Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2020) 28: 3389 – 3401

doi:10.3906/elk-2004-67

A regular expression generator based on CSS selectors for efficient extraction from HTML pages

Erdinç UZUN*
Department of Computer Engineering, Çorlu Faculty of Engineering, Tekirdağ Namık Kemal University, Tekirdağ, Turkey

Received: 09.04.2020 • **Accepted/Published Online:** 10.08.2020 • **Final Version:** 30.11.2020

Abstract: Cascading style sheets (CSS) selectors are patterns used to select HTML elements. They are often preferred in web data extraction because they are easy to prepare and have short expressions. In order to be able to extract data from web pages by using these patterns, a document object model (DOM) tree is constructed by an HTML parser for a web page. The construction process of this tree and the extraction process using this tree increase time and memory costs depending on the number of HTML elements and their hierarchies. For reducing these costs, regular expressions can be considered as a solution. However, preparing regular expression patterns is a laborious task. In this study, a heuristic approach, namely Regex Generator (REGEXN), that automatically generates these patterns through CSS selectors is introduced and the performance gains are analyzed on a web crawler. The analysis shows that regular expression patterns generated by this approach can significantly reduce the average extraction time results from 743.31 ms to 1.03 ms when compared with the extraction process from a DOM tree. Similarly, the average memory usage drops from 1054.01 B to 1.59 B. Moreover, REGEXN can be easily adapted to the existing frameworks and tools in this task.

Key words: Web data extraction, computational efficiency, regular expressions, heuristic algorithms

1. Introduction

A web page has different layouts containing menus, advertisements, banners, footers, sitemaps, title, summary, main text, price, description, reviews, images, etc. Obtaining these layouts has become very crucial for text processing applications such as search engines, sentiment analysis, recommendation systems, trend detection/monitoring, and e-commerce market monitoring. Many studies [1–4] in the literature focus on extracting parts of the title, summary, and main text from web pages automatically. Some studies [5, 6] are about obtaining the review part automatically. One study [7] focused on increasing the effectiveness and efficiency of the crawling process by separating the menu section into different groups. Most of these studies are about predicting these parts while ignoring the acceleration of the extraction process. Unlike other studies, this study introduces a heuristic approach that completes the extraction process in a shorter time and uses fewer resources.

Most web extraction tools known in this task use the HTML parsers that are based on the document object model (DOM) tree. Programming languages have several HTML parsers that can be utilized for this task. The following list presents several parsers for four well-known programming languages.

- Python: lxml, HTML Parser of The Standard Library, HTML5lib, HTML5-parser, AdvancedHTMLParser

- Node.js: Cheerio, Jsdom, HTMLparser2, Parse5

*Correspondence: erdincuzun@nku.edu.tr

- Java: jsoup, Lagarto, HTMLCleaner
- .Net: HAP (HTML Agility Pack), AngleSharp, Microsoft HTMLDocument

These parsers first create a DOM tree for a web page. CSS selectors or XML Path Language (XPath) are both capable of finding element/s containing data on this tree for the extraction process. Uzun et al. [8] compare three different well-known .NET parsers, including HAP[1], AngleSharp[2], and Microsoft (MS) HTMLDocument[3] to extract data from web pages. They use XPath patterns for this task. Uzun et al. [9] try three different parsers (HTML5parser, HTML5lib, lxml) and regular expression patterns in Python. lxml yields the best results as suggested in the Python community[4]. The extraction time of regular expression patterns prepared manually is much better than these parsers. However, only 43.5% of the regular expression patterns are suitable for the correct extraction process because the closing tag is not known. In this study, an approach that solves the ambiguity of the closing tag and automatically generates regular expression patterns is developed. Moreover, the extraction time results of lxml and regular expression patterns are compared in order to understand the time efficiencies of extraction methods.

A regular expression is a search pattern that is utilized by string searching algorithms in order to find, replace, or validate operations on strings. Regular expressions are a traditional search and replace technique used in word processors and text editors. Many programming languages allow regular expressions capabilities for performing these techniques. Moreover, web developers often prefer regular expressions for the client and server-side for input validation. Apart from these techniques, regular expressions can be also used naturally to extract data from web pages. Preparing a regular expression suitable for this task is a monotonous and error-prone operation [10]. For this reason, preparing this expression automatically can be considered as a solution. In the literature, several approaches for this task have been proposed for solving different issues. Bartoli et al. [11, 12] use annotated examples in text for generating regular expressions. Kushman and Barzilay [13] translate natural language text queries into regular expressions obtained from Amazon Mechanical Turk. Locascio et al. [14] use an LSTM-based sequence to sequence a neural network for specialized domain knowledge. Flores et al. [15] develop an algorithm for automatically generating regular expressions from biomedical texts using a coarse-to-fine text aligning method. Cui et al. [16] design an efficient novel regular expression-based text classifier. This classifier is tested on real-world medical data. All of these studies utilize different data sets that are not suitable for our study. Besides, they need corpus and more examples. Zhang et al. [17] deal with noisy web data, but they perform traditional regular expression operations including entity types in web documents, such as dates, times, email addresses, and course numbers. Our study focuses on the extraction of data between HTML tags. Moreover, a heuristic approach has been developed that can be used for web data extraction and this approach needs only one example for a website.

The rest of the study is organized as follows. The second section gives information about HTML, DOM tree, CSS selectors, regular expressions, and the shortcomings of regular expressions. The third section introduces our heuristic approach that converts a CSS selector to a regular expression pattern. The fourth section is dedicated to the experiments for comparing our heuristic approach with traditional extraction techniques and assessing the efficiencies of our approach. The last section provides our conclusion and future studies.

[1]HTML Agility Pack (HAP) (2020). An HTML parser written in C Sharp programming language [online]. Website https://HTML-agility-pack.net/ [accessed 22 August 2020].

[2]AngleSharp (2020). Explore the DOM [online]. Website https://anglesharp.github.io/ [accessed 22 August 2020].

[3]MS HTMLDocument (2020). Programmatic access to an HTML document [online]. Website https://docs.microsoft.com/en-us/dotnet/api/system.windows.forms.HTMLdocument?view=netframework-4.8 [accessed 22 August 2020].

[4]Beautiful Soup (2020). Python library for pulling data out of HTML files [online]. Website https://www.crummy.com/software/BeautifulSoup/bs4/doc/#installing-a-parser [accessed 22 August 2020].

2. Background

An HTML document contains content including HTML tags and other strings. A web browser renders the document and constructs the DOM tree for accessing/manipulating this content. CSS selectors are widely used for accessing operation. Moreover, regular expressions can be used for this task. However, regular expressions have some shortcomings for obtaining the appropriate content. This section covers technical information on these topics.

2.1. An HTML document

An HTML document includes HTML elements, $E = (e_1, e_2, , e_n)$ that are related with each other. An HTML element has three parts: opening tag (OT), content (C), and closing tag (CT) as represented $e_x = (OT_x, C_x, CT_x)$ where x is one element in the HTML elements. C may also contain inner HTML elements. All tags, $T = (t_1, t_2, , t_n)$, are maintained by the World Wide Web Consortium (W3C) organization that is an international community responsible for developing the Web standards. The opening tag of an element may have attributes (A) and their values (V) that are defined as $OT_x = (t_x, a_1 = v_1, a_2 = v_2, , a_n = v_n)$. Attributes (a) such as id, class, and itemprop provide extra information about elements. The closing tag of an element has only tag information that is defined $CT_x = (t_x)$. As a result, an HTML element can be expressed as follows:

$$e_x = ((t_x, a_1 = v_1, a_2 = v_2, , a_n = v_n), C_x, t_x) \tag{1}$$

2.2. A DOM tree

A web browser parses E in an HTML document and renders this document. The web browser's first task in the rendering process is to construct the DOM tree that is a hierarchical representation of an HTML document. Programming languages can access E on this tree. In web data extraction, researchers mostly use this technique. Each branch of this tree ends in e_x, and each e_x contains content or other e_x.

Figure 1 shows the simple layout example from an HTML document and the DOM tree constructed from this document.

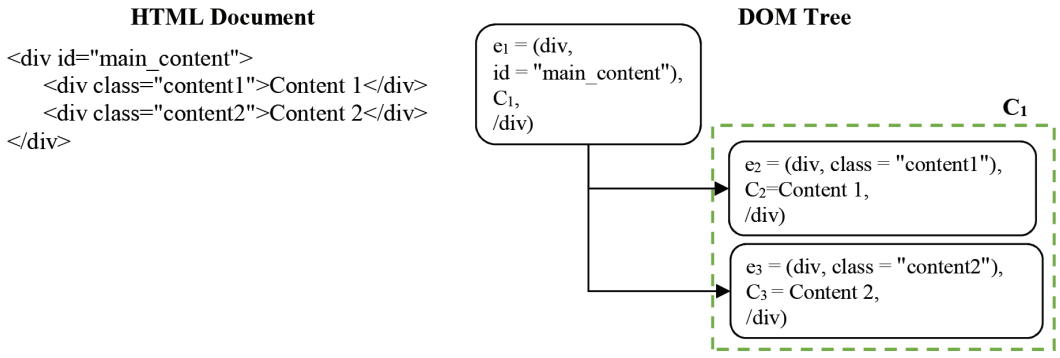

Figure 1. Simple layout example from an HTML document and its DOM tree.

2.3. CSS selectors

CSS selectors are patterns used to access the e_x with a tag name, some attributes and their values. Table 1 shows CSS selectors for accessing the elements or its content in Figure 1.

Table 1. CSS selectors and its results in Figure 1.

CSS selector	Result
div#main_content	e_2 and e_3
div.content1	Content 1
div.content2	Content 2

In Table 1, *div#main_content* pattern is used to select the element with tag *name* = *div* and *id* = ''main''. div.content1 is utilized to select elements with tag *name* = *div* and *class* = ''main''. The id attribute identifies a unique element, whereas a class attribute can be utilized to specify more than one. The id and class are widely used attributes. However, there are standard attributes such as accesskey, align, background, bgcolor, class, height, hidden, item, itemprop, style, subject, and width that can be used for accessing elements. Moreover, custom data attributes are defined in HTML5 that is a new property. These attributes and their values are expressed in square brackets. For example, *div*[*itemprop* = ''test''] pattern is used to select all div elements which contain *itemprop* = ''test''.

2.4. Regular expressions

Regular expressions can be utilized to match text strings including phone numbers, email addresses, dates, IP addresses, and HTML tags. Manual preparation of a regular expression pattern is a widely practical solution to extract the desired data from a text. Preparing precise regular expression patterns requires experts who know the data. In order to prepare regular expression patterns for web data extraction, there is a need for an expert who knows the HTML tags and the structure of the web page. Table 2 indicates the regular expression patterns for extracting data and its results.

Table 2. Regular expression patterns and its results in Figure 1.

No.	Regular expression pattern	Output data
1	<div\sclass=''content1''>(.*?)	Content 1
2	<div\sclass=''content2''>(.*?)	Content 2
3	<div\sid=''main_content''>(.*?)	<div class=''content1''>Content 1 <div class=''content2''>Content 2

In Table 2, a space character \s will match any of the specific spaces. For extracting data, the special parentheses (and) metacharacters are used for defining groups of characters and capturing them. The dot . is a special character utilized to match any one character. The character $*$ matches the preceding character zero or many times. The character ? means "match zero or one occurrence of the regular expression". In Table 2, an expert developer appends one closing tag ( ) to example no. 1 and 2 in order to obtain the appropriate output while in example no. 3, a single closing tag is not enough. Therefore, this expert adds two closing tags ( ) for achieving the right output. If this expert makes a mistake during the preparation of this pattern, the output data is "</div class="content1"> Content 1" because the number of opening and closing tags is crucial for proper web data extraction. Moreover, preparing a CSS selector pattern is much easier than preparing a regular expression pattern for a developer. Therefore, developers and tools prefer DOM-based extraction for this task.

3. Heuristic approach: REGEXN

Traditional regular expression generators [10–17] focus on trying all variations to obtain the most suitable pattern and ignore time efficiency. Moreover, these generators are suitable for different tasks. In this study, a heuristic approach, namely Regex Generator (REGEXN), is proposed to speed up the process of generating a regular expression pattern and extract data from web pages. Figure 2 compares the traditional web data extraction with REGEXN approach in two separate parts. The traditional web data extraction needs an HTML parser to construct a DOM tree and find element/s using a CSS Selector in this tree. Moreover, Figure 2 shows REGEXN approach that generates a regular expression pattern using a CSS Selector and an example web page for improving the time and memory costs of this task.

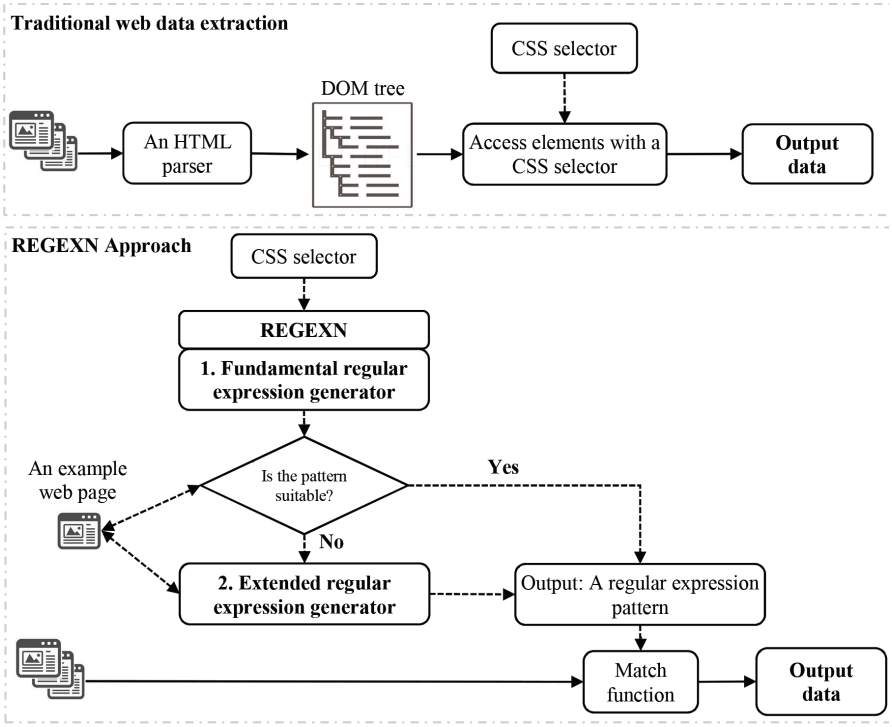

Figure 2. The flowcharts of traditional web data extraction and REGEXN approach.

REGEXN has two main steps for this task. The first step, namely fundamental regular expression generator, generates a candidate pattern. The second step, namely extended regular expression generator, checks whether this pattern is suitable or not through an example web page. If this pattern is not suitable, the second step modifies this pattern according to this example web page. On the other hand, the pattern obtained from the first step is used for this task.

3.1. First step: fundemantal expression generator

In this generator, four static patterns are prepared to express different situations. These patterns and their task are given in Table 3.

Square brackets [ ] define a character set to match only one out of several characters. Minus character − (between $a-z$, $A-Z$, $0-9$) specifies a range of characters. These patterns are used in Algorithm 1 that parses a given CSS Selector and generates a regular expression pattern for this selector.

Table 3. Patterns used in Algorithm 1.

Variable name	Pattern	Description
pattern_anycharacter	$[\backslash s\ a-zA-Z0-9\_-]*?$	Zero or more occurrences of a pattern including space, lower letters, upper letters, numeric characters, underscore, minus
pattern_space	'\s*'	One or more than one spaces
pattern_start	\s*?[' ']	Zero or more spaces and single or double quotation
pattern_end	[' '].*?	Single or double quotation and zero or more any characters

Algorithm 1: Fundemantal Expression Generator

Data: css_selector: A CSS Selector
Result: regex: A regular expression pattern

```
pos_id = find(#) in css_selector
pos_class = find(.) in css_selector
pos_other_attributes = find([) in css_selector
regex = Empty
if pos_class != -1 or pos_id != -1 or pos_other_attributes != -1 then
    d = 'id': pos_id, 'class': pos_class, 'other': pos_other_attributes
    d = eliminate(d.keys) where key < 0
    key_min = min(d), val = d[key_min]
    tagname = subtring of css_selector between 0, val
    regex = '<' + tagname + pattern_space
    while len(d) > 0 do
        pop(key_min) from d
        if len(d) > 0 then temp_key_min = min(d), temp_val = d[temp_key_min]
        else temp_val = last char index of ccs_selector
        if key_min = 'id' or key_min = 'class' then temp_css = subtring of css_selector between
          val+1, temp_val
        else temp_css = subtring of css_selector between val+1, find(]) in css_selector
        if key_min = 'id' or key_min = 'class' then
            if key_min = 'id' then rep_char = '#' else rep_char = '.'
            regex += key_min + '=' + pattern_start + pattern_anycharacter
            regex += replace(rep_char, pattern_anycharacter) in temp_css
            regex += pattern_anycharacter + pattern_end
        else
            regex += replace(''or ', pattern_start) in temp_css
        end
        if len(d) > 0 then key_min = temp_key_min, val = temp_val
    end
else
    regex = '<' + css_selector + '>(.*?)</' + css_selector + '>'
end
return regex
```

In Algorithm 1, the find method determines if the character ("#", ".", or "[", i.e. id attribute, class attribute, or other attributes of ccs_selector) occurs in css_selector, it returns the index of the first occurrence of css_selector. Otherwise, it returns –1 that means the given character is not found [Line 1, 2, 3]. If css_selector does not contain any attribute, the tag name is utilized for preparing a regular expression pattern [Line 28]. Otherwise, the algorithm generates a proper pattern for attribute/s. d is the dictionary that holds key-value pair whereas each key-value pair maps the key to its associated value [Line 6]. Keys can be id, class, or other attributes and its value is the index of the attribute in css_selector. A value is obtained from a dictionary by determining its corresponding key in square brackets [Line 8, 13]. The eliminate method removes attribute values less than 0 from d [Line 7]. The min method finds the smallest value in d and returns the key [Line 8, 13]. The substring method extracts the characters from css_selector, between two specified indices, and returns a new string [Line 9, 15, 16]. temp_val variables are used in the substring method [Line 13, 14, 15]. The replace method searches a character (#, ., ', '') for the part of ccs_selector, and returns a regular expression pattern where the specified characters are replaced [Line 20, 23]. Moreover, some patterns append to this regular expression pattern [Line 19, 21]. Table 4 shows the output generated against a given CSS selector in Algorithm 1.

Table 4 indicates that Algorithm 1 is sufficient for simple web data extraction. However, the second example in Table 4 is not proper for HTML Document in Figure 1. The web data extraction process is incorrect for this example because the number of opening and closing tags is not appropriate.

Table 4. Example CSS selectors and its outputs obtained from Algorithm 1.

CSS selectors	Pattern
div	(.*?)
div#main_content	<div\s*id=\s*?[' '][\sa-zA-Z0-9_-]*?main_content[\sa-zA-Z0-9_-]*?[' '].*?>(.*?)
*div.content*1	<div\s*class=\s*?[' '][\sa-zA-Z0-9_-]*?content1[\sa-zA-Z0-9_-]*? [' '].*?>(.*?)
*div.content*2	<div\s*class=\s*?[' '][\sa-zA-Z0-9_-]*?content2[\sa-zA-Z0-9_-]*?[' '].*?>(.*?)

3.2. Second step: extended regular expression generator

The extended regular expression generator consists of two complementary algorithms. The first algorithm (Algorithm 2) determines whether the end pattern needs to be changed and returns the position (index) of the appropriate end pattern. That is, it finds the correct closing tag. The second algorithm (Algorithm 3) modifies the pattern obtained in Algorithm 1 according to the closing tag produced by Algorithm 2.

In Algorithm 2, there are two inputs as pattern obtained from Algorithm 1 and HTMLdoc is an HTML document. Algorithm 2 parses the pattern and carries out three variables: element_name contains tag name and attributes [Line 1], start_tagname is the tag of element_name [Line 2], and end_tagname is the closing tag of element_name [Line 3]. For example, div#main_content in Table 4, element_name is <div id=''main_content''>, start_tagname is <div, and end_tagname is </div for the generated pattern from Algorithm 1. The find method in Algorithm 2 [Line 4, 6, 14] returns an integer value that is the lowest index of the substring if it is found in given string. This method searches the string value to be searched in HTMLdoc. Moreover, this method searches from the starting index. The algorithm continues until the number of opening tags equals (start_tag) the number of closing tags (end_tag). The count method returns the number of occurrences of an opening tag in the substring obtained from HTMLdoc [Line 11]. The algorithm returns

Algorithm 2: Find the position of an ending HTML tag in a given web page

```
Data: pattern: obtained from Algorithm 1, HTMLdoc: a web page
Result: end_index: the position of a unique ending pattern, innertag: Boolean value
1  element_name = resolve from pattern
2  start_tagname = resolve from tagname
3  end_ tagname = append / character to start_tagname
4  start_index = find(element_name, 0) in HTMLdoc
5  if start_index != -1 then
6      end_index = find(end_ tagname, start_index + len(pattern)) in HTMLdoc + len(end_
        tagname)
7      innertag = False, start_tag = 0, end_tag = 0, result_tmp = Empty
8      while True do
9          sub_tmp = substring of HTMLdoc between start_index, end_index
10         end_tag += 1
11         start_tag += count of start_tagname in sub_tmp
12         if start_tag != end_tag then
13             start_index = end_index
14             end_index = find(end_tagname, start_index + len(pattern)) in HTMLdoc +
                len(end_tagname)
15             innertag = True
16         else
17             return end_index, innertag
18         end
19     end
20 end
```

two outputs: end_index is the index of the correct closing tag and innertag is a Boolean value that indicates whether the pattern contains an inner tag. If innertag is False, it means that the current pattern is suitable for web data extraction. Algorithm 2 is the first step of Algorithm 3 that finds a unique ending pattern for HTMLdoc.

In Algorithm 3, if innertag value is True, the algorithm continues to find the end character (>) of a tag in HTMLdoc that starts at start_index [Line 3] and append the substring obtained from HTMLdoc to end_pattern [Line 5]. If there is only one end_pattern in HTMLdoc [Line 8], this string can be used as an end pattern for a given pattern. Table 5 indicates the pattern results for three main algorithms of REGEXN.

According to the sample web page in Figure 1, the output produced by Algorithm 1 in Table 5 is not sufficient for the CSS selector: div#main_content. Algorithm 2 finds the appropriate closing tag index value using the web page (HTMLdoc). Algorithm 3 tries to append tags and create a unique closing end pattern. In this example, the end pattern is $.</div>$ for this pattern and the web page in Figure 1. Finally, Algorithm 3 generates a new pattern that is suitable for web data extraction.

4. Experiments

In this section, firstly general information about the dataset[5] and extraction patterns is given. The second subsection is dedicated to the results of extraction time and memory usage for a DOM-based approach. The

[5]REGEXN Dataset (2020) [online]. Website https://adys.nku.edu.tr/Datasets/regexn.zip [accessed 22 August 2020].

Algorithm 3: Find a unique ending pattern in a given web page

```
Data: pattern: obtained from Algorithm 1, HTMLdoc: a web page
Result: new pattern
end_index, innertag = Algorithm 2(pattern, HTMLdoc)
if innertag == False then return pattern
start_index = end_index + 1
while True do
    end_index = find(>, start_index) in HTMLdoc
    if end_index != -1 then
        end_pattern += substring of HTMLdoc between start_index, end_index
        if count end_pattern in HTMLdoc = 1 then
            end_pattern = end_pattern.replace(" ", \"s").replace("\n",".")
            return pattern + end_pattern
        else
            start_index = end_index + 1
        end
    end
end
```

Table 5. The outputs of Algorithm 1, Algorithm 2 and Algorithm 3 for the given CSS selector

CSS Selector	div#main_content
Algorithm 1	<div\s*id=\s*?[' "][\sa-zA-Z0-9_-]*?main_content[\sa-zA-Z0-9_-]*?[' "].*?>(.*?)
Algorithm 2	110, True
Algorithm 3	<div\s*class=\s*?[' "][\sa-zA-Z0-9_-]*?content2[\sa-zA-Z0-9_-]*?[' "].*?>(.*?) .

third subsection covers these results for our heuristic approach. A crawler[6] is developed in Python 3 and all experiments are carried out on an Intel Core i7-8550U 16 GB RAM computer with Windows 10 operating system.

4.1. Dataset and CSS selectors

For testing extraction techniques, we need more than one web page per website. Therefore, a crawler has been prepared to gather web pages of a website. This crawler downloaded 51 web pages for 50 different websites that contain different languages including Turkish, Bosnian, English, Spanish, German, Albanian and fields such as article, dictionary, health, movie, newspaper. Moreover, 4-5 CSS selectors were prepared for each website. The statistical results obtained from web pages and CSS selectors/Patterns are given in Tables 6 and 7, respectively.

Table 6 shows that the average size of a web page is about 111,800 B and contains about 1623 HTML tags. The average number of a tag, which is used to link from one page to another, is about 174 hyperlinks for a web page. The average number of div tags, which defines a division or a section in an HTML document, is about 196 tags for a web page.

In Table 7, 241 CSS selector patterns, which are suitable for a given website, are prepared to obtain data from the web page. Table 7 contains information about these patterns. In addition, Table 7 has information about regular expression patterns obtained from Algorithm 1 and Algorithm 3. In this table, the HTML tags are grouped by the amount of data. The tags of div, main, section, table, body (Group 1) contain the most

[6]REGEXN (2020). Source codes [online]. Website https://github.com/erdincuzun/REGEXN [accessed 22 August 2020].

Table 6. Dataset obtained from the crawling process.

	File size (bytes)	Number of the HTML tags	Number of 'a' tags	Number of 'div' tags
Average	111,800	1623.60	173.94	195.48
Minimum	93,861	2242	145	75
Maximum	4,438,682	3566	1263	4492
Standard deviation	8927.5	967.00	262.00	64.00

Table 7. Information about CSS selectors prepared by experts and patterns generated by REGEXN.

HTML tag/s	Number of HTML tags	Number of attributes	ACL of CSS selector	ACL of REP	The First step is enough	The second step is needed
Group 1	107	131	17.38	124.46	39	68
Group 2	73	80	14.45	82.66	72	1
Hyperlink	61	0	1.00	33.00	61	0
	241	211	12.35	88.65	172	69
Group 1: div, main, section, table, body Group 2: h1, h2, h3, h4, h5, h6, ul, header, p, span, strong Hyperlink: a ACL: average character length REP: regular expression patterns						

amount of data, while other tags contain much less. Group 2 contains data consisting of few characters such as title, author name, day, and time. Hyperlink, a, is used to link from one page to another. Attributes in HTML are additional values that are often used to extract the proper data in the web data extraction process. In our CSS selectors, there are 211 attributes. A CSS Selector pattern consists of an average of 12 characters. On the other hand, regular expression patterns obtained from our generator contain about 89 characters. This generator has two steps for this task. While only a CSS selector pattern is sufficient for Step 1, a sample web page is also needed for Step 2. In Table 7, Step 2 should be applied for 69 CSS selectors whereas Step 1 is adequate for 172 CSS selectors.

4.2. The results of DOM-based approach

Most of the well-known programming languages have HTML parsers for operations on web pages. In this study, lxml is preferred as a Python Library recommended by the Python community for web data extraction as a well-known solution. The popular frameworks and tools use this library for this task. For example, Scrapy[7] is a popular framework for crawling websites and extracting web data from them. For extraction operation, this framework uses lxml library that supports CSS selectors. Moreover, the results of this library are the baseline experiment for our study. Table 8 indicates the average memory usage and extraction times obtained from the dataset for this library.

lxml is a DOM-based approach, so it primarily constructs a DOM tree. In Table 8, it increases the memory usage by an average of 1053.96 B. On the other hand, it takes an average creation time of 278.88 ms to construct a DOM tree for all web pages in the dataset. Since Group 1 contains more data, an average of

[7]Scrapy (2020). An open-source and collaborative framework for extracting the data [online]. Website https://scrapy.org/ [accessed 22 August 2020].

Table 8. Results of the average memory usage and extraction time for lxml.

	Average memory (Byte)	Average extraction time (ms)	Max	Min	Standard deviation
DOM tree construction	1053.96	278.88	1530.93	201.74	71.73
Group 1	0.07	520.48	3259.61	5.62	234.64
Group 2	0.01	437.05	3553.86	5.62	201.52
Hyperlinks	0.28	242.46	4692.38	6.27	359.86
CSS selectors with No Attribute	0.11	271.36	4692.38	6.27	228.99
CSS selectors with more than one attribute	0.04	499.50	3553.86	5.62	227.20
All extraction results	0.05	464.43	4692.38	5.62	231.15

520.48 ms is obtained in terms of extraction time. In Group 2, this time decreases to 437.05 ms. While the amount of memory used in Group 1 and Group 2 varies slightly, more memory is required to obtain hyperlinks. This is because a web page contains many data. However, since a hyperlink consists of a few characters, the average extraction time is 242.46 ms. Some CSS selectors do not contain any attributes. Using CSS selectors with no attribute allows very fast access to CSS selectors with multiple attributes. The reason for this is the number of characters in the result. As a result, a DOM-based approach requires about 278.88 ms for DOM tree construction and about 464.43 ms for an extraction process. This indicates that it needs an average time of 743.31 ms for an extraction process.

4.3. The results of REGEXN

REGEXN does not need a hierarchical structure like a DOM tree. It can be applied directly to the text. Table 9 shows that the improvements provided by REGEXN are much better than DOM-based approaches.

Table 9. Results of the average memory usage and extraction time for REGEXN.

	Average memory (Byte)	Average extraction time (ms)	Max	Min	Standard Deviation
Group 1	2.54	1.13	34.44	0.80	0.74
Group 2	0.37	0.93	5.58	0.80	0.13
Hyperlinks	4.23	0.92	1.53	0.80	0.12
Algorithm 1	0.88	0.96	5.58	0.80	0.18
Algorithm 3	3.38	1.20	34.44	0.81	0.93
All Extractions	1.59	1.03	34.44	0.80	0.53

Table 9 shows that REGEXN uses more memory during extraction when compared with the DOM-based approach. However, the DOM-based approach uses more memory during the construction of a DOM tree. On the other hand, according to the DOM-based approach in terms of time efficiency, the REGEXN technique appears to be significantly faster. As with the DOM-based approach, REGEXN is affected by the number of characters.

REGEXN has two steps introduced in Section 3 to generate a regular expression pattern. Step 1 is sufficient for situations that do not contain more than one closing tag against a start tag. However, if there is

more than one tag, it is necessary to find the appropriate closing tag on a sample web page. Step 2 decides the appropriate closing pattern. In Step 2, REGEXN appends a new pattern to the result obtained from Step 1. In other words, the number of characters in the regular expression pattern increases. As expected, this situation affects the extraction time. However, REGEXN still has a very good extraction time than the DOM-based approach. Consequently, REGEXN is about 722 times better than the DOM-based approach.

The dataset used in this study consists of well-formed HTML documents. Professional websites give importance to their HTML codes, and the error in these codes is low. However, some websites may contain some errors caused by web designers. Some HTML parsers have repair mechanisms for fixing these errors. lxml, used in this study, has recover mode for parsing through broken HTML. However, this mode is set to false in order to avoid wasting time. Similarly, there is no error correction mechanism in REGEXN. This issue can be considered as future work for fixing the error of a certain text in web pages.

5. Conclusion

Most previous studies and tools on web data extraction ignore time efficiency. They use traditional methods based on a DOM tree which is a common solution in web applications. This solution takes all elements in a web page into account. Our experiments show the average memory consumption and the results of average extraction time when using this solution. Regular expression can be considered as another solution for improving these results. However, preparing a regular expression pattern is a time-consuming operation. Moreover, regular expression patterns are more complex than DOM-based approaches. In DOM-based approaches, CSS selectors make it very easy to access elements in the DOM tree. A CSS selector pattern is much simpler and easier to write. In this study, REGEXN, which converts CSS Selector patterns to regular expression patterns, is introduced. REGEXN gives better extraction time efficiency than the DOM-based approach with approximately 722 times. Moreover, the memory usage of REGEXN is about 663 times better than the DOM-based approach.

The amount of data on the Internet is increasing rapidly. It has become an important issue to crawl web pages and scrap this data quickly. REGEXN is an open-source project and available via the web page https://github.com/erdincuzun/REGEXN. It can easily be integrated into other web data extraction tools for improving the time efficiency of the extraction operation.

This study is about text on web pages. Ajax-based web applications update a web page without reloading the page and need a DOM tree of a web page. Crawling Ajax-based web applications [18] is a popular issue in the literature. Traditional solutions [19] require an embedded browser to execute Ajax-requests. These solutions are expensive to execute, in terms of time and network traffic. In the future, we plan to solve these requests without rendering a web page. Moreover, we will develop an effective and efficient web scraper that can create datasets automatically for different purposes.

References

[1] Uzun E, Agun HV, Yerlikaya T. A hybrid approach for extracting informative content from web pages. Information Processing & Management 2013; 49(4): 928-944. doi: 10.1016/j.ipm.2013.02.005

[2] Ferrara E, Meo PD, Fiumara G, Baumgartner R. Web data extraction, applications and techniques: A survey. Knowledge-Based Systems 2014; 79: 301-323. doi: 10.1016/j.knosys.2014.07.007

[3] Liu Q, Shao M, Wu L, Zhao G, Fan G et al. Main content extraction from Web Pages based on Node Characteristics. Journal of Computing Science and Engineering 2017; 11(2): 39-48. doi: 10.5626/JCSE.2017.11.2.39

[4] Alarte J, Silva J, Tamarit S. What Web template extractor should I use? A Benchmarking and Comparison for Five Template Extractors. ACM Transactions on the Web 2019; 13(2): 1-19. doi: 10.1145/3316810

[5] Suleman K, Vechtomova O. Discovering aspects of online consumer reviews. Journal of Information Science 2016; 42(4): 492-506. doi: 10.1177/0165551515595742

[6] Uçar E, Uzun E, Tüfekci P. A novel algorithm for extracting the user reviews from web pages. Journal of Information Science 2017; 43(5): 696-712. doi: 10.1177/0165551516666446

[7] Uzun E, Güner ES, Kılıçaslan Y, Yerlikaya T, Agun HV. An effective and efficient web content extractor for optimizing the crawling process. Software: Practice and Experience 2014; 44(10): 1181-1199. doi: 10.1002/spe.2195

[8] Uzun E, Buluş HN, Doruk A, Özhan E. Evaluation of Hap, AngleSharp and HTMLDocument in web content extraction. In: International Scientific Conference UNITECH; Gabrovo, Bulgaria; 2017. pp. 275-278.

[9] Uzun E, Yerlikaya T, Kırat O. Comparison of Python libraries used for Web data extraction. In: 7th International Scientific Conference "TechSys 2018" – Engineering, Technologies and Systems; Plovdiv, Bulgaria; 2018. pp. 108-113.

[10] Friedl JEF, Oram A. Mastering Regular Expressions. 2nd ed. Sebastopol, CA, USA: O'Reilly, 2002.

[11] Bartoli A, Davanzo G, Lorenzo AD, Medvet E, Sorio E. Automatic Synthesis of regular expressions from examples. Computer 2014; 47(12): 72-80. doi: 10.1109/MC.2014.344

[12] Bartoli A, Lorenzo D, Medvet E, Tarlao F. Inference of regular expressions for text extraction from examples. IEEE Transactions on Knowledge and Data Engineering 2016; 28(5): 1217-1230. doi: 10.1109/TKDE.2016.2515587

[13] Kushman N, Barzilay R. Using semantic unification to generate regular expressions from Natural Language. In: Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies; Atlanta, GA, USA; 2013. pp. 826-836.

[14] Locascio N, Narasimhan K, Leon ED, Kushman N, Barzilay R. Neural generation of regular expressions from Natural Language with Minimal Domain Knowledge. In: Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing; Austin, TX, USA. 2016. pp. 1918-1923.

[15] Flores CA, Figueroa RL, Pezoa JE, Zeng-Treitler Q. CREGEX: A biomedical text classifier based on Automatically Generated Regular Expressions. IEEE Access 2020; 8: 29270-29280. doi: 10.1109/ACCESS.2020.2972205

[16] Cui M, Bai R, Lu Z, Li X, Aickelin U et al. Regular expression based medical text classification using constructive heuristic approach. IEEE Access 2019; 7: 147892-147904. doi: 10.1109/ACCESS.2019.2946622

[17] Zhang S, He L, Vucetic S, Dragut E. Regular expression guided entity mention mining from Noisy Web Data. In: Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing; Brussels, Belgium. 2018. pp. 1991-2000.

[18] Khalid S, Khusro S, Ullah I. Crawling Ajax-Based Web Applications: Evolution and state-of-the-art. Malaysian Journal of Computer Science 2018; 31(1): 35-47. doi: 10.22452/mjcs.vol31no1.3

[19] Fayzrakhmanov RR, Sallinger E, Spencer B, Furche T, Gottlob G. Browserless web data extraction: challenges and opportunities. In: WWW'18: Proceedings of the 2018 World Wide Web Conference on World Wide Web; Lyon, France. 2018. pp. 1095-1104. doi: 10.1145/3178876.3186008