# WikiScrape Project [Benchmarks - Rationale]

## Section 0: Coming up with ideas

### Rationale

Before we even decided to do anything to do with Wikipedia, we thought about different things we could do as a small side-project, and the topic of web scraping came up as a by-product of an idea to automatically select lyrics from songs on Genius.

From this, we decided to make a Wikipedia webscraper, where we would search through a Wikipedia page for all of its links just to see if we could.

To do this, we have used Python 2.7, and an external library BeautifulSoup4 to handle all webscraping.

## Section 1: Retrieving and sorting links

### Rationale

This would be the main part of the whole project, and the one for which we had the most learning to do, so we started with using BeautifulSoup to attempt to harvest every relevant link in a page.

To start off, we needed a page on Wikipedia that was small enough to test practically, but also large enough to have a reasonable number of links, and a number of situations, e.g. having an intro paragraph, a contents table, and further sections. To get this page, we used `Alt+X` until a fitting page was loaded.

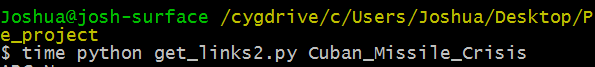
#### Method 1

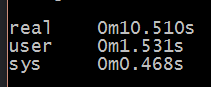
To start with, we created a file `get\_links.py`, which would use BeautifulSoup to read from the Wikipedia page, and output the links to standard output. Then, a bash file `main` would redirect the output into a file `bin.txt`, which then got its duplicate entries removed and then sorted through another bash file, `streamline.sh`. The purose of `main.sh` was to have the whole process automated by a single terminal command: `./main.sh Wiki\_Article\_Name` for ease of use.

#### Method 2

However, we left this direction to try to unify all of the running and processing into a single Python file. Enter `get\_links2.py`. Everything mentioned in the previous paragraph is handled in this file. Howver, rather than having an intermediate output (`bin.txt`), it would store all the links internally in a set (which automatically filters out duplicates), and still outputs to standard output.

### Benchmark 1





Since method 2 is all streamlined into one program, we can test our program’s runtime using *time*. We chose [the Cuban Missile Crisis](https://en.wikipedia.org/wiki/Cuban_Missile_Crisis) because it’s a very long and dense article, and this would be useful for checking how well the program scales with large articles.

For now, we aren’t thinking about optimisation yet, as we cannot identify any avoidable overhead using our current method. In future, we will need to start thinking about optimisation as we begin recursively searching links.

## Section 2: Introduction to Recursion

### Rationale

To further test the webscraping, and to see whether we could gravitate towards a possible implementation of the Wikipedia Game, we made the whole process recursive. That is, for each link, scrape its Wikipedia page for all of its links. To avoid potential unwanted computation, we store a set of all the links that have been visited, and use that to decide whether to pursue a rabbit hole of links. As of now, the recursive function has been depth-limited to two (where the highest depth is zero), to demonstrate that the recursion does indeed function properly. As of now, this recursive function mimics exploring Wikipedia as a depth-first search in lexicographical order, i.e. for (each item in this page, if not visited, get\_links() by alphabetical order).

Right now, there is a somewhat-pretty-printing going on to assist visualisation to ensure that what we are doing is what we want the program to do.

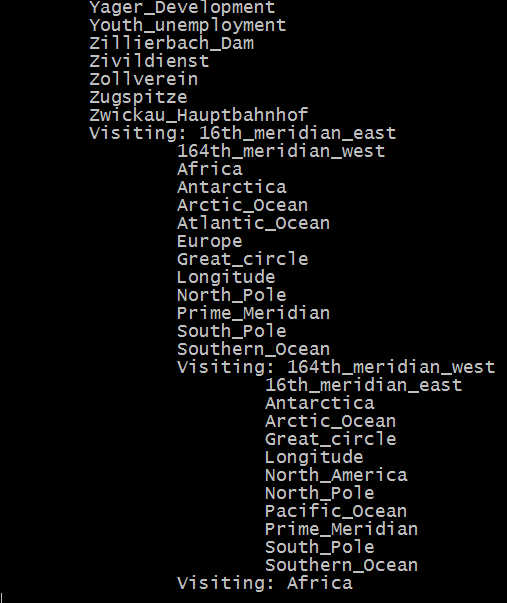
A parameter is used in get\_links() to keep track of the maximum depth, and another to keep track of the current depth. Right now, there are two base cases to the recursion: once the depth hits the limit, or when all outgoing links on the current page have already been visited.

The variable that keeps track of current depth is also used to assist in the somewhat-pretty-print, by indenting that many times, visually representing a directory tree by depth. This, every time the exploration goes further “down”, the indent will increase, and every time it comes back “up”, the ident will decrease.

Currently, format is as such:

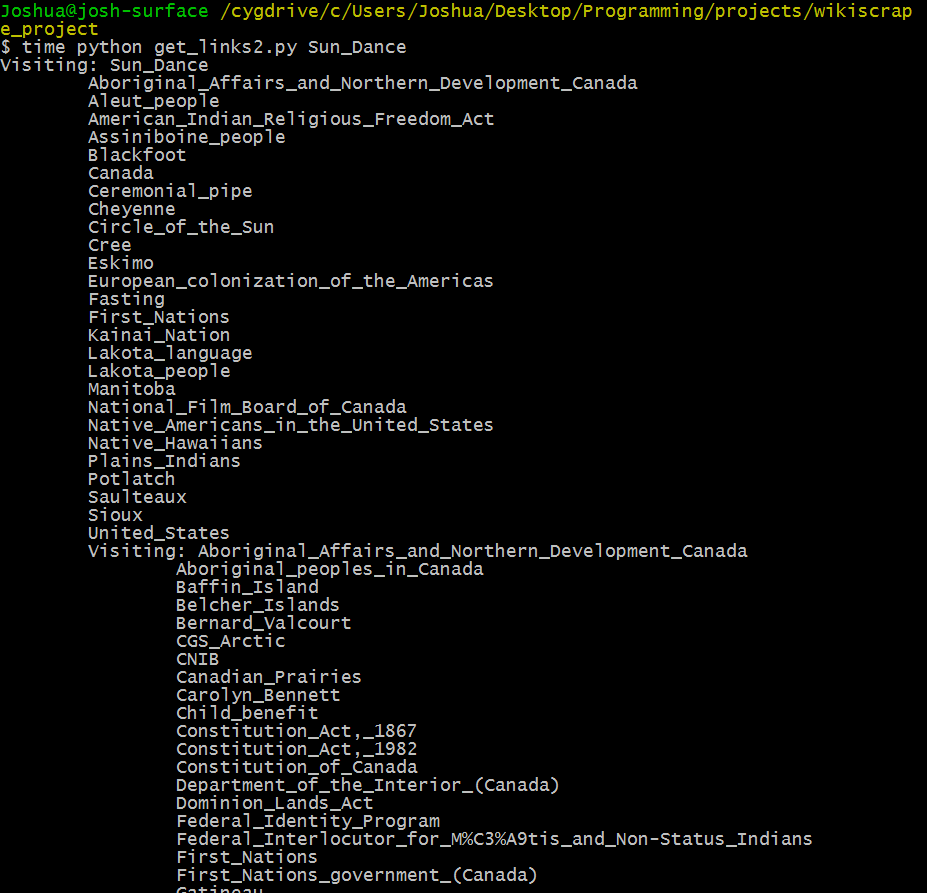
Name of page being searched

{depth indents} List of links from that page.



One issue we are running into is that there are still some useless/irrelevant pages that are being explored by the scraper. So far, we’ve seen a bunch of “Wikipedia:NOTRS”, “Help: Installing Japanese Keyboard” or similar pages. Gots to filter they out.

### Benchmark 1



Above is an example of the formatting.

|  |  |
| --- | --- |
|  |  |
| Depth = 2 | Depth = 3 |

Above are the runtimes with different depths of the [Wikipedia page Sun Dance](https://en.wikipedia.org/wiki/Sun_Dance). This page is relatively small compared to the one we used for benchmark 1, but it’s noticeable that some of the pages that it links to are quite big (e.g. during runtime, we noticed that the page for Canada took significantly longer than some of the others. At this moment, we still have yet to think of optimisations.

## Section 3: Switching to BFS

### Rationale

For the next benchmarks, we are looking towards changing the search methods from DFS to BFS, and adding some objectives to add to the program, for example, starting from one page, search for a path to another page, and additionally, printing the output to text files instead of standard output, in a neat, formatted manner.

The reason we will switch to BFS is due to the nature of the links. The “cost” of moving from a link to another is just one, and BFS will always find the optimal solution (if there is one) for uniform cost movements. This way, if we implement The Wikipedia Search Game, the first solution we come across will be the optimal solution, whereas DFS might find *a solution*, which might not be the best.

To prevent code re-use, we have delegated the webscraping part to a helper function called get\_links(), and renamed the old get\_links() function to dfs. This way, both dfs and bfs can make use of the utility function.

BFS will perform target searching (i.e. confirming exit stage) upon expanding outgoing links. This prevents potentially wasted computation from expanding every link on a level before registering a child node is a target.

### Benchmark 1

## Section 4: Introducing the Strainer

### Rationale

Soup strainer blah blah something optimization

### Benchmark 1

|  |  |
| --- | --- |
| Testing the change in runtime with SoupStrainer, worst time of 3 runs chosen from each test, DFS used with depth=2 and no target page set. Run on Josh’s SP3 on University internet | |
|  |  |
| Depth = 2 with SoupStrainer | Depth = 2 without SoupStrainer |

An additional observation is that the DFS performs significantly faster compared to the test in Section 2, Benchmark 1 (down from ~35 seconds). This is most likely due to fewer programs being run on Josh’s Surface at the time of execution, and also the university’s network being faster than Josh’s home internet. From now on we will log test case conditions as well.

Testing for further optimisations should be more comprehensive – we should look into making automated two-sample t-tests in future.

### Benchmark 2

Fix benchmark 1 later