Assignment 1

CS834-F16: Introduction to Information Retrieval

Fall 2016

Erika Siregar

CS Department - Old Dominion University

September 22, 2016

Question 1.4

```
List five web services or sites that you use that appear to use search, not
including web search engines. Describe the role of search for that service.
Also describe whether the search is based on a database or grep style of matching,
or if the search is using some type of ranking.
```

Answer

Five web services that I use that appear to use search:

1. `https://norfolk.craigslist.org/`
   - Craiglist is a classified advertisements website with sections devoted to jobs, housing, personals, for sale, items wanted, services, community, gigs, résumés, and discussion forums [1]. User can type any query (something like 'apartment near ODU' or 'women bike') and Craiglist will return the search results ranked by their relevance or prices.
2. `https://www.amazon.com/`
   - Amazon is the world's largest online retailer. It works in a similar way to Craiglist. User type a query of the thing that they want to buy online such as 'waterproof women shoes'. Amazon responds to the query by showing the list of 'waterproof women shoes' ranked by their relevance or prices.
3. `https://www.usps.com/`
   - The search role of this service is to track a package. A user can enter the tracking number and the system will return the current status and location of the package.
   - The search service that this system provides is clearly based on a database or grep style of matching. The system will take the number input by the user and find a matching record in the database.
4. `http://www.urbandictionary.com/`
   - The search role in `http://www.urbandictionary.com/` is to find definition of english slang words. The search results are ranked based on the number of votes that each definition has. The definition with the most votes will show up on the first order.
5. `https://www.kayak.com/`
   - `https://www.kayak.com/` is a travel site that aggregates all information from other travel sites and provides a recommendation for buying flight tickets. User can type the flight route and date and Kayak will give a suggestion about which ticket that user should buy. The search results are ranked based on the price, duration, and the number of stops.

Question 3.6

```
How would you design a system to automatically enter data into web forms in order
to crawl deep Web pages? What measures would you use to make sure your crawler's
actions were not destructive (for instance, so that it does not add random blog
comments).
```

Answer:

Web forms is one category of the deep web, which can only be accessed by filling out web forms. We cannot obtain the data if there is no human interaction with the web forms. In this case, the challenge is *"forms are designed to be handled by human, so how can we automate it?"*.
The simplest solution will be submitting the combination of all possible field values in cartesian product. However, this solution is not feasible when the number of fields and possible values are large. Thus, the goal is not to find all possible values, but to select a subset of values so as to minimize the number of submissions and maximize the coverage (retrieve more distinct data behind the form) [2]. Kantorski [3] conducted a survey to compare 15 approaches for handling automatic web form filling. All these methods have a common thread regarding how they address the problem of web form filling. They always have to deal with:

1. form detection: determine the field and domain of the form.

2. initial value generation: select words from the initial page, where the form is located

3. value generation: iteratively submitting queries using values obtained in previous iterations.

One method that is really interesting for me is the one proposed by Barbosa and Freire [4] about using keywords to siphon the hidden data. Instead of blindly issuing queries, they proposed a sampling-based approach to discover words that result in high coverage. The intuition is that words in the actual database or document collection are more likely to result in higher coverage than randomly selected words. Barbosa and Freire [4] come up with the idea of using sampling and probing to generate high-coverage query. Sampling is used to find high-frequency keywords (the candidate keywords). These candidate keywords will be combined into a query and used in site probing to determine the query cardinality. Queries with low cardinality are deemed not effective and removed from the database. Figure 1 summarize the algorithm.

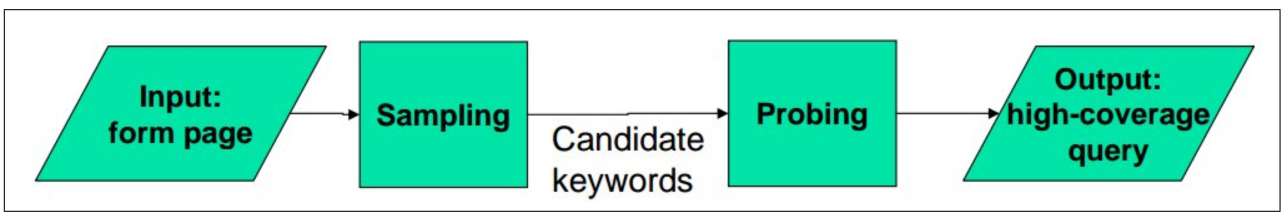

Figure 1: Sampling and probing to generate high-coverage query. Adapted from `https://vgc.poly.edu/~juliana/pub/sbbd2004-talk.pdf`

Furthermore, figure 2 illustrates the processes that happen in 'sampling'. Basically, sampling is a process to turn the input 'terms in the form page' into the output 'candidate (high-frequency) keywords'. Candidate keywords are produced by iteratively submitting queries using values obtained in previous iterations based on term frequency.

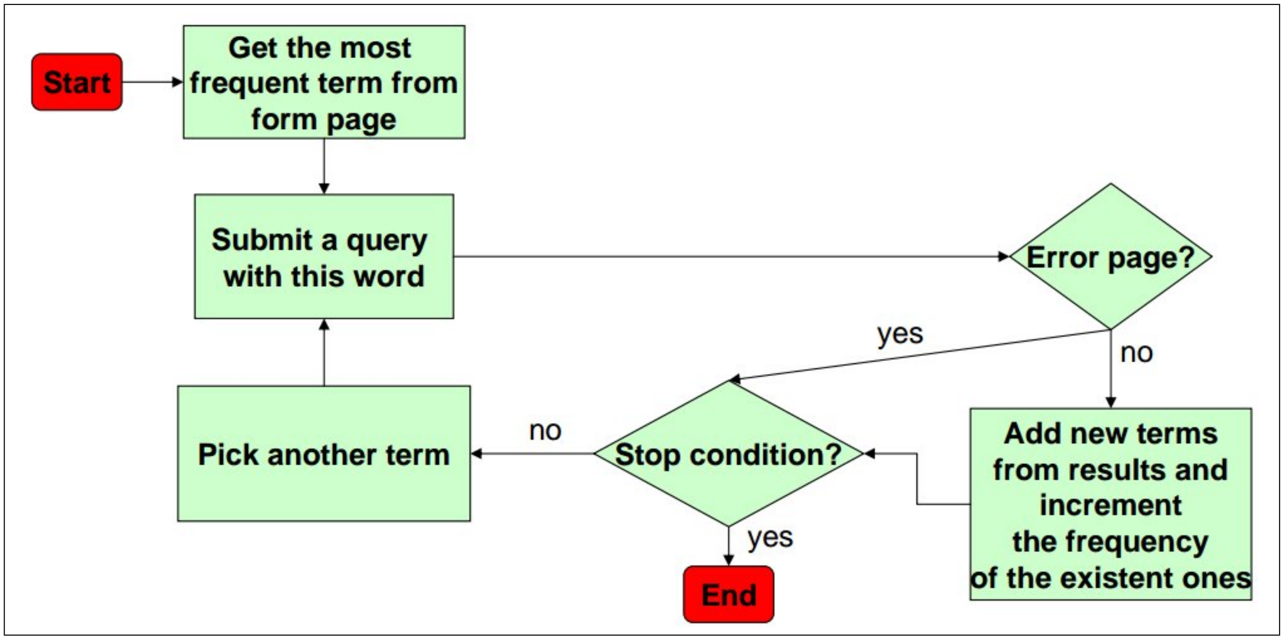

Figure 2: Sampling algorithm for building candidate keywords. Adapted from `https://vgc.poly.edu/~juliana/pub/sbbd2004-talk.pdf`

The response page for a query may contain information that is not relevant to the actual query (e.g. ads, navigation bars) that is not part of the site. This information may negatively impact the keyword selection process. Hence, it must be removed before selecting the candidate words. We also have to implement additional techniques to ensure that our design is not destructive such as removing stop words and stemming.

Question 3.9

```
Write a simple single-threaded web crawler. Starting from a single input URL
(perhaps a professor's web page), the crawler should download a page and then
wait at least five seconds before downloading the next page. Your program should
find other pages to crawl by parsing link tags found in previously crawled
documents.
```

Answer

Web crawler is a program that find and download web pages automatically. A web crawler is designed to follow the links on web pages to discover and download new pages. This program runs recursively, meaning that it will repeat itself everytime it download a new page. This process continues until the crawler either runs out of disk space to store pages or runs out of useful links to add to the request queue [5]. For the purpose of this assignment only, the user will be asked to enter the maximum number of recursive calls for the crawling process. To create the web crawler program, I modified the program that I made in web science class about extracting 1000 links from Twitter. Therefore, I once again utilize useful python libraries named Beautiful Soup [6] and Requests [7].

The web crawler program can be summarize into this algorithm:

1. Capture the keyboard input. User will be prompted to enter :
   - The initial URI (the seed).
   - The number of recursive calls (the level) that they want to make.
   - The path to output folder
2. Strip the seed URI.
3. Crawl the seed URI (follow the link and download the page) using library ‘Requests’.
4. If the URI’s status code is 200:
   - Parse HTML with beautiful soup.
   - Find all <a> tags.
   - Get href attribute from tag <a>.
   - Sometimes url in tag <a> is not a full url (e.g: ...). So, we have to add the schema and host to create an absolute URI.
   - Add these new found URIs to the frontier.
5. While our current level is less than depth (level < depth), then repeat the crawling process again using the URIs in the frontier.
6. Wait for 5 seconds before downloading the next page.
7. The outputs of this program are the downloaded html page and the list of extracted links.

Listing 1 shows the complete code for the web crawler.

```
#!/usr/bin/python

import hashlib
import os
from urlparse import urljoin

import errno
import requests as requests
import time
from bs4 import BeautifulSoup
from requests.exceptions import InvalidSchema


def open_url(level, idx, url, outdir):
    url = url.strip()

    # Hashing URL as a output file
    outfile = os.path.join(outdir, hashlib.md5(url).hexdigest() + '.html')
    listfile = os.path.join(outdir, 'urls.csv')

    # Crawl URL with requests, methods GET
    print('Debug : {}.{} Opening URL {}'.format(level+1, idx+1, url))

    try:
        resp = requests.get(url)

        # Process only if status code 200
```

```
        if resp.status_code == 200:
            print('Debug : {}.{} URL {} is opened'.format(level+1, idx+1, url))

            html = resp.text

            # Save html to outfile
            with open(outfile, 'w') as of:
                of.write(html.encode('utf-8'))
                print('Debug : {} is saved into {}'.format(url, outfile))

            # Save url to list
            with open(listfile, 'a') as f:
                f.write(url + '\n')

            # Parse HTML with beautiful soup
            soup = BeautifulSoup(html, 'html.parser')
            # Find all anchors, <a> tag
            links = soup.find_all('a')

            # Get href attribute from tag <a>
            hrefs = []
            for link in links:
                href = link.get('href')

                # Make url absolute
                # Sometimes url in tag <a> is not a full url, without schema and host
                # e.g: <a href="/folder/a.html">...</a>
                href = urljoin(url, href)

                hrefs.append(href)

            return hrefs
        else:
            print('Debug : {}.{} Cannot open URL {}, Status code: {}'.format(level+1, idx+1, url,
                                                                              resp.status_code))
    except:
        pass


# Capture input from keyboard
url = raw_input("Enter a URL: ")

depth = raw_input("Enter crawl depth or level: ")
depth = int(depth)

outdir = raw_input("Enter output directory: ")
# Make directory if not exists
try:
    os.makedirs(outdir)
except OSError as exc:
    if exc.errno != errno.EEXIST:
        raise

links = [url, ]

level = -1
while level < depth:
```

```python
    # Prepare childre_links to save newly founded links
    children_links = []
    for idx, link in enumerate(links):
        # open_url will return new list of links founded in link
        new_links = open_url(level, idx, link, outdir)
        if new_links:
            print('Debug : Found {} links'.format(len(new_links)))
            children_links = children_links + new_links

        # Sleep for 5 seconds
        print('Debug : Sleep for 5 seconds')
        time.sleep(5)

    # After all links are processed, set links with children_links and increase level
    links = children_links
    level += 1
```

Listing 1: Simple single-threaded web crawler

Figure 3 shows the crawling process that happens when we run the crawler using `http://www.cs.odu.edu/~mln/` as the seed and the number of recursive calls = 1.

```
/usr/bin/python2.7 "/media/erikaris/DATA/ODU/Semester 3/intro_to_info_retrieval/assignments/a1/code_report/3.9_single-threaded-crawler.py"
Enter a URL: http://www.cs.odu.edu/~mln/
Enter crawl depth or level: 1
Enter output directory: /media/erikaris/DATA/ODU/Semester 3/intro_to_info_retrieval/assignments/a1/code_report/crawling_output
Debug : 0.1 Opening URL http://www.cs.odu.edu/~mln/
Debug : 0.1 URL http://www.cs.odu.edu/~mln/ is opened
Debug : http://www.cs.odu.edu/~mln/ is saved into /media/erikaris/DATA/ODU/Semester 3/intro_to_info_retrieval/assignments/a1/code_report/c
Debug : Found 26 links
Debug : Sleep for 5 seconds
Debug : 1.1 Opening URL http://www.cs.odu.edu/~mln/
Debug : 1.1 URL http://www.cs.odu.edu/~mln/ is opened
Debug : http://www.cs.odu.edu/~mln/ is saved into /media/erikaris/DATA/ODU/Semester 3/intro_to_info_retrieval/assignments/a1/code_report/c
Debug : Found 26 links
Debug : Sleep for 5 seconds
Debug : 1.2 Opening URL http://www.cs.odu.edu/
Debug : 1.2 URL http://www.cs.odu.edu/ is opened
Debug : http://www.cs.odu.edu/ is saved into /media/erikaris/DATA/ODU/Semester 3/intro_to_info_retrieval/assignments/a1/code_report/crawli
Debug : Found 90 links
Debug : Sleep for 5 seconds
Debug : 1.3 Opening URL http://www.odu.edu
Debug : 1.3 URL http://www.odu.edu is opened
Debug : http://www.odu.edu is saved into /media/erikaris/DATA/ODU/Semester 3/intro_to_info_retrieval/assignments/a1/code_report/crawling_o
Debug : Found 117 links
Debug : Sleep for 5 seconds
Debug : 1.4 Opening URL http://www.cs.odu.edu/~mln/
Debug : 1.4 URL http://www.cs.odu.edu/~mln/ is opened
Debug : http://www.cs.odu.edu/~mln/ is saved into /media/erikaris/DATA/ODU/Semester 3/intro_to_info_retrieval/assignments/a1/code_report/c
Debug : Found 26 links
Debug : Sleep for 5 seconds
```

Figure 3: The crawling process

All outputs resulted from the crawling process (including the downloaded html pages and the list of extracted links) are uploaded to github under directory assignments/a1/crawling_output.

Question 3.12

`Design a compression algorithm that compresses HTML tags. Your algorithm should detect tags in an HTML file and replace them with a code of your own design that is smaller than the tag itself. Write an encoder and decoder program.`

Answer

To solve this problem, the first thing that we have to do is to identify all tags in the input HTML file. The tags in HTML file can be distinguished into 2 categories: 'start_tag' and 'end_tag'. We can take advantage of HTMLParser [8], a python library for parsing text files formatted in HTML (HyperText Mark-up Language) and XHTML. HTMLParser provides complete functions to handle start_tag, end_tag, and data. For this assignment, I will create two programs: the encoder and the decoder.
Here are the general algorithm to tackle this problem. We start with creating the encoder program:

1. Create a class *HTMLEncoder* which is an extension of *HTMLParser*.
2. Scan the input HTML file.
3. Parse the HTML file and identify all tags (start_tag and end_tag) and data in the HTML file. We do this using 'feed' function from HTMLParser.
4. Store the list of the tags into an array.
5. Compress the tags by replacing them with character smaller than the tag itself. Use the character that represents the tag index in the array. For example:
   - if index = 0, then char = a.
   - if index = 1, then char = b.
     etc.
6. Concatenate the new compressed tags and the data and store them in a string named 'minified_html'.
7. Create a HTML comment contains the mapping relation between the original tags and the compressed tags. Append this comment to the string 'minified_html'.
8. Write down the 'minified_html' into a file.

For the decoder program, we basically do the similar things as we did in the encoder program, but in reverse order:

1. Read the comment in 'minified_html' that contains the mapping between the compressed tags and the original tags.
2. Parsing the compressed HTML file to identify all compressed tags in that HTML file.
3. Replace the compressed tags with the original tags.

Figure 4 shows the terminal output for the 'Compress HTML Tags' program.

```
/usr/bin/python2.7 "/media/erikaris/DATA/ODU/Semester 3/intro_to_info_retrieval/a
Enter a URL: http://www.cs.odu.edu/
Enter output file for encoded html: csodu_encode.html
Enter output file for decoded html: csodu_decode.html

Process finished with exit code 0
```

Figure 4: Terminal output for the 'Compress HTML tags' program

The complete encoder and decoder code for HTML-tag compressing can be seen in listing 2 and listing 3, respectively.

```python

import json
import string
from HTMLParser import HTMLParser

import htmlmin as htmlmin
import requests


# HTMLEncoder is extent of HTMLParser
class HTMLEncoder(HTMLParser):
    _tags = []

    def handle_starttag(self, tag, attrs):
        # Append to array, with type 'starttag'
        self._tags.append(((tag, attrs), 'starttag'))

    def handle_endtag(self, tag):
        # Append to array, with type 'endtag'
        self._tags.append((tag, 'endtag'))

    def handle_data(self, data):
        # Append to array, with type 'data'
        self._tags.append((data, 'data'))

    def make_chars(self, number):
        num_char = int(number/len(string.lowercase))
        rem = number - (num_char * len(string.lowercase))

        chars = ''.join([string.lowercase[len(string.lowercase)-1] for i in range(0, num_char)]) + \
                string.lowercase[rem]

        return chars

    def encode(self, html):
        # Process normal html with HTMLParser
        self.feed(html)
        self.close()

        # After parsing is done, process array _tags
        tag_list = []
        minified_html = ''
        for data, type in self._tags:
            if type == 'starttag':
                (tag, attrs) = data

                if not tag in tag_list:
                    tag_list.append(tag)

                # Process attrs of tag, e.g:
                # [('href', 'http://...'), ('title', 'Some link')] become
                # <a href='http://...' title='Some link'>
                str_attrs = ' '.join(['{}="{}"'.format(name, val) for name, val in attrs])

```

```
                # Append encoded tag and it's attrs into var html
                encoded_tag = self.make_chars(tag_list.index(tag))
                minified_html += '<{}{}>'.format(
                    encoded_tag, (' ' if str_attrs else '') + str_attrs
                )
            elif type == 'endtag':
                # Append encoded end-tag into var html
                encoded_tag = self.make_chars(tag_list.index(tag))
                minified_html += '</{}>'.format(encoded_tag)
            elif type == 'data':
                # Append data into var html
                minified_html += data

        # Process json of definition as a comment
        definitions = '<!--{}-->'.format(
            json.dumps({ self.make_chars(key): val for key, val in enumerate(tag_list) })
        )

        # Return definition and minified html
        return definitions + minified_html
```

Listing 2: Encoder for HTML-tag Compressing

```

decoder# HTMLDecoder is extent of HTMLParser
class HTMLDecoder(HTMLParser):
    _tag_map = {}
    _tags = []

    def handle_comment(self, data):
        # Comment contains mapper of html tags
        original_tag_map = json.loads(data)

        # There are more than one comments
        # Process only comment type json
        if type(original_tag_map) == dict:
            self._tag_map = original_tag_map

    def handle_starttag(self, tag, attrs):
        # Append to array, with type 'starttag'
        self._tags.append(((tag, attrs), 'starttag'))

    def handle_endtag(self, tag):
        # Append to array, with type 'endtag'
        self._tags.append((tag, 'endtag'))

    def handle_data(self, data):
        # Append to array, with type 'data'
        self._tags.append((data, 'data'))

    def decode(self, enc_html):
        # Process normal html with HTMLParser
        self.feed(enc_html)
        self.close()

        # After parsing is done, process array _tags
        html = ''
        for data, type in self._tags:
            if type == 'starttag':
```

```
                (tag, attrs) = data

                # Process attrs of tag, e.g:
                # [('href', 'http://...'), ('title', 'Some link')] become
                # <a href='http://...' title='Some link'>
                str_attrs = ' '.join(['{}="{}"'.format(name, val) for name, val in attrs])

                # Append decoded tag and it's attrs into var html
                html += '<{}{}>'.format(
                    self._tag_map[tag],
                    (' ' if str_attrs else '') + str_attrs
                )
            elif type == 'endtag':
                # Append decoded end-tag into var html
                html += '</{}>'.format(self._tag_map[tag])
            elif type =='data':
                # Append data into var html
                html += data

        return html
```

Listing 3: Decoder for HTML-tag Compressing

I tested the encoder and decoder program using URI: `http://www.cs.odu.edu/`. Figure 5 shows the HTML output for tag-compressed `http://www.cs.odu.edu/` and figure 6 shows the HTML file after being decoded to its original tags.

```
<!--{"a": "html", "c": "meta", "b": "head", "e": "link", "d": "title", "g": "script", "f": "style", "i": "div", "h": "body", "k":
"u": "ul", "t": "strong", "w": "p", "v": "li"}-->


<a xmlns:st1="urn:schemas-microsoft-com:office:smarttags">
<b>
<c name="verify-v1" content="CXMn8RoyhZpl9fsKpbgxtiFw3kIdHD51r/ntbf1Rrcw=">
<c name="have-i-been-pwned-verification" value="683c2ed908b8c9136b82b07abeccef4d">
<d>Department Of Computer Science</d>
<c http-equiv="Content-Type" content="text/html; charset=windows-1252">
<c content="College of Sciences" name="Description">
<c content="College of Sciences" name="Keywords"><e title="style" href="files/style.css" type="text/css" rel="stylesheet">
<e title="style" href="files/screen.css" type="text/css" rel="stylesheet">
<f type="text/css">BODY {
    MARGIN: 7px 0px 0px
}
</f>
<g language="JavaScript">
function OnSubmitForm()
{
  if(document.myform.operation[0].checked == true)
  {
    document.myform.action ="http://www.google.com/search";
  }
  else
  if(document.myform.operation[1].checked == true)
  {
    document.myform.action ="search_user.php";
  }
  return true;
}
</g>

<g language="JavaScript" src="files/university.js"></g>

<g language="JavaScript" src="files/mm_menu.js"></g>

<g language="JavaScript" src="files/jquery.js"></g>

<g language="JavaScript" src="files/easySlider1.7.js"></g>

<g language="JavaScript">
$(document).ready(function(){
    $("#slider").easySlider({
        auto: true,
        continuous: true,
        pause: 7000,
        controlsShow: false
```

Figure 5: Encoding output (tag-compressed) for `http://www.cs.odu.edu/`

```
<head>
<meta name="verify-v1" content="CXMn8RoyhZpl9fsKpbgxtiFw3kIdHD51r/ntbf1Rrcw=">
<meta name="have-i-been-pwned-verification" value="683c2ed908b8c9136b82b07abeccef4d">
<title>Department Of Computer Science</title>
<meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
<meta content="College of Sciences" name="Description">
<meta content="College of Sciences" name="Keywords"><link title="style" href="files/style.css" typ
<link title="style" href="files/screen.css" type="text/css" rel="stylesheet">
<style type="text/css">BODY {
    MARGIN: 7px 0px 0px
}
</style>
<script language="JavaScript">
function OnSubmitForm()
{
  if(document.myform.operation[0].checked == true)
  {
    document.myform.action ="http://www.google.com/search";
  }
  else
  if(document.myform.operation[1].checked == true)
  {
    document.myform.action ="search_user.php";
  }
  return true;
}
</script>

<script language="JavaScript" src="files/university.js"></script>

<script language="JavaScript" src="files/mm_menu.js"></script>

<script language="JavaScript" src="files/jquery.js"></script>

<script language="JavaScript" src="files/easySlider1.7.js"></script>

<script language="JavaScript">
$(document).ready(function(){
    $("#slider").easySlider({
        auto: true,
        continuous: true,
        pause: 7000,
        controlsShow: false
```

Figure 6: Decoding output (tag-uncompressed) for `http://www.cs.odu.edu/`

Question 3.16

`Give a high-level outline of an algorithm that would use the DOM structure to identify content information in a web page. In particular, describe heuristics you would use to identify content and non-content elements of the structure.`

Answer

According to Lopez et al [9], content extraction could be done using CNR (Chars-nodes ratio) algorithm, which shows the relation between text content and tags content of each node in the DOM tree. CNR considers nodes as blocks where the internal information is grouped and indivisible using the DOM structure. CNR of an internal node takes into account all the texts and tags included in its descendants.

Steps for content extraction:

1. Compute the CNR for each node in the DOM tree.
   It ignores irrelevant code that should not be counted as text called "nonContentNode", for instance, nodes without text (e.g., img), nodes mainly used for menus (e.g., nav and a) and irrelevant nodes (e.g., script, video and svg). Computation of CNRs is done with a cumulative and recursive process that explores the DOM tree counting the text and descendants of each node. This step recursively obtains the CNR of each node starting at the root node of the DOM tree. At each node it adds three new attributes to the node with the computed weight (weight), the number of characters it contains (textLength), and the CNR (CNR). Figure 7 shows the algorithm to compute CNR.

2. Select those nodes with a higher CNR.

3. Starting from step 2, traverse the DOM tree bottom-up to find the best container nodes (e.g., tables, divs, etc.) that, roughly, contain as more relevant text as possible and less nodes as possible. Each of these container nodes represents an HTML block.
   In this step, it removes all the nodes in the set that are descendant of other nodes in the set. Then, it proceeds bottom-up in the tree by discarding brother nodes and collecting their parent until a fix point is reached. This process produces a final set of nodes that represent blocks in the webpage. Figure 8 shows the algorithm to identify main content blocks.

4. Choose the block with more relevant content. The final block contains more text (in the subtree rooted at that node).

Algorithm 1 Algorithm to compute chars-nodes ratios

Input: A DOM tree $T = (N, E)$ and the root node of T, $root \in N$
Output: A DOM tree $T' = (N', E)$

```
computeCNR(root)

function ComputeCNR(node n)
 case n.nodeType of
 "textNode":
    n.addAttribute('weight',1);
    n.addAttribute('textLength',n.innerText.length);
    n.addAttribute('CNR',n.innerText.length);
    return n;
 "nonContentNode":
    n.addAttribute('weight',1);
    n.addAttribute('textLength',0);
    n.addAttribute('CNR',0);
    return n;
 otherwise:
    descendants = 1;
    charCount = 0;
    for each child ∈ n.childNodes do
       newChild= ComputeCNR(child);
       charCount = charCount + newChild.textLength;
       descendants = descendants + newChild.weight;
    n.addAttribute('weight',descendants);
    n.addAttribute('textLength',charCount);
    n.addAttribute('CNR',charCount/descendants);
    return n;
```

Figure 7: Algorithm to compute CNR. Adapted from [9]

Algorithm 2 Identifying main content blocks

Input: A DOM tree $T = (N, E)$ and a set of nodes $S \subset N$
Output: A set of nodes $blocks \subset N$
Initialization: $blocks = S$

(1) $blocks = blocks \backslash \{b \mid (b' \to b) \in E^* \text{ with } b, b' \in blocks\}$
(2) while $(\exists n \in N \,.\, (n \to b), (n \to b') \in E \text{ with } b, b' \in blocks)$
(3) $\quad blocks = (blocks \backslash \{b \mid (n \to b) \in E\}) \cup \{n\}$

return $blocks$

Figure 8: Algorithm to identify main content blocks. Adapted from [9]

References

[1] Wikipedia. Craigslist. `https://en.wikipedia.org/wiki/Craigslist`, 2016. [Online; accessed 20-September-2016].

[2] Gustavo Zanini Kantorski and Carlos Alberto Heuser. Automatic filling of web forms. In *AMW*, pages 215–219, 2012.

[3] Gustavo Zanini Kantorski, Viviane Pereira Moreira, and Carlos Alberto Heuser. Automatic filling of hidden web forms: A survey. *SIGMOD Rec.*, 44(1):24–35, May 2015.

[4] Luciano Barbosa and Juliana Freire. Siphoning hidden-web data through keyword-based interfaces. *Journal of Information and Data Management*, 1(1):133, 2010.

[5] Bruce Croft, Donald Metzler, and Trevor Strohman. *Search Engines: Information Retrieval in Practice*. Addison-Wesley Publishing Company, USA, 1st edition, 2009.

[6] Leonard Richardson. Beautiful Soup. `https://www.crummy.com/software/BeautifulSoup/`, 2016. [Online; accessed 20-September-2016].

[7] Kenneth Reitz. Requests: HTTP for Humans. `http://docs.python-requests.org/en/master/`, 2016. [Online; accessed 20-September-2016].

[8] Python Software Foundation. HTMLParser — Simple HTML and XHTML parser. `https://docs.python.org/2/library/htmlparser.html`, 2016. [Online; accessed 18-September-2016].

[9] Sergio López, Josep Silva, and David Insa. Using the dom tree for content extraction. *arXiv preprint arXiv:1210.6113*, 2012.