **Implemented** robust start, stop, and restarting controls based on user page refresh and re-entering user input	**20**

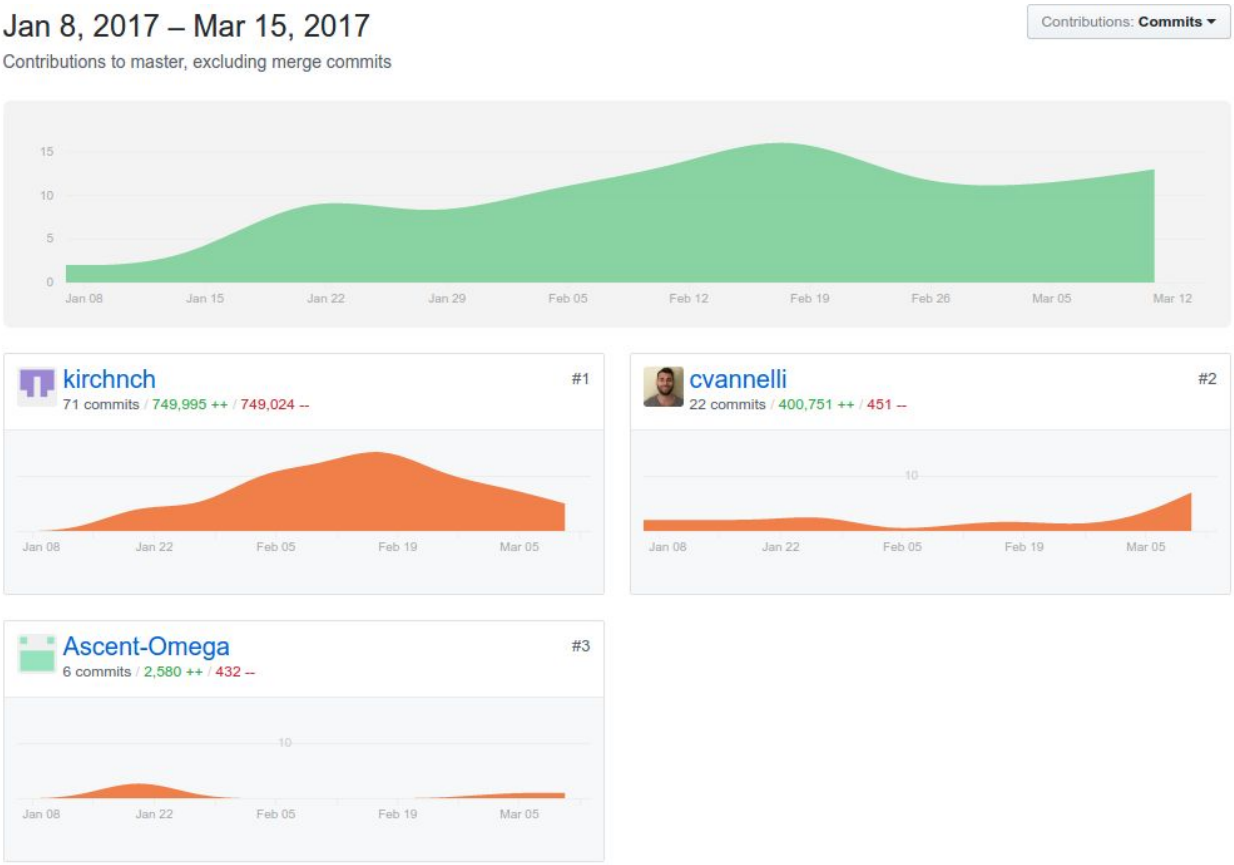
	Integrated casperjs scraper with Node.js and d3	
Week 6	**Helped** draft Mid-Point check report **Helped** host semi-functional package for Mid-Point check **Refactored** dynamic d3 code for force layout to create collapsible functionality from dynamic json hierarchical ADT **Wrote** setup script so project package and prerequisites can be easily installed automatically (includes venv and pip packages)	**20**
Week 7	**Played** with bounding box for force layout **Fixed** major difficult force layout bugs **Optimized** force layout behavior with zooming and graph repositioning **Debugged** multithreaded python scraper for memory leak using memory profiling tools **Worked** on second scraper (scrapy) after encountering memory leak in python's multithreading library	**15**
Week 8	**Fixed** additional force layout bugs **Integrated** casperjs scraper with updated d3 code **Debugged** multithreaded python scraper more for memory leak using memory profiling tools, persistent queue, separating lxml parsing in a separate process, and even replacing python requests with a curl subprocess, to no avail **Implemented** faster node expanding functionality **Added** coloring and additional highlighting to force layout	**15**
Week 9	**Added** node titles to force layout **Merged** everyone's work onto master with data transfer integration between frontend and scrapers **Setup** casperjs to run on server **Gave** week 9 update	**5**
Week 10	**Draft and finalize** report	**10**
	Total	**135**

Changes to Original Plan

We originally planned to use Jaunt , a java based web crawler. However, a multithreaded html python scraper was developed instead, with the use of a child_process wrapper for python (PythonShell library). Javascript can hide links, so we also decided to implement casperjs in

order to scrape more links. Extensive research and prototype coding with various scrapers was performed beforehand for other libraries such as scrapy-splash (python), requests/cheerio, x-ray, nightmare, scrapman, and htmlunit to see what would work best. Scrapy python was also utilized as a more production level scraper, especially after memory leaks were found in the python scraper after much effort to resolve them.
We also decided to remove the functionality of the casper scraper on the website due the RAM issues and casper crashing the server if it is being used with the other scrapers. Much of this was discussed earlier in the Casper Scraper Discussion. Issues such as these were precisely why multiple crawlers were developed.

User Github Activity



Major Sources/Help

https://github.com/d3/d3-3.x-api-reference/blob/master/Force-Layout.md
https://d3-wiki.readthedocs.io/zh_CN/master/Pack-Layout/
http://tech.labs.oliverwyman.com/blog/2008/11/14/tracing-python-memory-leaks/
http://bl.ocks.org/Caged/6476579
https://jsfiddle.net/t4vzg650/6/

https://roshansanthosh.wordpress.com/2016/09/25/forces-in-d3-js-v4/
http://stackoverflow.com/questions/19291316/force-graph-in-d3-js-disappearing-nodes
http://docs.casperjs.org/en/latest/quickstart.html
https://bl.ocks.org/mbostock/7607535
https://www.dashingd3js.com/

Conclusion

The graphical crawler is an inspirational project because it unifies the art of coding with the art of graphics design. The idea for the project had an interesting rudimentary concept and required the team to brainstorm quite a lot. Once the basic requirements were completed, we continued in expanding visualizations and efficiency in scraping. We are very proud of the work we have done and are excited with how to turned out. We are hoping that this project will continue to grow outside of this class and that we can further expand the user experience and functionality of the crawler over time.