Mid-Point Check: Graphical Crawler Application

Team Grus: Aaron Berns, Jacqueline Bonner, Hunter Schallhorn

Project Code: https://github.com/anberns/grus

Project Link: https://graphicalwebcrawler.herokuapp.com/

Project Status

Our team has accomplished several of the tasks necessary to complete our project. Progress in each area is detailed below.

1. **UI**

   The first major piece that is up and running is the app's home page. It consists of a header and a form that submits user choices to the app's server for the starting URL, optional keyword, page limit and search type. To the right of the form is a table which displays the previous crawls that have been done from the current browser. Clicking on the URL of a previous crawl queries the database for its saved crawl results, resulting in a graphical display of the crawl without running the crawler function. The page has been styled with bootstrap.

   The future work planned for the main UI is to add an error message so the server can relay back to the user when there has been an issue. We also plan to reorder the previous searches so that the most recent search is listed at the top.

2. **Data Visualizer**

   The second major portion of the app is the visualization page. The header displays information about the crawl, such as the type of search, the starting page, and the keyword searched for (if there is one). The visualization is a simple d3js force directed graph. The visited sites are displayed as circles or nodes and the path taken through the sites is shown as bold black arrows, from one node to another. The title of each visited page is also shown and the url of the page is shown when user hovers over the node or title. The other discovered links are also displayed, but with greyed out nodes and links to indicated that, while these urls were found in the crawl, they were never visited.

   Future work for the visualization includes navigating to the selected url when the user clicks on a page title or node, adding a highlight to indicate the starting page and the page where the keyword was found, and making the visualization zoomable. We also plan to add a websocket to push data to the visualization as it comes in from the crawler. The d3js code is already set up to be updated dynamically and pushing data to it via the websocket will solve some of our timeout issues. There are also a few bugs that need fixing, such as links overlapping each other and nodes being forced off the sides of the visualization.

3. **Server**

   The third major portion of the app that is functional is the server. Our app is hosted on heroku and can be accessed at https://graphicalwebcrawler.herokuapp.com/ . The server is written in Python and uses the Flask library to enhance its functionality. It uses Flask_Pymongo to connect with the MongoDB database. The server has a route for the index page, which stores a new uuid4 cookie to begin to track previous crawls, or reads an existing one and queries the database for past crawls to show on the app's home page. The server also has a submit route which accepts posted data from the home page, calls the crawler function from the imported crawler module, stores the crawl parameters and results in the database and renders the visualizer template, passing the crawl data to the visualizer implementation. A third route handles the requests for previous crawl data by querying the database and rendering the same visualizer template with the results, bypassing the crawler.

   Future work on the server will implement a WebSocket connection between the Crawler and the d3.js data visualizer for real-time transfer of data, allowing the visual graph to grow as the crawler crawls. Any additional features that are added to the app in the coming weeks will be supported by server additions where necessary.

4. **Crawler**

   The crawler functionality has also been written, which currently instantiates either a BFS or DFS Spider object in order to perform the crawl with a given starting url, depth limit and optional keyword. Both types of searches begin by parsing the initial page using Requests and BeautifulSoup in order to find all of the links on the page and the title of the page. Then, they follow the links found in two different ways, as discussed below, before returning the results to the server.

   a. **Breadth First Search Functionality:** The BFS starts by saving off information about the current page in a dictionary format - the url, title of the page, list of links available on that page, the current depth of that page, and the parent of that page. It then goes through all of the links that were found and adds them to a queue of links to be visited, along with pertinent information about those urls. It loops back around until it either has an empty queue, has found the keyword, or it reaches the user-specified depth limit. At the end, it has built a dictionary to be returned to the server that contains all of the urls that have been visited, along with all information saved about them.

b. **Depth First Search Functionality:** The DFS starts off by saving off the information about the current page, in the same way as the BFS. However, instead of going through all of the links found, it picks a next link to follow at random, discarding the rest. This loops back around until it either has no next link to follow, the keyword has been found, or it reaches the depth limit set by the user. At the end, it has built the dictionary to be returned to the server, like in the BFS.
c. **Keyword Functionality:** At the end of each loop in both the DFS and the BFS, there is a check to search through the parsed page using BeautifulSoup's find_all function. In the function, we passed in a string argument set equal to the keyword passed in by the user as a regular expression using the re Python library. The regular expression was added to ensure that, for example, if the keyword were "foo", and the string "foo bar" was found, it would still match.
d. **Error Handling:** The crawler is currently set to error check for a valid url using the validators library before adding it to the list of urls to crawl. It also has some error checking built in when the url is requested - as the status code is checked before the page is parsed. The crawler also attempts to check for relative urls and joining them with a base url if necessary and checks for repeated urls, so as not to crawl the same site over and over. In addition, if the crawler runs out of valid links to follow, it will exit early and provide the results that it was able to retrieve.

The next step for the crawler is to refactor it to use sockets in order to send pieces of data to the server instead of sending the entire built structure at once. This is important because, currently, our app is timing out when trying to conduct almost all BFS crawls and also DFS crawls of a depth larger than about 10 or so. This appears to be a limitation of Heroku.

Also, the crawler cannot currently find all of the links on pages that have JavaScript generated content. A solution (using Selenium and Chromedriver) had been used in an attempt to handle that issue, but it ended up being so slow, that it hampered progress. So, more research needs to be done in how to appropriately handle those pages.

In addition, more error handling needs to be added. For example, during testing, it was found that if, when requesting the page, a bad status is returned, the crawler can sometimes bug out and get stuck in a loop. So, to prevent this while a fix is being designed, the crawl may stop early.

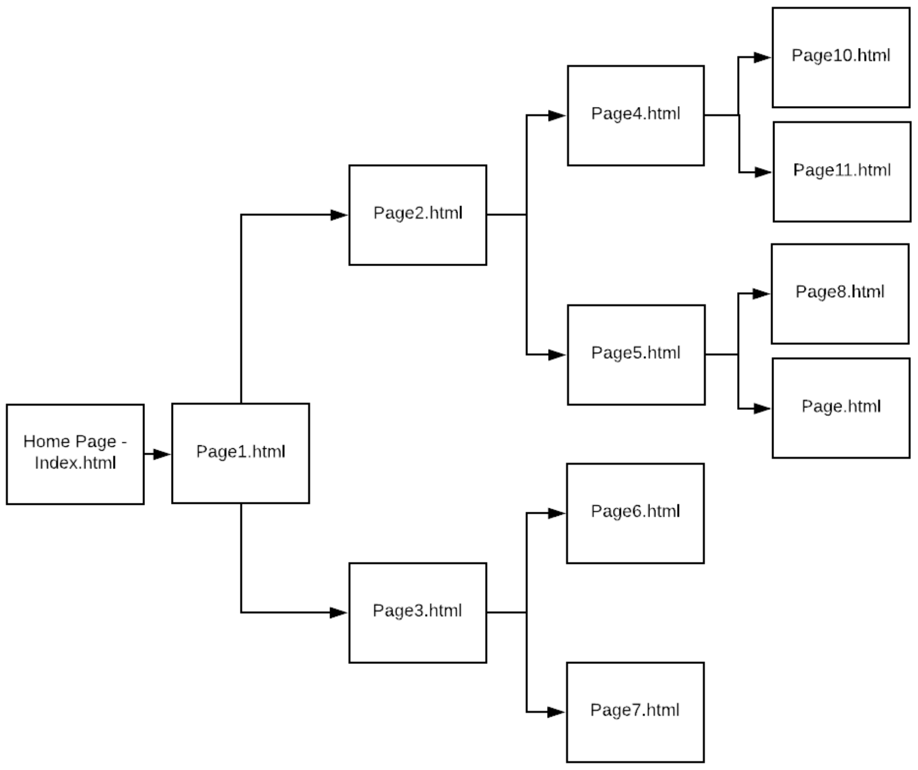
5. **Database**

   The database type is MongoDB, hosted by mLab through heroku. It is functional and set up with a test collection for storing test crawl data. The server is able to insert and query the database and basic testing has shown that each new crawl is stored under a unique document id and that all intended crawl data is being stored properly and is being retrieved intact. In the future, any data modeling and indexing that becomes necessary will be implemented.

6. **Testing**

   For initial testing of crawls, a testing site was designed to show a small crawl. This was done so that small-scale unit testing could be done in a known environment. It was also done to test the breadth-first search functionality as the application is currently having timeout issues with breadth-first searches of sites with many links, as discussed above. The structure of the test site, found at https://www.bomanbo.com/ is shown graphically below.

   Keyword functionality can also be tested using this site as the word "keyword" was put onto page 4 for testing and the word "page" can be found on all of the pages.



User Instructions

1. Navigate to https://graphicalwebcrawler.herokuapp.com/
2. Choose the search type: depth-first search or breadth-first search.
    a. For a depth-first search, you may enter any valid website url, an optional keyword and a link limit of up to 10.
    b. For a breadth-first search, please enter in www.bomanbo.com as the website url, an optional keyword and any link limit (it will stop on its own once it runs out of valid links to follow).

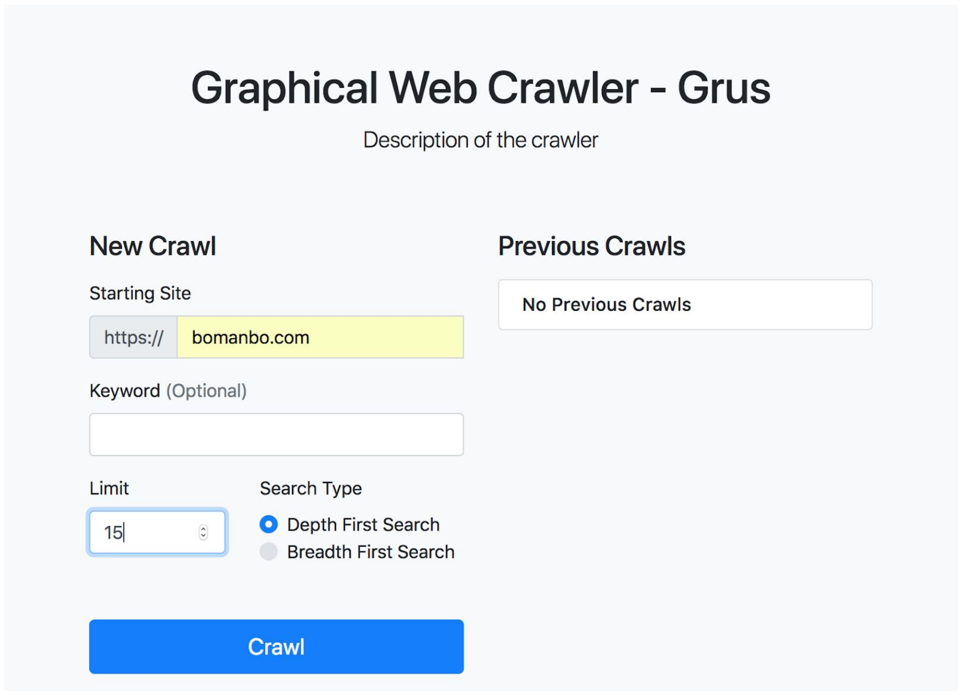
    NOTE: The current link limit has not been set. However, until we have our site set up to work with web sockets, we are currently experiencing limitations due to Heroku's timeout policy. So, we have to set a pretty low limit for the depth. This is also why our breadth-first search is not working on all websites (too vast to process in the time allotted). Please follow the limits as specified.

3. Click 'submit'.
4. Your choices will be posted to the server and the crawl results will be shown graphically when the crawl has finished.

    NOTE: At this point the results are shown all at once. The next step is to have the results update continuously via a web socket.

5. Navigate again to https://graphicalwebcrawler.herokuapp.com/. If you use the back button to do this, your previous searches will not show up unless you hit refresh.
6. The crawl that was just completed will now be shown in the "Previous Crawls" section of the index page.
7. Click on the the url of the previous crawl and the results will be loaded, just as before. This time, they are retrieved from the database and are the exact results, not a re-crawl.

Screenshots


Graphical Web Crawler - Grus

Description of the crawler

New Crawl

Starting Site: https:// bomanbo.com

Keyword (Optional):

Limit: 15

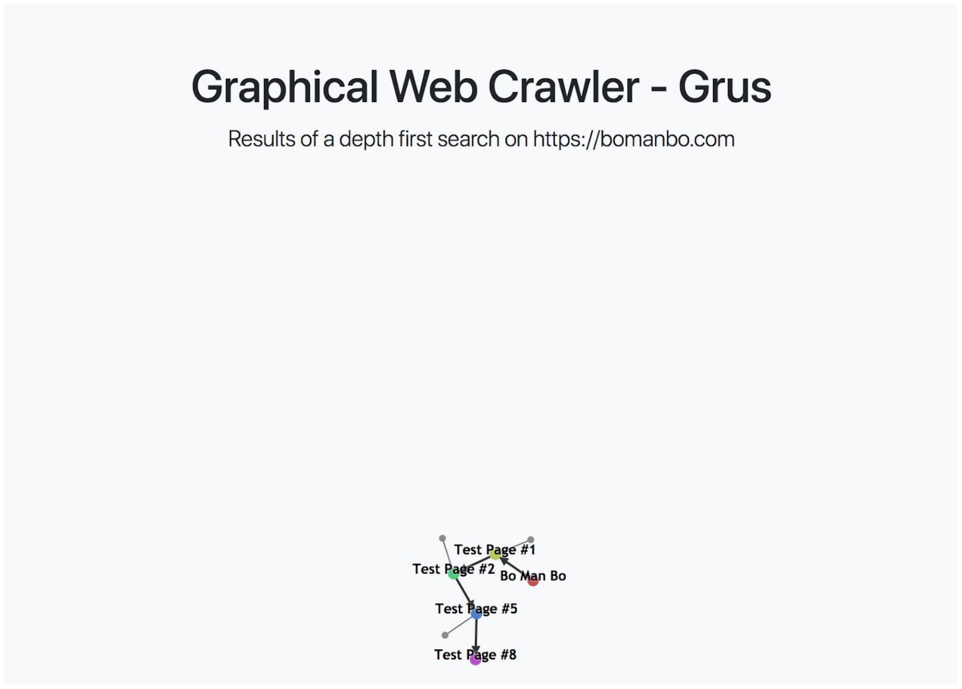
Search Type: [✓ Depth First Search] Breadth First Search

Crawl

Previous Crawls

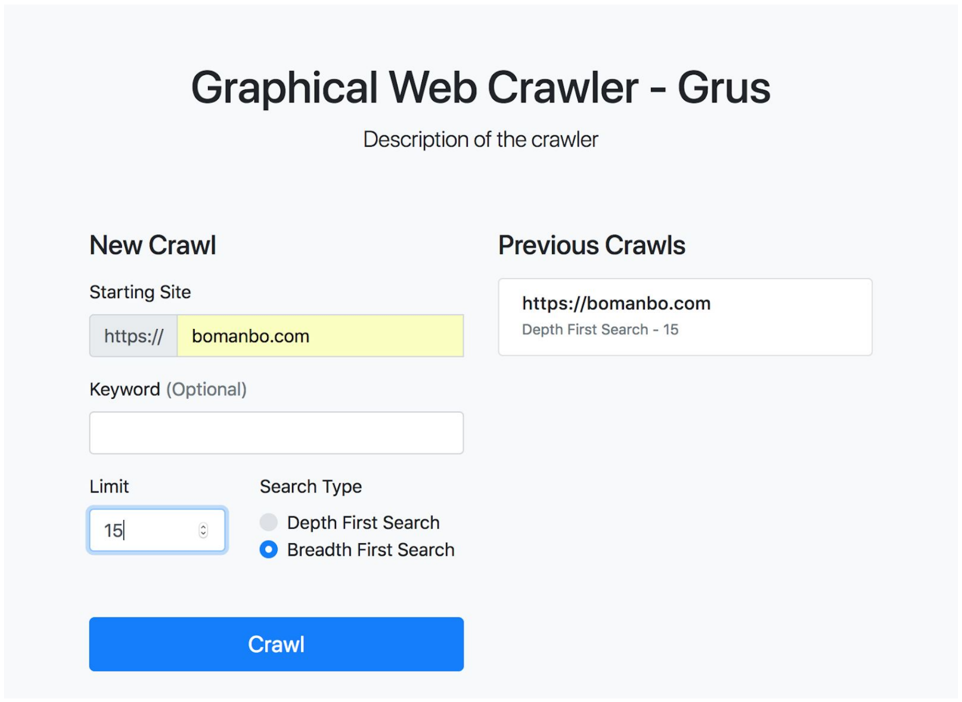
No Previous Crawls

Figure 1. Start of a depth first search on a test website with no keyword and with no previous searches stored on the user cookie.


Graphical Web Crawler - Grus

Results of a depth first search on https://bomanbo.com

Figure 2. Results of a depth first search on a test website with no keyword.


Graphical Web Crawler - Grus

Description of the crawler

New Crawl

Starting Site: https:// bomanbo.com

Keyword (Optional):

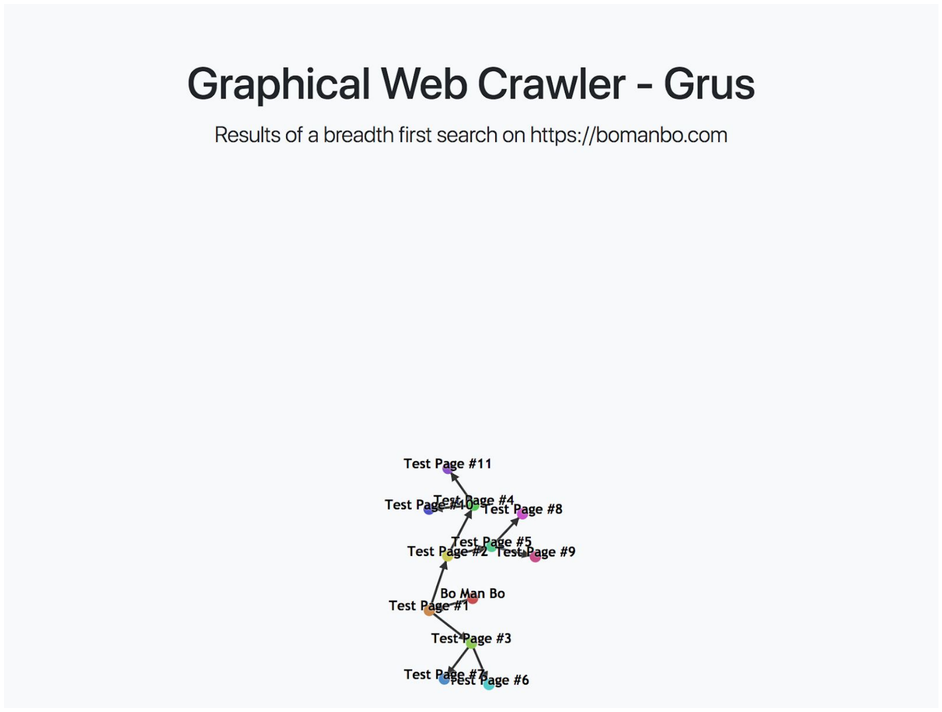
Limit: 15

Search Type: Depth First Search [✓ Breadth First Search]

Previous Crawls

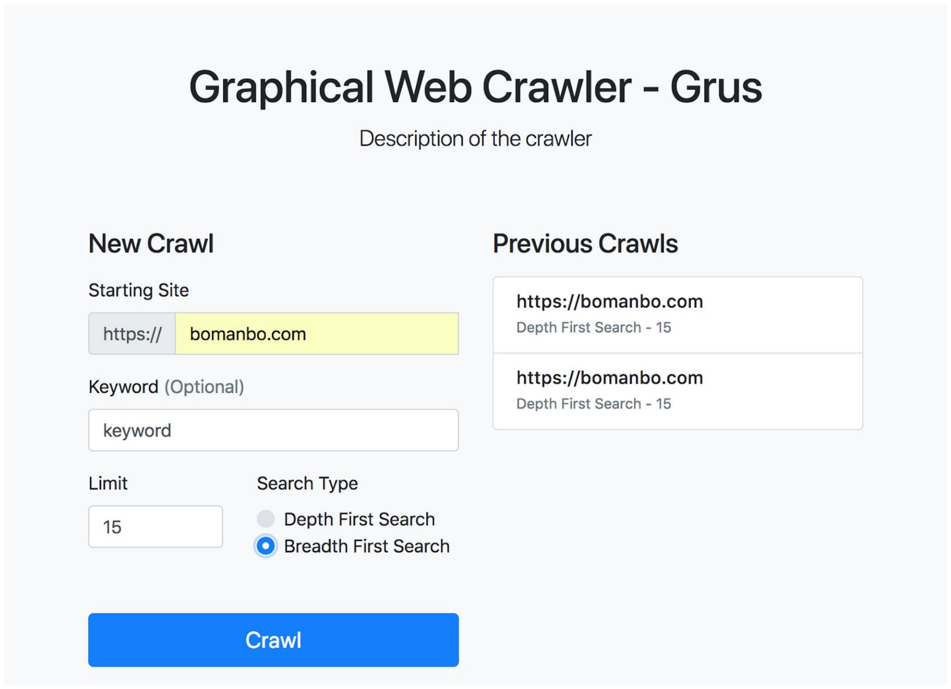
https://bomanbo.com
Depth First Search - 15

Figure 3. Start of a breadth first search on a test website with no keyword and with the previous search stored on the user cookie.


Graphical Web Crawler - Grus

Results of a breadth first search on https://bomanbo.com

Figure 4. Results of a breadth first search on a test website with no keyword.



Graphical Web Crawler - Grus

Description of the crawler

New Crawl

Starting Site: https:// bomanbo.com

Keyword (Optional): keyword

Limit: 15

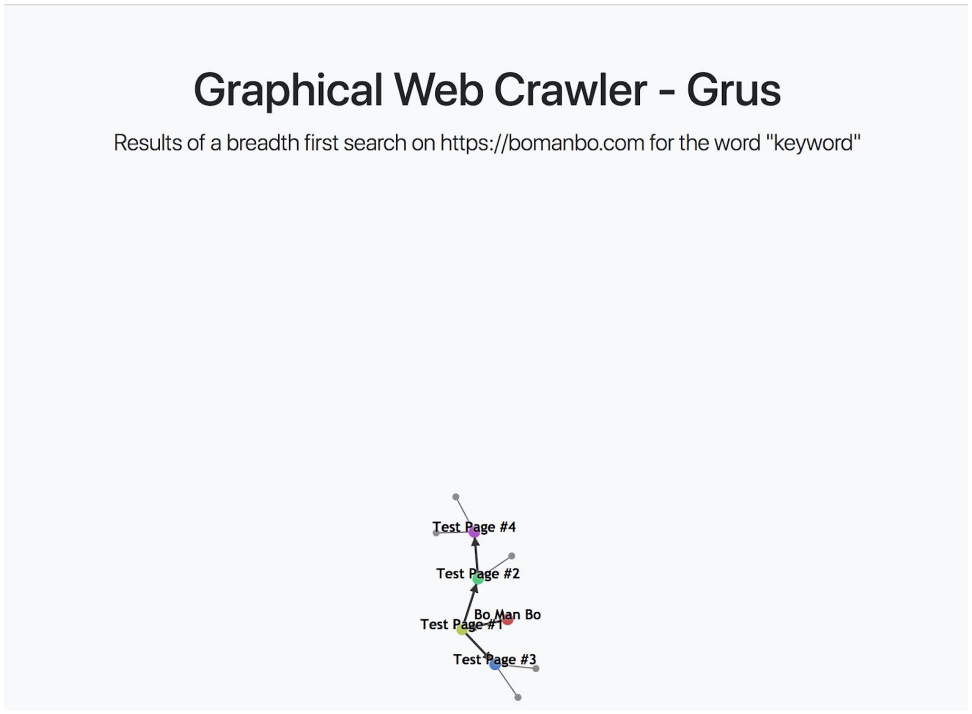
Search Type: Depth First Search [✓ Breadth First Search]

Previous Crawls

https://bomanbo.com
Depth First Search - 15

https://bomanbo.com
Depth First Search - 15

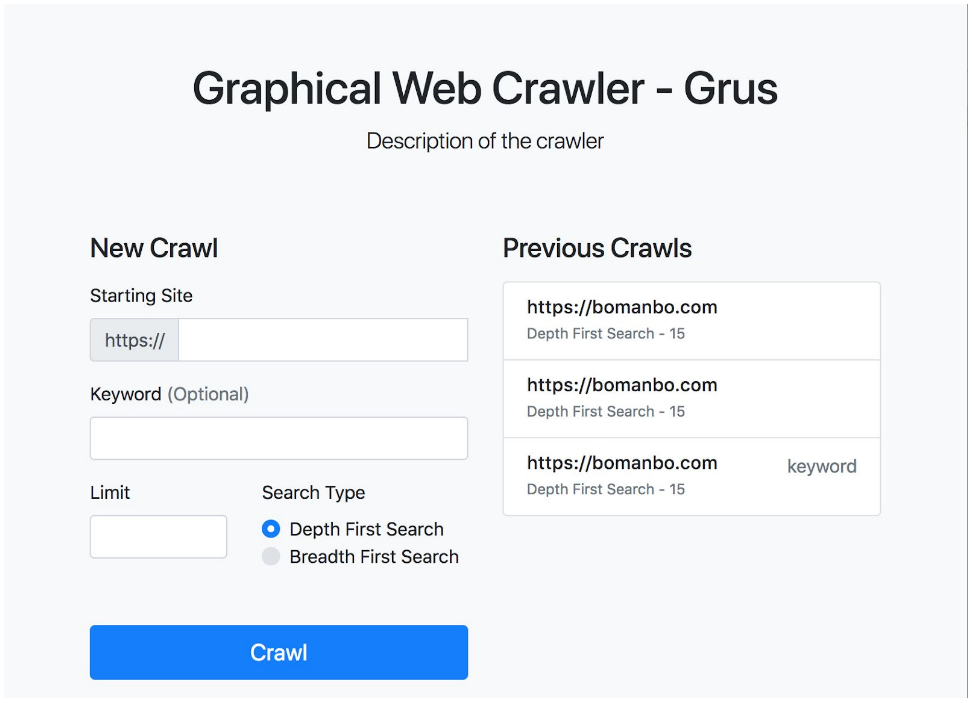
Figure 5. Start of a breadth first search on a test website with a provided keyword and with both previous searches stored on the user cookie.



Graphical Web Crawler - Grus

Results of a breadth first search on https://bomanbo.com for the word "keyword"

Figure 6. Results of a breadth first search on a test website with a provided keyword which is found on Test Page #4. Notice the search stops short this time.



Graphical Web Crawler - Grus

Description of the crawler

New Crawl

Starting Site

https://

Keyword (Optional)

Limit

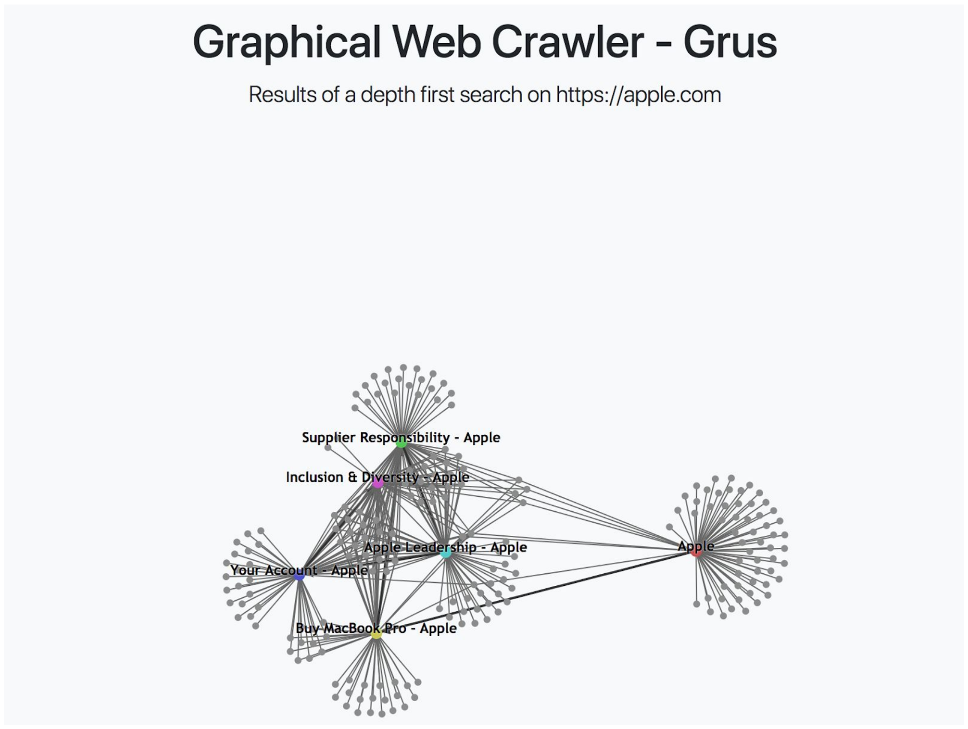
Search Type

[✓ Depth First Search] Breadth First Search

Previous Crawls

https://bomanbo.com Depth First Search - 15	
https://bomanbo.com Depth First Search - 15	
https://bomanbo.com Depth First Search - 15	keyword

Figure 7. All previous searches listed on the main UI page.



Graphical Web Crawler - Grus

Results of a depth first search on https://apple.com

Figure 8. A depth first search on a larger site.

References

https://prateekvjoshi.com/2016/03/08/how-to-create-a-web-server-in-python-using-flask/

http://flask.pocoo.org/

http://www.bogotobogo.com/python/MongoDB_PyMongo/python_MongoDB_RESTAPI_with_Flask.php

https://progblog.io/How-to-deploy-a-Flask-App-to-Heroku/

http://flask-pymongo.readthedocs.io/en/latest/

https://github.com/heroku/heroku-buildpack-google-chrome

https://stackoverflow.com/questions/41059144/running-chromedriver-with-python-selenium-on-heroku

https://pythonspot.com/category/selenium/

https://stackoverflow.com/questions/26745519/converting-dictionary-to-json-in-python

https://docs.python.org/3.1/library/uuid.html

http://www.pythonforbeginners.com/beautifulsoup/beautifulsoup-4-python

https://beautiful-soup-4.readthedocs.io/en/latest/

http://validators.readthedocs.io/en/latest/

http://docs.python-requests.org/en/master/

https://docs.python.org/3/library/re.html