Implementation of Breadth-First Search Web Crawler and String Matching in Developing a Simple Search Engine Optimization Analyzer

Manasye Shousen Bukit / 13516122
Informatics Undergraduate Program
School of Electrical Engineering and Informatics
Bandung Institute of Technology, Ganesha Street 10 Bandung 40132, Indonesia
13516122@std.stei.itb.ac.id
manasyebukit@gmail.com

***Abstract*—Web crawlers have a long and interesting field in computer science's history. Early, web crawlers collected statistics about websites. In addition to collecting statistics about the web and indexing the applications for search engines, modern crawlers could also be used to perform accessibility and vulnerability checks on the application. Quick expansion of the web, and the complexity added to web applications have made the process of crawling more challenging. Throughout the history of web crawling, many researchers and industrial groups addressed different issues and challenges regarding web crawlers. Different solutions have been proposed to reduce the time and cost of crawling by using many different algorithm. What follows is one of the most popular algorithm to implement web crawler, breadth-first search. In this article, we will also build a simple search engine optimization analyzer using crawled data and string matching.**

Keywords— crawling, algorithm, webpage, SEO

I. INTRODUCTION

A web crawler or also known as a web spider is a program that is capable of iteratively and automatically downloading web pages, extracting URLs from their HTML and fetching them. Many legitimate sites in particular search engines use web crawling to provide up-to-date data to the users. Web crawlers are mainly used to create a copy of all the visited pages for later processing by a search engine, that will index the downloaded pages to provide fast searches. [3]

Crawlers can also be used for automating maintenance tasks on a webpage, such as checking links or validating HTML code. In addition, crawlers could be used to gather specific types of information from web pages, such as harvesting e-mail addresses (usually for spam). Knowing the role of web crawling in modern web development, designing a suitable web crawling is a challenging task. Several criteria that need to be considered when building a web crawler are quality of information taken from the webpage, speed of gathering data without burdening the website's traffic.

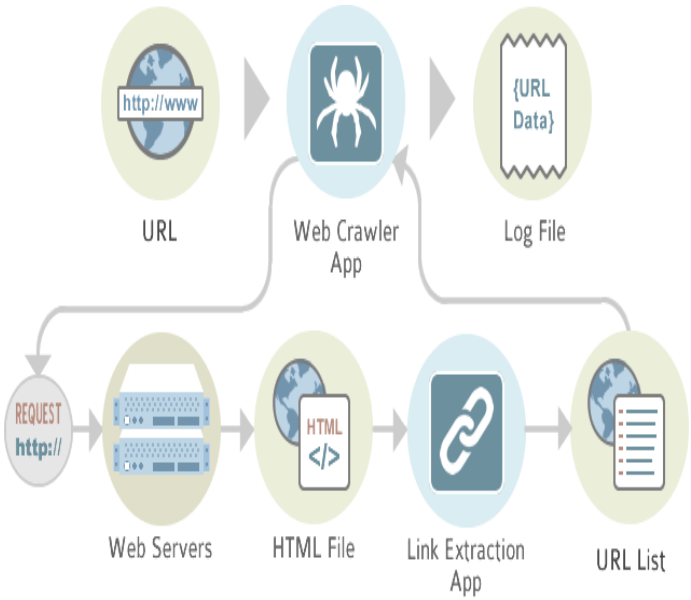

Figure 1. How web crawling works
Source: https://seopressor.com/blog/how-to-control-web-crawlers/

Regarding crawling's speed, algorithm play important role in maximizing number of pages crawled in an amount of time. Therefore, determining the right web crawler's algorithm is an aspect that need to considered. Since HTML is a tree-structured data structure, algorithm's options could be narrowed. One of the most suitable algorithm and easy-to-understand algorithm is breadth-first search.

Search engine use web crawler to gather data from web pages and decide what web page have the most match with search keyword and display it to the users. Even though modern search engine have many other criteria, we could build our own search engine optimization analyzer using data crawled from web pages (in this example all links gathered from a homepage) and string matching to check certain keyword's appearance in this website.

II. BASIC THEORIES

A. *Breadth-First Search (BFS)*

Breadth-first search or shortly BFS is an important graph search algorithm that is used to solve many problems including finding the shortest path in a graph and solving puzzle games. Various problems in computer science can be solved by using data structure graph. For instance, analyzing networks, mapping routes, and scheduling are graph-related problems. Graph search algorithms like breadth-first search are useful for analyzing and solving this graph problems.[1]

Breadth-first search starts by searching a start node, followed by its adjacent nodes, then all nodes that can be reached by a path from the start node containing two edges, three edges, and so on. Formally, the BFS algorithm visits all vertices in a graph **G** that are **k** edges away from the source vertex before visiting any vertex **k + 1** edges away. This is done until no more vertices are reachable from **S**.

For a graph **G = (V, E)** and a source vertex **v**, BFS search traverses the edges of **G** to find all reachable vertices from **v**. It also computes the shortest distance to any reachable vertex if necessary. Any path between two points in a breadth-first search tree corresponds to the shortest path from the root **v** to any other node **S**. By definition, there are three types of vertices in BFS: *tree* vertices, vertices that have been visited; *fringe* vertices, those adjacent to tree vertices but not yet visited; and *undiscovered* vertices, those that we have not been encountered yet.

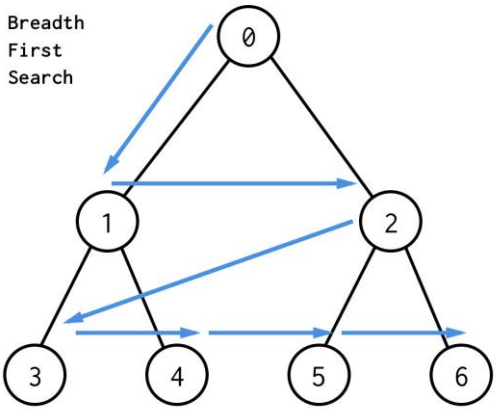

Figure 2. Breadth-First Search Algorithm
Source: http://mishadoff.com/images/dfs/binary_tree_search.png

B. *Depth-First Search (DFS)*

Depth-first Search (DFS) is an algorithm for searching a graph or tree data structure. The algorithm starts at the root (top) node of a tree and goes as far as it can down a given branch (path), and then backtracks until it finds an unexplored path, and then explores it. The algorithm does this until the entire graph has been explored. Many problems in computer science can be thought of in terms of graphs. For example, analyzing networks, mapping routes, scheduling, and finding spanning trees are graph problems. To analyze these problems, graph search algorithms like depth-first search are useful.[2]

Depth-first searches are often used as subroutines in other more complex algorithms. For example, the matching algorithm, Hopcroft–Karp uses a DFS as part of its algorithm to help find a matching in a graph. DFS is also used in tree traversal algorithms, also known as tree searches, which have applications in the the travelling salesman problem and the Ford Fulkerson's algorithm. The main strategy of depth-first search is to explore deeper into the graph whenever possible. Depth-first search explores edges that come out of the most recently discovered vertex, s. Only edges going to unexplored vertices are explored. When all of s's edges have been explored, the search backtracks until it reaches an unexplored neighbor. This process continues until all of the vertices that are reachable from the original source vertex are discovered. If there are any unvisited vertices, depth-first search selects one of them as a new source and repeats the search from that vertex. The algorithm repeats this entire process until it has discovered every vertex. This algorithm is careful not to repeat vertices, so each vertex is explored once.

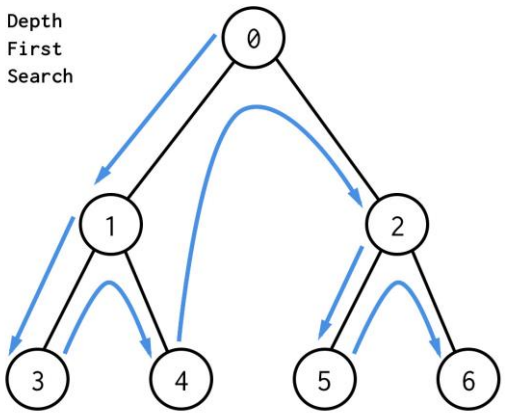

Figure 3. Depth-First Search Algorithm
Source: http://mishadoff.com/images/dfs/binary_tree_search.png

C. *Hypertext Markup Language (HTML)*

HTML is the common markup language for creating and developing a web pages. A markup language is a language used for communication to a web browser about how the contents of a web page will be displayed. HTML is written in the form of "tags" that are surrounded by angle brackets like start tag ***<html>*** and end tag ***</html>***. An HTML file have an **.htm** or **.html** file extension as in index.html , about_us.htm, which can identify that the page is a web page or HTML Documents. [4]

Each HTML document can actually be referred to as a document tree. We describe the elements in the tree like we would describe a family tree. There are ancestors, descendants, parents, children and siblings. It is important to understand the document tree because CSS selectors use the

document tree. Use the sample HTML document below for these examples. The **<head>** section of the document is omitted for brevity. Use the sample HTML document below for these examples. The **<head>** section of the document is omitted for brevity.

```
<body>
  <div id="content">
    <h1>Heading</h1>
    <p>Paragraph 1</p>
    <p>Another <em>Paragraph</em> </p>
    <hr>
  </div>
  <div id="nav">
    <ul>
      <li>item 1</li>
      <li>item 2</li>
      <li>item 3</li>
     </ul>
  </div>
</body>
```

A diagram of the above HTML document tree would look like this.

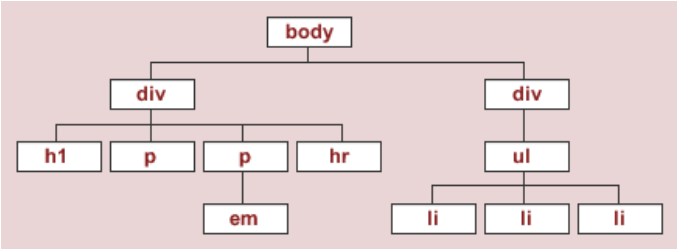

Figure 4. Tree-structure HTML
Source: https://www.w3schools.com/js/pic_htmltree.gif

An ancestor refers to any element that is connected but further up the document tree, no matter how many levels higher. In the diagram above, ***<body>*** element is the ancestor of all other elements on the page. A descendant refers to any element that is connected but lower down the document tree, no matter how many levels lower. In the diagram above, all elements that are connected below the *** *** element are descendants of that *** .***

A parent is an element that is directly above and connected to an element in the document tree. In the diagram above, the *** *** is a parent to the ***<ul>***. A child is an element that is directly below and connected to an element in the document tree. In the diagram above, the ***<ul>*** is a child to the *** ***. A sibling is an element that shares the same parent with another element. In the diagram above, the ***<li>***'s are siblings as they all share the same parent, the ***<ul>***.

D. Search Engine Optimization(SEO)

Search engine optimization or SEO for short is the practice used to increase traffic to website by using search engine results. Search engine like google, yahoo, bing has a web crawler that gather information about all content of web pages on the internet.

After gathering data they needed, search engine have a formula to check how suitable that web pages' content with search keyword user typed. This formula, however vary according to search engine. That is the reason why you could see the same keyword input but search result vary in different search engine. However, we can establish a simple formula to calculate the value of a web page.

Tag	Value
h1	10
h2	5
h3	2
p	1

Table 1. Tag value in our SEO analyzer
Source: author's documentation

Understanding how search engine works could help website developer to gain advantages in term of business. A website that tend to be at top of the search result have a bigger chance to get visited, and therefore optimizing website's business.

III. IMPLEMENTATION

A. Breadth-First Search Web Crawler

The first thing that we need to do is search for a web page that wanted to be crawled and search keyword within the website. To simplify, we choose one website that we would like to gather links and heading from that website and crawl that links again until no more links to be crawled. Since, the links could be in a huge number, it's better to save links that already crawled and want to be crawled in a text file because we could resume the crawling anytime we want.

Before implementing the BFS, we need to make a LinkFinder class that handle what information we want crawl. In our own web crawler, we only care if the tag is an anchor tag or a heading tag, so we can later process this information.

```
# When we call HTMLParser feed() this function is called
def handle_starttag(self, tag, attrs):

    # Search other link
    if tag == 'a':
        for (attribute, value) in attrs:
            if attribute == 'href':
                url = parse.urljoin(self.base_url,value)
                self.links.add(url)

    # Gather all heading tag for SEO purposes
    if tag == 'h1' or tag == 'h2'or tag == 'h3':
        for (attribute, value) in attrs:
            data = '' + tag + ' ||| ' + value
            if attribute == 'title':
                if not os.path.exists('heading.txt'):
                   write_file('heading.txt', data)
                else:
                   append_to_file('heading.txt', data)
```

If the tag is a heading (in our example we only count h1, h2, and h3), we append it directly to a file heading.txt . If we encounter an anchor tag , we add it to a set data structure named links because with set it will automatically be unique and we just want to crawl an url once.

After we finish implementing LinkFinder, we create Spider class. This class is responsible for gathering links and heading by initiating LinkFinder and also added links to the set of crawled and queue. Spider have a method called gather_link that is responsible for requesting a website url and decode it to human-readable string.

```
# Converts raw response data into readable information
and checks for proper html formatting
@staticmethod
def gather_links(page_url):

    html_string = ''
    try:
        response = urlopen(page_url)
        header = response.getheader('Content-Type')
        if 'text/html' in header:
            html_bytes = response.read()
            html_string = html_bytes.decode("utf-8")
        finder = LinkFinder(Spider.base_url, page_url)
        finder.feed(html_string)

    except Exception as e:
        print(str(e))
        return set()

    return finder.page_links()
```

In Spider class, we also implementing breadth-first search in crawling and adding links to crawled.txt and queue.txt. Once we have all the links gathered from a web page, we first check whether that page already been crawled (we know this by searching in crawled.txt). If it's not yet been crawled, append it to last line of queue.txt. We could already see this crawling web page is using breadth-first search algorithm. The next link to crawled is taken from the head of the queue.

```
# Saves queue data to project files
@staticmethod
def add_links_to_queue(links):

   for url in links:
      if(url in Spider.queue) or (url in Spider.crawled):
          continue

      if Spider.domain_name != get_domain_name(url):
          continue

      Spider.queue.add(url)
```

We also limit links that want to be crawled if it has the same domain name as first web page crawled. We do this because we could encounter social media links, advertisement links, and other links that is not related to that website. The last thing we need is main file that generate first spider ,crawl that url. In this file, we also could implement multi-threading to our web crawler if we want to since all resources and static and could be accessed by many spiders. This could speed up the crawling, but we couldn't see exactly how the BFS works, but more like parallel searching. So for our program, we set number of threads to be 1.

```
import threading
from queue import Queue
from spider import Spider
from domain import *
from general import *

PROJECT_NAME = 'Links'
HOMEPAGE = 'http://www.goal.com/id'
DOMAIN_NAME = get_domain_name(HOMEPAGE)
QUEUE_FILE = PROJECT_NAME + '/queue.txt'
CRAWLED_FILE = PROJECT_NAME + '/crawled.txt'
NUMBER_OF_THREADS = 1
queue = Queue()
Spider(PROJECT_NAME, HOMEPAGE, DOMAIN_NAME)

# Create worker threads (will die when main exits)
def create_workers():
    for _ in range(NUMBER_OF_THREADS):
        t = threading.Thread(target = work)
        t.daemon = True
        t.start()

# Do the next job in the queue
def work():
    while True:
        url = queue.get()
        Spider.crawl_page(threading.current_thread().
        name, url)
        queue.task_done()

# Each queued link is a new job
def create_jobs():
    for link in file_to_set(QUEUE_FILE):
        queue.put(link)
    queue.join()
    crawl()

# Check if there are items in the queue
def crawl():
    queued_links = file_to_set(QUEUE_FILE)
    if len(queued_links) > 0:
        print(str(len(queued_links))
        + ' links in the queue')
        create_jobs()

create_workers()
crawl()
```

If we run main.py, it will start gathering all links and heading until no links is there to be found.

Figure 5. The process of the crawling
Source: author's documentation

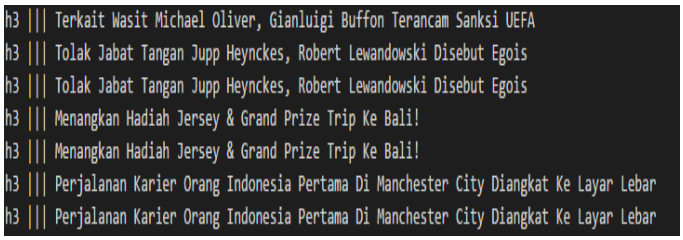

Figure 6. Header.txt content's example
Source: author's documentation

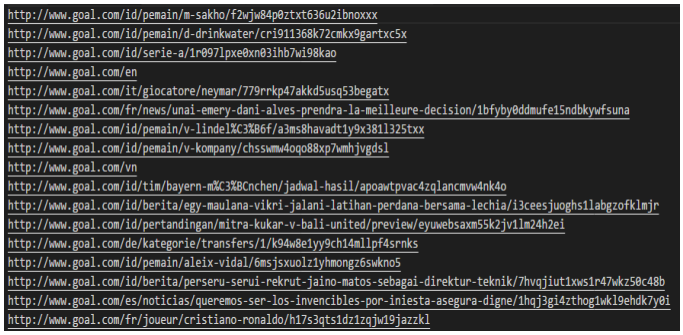

Figure 7. Links' example in crawled.txt
Source: author's documentation

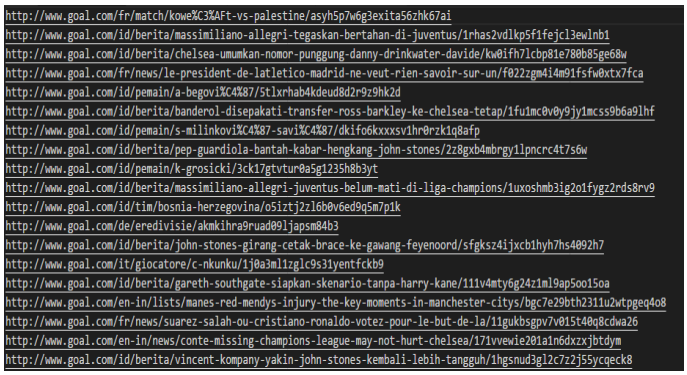

Figure 8. Links' example in queue.txt
Source: author's documentation

B. String Matching and SEO Analyzer

After gathering heading data in the crawled page, we could implement string matching to search given keyword. String matching algorithm may vary, we could use bruteforce, Knuth-Morris-Pratt, Boyer-Moore or regular expression. We use regular expression because by regular expression we could make our own pattern without having the boundary of exact matching like in Knuth-Morris-Pratt, Boyer-Moore, or bruteforce.

```
import re

def generate_pattern(pat):
    # Split each word when encounter space
    pat_split = (pat.lower()).split(" ")
    pattern = ""

    # Add all between words
    for i in range(len(pat_split)):
        pattern += (".*" + pat_split[i])
    pattern += ".*"
```

```
    # Compile pattern into regex pattern
    regex = re.compile(pattern)

    return regex
```

In generating pattern, we first lower case the keyword because we want the matching to be case-insensitive. After that, we split keyword when encounter space and add **.*** in between the splitted keyword. For instance, string "i buy" will find match in text "Therefore, I want to buy".

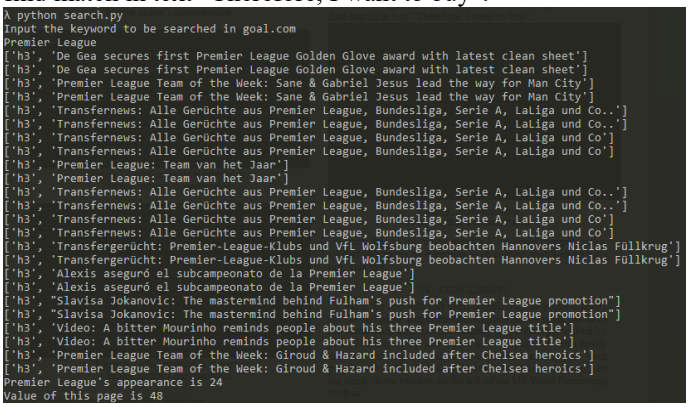

Figure 8. String matching
Source: author's documentation

According to our value in table 1, we could compute value of that web pages by summing each content matched with it's tag. This tag's value define how important this heading in search engine. Even though the formidable search engine's formula is more complex, we could use this program to check how "valueable" a web page is when search with a certain search keyword.

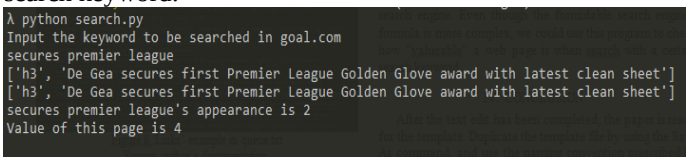

Figure 9. Non-exact string matching and page value
Source: author's documentation

IV. CONCLUSION

Web crawling is useful to have a better understanding in analyzing a web page regarding search engine optimization value. An algorithm for web crawling and string matching is discussed and implemented in this paper. However, the current implementation leaves much room for implementation, both to to the quality of the result and its efficiency.

V. APPENDIX

The author's implementation of the algorithm discussed in this paper and the sample data crawled can be accessed on Github (https://github.com/manasye/SEO_Analyzer) . It is written in python and require a html.parser, urllib, and re module to run.

ACKNOWLEDGMENT

The author thanks Dr. Nur Ulfa Maulidevi, S.T., M.Sc and Dr. Rinaldi Munir,M.T greatly as the instructors of IF2211 for giving the opportunity to write this paper. The author also thanks the author's parents for providing moral support throughout the writing process. The author thanks the authors of the references below, and numerous others, for their contributions to computer science and for giving insights to the author.

REFERENCES

[1] https://brilliant.org/wiki/breadth-first-search-bfs/ (Retrieved May 4, 2018, 16:54)

[2] https://brilliant.org/wiki/depth-first-search-dfs/ (Retrieved May 6, 2018, 16:29)

[3] https://www.sciencedaily.com/terms/web_crawler.htm (Retrieved May 9. 2018, 20:40)

[4] http://selfstudyathome.blogspot.co.id/2011/10/html-tutorial-theory-part-1.html (Retrieved May 9, 2018, 21:59)

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 12 Mei 2018

Manasye Shousen Bukit / 13516122