Graphical Web Crawler Final Report

CS467 Online Capstone
March 17, 2019

Kerensa Crump - Brent Freeman - Long Le

crumpke@oregonstate.edu
freembre@oregonstate.edu
lelon@oregonstate.edu

I. Introduction:

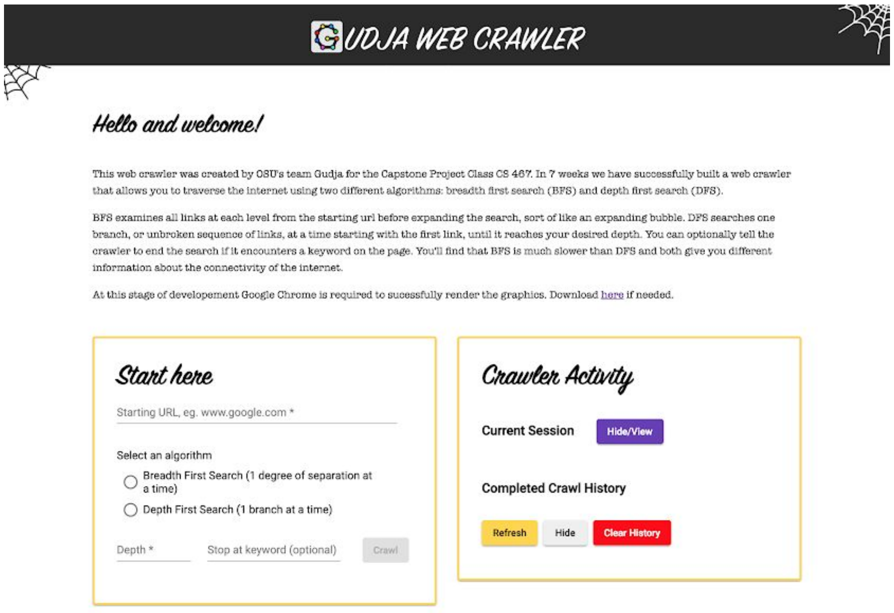
This Gudja online web crawler was created by OSU's team Gudja for the CS 467 Capstone project class. In less than 10 weeks, our team successfully built a single page application for a web crawler that allows the user to traverse the internet using two different algorithms: breadth first search (BFS) and depth first search (DFS).

BFS examines all links at each level starting from the user supplied URL. The algorithm expands the search at every level, like an expanding bubble. On the other hand, DFS searches one branch or unbroken sequence of links at a time. It begins with the first link and continues crawling until it reaches the desired depth. As an additional option, the user can tell the crawler to end the search if it encounters a keyword on a web page. You'll find out that BFS is much slower than DFS when you test out the web crawlers. Each algorithm will give you different information about the connectivity of the internet.

Our graphical web crawler allows users to visually see a web crawling program in action. The simulation will provide basic insight on how a real life web crawler, such as Googlebot, might visit pages on the web. The graphical result is a high level map of all the external pages that were visited by the web crawler.

The web crawler animation communicates the following information:

- Web page title (hover over node, view the tooltip)
- Web page URL (hover over node, view the tooltip)
- Access to the web page (hover over node, click, URL opens in new tab)
- Domain name by node color
- Links between web pages
- Directionality of the link, from source ----> target
- Node connectivity (number of links) reflected by relative diameter
- First instance of submitted keyword (if found there will be a pulsing blue highlight around the node, hover over node, view the tooltip “Keyword found here!”)



GUDJA WEB CRAWLER

Hello and welcome!

This web crawler was created by OSU's team Gudja for the Capstone Project Class CS 467. In 7 weeks we have successfully built a web crawler that allows you to traverse the internet using two different algorithms: breadth first search (BFS) and depth first search (DFS).

BFS examines all links at each level from the starting url before expanding the search, sort of like an expanding bubble. DFS searches one branch, or unbroken sequence of links, at a time starting with the first link, until it reaches your desired depth. You can optionally tell the crawler to end the search if it encounters a keyword on the page. You'll find that BFS is much slower than DFS and both give you different information about the connectivity of the internet.

At this stage of developement Google Chrome is required to successfully render the graphics. Download here if needed.

Start here

Starting URL, eg. www.google.com *

Select an algorithm

- Breadth First Search (1 degree of separation at a time)
- Depth First Search (1 branch at a time)

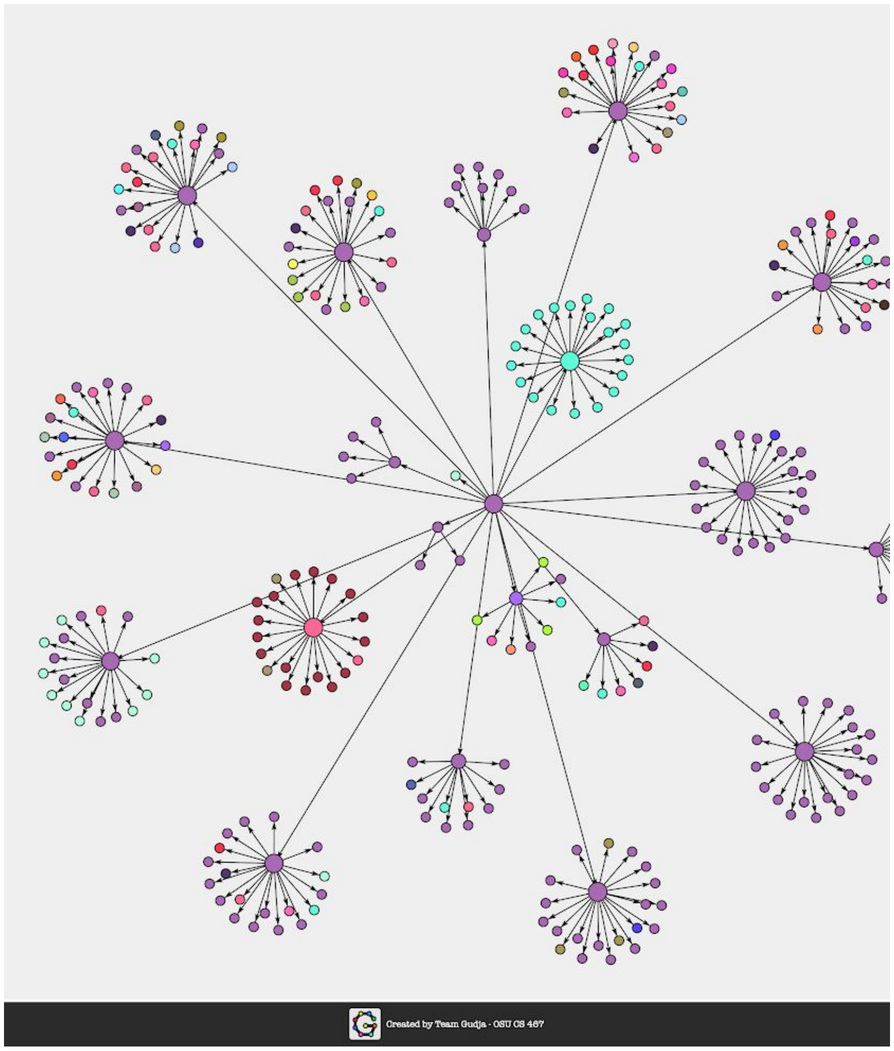
Depth * | Stop at keyword (optional) | Crawl

Crawler Activity

Current Session | Hide/View

Completed Crawl History

Refresh | Hide | Clear History



www.vox.com, BFS, Depth = 2, no keyword
(Screenshot Cuts off scrolling in the x direction inside the crawler component)

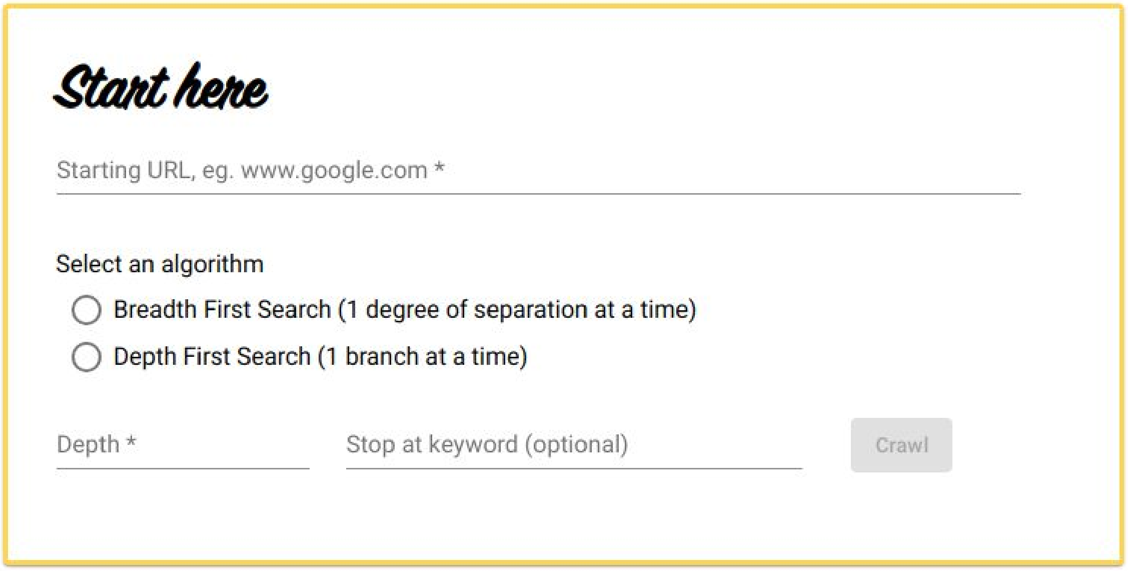
II. Setup and Use

The website is deployed on Google Cloud Platform at the following link:

https://gudja-webcrawler.appspot.com/

Please utilize the **Google Chrome** or **Chromium** web browser only. Other web browsers will not display the graphical design correctly and reduce user experience.

Website Instructions:


Start here

Starting URL, eg. www.google.com *

Select an algorithm

◯ Breadth First Search (1 degree of separation at a time)

◯ Depth First Search (1 branch at a time)

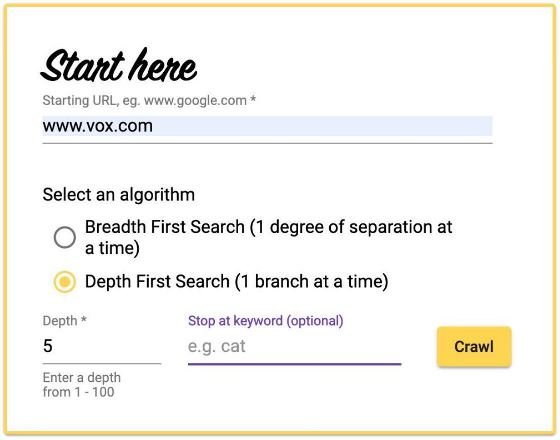
Depth *

Stop at keyword (optional)

1. Enter a starting URL for the web crawler to start.
2. Choose either Breadth First Search or Depth First Search.
3. Choose the depth level of the crawl.
4. Enter a keyword to search for on the web pages. (optional)
5. Click Crawl!
6. Look at the visualization - hover over nodes for URL tooltip.
7. Repeat steps 1 - 6

Form Validation:
Quality input is ensured through proper form validation. Here, the required fields are Starting URL, Algorithm, and Depth. If any of these is not properly entered, the “Crawl” button will remained greyed out and the form will not submit. Hints

beneath the input fields and text color changes in and below the input fields guide the user. Grey text is inactive, purple is active, and red is invalid. The depth field only accepts 1-4 for BFS and 1-100 for DFS. This reduces the risk of timing out for each algorithm. Other usability features include drop down menus on the URL and keyword fields as well as incremental arrows on the right side of the depth field. This makes repeated use of the crawler fast, intuitive, and pleasant.



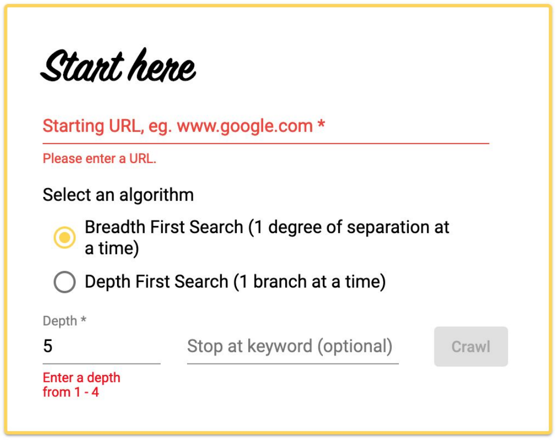
Start here

Starting URL, eg. www.google.com *
www.vox.com

Select an algorithm
Breadth First Search (1 degree of separation at a time)
[✓ Depth First Search (1 branch at a time)]

Depth *
5
Enter a depth from 1 - 100

Stop at keyword (optional)
e.g. cat



Start here

Starting URL, eg. www.google.com *
Please enter a URL.

Select an algorithm
[✓ Breadth First Search (1 degree of separation at a time)]
Depth First Search (1 branch at a time)

Depth *
5
Enter a depth from 1 - 4

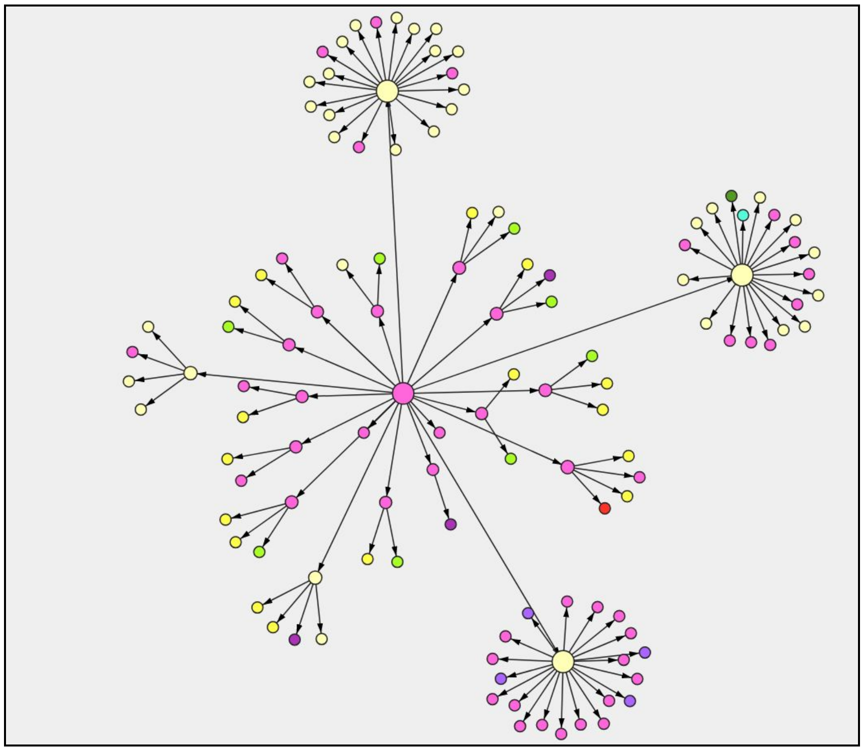
Stop at keyword (optional)

Waiting Spinner:

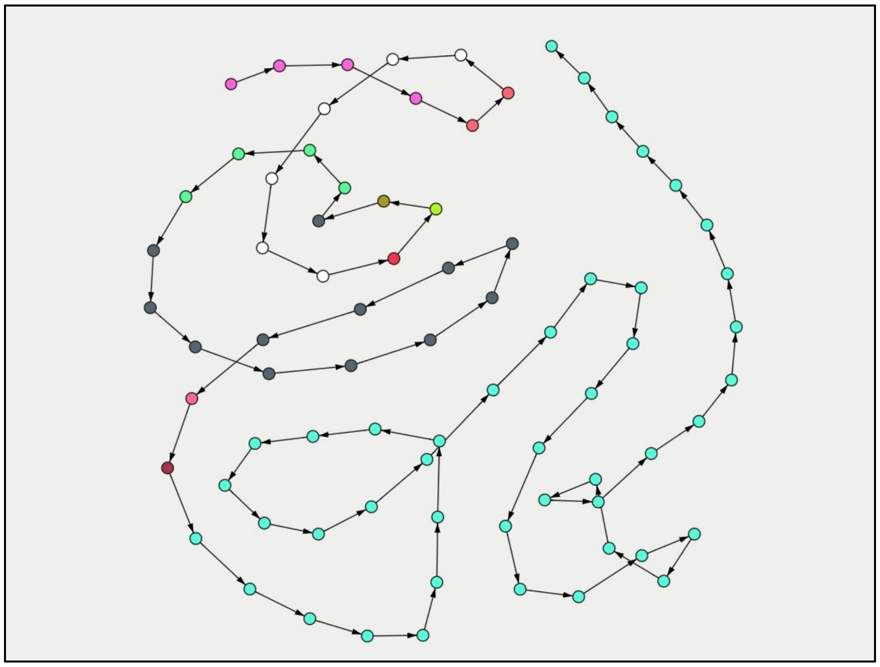
This waiting spinner animation is immediately displayed after clicking “Crawl” and disappears as soon as the data is received and the crawler graphics are rendered. This seamless transition gives the user useful feedback that their request has been submitted and is processing. The spinner will timeout after 10 minutes if data has not been received from the server and will present the error message “Error: timeout”.

Breadth First Search Result:

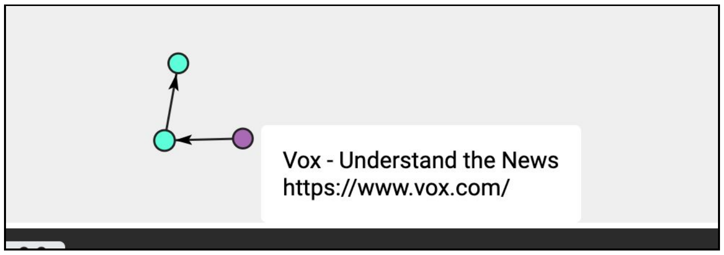
BFS results generally follow a fireworks shaped pattern, creating clusters of nodes that expand out from the starting URL by the number of links equal to the depth.



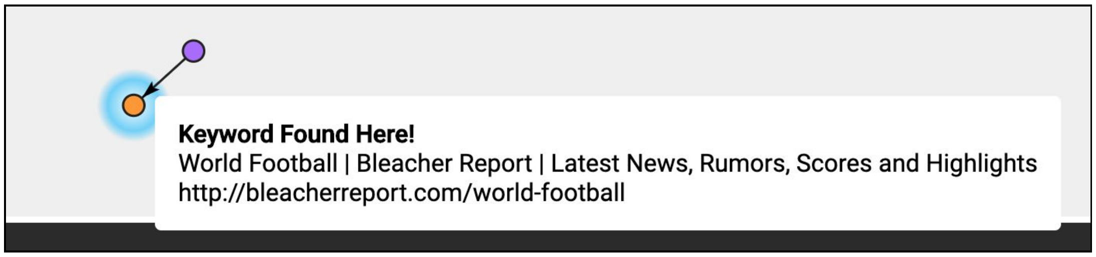
Depth First Search Result:
DFS results strictly follow a single branch pattern.



Tooltip: shows up when hovering over a node.
Click to open the link in a new tab

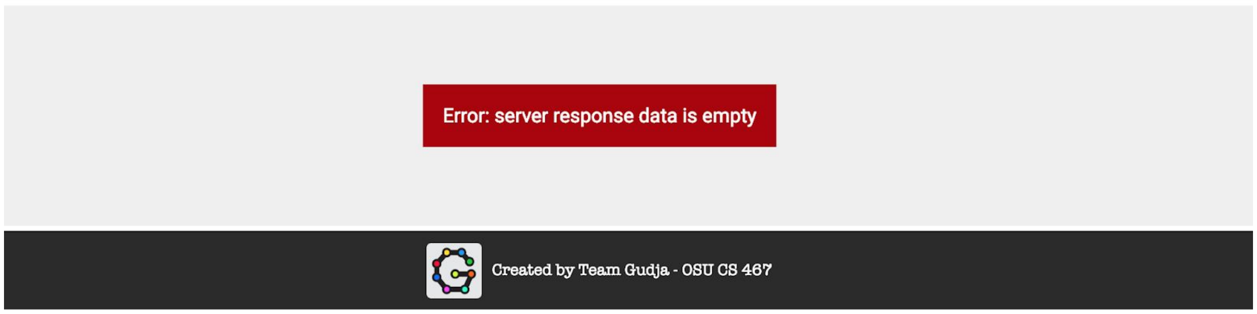


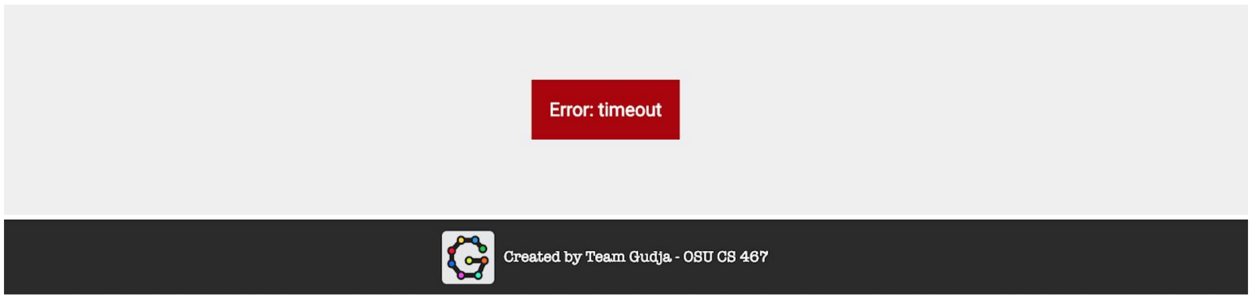
Found Keyword (halted program and highlighted URL):


Keyword Found Here!
World Football | Bleacher Report | Latest News, Rumors, Scores and Highlights
http://bleacherreport.com/world-football

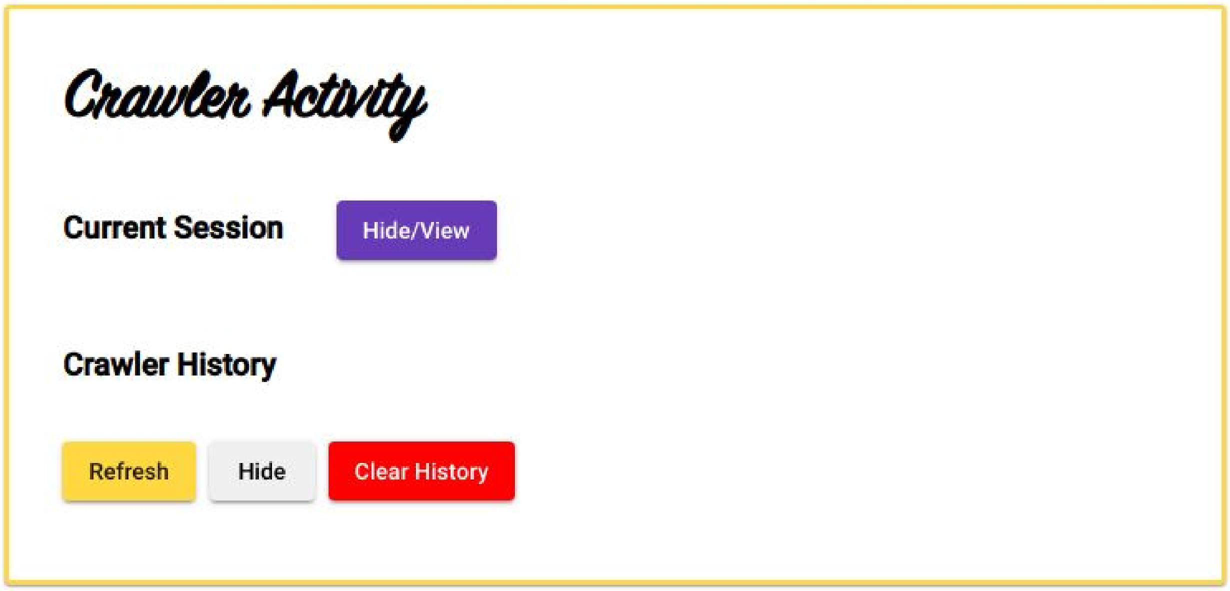
Error Messages:

On occasion during extensive crawls, the GCP cloud functions will malfunction and send back empty data, which cannot be rendered into a graphic. In this case, the error is displayed such as the image below. These errors can be caused by a few problems, notably timing out during a crawl. For both algorithms, and especially Breadth First Search, the Cloud functions attempt to make many http calls to the websites, if one of these http calls gets “hung up” it can cause a timeout which the Cloud Function can usually catch along with any other exceptions thrown by the Python function and will send an empty JSON. However, if the Google Cloud Function, on the Google business side has an issue, it should return but we have found instances where it does not return and does not timeout, which leaves the spinner spinning until 10 minutes is reached on the Angular timer and a timeout message is displayed. For further discussion on Google Cloud Functions see section III.iv and section V.i.


Error: server response data is empty



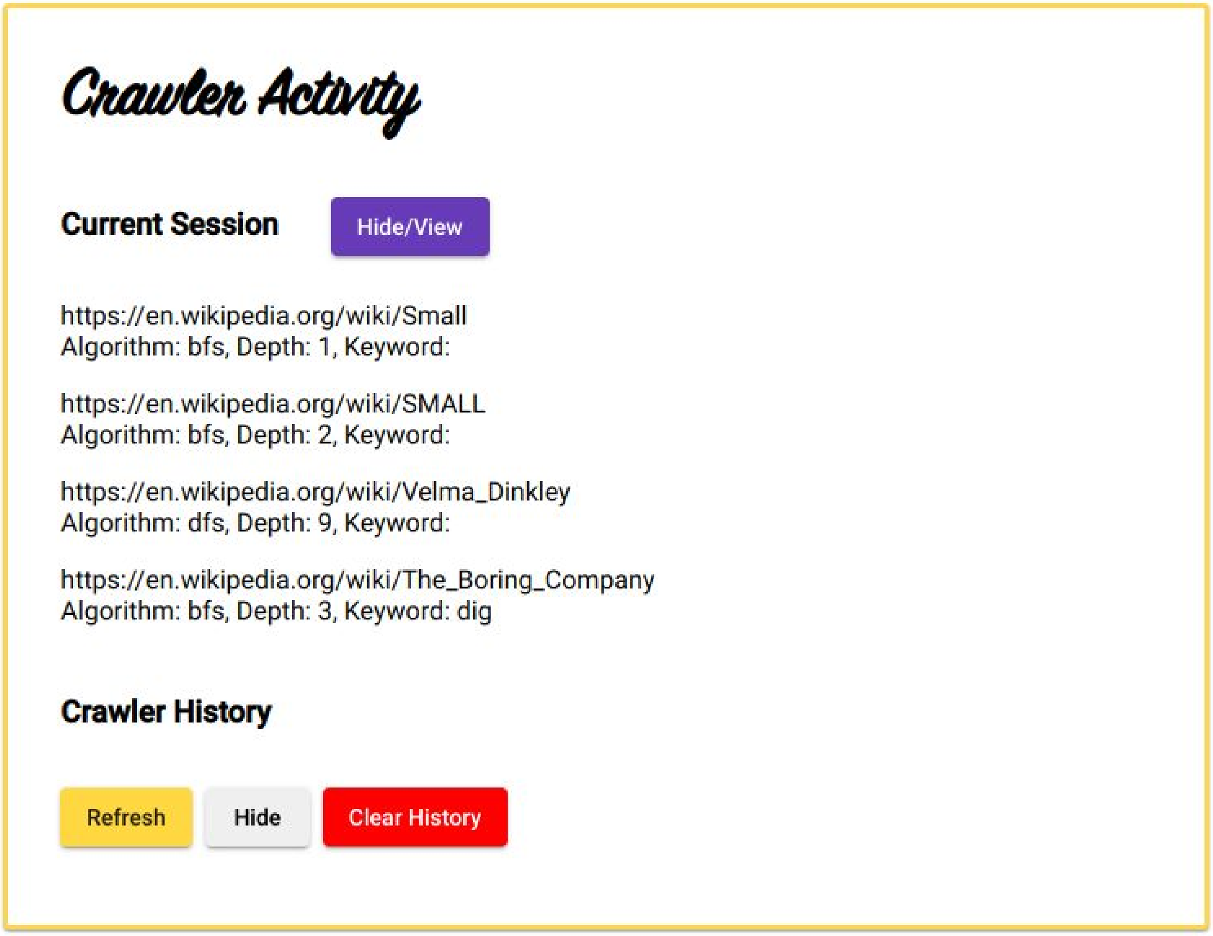
Session & History:



The Crawler Activity section keeps track of a user's URL current session activity all completed crawl history, even after closing the browser. Submitted form data will be saved to Angular "posts" data model which is immediately updated in the Current Session section. The urlHistory and keywordFoundURL cookies are sent from the server with a successful response and contain the updated url and keyword data. This completed crawl data is updated in the Crawler History section and is accessible after closing the browser. Click the Refresh button to reload and display the previously entered URLs. On the other hand, the Clear History button allows the user to delete all of the information on the cookie.

The Hide/View and Hide buttons are used for functional reasons as well as enhancing aesthetics. Hiding the session history information or the previous sessions' history allows the user to see more of the graphical results of the crawl without having to scroll further down.

Current Session Activity:



Crawler Activity

Current Session

https://en.wikipedia.org/wiki/Small
Algorithm: bfs, Depth: 1, Keyword:

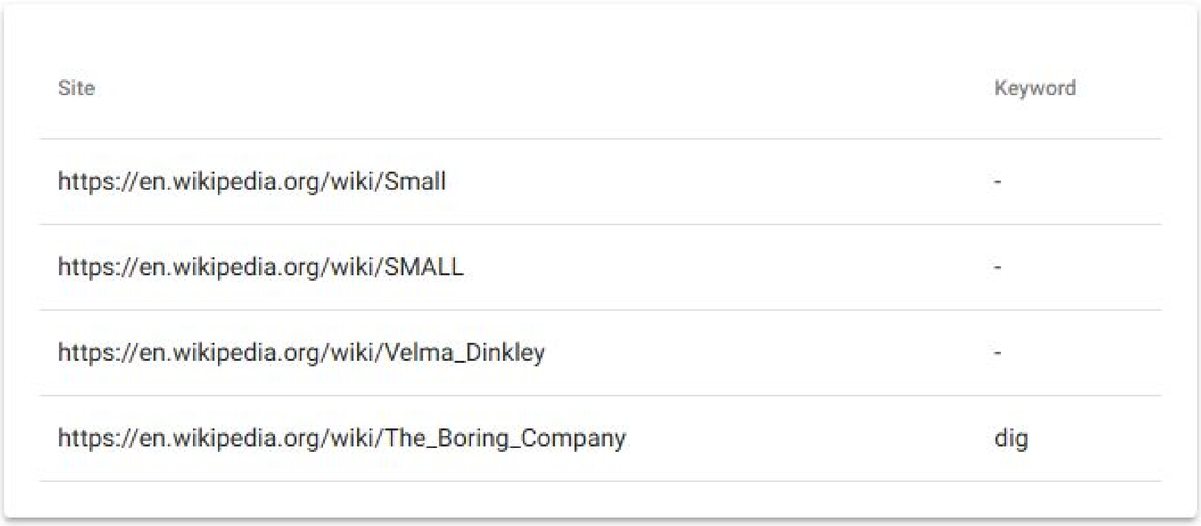
https://en.wikipedia.org/wiki/SMALL
Algorithm: bfs, Depth: 2, Keyword:

https://en.wikipedia.org/wiki/Velma_Dinkley
Algorithm: dfs, Depth: 9, Keyword:

https://en.wikipedia.org/wiki/The_Boring_Company
Algorithm: bfs, Depth: 3, Keyword: dig

Crawler History

History Recalled from Cookie:



Site	Keyword
https://en.wikipedia.org/wiki/Small	-
https://en.wikipedia.org/wiki/SMALL	-
https://en.wikipedia.org/wiki/Velma_Dinkley	-
https://en.wikipedia.org/wiki/The_Boring_Company	dig

Interesting Websites to crawl:

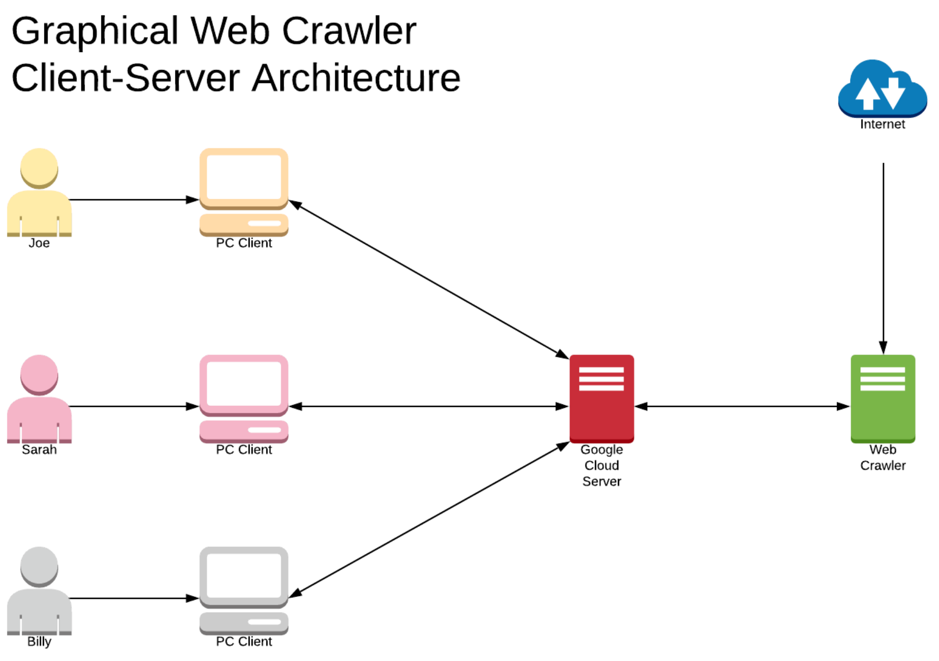
1. https://en.wikipedia.org/wiki/Main_Page
2. https://www.stackexchange.com
3. https://www.vox.com - Depth of 2 BFS

Download instructions:

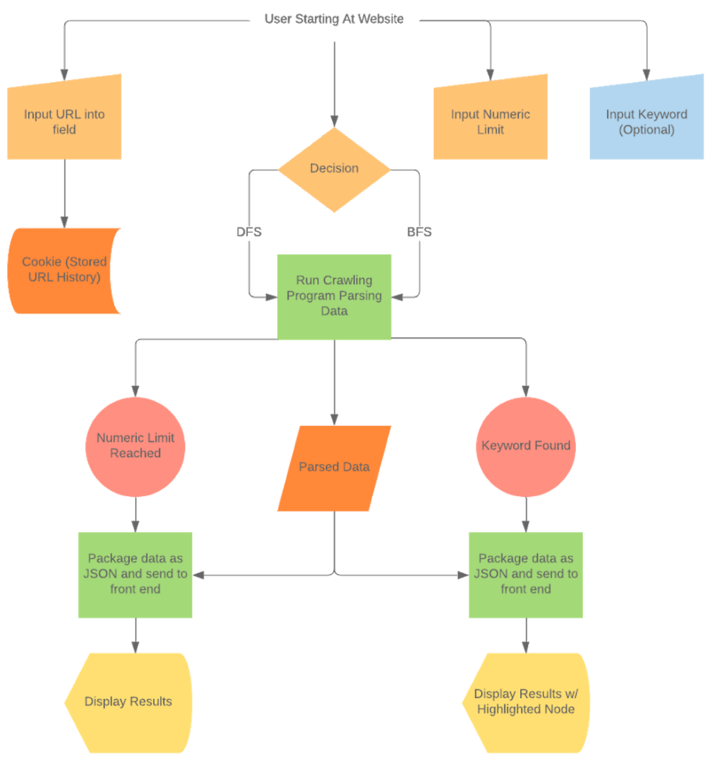
View GitHub Repo: GudjaWebCrawler

1. Clone master branch to desired directory
2. Navigate to new directory root
3. Run npm install
4. Run node index.js
5. Website will be viewable at port 3000

Gudja Web Crawler Design:



Website Flow Chart


User Starting At Website
Input URL into field
Input Numeric Limit
Input Keyword (Optional)
Decision
DFS
BFS
Cookie (Stored URL History)
Run Crawling Program Parsing Data
Numeric Limit Reached
Parsed Data
Keyword Found
Package data as JSON and send to front end
Package data as JSON and send to front end
Display Results
Display Results w/ Highlighted Node

III. Software Systems

i. Angular

Angular provided the framework to create a responsive single page application with two way data binding to maximize D3's graphical rendering. Angular's MVC structure was accomplished using models, components, and services. This structure utilized observables to update the Model and View interchangeable for form submissions, POST responses, updating cookies, and rendering D3 graphics. Each component controlled its appearance logic and made calls to service functions which handled the business logic, reducing coupling. Angular also integrates with our other tools especially D3 and Node, and is compatible for use on the Google Cloud Platform.

External packages

a) **Subject from rxjs**
Similar to an EventEmitter, it creates an observable for a selected data model that emits update notifications.

b) **Subscription from rxjs**
Subscribes to updates emitted by the selected data model Subject.

c) **CookieService from njx-cookie-service**
Convenient package for reading and writing to specific cookies.

d) **Angular Material**
Angular Material is a library of polished Angular components that can be used in place of traditional html elements and use custom directives.

ii. **D3**
D3 is a JavaScript library that uses native HTML, CSS, and Javascript without special engines to turn data into a wide array of visualizations. D3 worked well with the Document Object Model through its built-in data binding. The core of the network graph visualization used the forceSimulation module. Features like the tooltip, dynamic svg sizing animation, and pulsing highlight were managed separately using custom functions tied to events like mouseover and simulation tick. Critically, D3 is very fast which helps offset the inherent lag of crawling the web. Our application is only slow getting the data, not displaying it.

a) **forceSimulation module**
Network graph simulation utilized separate forces (negative charge force, link force, x and y spread forces) to render the position of the nodes and edges with a numerical integrator, creating a smooth animation.

iii. **Node.js**
The backbone of or app, Node is both familiar to the whole team and also simple to use. Node was very effective through all iterations of our integration with the Python based web crawler.

Node Packages

a) **Express**
Framework for creating routes.

b) **Cors**
Enables Cross-Origin Resource Sharing so that our front end can effectively communicate with our back end.

c) **Request-Promise**
Combines http request with a promise to easily enable asynchronous actions within the back end.

d) **Body-Parser**
Parses the requests to easily accessible objects.

e) **Cookie-Parser**
Used to manipulate cookies

iv. **Python 3.x**

Python was chosen as the programming language to develop the web crawler programs because it was an opportunity to learn something new. There are many positives to using Python which includes the following: clean looking code, higher levels of abstraction, fast development speed, great documentation, and a large number of useful libraries.

Python Libraries

a) **Beautiful Soup 4**
A library used to pull data from HTML and XML files.

b) **lxml**
Fast HTML parser.

c) **urllib.parse**
Interface to break URL strings into components.

d) **urllib.robotparser**
Module used to to assist with respecting robots.txt.

e) **NumPy**
Used as an efficient multi-dimensional container of generic data.

f) **requests**
HTTP library for Python.

g) **json**
JSON encoder and decoder library.

h) **sys**
System-specific parameters and functions.

v. **Google Cloud Functions**

Google Cloud Functions are Google's answer to serverless technology. We created our individual functions written out as Python methods, Google hosts these functions, and we can access them with a http request. See figure *Gudja Web Crawler and Google Cloud Platform* below.

1. **cloud_bfs**
A self contained Python function that receives input via an http POST request, performs the Breadth First Search and returns the results as a JSON.
2. **cloud_dfs**
A self contained Python function that receives input via an http POST request, performs the Depth First Search and returns the results as JSON.

vi. **Google App Engine**

Google App Engine serves as the host for our application. This platform allows for a relatively easy method to share our application with the public with less worry about maintenance. It too has the capability to support our different languages allowing us a lot of freedom during development. See figure *Gudja Web Crawler and Google Cloud Platform* below.

vii. **Slack**

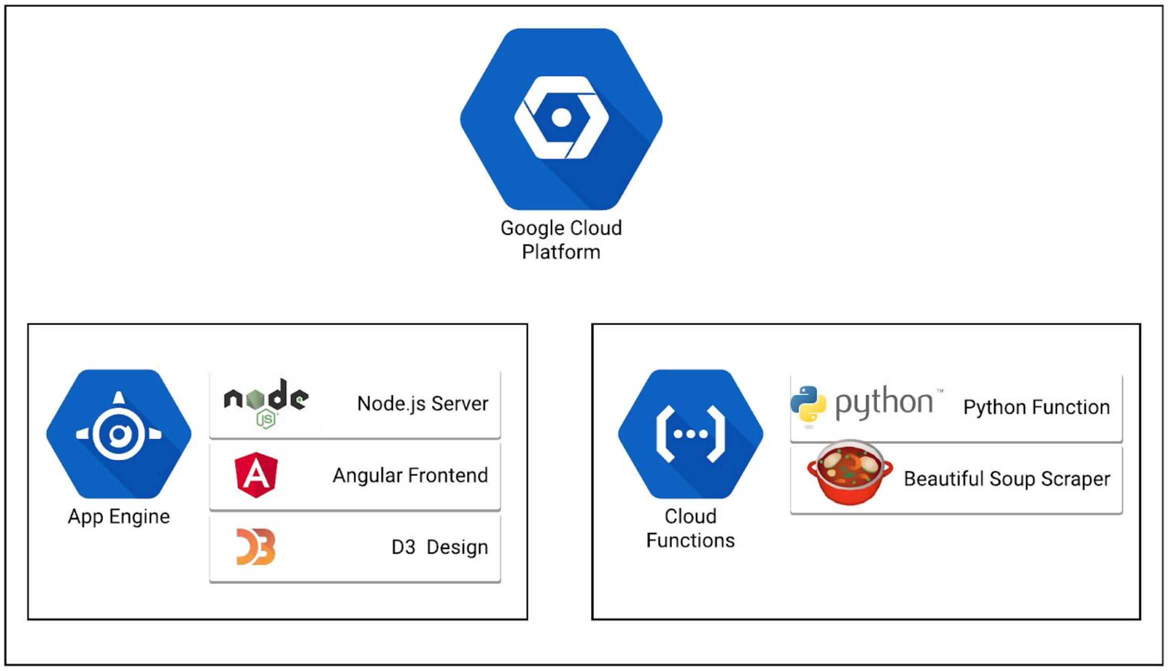
The go-to communication platform. Slack was instrumental in allowing us to share our progress, ideas and frustrations and fixes. This was particularly useful as we live and work in different areas, time zones and on different schedules such that real time communication would be difficult to plan and execute.

viii. **Git and GitHub**

GitHub was used for version control during the development of the Gudja Web Crawler. It was particularly useful for managing four different front ends that had to be compiled into Angular production code for three GCP testing sites plus the local testing environment http://localhost:3000. This made it convenient for each person to get the latest front end updates and deploy quickly to their testing site. The use of Git and GitHub was therefore essential for testing experimental branches and being able to quickly run the program on different platforms, while safely maintaining code.

https://github.com/lelon32/GudjaWebCrawler

Gudja Web Crawler and Google Cloud Platform



IV. Team Member Contributions

i. Kerensa Crump

The Web Dev Maestro and design Ace, Kerensa turned a bunch of code to a fully functioning application. She took the opportunity during this Capstone project to build her front-end skills by learning Angular and D3, producing 100% of the front-end design, logic, and animation. She was able to create a visually beautiful, and working web page on Chrome using Angular and will add browser compatibility as well as mobile responsive design in the near future. She learned the basics of Angular through a Udemy course[3] and several tutorials throughout development [4] - [6]. In creating the graphic display, Kerensa sifted through the D3 visualization library and online demos [7] - [13] for ways to make the data visually appealing and informative. She also contributed to the node.js server side code, particularly setting up the routing, making the crawler functions asynchronous, and handling cookies. During late nights when coding was no longer productive, she created the Gudja logo and favicon in Illustrator and added decorative svg elements to enhance to overall aesthetic. She contributed images, text, and suggestions to the poster and the write-up. Like her excellent teammates, Brent and Long, she was an active participant on Slack throughout the 10 weeks, handling broken graphics, promise bugs, and network timeouts as part of the team.

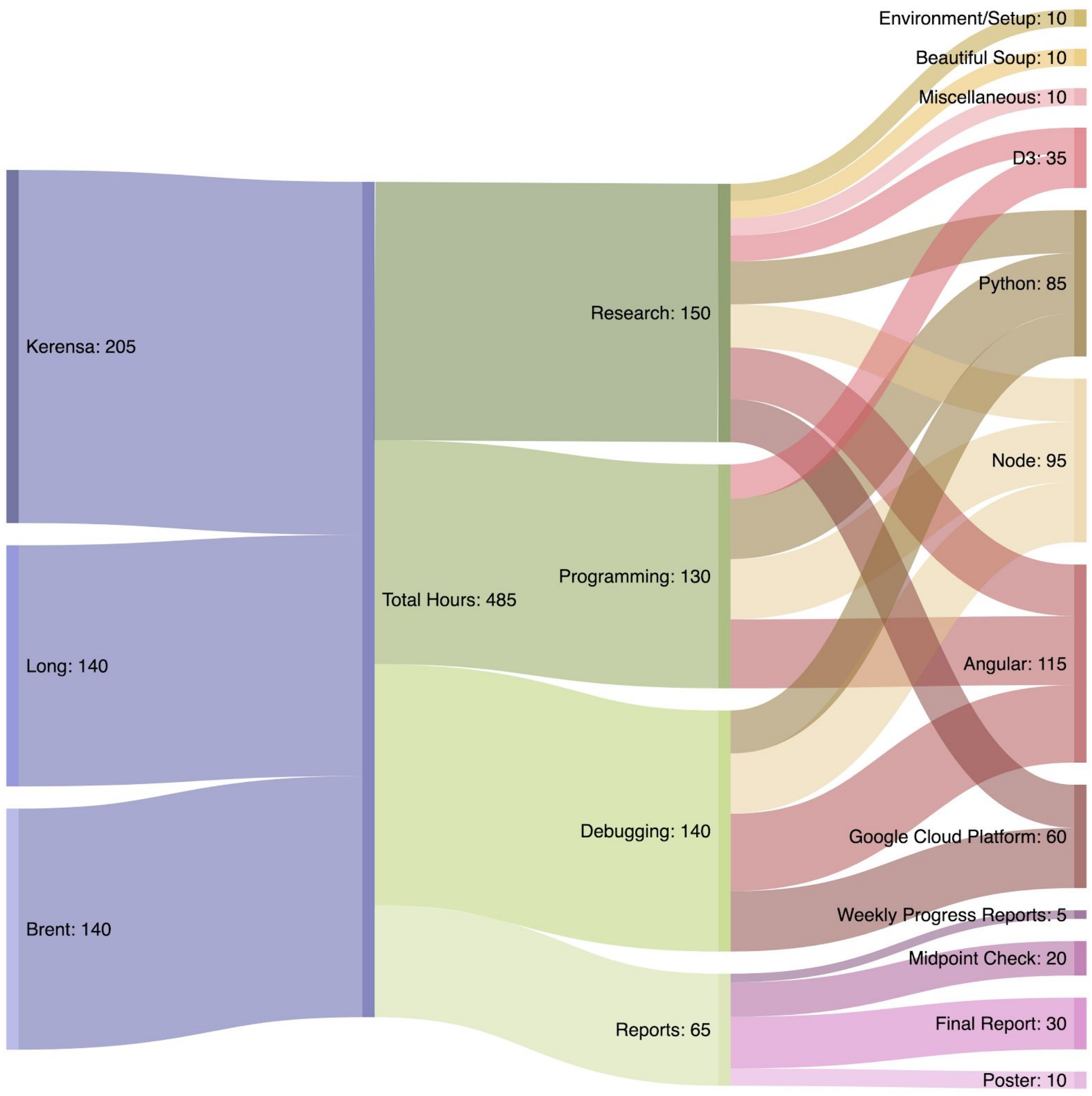
ii. Long Le

Enthusiastic team member and Python pro, Long primarily developed the breadth first search web crawling program and supported backend development for the website. Long researched, designed, and implemented the breadth first search algorithm in the Python language. He implemented the depth limit capability for BFS and the URL validation feature. He developed the initial user entered URL history feature using cookies in JavaScript. Long implemented the keyword search and program halting feature into BFS. Additionally, he researched and revised BFS to respect the robots.txt file while crawling web sites. He also implemented initial method to transfer data from node.js to Python child processes, as well as constructing the necessary JSON data to transfer back. Long’s other contributions include useful small functions to parse out URLs, setting up GitHub for the team, identifying bugs, cleaning up code, creating design diagrams, and contributing to and revising written assignments. Finally, he assisted to debug code as well as provided minor contributions to optimizing BFS to run faster.

iii. **Brent Freeman**

The Google researcher, trouble shooter, and rock of the team, Brent spent a lot of time just getting the web application to work (see Python to Front End). Originally hoping to spend more time on using Python, Brent created the initial WebCrawler class from which the Breadth First Search and Depth First Search (BFS and DFS) would call while also developing the depth first search. While having a fairly good understanding of Python, it was still a challenge to set up a proper environment that could be accurately reflect running it locally vs running the application as a Google app. The nuances of how Google hosts applications on its servers lead to many unexpected challenges that Brent worked through to get the app running live. The complete list of Brents work is: Initial WebCrawler class, DFS, hosting on Google App Engine, modifying BFS and DFS to run as Google Cloud Functions, creating initial methods in Node server to make http calls to the cloud functions and pass the data to the front end, creating the readme.md for the GitHub repository in markdown, created the hours worked flow chart, and wrote and created graphics for the reports/posters/progress videos.

iv. Hours Worked Flow Chart[14]



V. Deviations and Challenges

i. Data Flow: Node to Python to Node to Front End

Initial development of the application looked at using a Node back end with web crawling run by Python functions. The only question was how to get Node and Python to pass data between each other. A quick search determined that the plan was very doable with this blog post showing exactly how to do it[1]. This became our guide and we set out to make it work and we were quite successful. The method was that we would pass data using standard out and then create a data.json file that could be picked up by the front end.

This worked very well throughout our individual testing and we thought we had a winning plan. Contributing to our success was that not only did we each test locally, we also tested on Google Cloud Shell which used a “local” instance hosted by Google. Through the midpoint progress report the application was deployed but connectivity errors were focused on cross origin resource sharing problems that prevented the front-end from communicating with the back end. After the midpoint, those issues were resolved but there was still no communication between the back-end and the Python functions.

After some frustrating searches it was found in a google document related to PHP[2] that “ the local filesystem that your application is deployed to is not writeable.” No other documentation on Google discussed this issue. Luckily, there was a fix right below that section that discussed the use of Google Cloud Storage where our python functions could write our data.json and our Node server could retrieve it. Alas, this too was not meant to be.

While the team was able to successfully redirect output to the new Google Cloud Storage Bucket, they then realised that it would be difficult to keep track of multiple read writes using the same file name. While it would not be too difficult to append a unique identifier to each new file, this did not seem like an optimal solution. It was then that another option came to the fore from a previous search for solutions-- Google Cloud Functions.

Cloud Functions are a relatively new service using Python as an option and has only been available since July 2018[15]. Cloud Functions provide an up to date Python runtime that turned our Python programs into an http triggered function used by sending a request to a unique URL. A simple POST request to the selected url from the server passes the request information as a JSON. From there, the Cloud function runs on a Google server, providing us a lot of control over which libraries to use, the size of the return values and some of the timeout length (up to 9 minutes.) We quickly refactored to take advantage of these and found them very quick to deploy and relatively easy to use. The advantage of Cloud functions are that it allows much easier communication between languages, allows for cleaner code and a simpler back end, and is much easier to test on multiple computers and different users. The downsides, however are that we are reliant on Google to maintain its networks (which can be problematic as the Python runtime is in beta testing) and we have run into some issues with down time as we cannot control all of the environment (timeout length beyond 9 minutes, file size, python version)

ii. Time Outs and Web Crawling Woes

One major obstacle during development of our web crawler is the fact that the program times out. The front end was not able to wait long enough for data to be returned from the back end. The time limit on GCP is up to 9 minutes but node.js times out even earlier than this limit. On numerous occasions, the web crawling function completes the crawl and sends data back to the front end, but node.js has already timed out and thus cannot receive data.

There were a few quick fixes which helped to alleviate the problem in the short term. The first fix was to exclude parsing all internal links to allow the crawler to run faster. The additional design justification for this is to allow the web crawler to jump from external website to external website, which provides a more interesting path. The second quick fix was to limit each parsing execution in BFS to return a maximum of 20 URLs for each examined URL, rather than being unlimited. The third quick fix was to limit the user to a maximum of a depth level of 4 as the limit for BFS and 100 for DFS. Any more than 4 levels for BFS and the program and the application will time out. The fourth fix addressed node.js timing out at 2 minutes by disabling the node timeout entirely.

A more permanent fix for the time out issue is to implement a live data feature in the application, instead of sending static data to the front end after the web crawling program completes. The application should continuously send and receive updated data as the web crawler crawls the web. This idea is likely to involve a major design change to how the application works, which our team would not have time to implement. Mostly likely, our application would need to be running a database server where the web crawler would insert data continuously, and the front end can read data from a stream.

We also encountered other crawling issues besides timing out. Certain websites were keen on not allowing the web crawler to crawl the site. One fix was to revise the parsing feature of the web crawler to respect the robots.txt file. This fix helped helped to all the web crawling to be able to crawl many additional sites.

iii. Bug Report

Bug 1: DFS does not continue to requested depth

Developer Notes: DFS is designed to successfully complete in 3 ways, reaching the requested depth, finding the keyword in search, or running out of links (dead end) this is by design.

Corrective Actions: a future version should try to step back one link previous to try a different link to limit the number of dead ends. Additionally, the front end can display a message alerting users that a dead end was reached and this is normal.

Bug 2: App only works in Chrome

Developer Notes: The application was developed using the latest version of Angular and D3. This version takes advantage of the latest version of javascript supporting most modern browsers but fails for many old versions. Additionally, Team Gudja developers only used Chrome or older versions of Safari and did not test other browsers for compatibility in CSS and Javascript.

Corrective Actions: Application should modify the CSS and D3 code to work on all up-to-date browsers to include, Chrome, Edge, Firefox, Safari, Android and iOS.

Bug 3: BFS takes a long time to run

Developer notes: BFS examines every link on a website, and then does the same thing for each of those links and then again. The time increase is exponential as depth increases. The time it takes is at the whim of the speed of the networks, the speed of google processors where the BFS function is hosted and the number of links it needs to traverse.

Corrective Action: The BFS code was developed in Python using readily available libraries. Switching to other libraries or even different languages may improve speed, Additionally, due to the nuances of how Google hosts its applications, the BFS function needed to be moved and turned into a separate cloud function accessible by an http request. While this increases the flexibility and portability of the application it may also cause more slowdown. Future development should focus on hosting it on another site.

Bug 4: BFS did not keyword search root URL

Developer notes: Minor bug that was overlooked during the development of the keyword search halting program feature.
Corrective Action: The code in cloud_bfs.py did not perform the search on the root URL. Added this code in and utilized a flag variable to end the main loop, to ensure the JSON data was properly created on the root URL.

Bug 5: The node.js front end was timing out too soon

Developer notes: The front end was tested and it was consistently timing out at 2 minutes for extensive crawl requests. This appeared to be a front-end issue until it was discovered that the node server was sending an empty response after it timed out at 2 minutes.
Corrective Action: Disabled the node.js request timeout with req.setTimeout(0) which allowed the front end to run for much longer without timing out. The front-end timeout logic was capped at 10 minutes for usability. Angular displays a timeout error.

Bug 6: Cloud functions returns empty data

Developer notes: The cause seems to be isolated to the cloud functions. Testing locally does not return empty data. The issue only happens during deployment on GCP.
Corrective Action: Research and continued testing on the cloud functions call_bfs and call_dfs.

Bug 7: Svg element cut off large crawler animations

Developer notes: The forceSimulation animation was built to populate an svg with a fixed width and height.
Corrective Action: A custom differential function was written in D3 to adjust the svg width and height and translate the animation during each tick of the forceSimulation.

VI. Conclusions

As a team, the experience of developing a graphical web crawler gave us insight on the many difficulties of learning and implementing new technologies. The process has allowed us to see the effects of design flaws and well as the importance of development methodologies, such as agile. There were many unexpected adjustments that were needed to make our application work correctly. We overcame many of the challenges that we faced and created a working website that crawls the web and provides a graphical representation.

In the process of working together, we also discovered the importance of diverse knowledge as well as diligence, commitment to our work, and mutual respect for our teammates. We learned to play on the strengths of our teammates and depended on each other to provide expertise in their specialized area. This allowed us to concentrate on our own strengths to contribute to the project. As a team, you can complete tasks much faster and also provide a better end product.

Ultimately, our team has done our best to create an application that represents the original vision. While not perfect, every requirement was addressed and the website looks visually appealing. This iteration of application release can be considered alpha, in order to allow for user feedback and end user testing. We hope you enjoy using our graphical web crawler and are able to appreciate the amount of work that was put into it.

References:
[1] https://www.sohamkamani.com/blog/2015/08/21/python-nodejs-comm/
[2] https://cloud.google.com/appengine/docs/standard/php/googlestorage/
[3] Angular course - Udemy
https://www.udemy.com/angular-2-and-nodejs-the-practical-guide/
[4] Angular Material Components - Angular Material
https://material.angular.io/components/categories
[5] Angular table rendering - Stackoverflow
https://stackoverflow.com/questions/49284358/calling-renderrows-on-angular-material-table
[6] Angular add CSS classes - malcoded
https://malcoded.com/posts/angular-ngclass
[7] D3 force directed graph - bl.ocks
https://bl.ocks.org/mbostock/533daf20348023dfdd76
[8] D3 thinking with joins - bost.ocks
https://bost.ocks.org/mike/join/
[9] D3 force directed graph - youtube
https://www.youtube.com/watch?v=cJ6NdluzEG8&feature=youtu.be
[10] D3 conceptual guide - davidwalsh
https://davidwalsh.name/learning-d3
[11] D3 Radial gradient - bl.ocks
https://bl.ocks.org/pbogden/14864573a3971b640a55
[12] D3 arrow markers - Medium

https://medium.com/@raghothams/markers-with-d3-7b01b8be9efe
[13] D3 Pulse animations - bl.locks
https://bl.ocks.org/cherdarchuk/822ba3ead00a0ffdbcfd4a144e763e31
[14] Created with SankeyMATIC
http://sankeymatic.com/build/
[15]Google Cloud Function Release Notes
https://cloud.google.com/functions/docs/release-notes